Pattern Recognition Project

Apartment Rent Prediction

Presented by:

(Team ID: CS_9)

Name	ID	Section
Mina Edwar Dawood Elias	2021170565	7
Nada AbdElmoneem Ahmed	2021170583	7
Nardine Guirguis Edward Amin	2021170576	7
Youssef Emad El-din Ibrahim	2021170640	8
Youssef Hany Ezzat Aly	2021170653	8
Dalia AbdElazim Mohamed	2021170179	3

Contents

1.Introduction

1.1 Overview

In an era we are living in right now that is characterized by the unlimited access to data and computational power, The help of machine learning has become very important in various fields. This model presents an overview of Apartment Rent Prediction using different machine learning algorithms. We applied those algorithms on a real-world task using Apartment Rent Dataset.

In this task we aim to predict the rent category of apartment depending on some features inside the dataset used. We have faced a lot of challenges dealing with this data as it has some missing values, unclear formats …etc. We applied different machine learning algorithms dealing with each one to provide the best accuracy it can provide. Finally we chose the best 2 models with the best accuracy provided to start using it to get the best predictions.

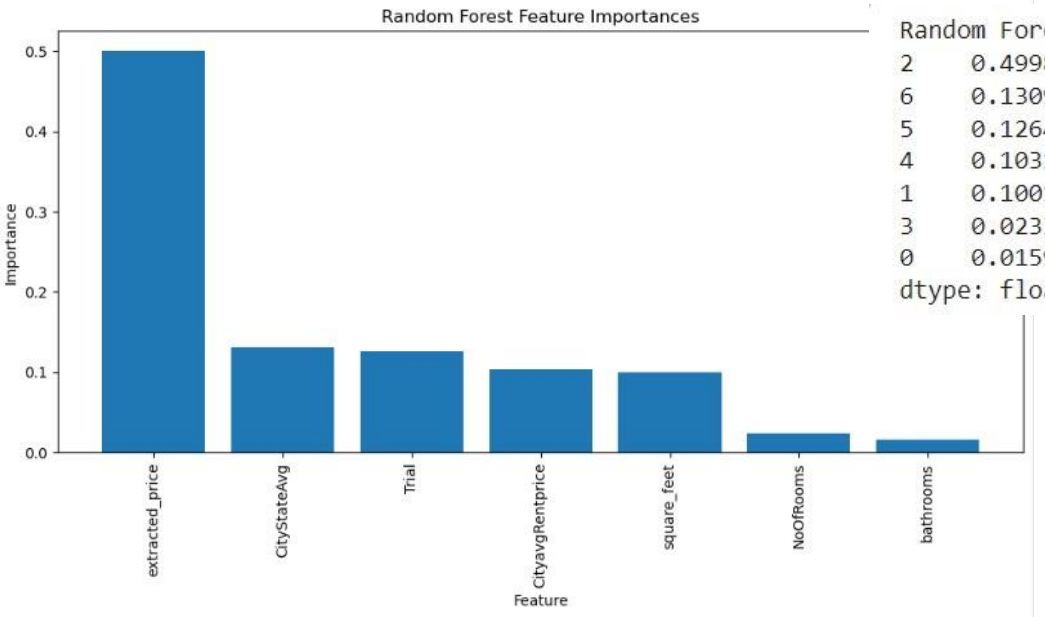
2.Data Preprocessing

2.1 Train Test Split

- we started to split the data to 70% for training and 30% for testing by using the train_test_split and giving it test_size=0.3 why 30%? because we had 9k rows.

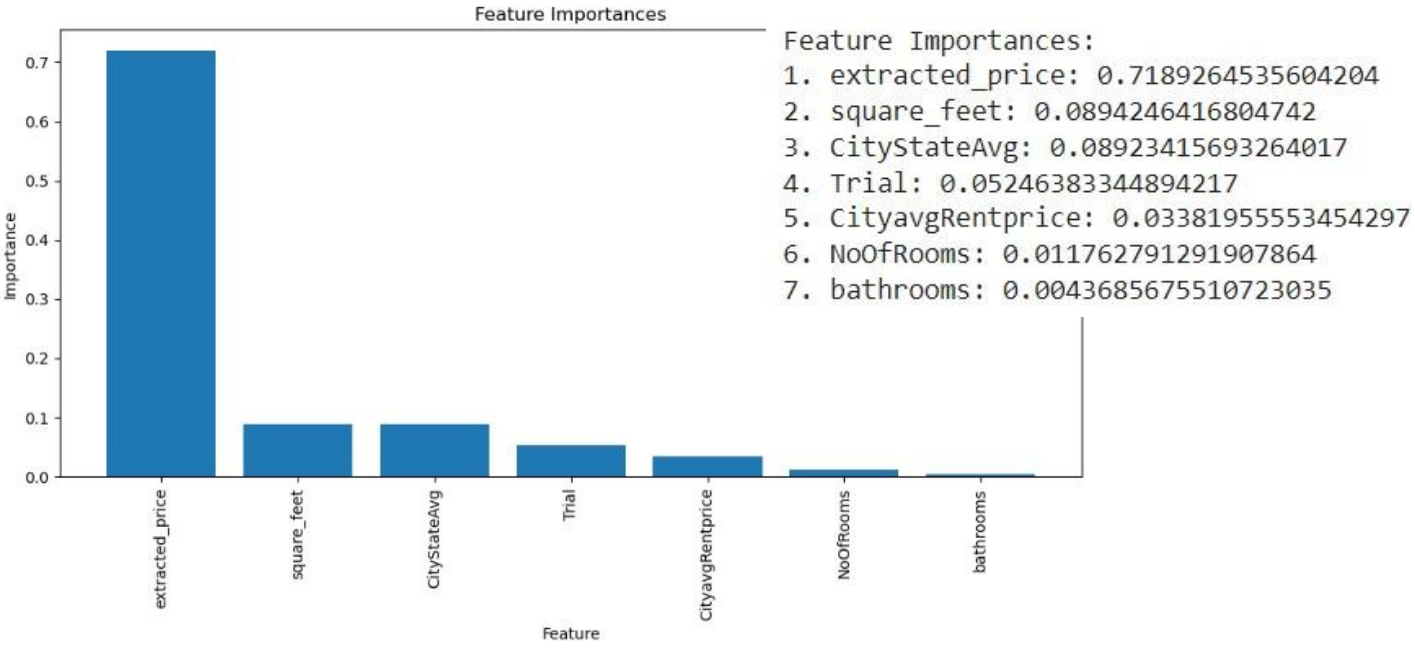
```
## 80% Train , 20% Test Mandatory ##
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,random_state=42)
```

After splitting data we apply **Random forest classification** for the most important features by calculate feature importance for each one , sorting all of this , print and plotting .




```
Random Forest feature importance:
2    0.499880
6    0.130944
5    0.126445
4    0.103151
1    0.100186
3    0.023394
0    0.015999
dtype: float64
```

Then we apply **Decision tree classifier** for the most important features



Finally, we are ready to use the data in our machine learning models.

2.2 Data Loading

We started our project by loading the dataset which is ApartmentRentPrediction.csv a comma separated values file into our environment in a pandas data frame (df) using the method read_csv().

```
######################Nulls Detection#######
df.isna().sum()
```

2.3 Data Exploration

We started printing our data frame now to see number of columns and rows of the data and some samples of them. We found that our data contains 9000 rows and 21 columns.

Diving into the dataset we started to get the number of unique values in each column which can be useful later. After that we started to check the data type of each column and we found that some of the columns are categorical columns which we cannot deal with them instantly.

We then started checking for missing values, duplicates and performed some statistical analysis on the numerical columns. We found that we had some missing values, and no duplicates in some columns as shown in the following figures.

```
####Nulls
df.isna().sum()
```

```
id                0
category          0
title             0
body              0
amenities      3185
bathrooms        30
bedrooms          7
currency          0
fee               0
has_photo         0
pets_allowed   3751
price             0
price_display     0
price_type        0
square_feet       0
address        2971
cityname         66
state            66
latitude          7
longitude         7
source            0
time              0
```

```
####Duplicates
duplicates = df[df.duplicated()]

if duplicates.empty:
    print("No duplicate rows found.")
else:
    print("Duplicate rows found:")
    print(duplicates)
```

```
No duplicate rows found.
```

```
####Checking the type of each column
df.dtypes
```

```
id                int64
category         object
title            object
body             object
amenities        object
bathrooms       float64
bedrooms        float64
currency         object
fee              object
has_photo        object
pets_allowed     object
price             int64
price_display    object
price_type       object
square_feet       int64
address          object
cityname         object
state            object
latitude        float64
longitude       float64
source           object
time              int64
```

```
###performing statistical analysis
df.describe()
```

	id	bathrooms	bedrooms	price	square_feet	latitude	longitude	time
count	9.000000e+03	8970.000000	8993.000000	9000.000000	9000.000000	8993.00000	8993.000000	9.000000e+03
mean	5.623668e+09	1.380769	1.744023	1487.286222	947.138667	37.67689	-94.778612	1.574906e+09
std	7.007402e+07	0.616171	0.942446	1088.561190	668.806214	5.51527	15.769232	3.755142e+06
min	5.508654e+09	1.000000	0.000000	200.000000	106.000000	21.31550	-158.022100	1.568744e+09
25%	5.509250e+09	1.000000	1.000000	950.000000	650.000000	33.66200	-101.858700	1.568781e+09
50%	5.668610e+09	1.000000	2.000000	1275.000000	802.000000	38.75550	-93.707700	1.577358e+09
75%	5.668626e+09	2.000000	2.000000	1695.000000	1100.000000	41.34980	-82.446800	1.577359e+09
max	5.668663e+09	8.500000	9.000000	52500.000000	40000.000000	61.59400	-70.191600	1.577362e+09

2.4 Data Cleaning

2.4.1 Handling Missing Values

As we saw before we have columns with lots of missing values, It's time to deal with them. Note we didn't drop any row as the dataset with 9k rows will be affected if we removed 3k rows, so we replaced them with different techniques in each column.

1.Column (amenities):

It contains categorical values so as a beginning we just filled them with 0 or we can replace them with None. We will further do some more preprocessing on this column which we will show later in this report.

```
df['amenities'] = df['amenities'].fillna(0)
```

2.Column (bathrooms) & Column (bedrooms):

They contain numerical values and there aren't a lot of missing values so we just replaced the missing values of each column with the mode(most frequent value) of this column.

```
#bathrooms and bedrooms since there are low number of missing values mean or mode replacement or we can drop them
df['bedrooms']=df['bedrooms'].fillna(df['bedrooms'].mode().iloc[0])
df['bathrooms']=df['bathrooms'].fillna(df['bathrooms'].mode().iloc[0])
```

3.Column (pets_allowed):

A categorical column and there are a lot of missing values. Here we just filled the missing values with 'No' and the other different values we mapped them to 'Yes' and we will further preprocess on this column.

```
df['pets_allowed'] = df['pets_allowed'].fillna('No').apply(lambda x: 'Yes' if x != 'No' else x)
df['pets_allowed']
#Converting any pets to Yes and Missing Values to No
```

4.Column (address):

A categorical column and there are a lot of missing values. Here we just filled the missing values with 'Unknown' and we will do some more preprocessing on it later.

```
###address
df['address']=df['address'].fillna("Unknown")
```

5.Column (cityname) & Column (state):

categorical columns and there aren't a lot of missing values. Here we just filled the missing values of each column with the mode of this column and we will do some more preprocessing on it later.

```
df['cityname']=df['cityname'].fillna(df['cityname'].mode().iloc[0])
df['state']=df['state'].fillna(df['state'].mode().iloc[0])
```

6.Column (longitude) & Column (latitude):

numerical columns and there aren't a lot of missing values. Here we just filled the missing values of each column with the mean(average) of this column.

```
####Long and Lat columns we can fill NaN with the mean.
df['longitude'] = df['longitude'].fillna(df['longitude'].mean())
df['latitude'] = df['latitude'].fillna(df['longitude'].mean())
```

Now we successfully replaced all missing values and have no more missing values as shown in the following figure:

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

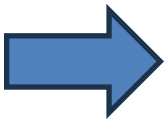
id	0
category	0
title	0
body	0
amenities	0
bathrooms	0
bedrooms	0
currency	0
fee	0
has_photo	0
pets_allowed	0
price	0
price_display	0
price_type	0
square_feet	0
address	0
cityname	0
state	0
latitude	0
longitude	0
source	0
time	0

2.4.2 Handling Incorrect Formats

In exploring the Rent category column which is our target column we found that the format is not clear as it must be numeric values containing no strings or symbols we found that it has values like $1,194 which must be 1194 clear. We used regular expression to just replace the '$' or ',' with an empty string also we made sure that we converted it's type to int after we did this as showed in the next figure.

```
print("before")
print(df['RentCategory'] )
```

```
before
0       medium-priced rent
1       medium-priced rent
2       medium-priced rent
3                 low rent
4               high rent
              ...
8995    medium-priced rent
8996            high rent
8997             low rent
8998    medium-priced rent
8999            high rent
```

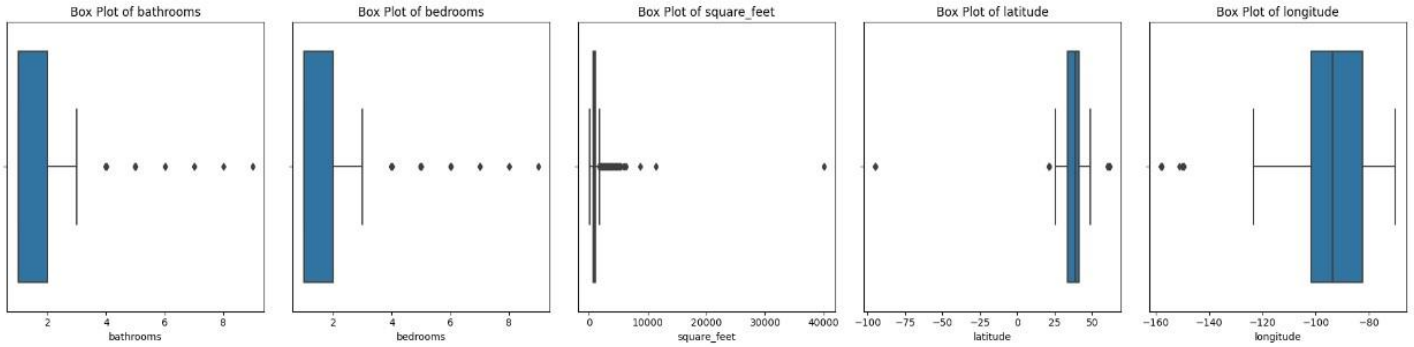


```
df['RentCategory'] = df['RentCategory'].apply(categorize_rent_category)
print("after")
print(df['RentCategory'] )
```

```
after
0       Medium
1       Medium
2       Medium
3          Low
4         High
         ...
8995    Medium
8996      High
8997       Low
8998    Medium
8999      High
```

2.4.3 Handling Outliers

- We first visualized the outliers of our numerical columns for outlier detection as shown in the figure:



- We made sure that our numerical columns outliers are replaced with the nearest nonoutlier value (Winsorization).

2.5 Processing Numerical Features

2.5.1 Different Techniques

Column (id): In this column we found that each row has a unique id but it's number is very high so we just started giving each row a unique id also but numbered from 0 to 8999 instead of the high numbers like 5668617853 so it looks better for eyes as it looks like an index for the row.

```
##id column can be mapped to index 0 till 8999 (useless it is integer anyways and unique and we won't use it)
##we just do it to make it look easier
unique_ids = df['id'].unique()
id_mapping = {original_id: new_id for new_id, original_id in enumerate(unique_ids)}
df['id'] = df['id'].map(id_mapping)
df['id']
```

```
0          0
1          1
2          2
3          3
4          4
        ...
8995    8995
8996    8996
8997    8997
8998    8998
8999    8999
Name: id, Length: 9000, dtype: int64
```

Column (time): In this column we have found that time of the reported row has a special format but we don't care about the time so we just scaled it to make it looks easier. We divided the time rows with minimum value of the time column.

```
#### time column
max_time = df['time'].max()
min_time = df['time'].min()
print(f"Maximum time in : {max_time}")
print(f"Minimum time in : {min_time}")
```

```
Maximum time in : 1577362186
Minimum time in : 1568743976
```

```
df['time']=df['time']/df['time'].min()
```

```
df['time']
```

```
0       1.000008
1       1.005273
2       1.005492
3       1.005492
4       1.005492
          ...
8995    1.000024
8996    1.005491
8997    1.005492
8998    1.000018
8999    1.005492
Name: time, Length: 9000, dtype: float64
```

Column(bedrooms): In this column we noticed that there are 171 values = 0 which is nonsense so we replaced all 0's with 1's.

```
df['bedrooms'].value_counts()[0]
```

```
171
```

so we did this:

```
df['bedrooms']=df['bedrooms'].replace(0,1)
```

2.6 Handling Categorical Features

2.6.1 Different Techniques

- Exploring the csv file we found that In the **"body" column**, we started thinking for tokenizing it but we found that most of the sentences already extracted in other columns like the amenities .etc. most of the apartments have either a fixed price listed or a price range, it always starts with "$(followed by numbers)We",or in case of the price range "$(followed by numbers) - $(followed by numbers)We".The next step is relatively simple, we used library re(regular expression) to identify the pattern we found then we just extract the price/minimum price in case of price range and put it in the "extracted_price" column, and then we use it as an indicator to the prices as shown in the function extract_price in the following fig **Note**: if no pattern noticed we just return0

```
######Extracting Price Info from body column######
def extract_price(text):
    prices = re.findall(r'\$(\d+(?:\.\d+)?)', text)
    if prices:
        return prices[0]
    else:
        return 0

df['extracted_price'] = df['body'].apply(extract_price)
```

```
df['extracted_price']=df['extracted_price'].astype(np.float64)
df['extracted_price']
```

```
0          0.0
1       1370.0
2       1009.0
3        695.0
4       3695.0
          ...
8995       0.0
8996    2035.0
8997     424.0
8998       0.0
8999    2398.0
Name: extracted_price, Length: 9000, dtype: float64
```

- Moving to **the state and cityname columns** we processed using the same idea in both, We take the mean of prices of each city, then we represent the city with its mean of prices, not its original name, in a new column called "city_price_mean", we then update the original column of "cityname" with the values of the "city_price_mean" column then we drop the "city_price_mean" column and then rename "cityname" column to "CityavgRentprice", doing the same to state column We take the mean of prices of each

state, then we represent the state with its mean of prices, not its original name, in a new column called “state_price_mean“, we then update the original column of “state” with the values of the “state_price_mean” column then we drop the “state_price_mean” column and then rename “state” column to “StateavgRentprice” as shown in the fig:

```
#grouping the state according to its names and replacing with the avg price of each state
state_encoding = df.groupby('state')['extracted_price'].mean().reset_index()
state_encoding.columns = ['state', 'state_price_mean']
df = pd.merge(df, state_encoding, on='state', how='left')
#df['state']=df['state_price_mean']
#df.drop(['state_price_mean'], axis=1, inplace=True)
```

```
#grouping the city according to its names and replacing with the avg price of each city
city_encoding = df.groupby('cityname')['extracted_price'].mean().reset_index()
city_encoding.columns = ['cityname', 'city_price_mean']
df = pd.merge(df, city_encoding, on='cityname', how='left')
#df['cityname']=df['city_price_mean']
#df.drop(['city_price_mean'], axis=1, inplace=True)
```

```
df = df.rename(columns={'city_price_mean': 'CityavgRentprice', 'state_price_mean': 'StateavgRentprice'})
```

2.6.2 Encoding

As we have a regression problem which depends on numerical continuous variables we can’t continue to our models with non-numerical features so Encoding just transfer them from the categorical to numerical in different ways depending on its type.

2.6.2.1 Label Encoder

Before we start encoding we found that some columns has different unique values but some of this unique values only happens once so we just replace with the most frequent one that appears the most in the data as shown in the next figures.

```
df['price_type'].value_counts()
##### We convert them to Monthly
```

```
price_type
Monthly          8998
Weekly              1
Monthly|Weekly      1
Name: count, dtype: int64
```

```
df['source'].value_counts()
```

```
source
RentLingo            6232
RentDigs.com         2474
ListedBuy             169
RealRentals            58
GoSection8             27
Listanza               21
RENTOCULAR             15
RENTCafé                1
tenantcloud             1
Real Estate Agent       1
rentbits                1
Name: count, dtype: int64
```

```
df['category'].value_counts()
##### We convert them to housing/rent/apartment
```

```
category
housing/rent/apartment     8997
housing/rent/short_term       2
housing/rent/home             1
Name: count, dtype: int64
```

Now jumping to the **label encoder** converts categorical data into a numerical representation by giving each category a special numerical label. We encoded the categorical columns into numerical ones by using object from the built-in LabelEncoder(). **We encoded columns (source,**

price_type, category, address, body, title, pets_allowed, has_photo, fee, currency, RentCategory) as shown in the figure below:

```
label_encoder = LabelEncoder()
df['source'] = label_encoder.fit_transform(df['source'])
df['price_type'] = label_encoder.fit_transform(df['price_type'])
df['category'] = label_encoder.fit_transform(df['category'])
df['address'] = label_encoder.fit_transform(df['address'])
df['body'] = label_encoder.fit_transform(df['body'])
df['title'] = label_encoder.fit_transform(df['title'])
df['pets_allowed'] = label_encoder.fit_transform(df['pets_allowed'])
df['has_photo'] = label_encoder.fit_transform(df['has_photo'])
df['fee'] = label_encoder.fit_transform(df['fee'])
df['currency'] = label_encoder.fit_transform(df['currency'])
```

2.6.2.2 Different Technique

- Exploring the csv file we can notice that the amenities are showed in the amenities column with splitting between each one of them with a comma (,) so we did a function that split the contents of each cell in the amenities column after each comma and then count the number and update the original column cells.

```
####amenities column we fill NaN with 0 and we count each amenity.
##  Ex: Gym , Parking , Pool = 3
def encode_amenities(value):
    items = value.split(',') if isinstance(value, str) else []
    count=len(items)
    return count


if df['amenities'].dtype == object:
    df['amenities'] = df['amenities'].apply(encode_amenities)


df['amenities']
```

```
0        6
1        9
2        7
3        0
4        0
        ..
8995     4
8996     0
8997     2
8998    11
8999     0
Name: amenities, Length: 9000, dtype: int64
```

We then replace the 0 values (Previously the NaN) with the mean or mode, which is 3 in our case because it is nonsense to have an apartment without any amenities.

```
####Replacing nan with 0 then with the mean or mode which is equal 3.
df.loc[df['amenities'] == 0, 'amenities'] = 3
df['amenities']
```

```
0        6
1        9
2        7
3        3
4        3
        ..
8995     4
8996     3
8997     2
8998    11
8999     3
Name: amenities, Length: 9000, dtype: int64
```

2.7 Dropping Columns

We started seeing what columns that doesn't have a real effect on our target variable or not meaningful to use it to predict the apartment rent price, so we dropped like 11 columns below is a summary of each column dropped with the reason of dropping it:

Column name	Reason of dropping
id	It is not more than an index (unique identifier) just for our row.
category	It has only one unique value which is 0 after encoding.
title	Non meaningful to take the title of the report to predict the price. Especially after encoding it to numbers which represent the number of characters in each row (doesn't make sense the more in title the more the price).
body	After exploring the body we found that all information is extracted in other columns so we don't need it anymore we encode it and drop it.
currency	It has only one unique value which is 0 after encoding.
fee	It has only one unique value which is 0 after encoding.
source	Non meaningful: We can't just use the source in which the report was reported to predict the price of an apartment.
time	After scaling the time we decided to drop it as it has no effect we don't predict the price according to the time it was reported.
price	After preprocessing the price_display column our target column so this price column is just a duplicate of our target column.
price_type	It has only one unique value which is 0 after encoding.
address	We decided to drop it as we already have the city, state, longitude and latitude so we can just figure out the area from them not with the address column which has maybe the no of apartment or no of the block.

Snapshots of the code where we dropped the columns and our dataframe now:

```
###Columns to drop : category, fee , currency , price_type  (as they have only one value)
##Then drop them
###We can drop title, body as they dont affect the data
###column price might be dropped as we already preprocessed price_display so it is a duplicated column
columns_to_drop = ['id', 'category', 'title', 'body', 'currency', 'fee', 'source', 'time','price','price_type','address']
df = df.drop(columns=columns_to_drop)
```

```
df.head(5)
```

amenities	bathrooms	bedrooms	has_photo	pets_allowed	price_display	square_feet	CityavgRentprice	StateavgRentprice	latitude	longitude	extracted_price
6	3	2	1	1	1194	800	1152.095238	1190.845000	35.7585	-78.7783	0.0
9	1	1	1	1	1370	795	1602.509259	1298.712121	43.0724	-89.4003	1370.0
7	1	1	1	1	1009	560	1287.750000	1532.148148	29.6533	-82.3656	1009.0
3	1	1	1	1	695	600	952.650794	939.966292	41.2562	-96.0404	695.0
3	3	3	1	0	3695	1600	3084.543046	2811.288528	34.0372	-118.2972	3695.0

2.8 Feature Engineering

In this section we will introduce you to new 5 features we have made it after exploring some of the data:

1. AvgAreaPrice: This feature we decided to use longitude and latitude values in a feature so we divided the longitude and latitude each one to 10 equal ranges(bins). After that we started grouping this area by the avg price of this area.

```
##############New Feature ALERTTTTT############ AvgAreaState
df['longitude_range'] = pd.cut(df['longitude'], bins=10)
df['latitude_range'] = pd.cut(df['latitude'], bins=10)


range_means = df.groupby(['longitude_range', 'latitude_range'])['extracted_price'].mean()


df['AvgAreaPrice'] = df.groupby(['longitude_range', 'latitude_range'])['extracted_price'].transform('mean')


df.drop(['longitude_range', 'latitude_range'], axis=1, inplace=True)
```

2.NoOfRooms: This feature we simply calculated the total number of rooms in an apartment by adding the no of bedrooms to bathrooms in each row.

```
##### We will see if the features we did before in MS1 will help us here as well or no#####
df['NoOfRooms']=df['bathrooms']+df['bedrooms']
df['bed/bathrooms']=df['bedrooms']/df['bathrooms']
#df['Trial'] = df['squar_feet']/df['NoOfRooms']
```

3.CityStateAvg: This feature we simply got the two columns we already had from the cityname and state which are CityavgRentprice and StateavgRentprice we added them together then divided them by 2.

```
##########NEW FEATURE ALERT!!!!!!!!!!!!################
df['CityStateAvg']=(df['CityavgRentprice']+df['StateavgRentprice'])/2
```

4.avgBathroomsPrice: This feature we grouped the unique values of no of bathrooms and decided to get for each different group (number of bathrooms) the avg price of this group for example the apartments which has 1 bathroom an avg price of 1266 and so on to all groups, price based on no of bathrooms.

```
#########NEW FEATURE ALERT!!!!!!!!!!!!################
bath_encoding = df.groupby('bathrooms')['extracted_price'].mean().reset_index()
bath_encoding.columns = ['bathrooms', 'avgBathroomsPrice']
df = pd.merge(df, bath_encoding, on='bathrooms', how='left')
```

5.Trial: Finally, our last feature and we have achieved it after so many trials so that's why we called it Trial. Exploring the data with the new features we introduced and trying to get a pattern which we can get from it closer to the target we have discovered a pattern which if we added the CityavgRentprice + square_feet/2 + avgBathroomsPrice/3 + bedrooms*20 it gets us closer to the target. We basically gave the bedrooms cost (20), divided the avgBathroomsPrice by 3 which is the mode of the bathrooms, also divided the square_feet by 2 then added all those to the CityavgRentprice forming our new feature Trial.

```
#########NEW FEATURE ALERT!!!!!!!!!!!!################
df['Trial'] = df['CityavgRentprice'] + df['square_feet']/2 + df['avgBathroomsPrice']/3 +df['bedrooms']*20
```

2.9 Feature Selection

- After Introducing new feature and handling their outliers, We are ready to select the best features to be ready to enter the machine learning models to predict the apartment rent prices. We started by feature selection using the SelectKBest method from scikit-learn. We do this by :
    1. Separating features and target columns into X which is include all column expect target column and set the target column in y
    2. Standardization : each feature in X has a mean of 0 and standard deviation of 1 by StanderScalar from scikit-learn
    3. We will save scalar object using joblib to be used later
    4. Now we will select k best features using "SelectKBest" based on scoring function = "f_classif" which computes the ANOVA F-value between the target variable and each feature and the choose the k best
    5. Then getting selectedfeatures by get_support() and and create dataframe for it as shown in the figure below:

	bathrooms	square_feet	CityavgRentprice	extracted_price	NoOfRooms	\
0	0.997783	1.098366	0.502177	0.367061	1.354174	
1	0.997783	1.354466	-0.318391	0.389550	1.354174	
2	0.997783	0.393043	-0.934441	-0.944821	0.629909	
3	2.651948	2.668550	3.748126	5.028925	2.078439	
4	-0.656382	-1.229620	0.703826	0.248991	-1.542885	

	CityStateAvg	Trial	RentCategory
0	0.502803	0.735055	1
1	-0.537019	0.093692	1
2	-0.999771	-0.707118	1
3	3.528614	4.485876	2
4	0.921553	0.204562	1

	Feature	Score
16	Trial	1020.562566
10	extracted_price	993.046057
5	square_feet	770.312608
14	CityStateAvg	682.439738
1	bathrooms	628.424165
6	CityavgRentprice	571.020985
11	NoOfRooms	558.271533
13	AvgAreaPrice	515.163637
7	StateavgRentprice	494.034107
15	avgBathroomsPrice	392.337369
2	bedrooms	371.172631
9	longitude	125.945806
0	amenities	49.287108
3	has_photo	17.648315
4	pets_allowed	13.355323
8	latitude	12.038639
12	bed/bathrooms	2.186928

- After many trials to perform better in the models we decided to take features into our X dataframe .So X has now the 7 features that is ready to predict the target variable which is in Y dataframe as in the following figure:

X:

	bathrooms	square_feet	CityavgRentprice	extracted_price	NoOfRooms	CityStateAvg	Trial
0	0.997783	1.098366	0.502177	0.367061	1.354174	0.502803	0.735055
1	0.997783	1.354466	-0.318391	0.389550	1.354174	-0.537019	0.093692
2	0.997783	0.393043	-0.934441	-0.944821	0.629909	-0.999771	-0.707118
3	2.651948	2.668550	3.748126	5.028925	2.078439	3.528614	4.485876
4	-0.656382	-1.229620	0.703826	0.248991	-1.542885	0.921553	0.204562
...	...	...	...	...	...	...	...
7195	-0.656382	1.604268	1.315047	1.280693	0.629909	0.809270	1.527604
7196	-0.656382	-0.463420	0.066229	-0.129580	-0.818620	-0.048199	-0.125796
7197	-0.656382	-0.278693	-0.536641	-0.968247	-0.094356	-0.457438	-0.576227
7198	-0.656382	-0.373156	0.209451	0.156223	-0.818620	0.316619	0.020967
7199	-0.656382	-0.232511	0.236580	1.023002	-0.818620	0.067438	0.080640

y:

```
0       1
1       1
2       1
3       2
4       1
       ..
7195    2
7196    0
7197    1
7198    1
7199    2
Name: RentCategory
```

3.Models

In this section in each model we quickly introduce the model, We then used also different performance metrics to try and evaluate our model train and test prediction abilities, the most common ones are MSE and R2-score(Coefficient of determination) which indicates how much of our target variable is represented by the independent variables, it represents how good our regression model fits the data. Finally, We visualize the model with different plots to see how well it's fitting the data.

3.1 Random Forest

3.1.1 Introduction

Random Forest is a versatile ensemble learning method for classification and regression task , it builds multiple decision trees by randomly selecting subsets of the training data with replacement . and it can measure the relative importance of each feature in prediction by computing based on the decrease in impurity caused by each feature across all decision trees in the forest.

We set max_depth = 15 and min_samples_split = 5 and n_estimators = 30 , this gave us test accuracy = 83% and train accuracy = 96% . then we change min_samples_split = 10, we find test accuracy = 83% and train accuracy = 95.5% . we try max_depth=5,then we find test accuracy = 79% and train accuracy = 86% . so we noted that when min_sample_split decrease this is gave us better accuracy and with max depth increasing too,so the best test accuracy = 83% and train accuracy = 96% with max_depth = 15 and min_samples_split = 5 and n_estimators = 30

3.1.2 Error/Accuracy

We used our training data to fit the model and then used the model to predict the test data , this model gave an accuracy of roughly 96% on the training data and 83% on the test data

```
RandomForest Accuracy gb test: 0.8333333333333334
RandomForest Accuracy gb train: 0.96875
```

3.2 Decision Tree

3.2.1 Introduction

For better predictions we apply different models . so we apply decision trees which is provide a simple yet powerful approach to modeling data, with a focus on interpretability, feature importance, and handling of complex relationships, albeit requiring attention to overfitting and potential

3.2.2 Error/Accuracy

The accuracy of train didn't differ a lot, but test accuracy improved alot .

```
Decision Tree Accuracy dt test: 0.8055555555555556
Decision Tree Accuracy dt train: 0.9984722222222222
```

3.3 Gradient Boosting

3.3.1 Introduction

Trying to improve the prediction performance of our model we used an ensemble technique which is gradient boosting . it builds a strong predictive model by combining multiple weak learners, typically decision trees, in a sequential manner. It works by base learners (weak) each learner attempts to correct the errors made by previous ones . it train sequentially . then gradient boosting minimizes a loss function and introduces a learning rate then The final prediction is the sum of predictions from all weak learners, weighted by the learning rate .

We set learning_rate = 0.5 and n_estimators = 30 , we find test accuracy = 85% and train accuracy = 94% , then we change learning_rate = 1.1, so gave us test accuracy = 82% and train accuracy = 96% , then we set learning_rate = 0.001,

we find test accuracy = 54% and train accuracy = 54% . we try learning_rate = 1 and , find test accuracy = 83% and train accuracy = 96% . so we noted that we find good accuracies with learning_rate between [0.1 , 1] , finally the best test accuracy = 85% and train accuracy = 94% with learning_rate = 0.5 and n_estimators = 30

3.3.2 Error/Accuracy

We can notice that the accuracy is improved in both (train & test) .

```
Gradient Boosting Accuracy gb test: 0.8583333333333333
Gradient Boosting Accuracy gb train: 0.9434722222222223
```

3.4 SVM

3.4.1 Introduction

Support Vector Machine (SVM) is a supervised machine learning algorithm . It works by finding the hyperplane that best separates different classes in the feature space .

There were many trials in changing the hyperparameters but it is not one of our main/top with the best accuracies.

3.4.2 Error/Accuracy

We can see good accuracies by there the better more than it .

```
SVM Accuracy test: 0.8188888888888889
SVM Accuracy train: 0.8730555555555556
```

3.5 KNN

3.5.1 Introduction

It makes predictions based on the similarity of a new data point to the labeled data points (also called neighbors) in the training dataset .

hyperparameter tuning : the choice of k is crucial in KNN. A smaller k value leads to a more flexible model with a complex decision boundary, potentially prone to overfitting , Conversely, a larger k value leads to a smoother decision boundary but may cause underfitting.

After a lot of trials it was not so good in test accuracy so it was not one of our main models.

3.5.2 Error/Accuracy

This is gave us best accuracy train , but didn't improve accuracy test

```
KNN Accuracy test: 0.7694444444444445
KNN Accuracy train: 0.9984722222222222
```

3.6 XGBoost

3.6.1 Introduction

Using another ensemble technique, extreme gradient boosting which is similar to gradient boosting but with some improved aspects like regularization (adding a penalty term like L1 or L2 regression) and parallelization that speeds up the training process, since the GBM performed well we tried this model which gave a better accuracy of 98% for the train data and 92% for the test data.

We set n_estimators which is number of rounds or trees to build with value 200 and set gamma = 0 , learing_rate = 0.1 , max_depth = 10 , then we find test accuracy = 83% and train accuracy = 99% . then we change gamma = 1 then we see our test accuracy = 84% and train accuracy = 95% . then we set gamma = 0 and learning rate = 0.01 and max_depth = 10 , we noted the test accuracy = 83% and train accuracy = 95% . we change n_esctimators = 100 and gamma = 0 , learning rate = 0.5 , max_depth = 10 , we find test accuracy = 83% and train accuracy = 99% ,we then try max_depth=15 and n_estimators=200,we find test accuracy=83% and train accuracy=99% ,now try gamma=1 we find test accuracy=83% and train accuracy=95%. We noted that if learin when gamma != 0 the accuracy of train decrease . n_estimators don't affect on accuracies too much . and max_depth in range [10, 15] gave us good accuracies . so . finally the best accuracy for test = 83% and train accuracy = 99% with gamma = 0 , learning rate = 0.1 , max_depth = 10 and n_estimators = 200 .

3.6.2 Error/Accuracy

We noticed that it improved the accuracy of both training model and testing model compared to the Gradient Boosting.

```
XGB Accuracy test: 0.8355555555555556
XGB Accuracy train: 0.9938888888888889
```

We have finished our models, Now let's use all of this models to make important topic .... **Ensemble learning**

Ensemble learning is a machine learning technique where multiple models are combined to improve the overall performance and robustness of the prediction. Instead of relying on a single model, ensemble methods leverage the wisdom of crowds by aggregating the predictions of multiple models . it aim to build diverse models that make different types of errors on the dataset. By combining these diverse models . it is useful to improve performance and generalize will to unseen data We use to types :

1- **Stacking (Meta-Learning) :** It involves training multiple base models and then training a meta-model to combine their predictions. The meta-model learns how to best combine the base models' predictions. We use this with models (Decision Tree & Random Forest & XGBoost) and final model is simplest (logistic regression) .

```
stackingclass = StackingClassifier(estimators=estimators,final_estimator=LogisticRegression(random_state= 42))
stackingclass.fit(X_train,y_train).score(X_test,y_test)
```

```
stacking_test = accuracy_score(y_test, y_pred)

print("stacking Accuracy test:", stacking_test)
```

And this is gave us the accuracy for test 84%

```
stacking Accuracy test: 0.8427777777777777
```

2- **Voting :** each model in the ensemble makes a prediction, and the final prediction is determined by majority voting (for classification) . We use this with models (SVM & KNN & Gradient boosting)

```
voting_clf = VotingClassifier(estimators=estimators,voting="hard")
```

```
voting_test = accuracy_score(y_test, y_pred)
print("voting Accuracy test:", voting_test)
```

and this is gave us the accuracy for test 82%

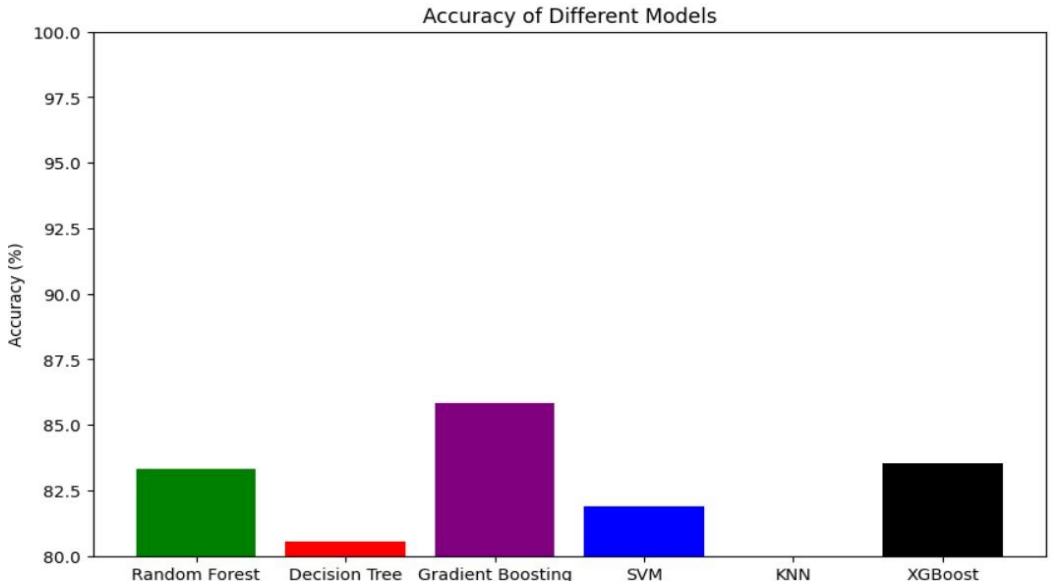
```
voting Accuracy test: 0.825
```

Now let's Visualize our accuracies models

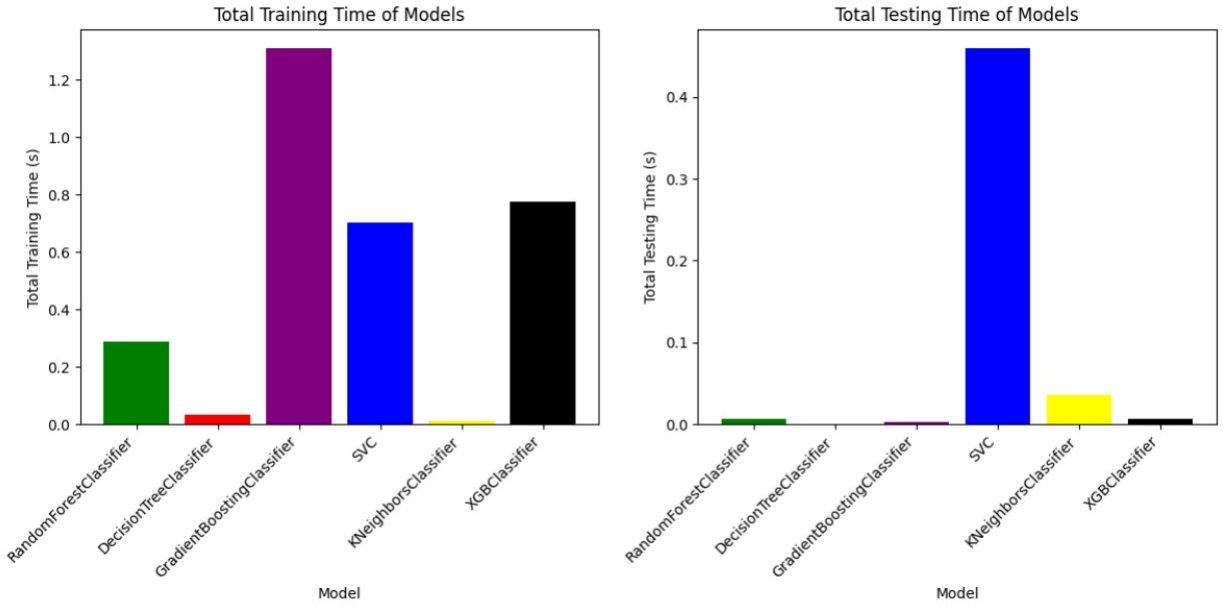
4.Conclusion

- After comparing the seven models, We noticed that the best generalized one was XGBoost Model . and the best model for train accuracy is KNN and Decision Tree and the best model for test accuracy is XGBoost .

Model	Train accuracy	Test accuracy
Random Forest	96.875000	83.333333
Decision Tree	99.847222	80.555556
Gradient Boosting	94.347222	85.833333
SVM	87.305556	81.888889
KNN	99.847222	76.944444
XGBoost	98.500000	83.944444

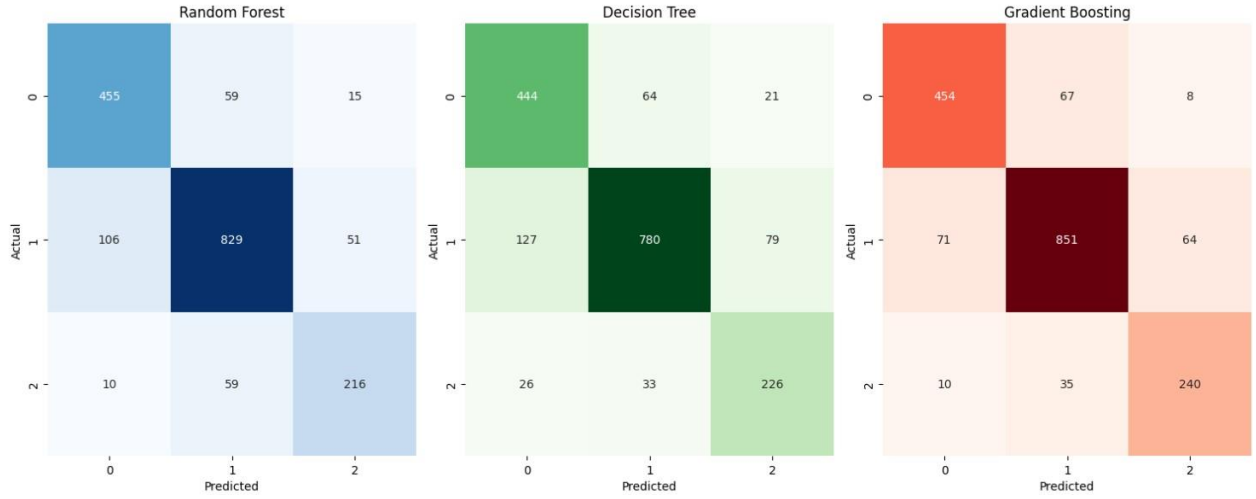


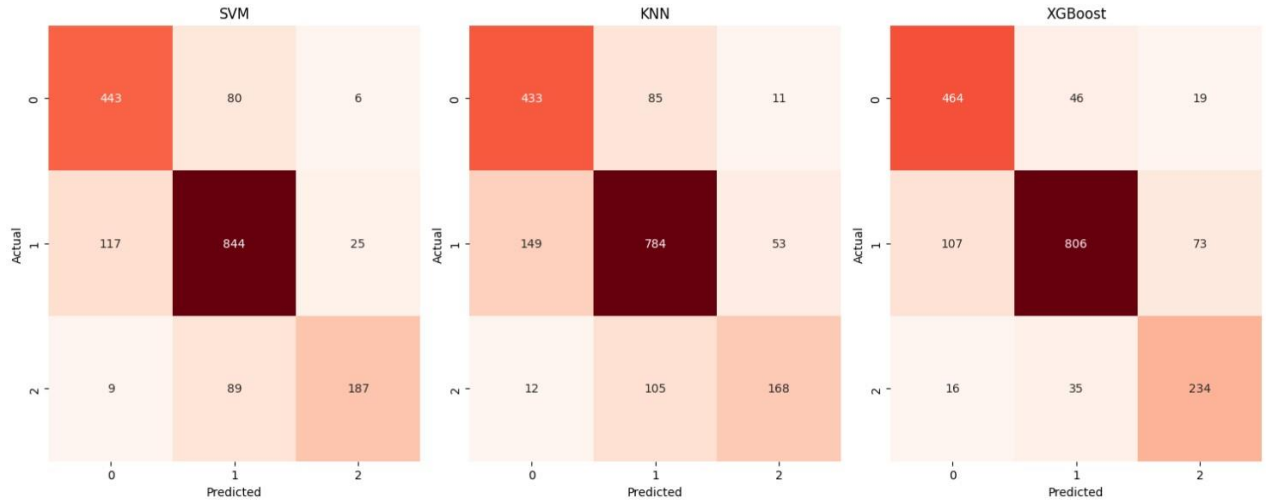
- Then we can see total train and test time for each model . we can note that the models works with ensemble method take large time in train but very small time in test , this is because the ensemble works on more than 1 model so it takes large time in training



Now we will see the confusion matrix for each model

This is show the relationship between actual value and predicted value





Now we will see classification report for each model

SVM Classification Report:

	precision	recall	f1-score	support
0	0.78	0.84	0.81	529
1	0.83	0.86	0.84	986
2	0.86	0.66	0.74	285
accuracy			0.82	1800
macro avg	0.82	0.78	0.80	1800
weighted avg	0.82	0.82	0.82	1800

RandomForest Classification Report:

	precision	recall	f1-score	support
0	0.74	0.90	0.81	529
1	0.90	0.82	0.86	986
2	0.83	0.76	0.80	285
accuracy			0.83	1800
macro avg	0.82	0.83	0.82	1800
weighted avg	0.84	0.83	0.83	1800

Decision Tree Classification Report:

	precision	recall	f1-score	support
0	0.74	0.84	0.79	529
1	0.89	0.79	0.84	986
2	0.69	0.79	0.74	285
accuracy			0.81	1800
macro avg	0.78	0.81	0.79	1800
weighted avg	0.82	0.81	0.81	1800

Gradient Boosting Classification Report:

	precision	recall	f1-score	support
0	0.81	0.84	0.83	529
1	0.89	0.83	0.86	986
2	0.73	0.84	0.78	285
accuracy			0.84	1800
macro avg	0.81	0.84	0.82	1800
weighted avg	0.84	0.84	0.84	1800

XGBOOST Classification Report:

	precision	recall	f1-score	support
0	0.80	0.86	0.83	529
1	0.90	0.83	0.86	986
2	0.74	0.82	0.78	285
accuracy			0.84	1800
macro avg	0.81	0.84	0.82	1800
weighted avg	0.84	0.84	0.84	1800

KNN Classification Report:

	precision	recall	f1-score	support
0	0.72	0.82	0.77	529
1	0.79	0.81	0.80	986
2	0.74	0.48	0.59	285
accuracy			0.76	1800
macro avg	0.75	0.70	0.72	1800
weighted avg	0.76	0.76	0.76	1800

5.saving and load models

We save our models by dump () which is a convenient function for saving Python objects to disk in a binary format .

```
dump(rf_classifier, rf_model_path)
dump(dt_classifier, dt_model_path)
dump(gb_classifier, gb_model_path)
dump(svm_classifier, svm_model_path)
dump(knn_classifier, knn_model_path)
dump(xgb_classifier, xgb_model_path)


print("Models saved successfully!")
```

```
Models saved successfully!
```

And the we load pre-trained machine learning models from files using **joblib**'s **load()** function ang If the models are not found, it falls back to training the models from scratch

```
    print("Models loaded successfully!")
except FileNotFoundError:
    print("Saved models not found. Training models from scratch...")

    rf_classifier = RandomForestClassifier(max_depth= 10, min_samples_split= 5, n_estimators= 30)
    rf_classifier.fit(X_train, y_train)

    dt_classifier = DecisionTreeClassifier(min_samples_split= 2, min_samples_leaf= 1, max_features= 5)
    dt_classifier.fit(X_train, y_train)

    gb_classifier = GradientBoostingClassifier(learning_rate= 1, n_estimators= 30)
    gb_classifier.fit(X_train, y_train)

    svm_classifier = SVC(kernel='rbf',C=10,gamma='scale',random_state=42)
    svm_classifier.fit(X_train, y_train)

    knn_classifier = KNeighborsClassifier(n_neighbors= 9, p= 1, weights= 'distance')
    knn_classifier.fit(X_train, y_train)

    xgb_classifier = xgb.XGBClassifier(random_state=42,gamma= 0, learning_rate= 0.1, max_depth= 7, n_estimators= 200)
    xgb_classifier.fit(X_train, y_train)

    print("Models trained successfully!")
```