Ehud Vaknin 209479088

Moshe Bercovich 206676850

House Pricing Predictions

Problem –

In the real estate market, accurately forecasting apartment prices is a critical task for various stakeholders, including sellers, brokers, and contractors. The challenge lies in understanding the vast number of factors that influence apartment prices, such as location, size, facilities, and market trends. These factors can vary significantly from one region to another, making price estimation complex and often subjective.

This project aims to develop predictive models that can assist in estimating the future selling prices of apartments. By leveraging historical data and machine learning techniques, these models can provide data-driven insights, helping sellers to set competitive prices, brokers to offer realistic price ranges to clients, and contractors to plan financial aspects of new developments. Ultimately, this can lead to more informed decision-making and a more transparent real estate market.

Methodology –

- We use machine learning techniques to build a predictive model for house prices. The process involves data preprocessing, model selection, training, and evaluation.
- Programming language – python
- Model types - Random Forest Regressor, Decision Tree Regressor, Linear Regression

Dataset description –

The dataset contains 21k rows of house sales details. Each sale have the following parameters – id, date, price, bedrooms, bathrooms, square footage of the home, square footage of the lot, floors, waterfront, view, condition, grade, footage of house apart from basement, square footage of the basement, Built Year, Year when house was renovated, zip code, Latitude coordinate, Longitude coordinate, Living room area in 2015, lot Size area in 2015.

Pre-processing (price prediction) –

1. Drop unnecessary features – date, zip code, id.
2. Find each house location by its “lat” and “long” values using geopy library.

```python
# Analyze lat,long columns ? keep : remove
import geopy
from geopy.geocoders import Nominatim
from typing import Optional
geolocator = Nominatim(user_agent="house_pricing")
def reverse_geocode(lat: float, long: float, state_city: Optional[str] = None) -> str:
    if state_city is not None and state_city not in [np.nan]:
        print(f"Location of {lat}, {long} already exists: {state_city}")
        return state_city
    location = geolocator.reverse((lat, long), timeout=1000)
    address = location.address
    # Extract city and state from the address
    city = location.raw['address'].get('city', '')
    state = location.raw['address'].get('state', '')
    print(f"Location of {lat}, {long}: {state}, {city}")
    return f"{state}, {city}"

lat_col = df['lat'][0]
long_col = df['long'][0]

for start_index in range(0, len(df)//1000, 10):
    end_index = start_index + 10
    df.loc[start_index:end_index, 'state_city'] = df.loc[start_index:end_index].apply(
      lambda row: reverse_geocode(row['lat'], row['long'], row.get('state_city')),axis=1)

    df.to_csv("/content/house_locations.csv")
```

3. All the locations are in Washington, so they don’t contribute to the prediction process.
4. Drop the “lat” and “long” columns, so the data remain with 15 columns and the target column – price.
5. Separate the “price” column from the data – it will be the target class
6. Split the data to train and test using the method “train_test_split” with test_size = 0.2 and random_state = 1.
7. The train set contain 17290 rows and the test set contain 4323 rows.

8. Build a function that returns a list of ranges according to the requested number of intervals

```python
def getIntervals(data:pd.Series, num_intervals) -> list:
  return [(x,x+int((data.max()-data.min())/num_intervals)) for x
          in range(int(data.min()), int(data.max()), int((data.max()-data.min())/num_intervals))]
```

9. Building functions that go through the entire series and change the values to their intervals according to the ranges

```
def findInterval(intervals:list, price):
  for interval in intervals:
    if interval[0] <= price <= interval[1]:
      return intervals.index(interval)
  return -1           # not found


def convert_prices(data:pd.Series, number_of_intervals:int)->pd.Series:
  intervals = getIntervals(data, number_of_intervals)
  print("intervals: ", intervals)
  data_intervals = data.apply(lambda x: findInterval(intervals, x))
  return data_intervals

target_intervals = convert_prices(target_class, 10)
print(set(target_intervals.values))
```

```
intervals:  [(75000, 837500), (837500, 1600000), (1600000, 2362500), (2362500, 3125000), (3125000, 3887500), (3887500, 4650000),
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

10. Split the new series (the interval one) to fit to the train data and test data

```
# split target_intervals to train, test
target_train_intervals, target_test_intervals = train_test_split(target_intervals, test_size=0.2, random_state=1)
print(target_train_intervals.size)
print(target_test_intervals.size)
```

```
17290
4323
```

11. Get the majority rule of each one of the subsets.

```
# Get the majority role for each one of the target sets
print("train: ", target_train_intervals.value_counts())
print("test: ", target_test_intervals.value_counts())
```

```
train:  price
0    15224
1     1736
2      238
3       67
4       18
6        3
5        3
7        1
Name: count, dtype: int64
test:  price
0    3798
1     429
2      60
3      23
4       8
9       2
5       1
8       1
6       1
Name: count, dtype: int64
```

1. Initialize and fit a Linear Regression model to predict the exact prices.
2. The prediction gave an array with float numbers, so we apply the round method to get the integer values

```
# Init LinearRegression classifier
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
price_reg = reg.fit(data_train, target_train)
price_pred = price_reg.predict(data_test)
price_pred
```

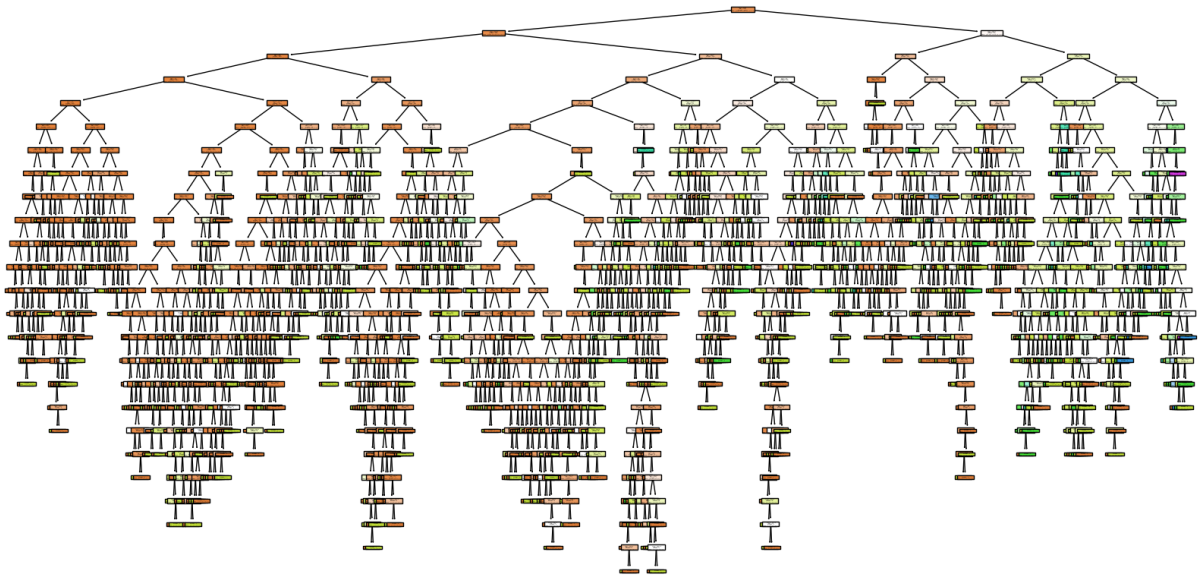
```
array([ 727297.68410977,  312986.54189445,  853519.48368193, ...,
        463062.87675236, 1381599.57246012,  276748.07161185])
```

3. Get the R^2 and the MSE:

```
[37] price_reg_r2 = r2(target_test, price_pred_test)
     price_reg_mse = mse(target_test, price_pred_test)
     print("R2 : ", price_reg_r2)
     print("MSE : ", price_reg_mse)
```

```
R2 :  0.6534026663127288
MSE :  59823036364.11011
```

4. Initialize the RandomForestClassifier model and fit to the train data.



5. Predict the test data and the train data using “predict” function.
6. Get the test accuracy score– 0.919500346981263, train accuracy score – 0.9998843262001157 -> No over-fitting :) .
7. Checking the dependency between the number of the model’s estimators and the accuracy with for loop and dictionary that contains all the results.

```
[ ]  # Check accuracies of different classifiers by num of estimators
     accuracies = {}
     for i in range(100, 1001, 100):
       rfc = RFC(n_estimators=i, random_state=1)
       rfc.fit(data_train, target_train_intervals)
       # target prediction
       target_pred_intervals = rfc.predict(data_test)
       # train prediction
       train_pred_intervals = rfc.predict(data_train)
       accuracies[i] = (accuracy_score(target_train_intervals, train_pred_intervals), accuracy_score(target_test_intervals, target_pred_intervals),
                        accuracy_score(target_train_intervals, train_pred_intervals) > accuracy_score(target_test_intervals, target_pred_intervals) + 0.1)
       print(f"Accuracy score [{i}]: ", accuracies[i])
     print(accuracies)

     Accuracy score [100]:  (0.9998843262001157, 0.919500346981263, False)
     Accuracy score [200]:  (0.9998843262001157, 0.919500346981263, False)
     Accuracy score [300]:  (0.9998843262001157, 0.9211195928753181, False)
     Accuracy score [400]:  (0.9998843262001157, 0.9211195928753181, False)
     Accuracy score [500]:  (0.9998843262001157, 0.9204256303492945, False)
     Accuracy score [600]:  (0.9998843262001157, 0.9199629886652787, False)
     Accuracy score [700]:  (0.9998843262001157, 0.9201943095072866, False)
     Accuracy score [800]:  (0.9998843262001157, 0.9201943095072866, False)
     Accuracy score [900]:  (0.9998843262001157, 0.9192690261392551, False)
     Accuracy score [1000]:  (0.9998843262001157, 0.9197316678232709, False)
     {100: (0.9998843262001157, 0.919500346981263, False), 200: (0.9998843262001157, 0.919500346981263, False), 300: (0.9998843262001157, 0.921119592875318
```

8. The best score is :

```
[ ]  max_key = max(accuracies, key=lambda k: accuracies[k][1])
     print("The best accuracy is when the classifer used ", max_key, " estimators with accuracy: " ,accuracies[max_key])

     The best accuracy is when the classifer used  300  estimators with accuracy:  (0.9998843262001157, 0.9211195928753181, False)
```

9. Checking the influence between each feature and the accuracy with for loop and dictionary that contains all the results.

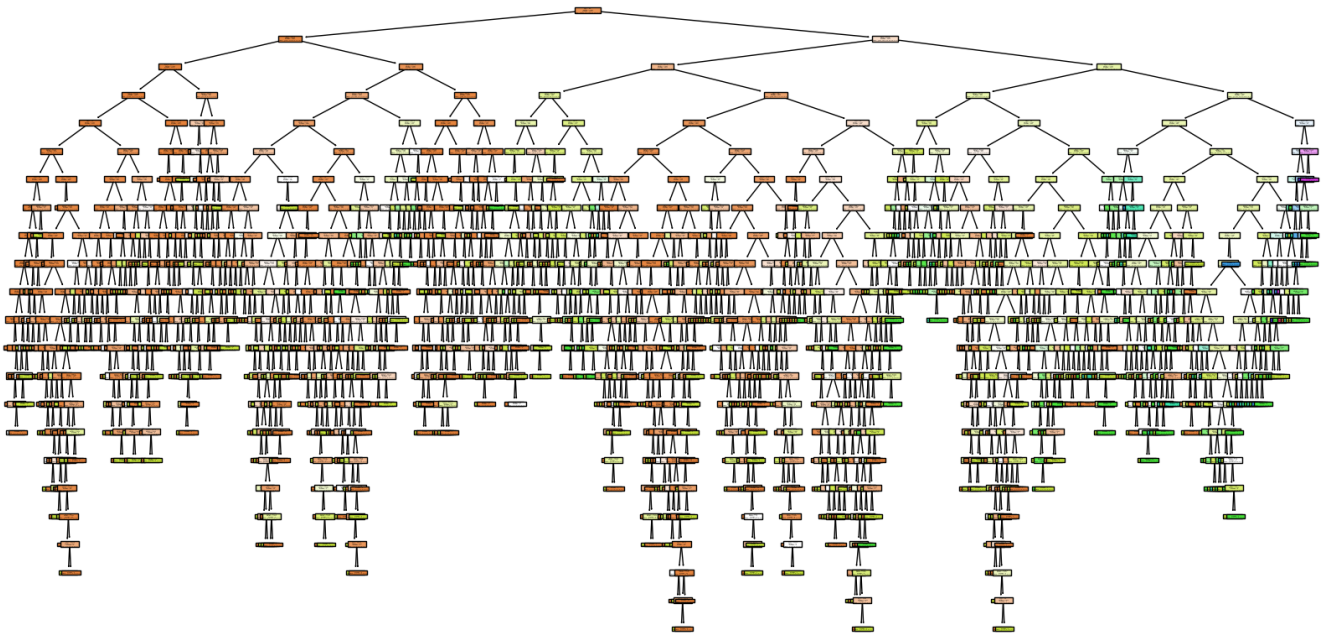
```
[ ]  # Check accuracies of different classifiers by dropping one feature each iteration
     accuracy_by_feature = {}
     for col in data_train.columns:
       rfc = RFC(n_estimators=300, random_state=1)
       rfc.fit(data_train.drop(col,axis=1), target_train_intervals)
       # test prediction
       target_pred_intervals = rfc.predict(data_test.drop(col,axis=1))
       # accuracy of test prediction
       accuracy_score_test = accuracy_score(target_test_intervals, target_pred_intervals)
       # train prediction
       train_pred_intervals = rfc.predict(data_train.drop(col,axis=1))
       # accuracy of train prediction
       accuracy_score_train = accuracy_score(target_train_intervals, train_pred_intervals)
       # over-fitting
       over_fitting = accuracy_score_train > accuracy_score_test + 0.1
       accuracy_by_feature[col] = (accuracy_score_train, accuracy_score_test, over_fitting)
       print(f"Accuracy score [{col}]: ", accuracy_by_feature[col])

     print(accuracy_by_feature)
```

The results:

```
Accuracy score [bedrooms]:  (0.9998843262001157, 0.9192690261392551, False)
Accuracy score [bathrooms]:  (0.9998843262001157, 0.9188063844552394, False)
Accuracy score [sqft_living]:  (0.9998843262001157, 0.9227388387693731, False)
Accuracy score [sqft_lot]:  (0.9998843262001157, 0.9201943095072866, False)
Accuracy score [floors]:  (0.9998843262001157, 0.9218135554013417, False)
Accuracy score [waterfront]:  (0.9998843262001157, 0.9197316678232709, False)
Accuracy score [view]:  (0.9998843262001157, 0.9206569511913023, False)
Accuracy score [condition]:  (0.9998843262001157, 0.9208882720333103, False)
Accuracy score [grade]:  (0.9998843262001157, 0.9151052509831136, False)
Accuracy score [sqft_above]:  (0.9998843262001157, 0.9201943095072866, False)
Accuracy score [sqft_basement]:  (0.9998843262001157, 0.9215822345593337, False)
Accuracy score [yr_built]:  (0.9998843262001157, 0.9155678926671293, False)
Accuracy score [yr_renovated]:  (0.9998843262001157, 0.9192690261392551, False)
Accuracy score [sqft_living15]:  (0.9998843262001157, 0.9125607217210271, False)
Accuracy score [sqft_lot15]:  (0.9998843262001157, 0.9204256303492945, False)
```

According to the results, no feature has massive influence on the accuracy score, so we don't need to drop non of the features to improve the prediction skills.

10. Initialize a Decision Tree Classifier (using ID3) and fit it with the data.



11. Predict the test data and the train data using “predict” function.
12. Get the test accuracy score– 0.8924358084663429, train accuracy score – 0.9998843262001157 -> the distance between the accuracies is 0.107 so we assume there is no over-fitting.

Pre-processing (bedroom prediction) –

1. Reopen the dataset to make a prediction on the bedroom column.
2. Drop the unnecessary columns (id, zipcode, lat, long, date).
3. Find the distribution of the values in the target class.

```
# Get the value counts for each class
value_counts = target_class.value_counts()
print(value_counts)
# Identify the majority class
majority_class = value_counts.idxmax()
majority_count = value_counts.max()
print("Majority Class:", majority_class)
print("Count of Majority Class:", majority_count)
```

```
bedrooms
3     9824
4     6882
2     2760
5     1601
6      272
1      199
7       38
0       13
8       13
9        6
10       3
11       1
33       1
```

4. Checking the influence of different number of cv on the accuracy scores.
   First, we run the for loop on numbers between 3 to 10 and the results was around 0.42. Next, we run the for loop on numbers between 100 to 1000 with steps of 100.

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score
gnb = GaussianNB()
scores_by_cv = {}

for i in range(100, 1001, 100):
  scores = cross_val_score(gnb, data, target_class, cv=i)
  scores_by_cv[i] = (scores, scores.mean())
for cv in scores_by_cv:
  print(f"Accuracy score [{cv}]: ", scores_by_cv[cv][1])
```

```
Accuracy score [100]:  0.44121927803379407
Accuracy score [200]:  0.44111960584437654
Accuracy score [300]:  0.441722475900558
Accuracy score [400]:  0.4421439393939394
Accuracy score [500]:  0.44246194503171243
Accuracy score [600]:  0.4422384884884885
Accuracy score [700]:  0.4430967741935483
Accuracy score [800]:  0.442756283068783
Accuracy score [900]:  0.44304814814814814
Accuracy score [1000]:  0.44332467532467534
```

5. Split the data to train and test.
6. Initialize Linear Regressor model and fit to the train data.
7. Predict the test data – the model returned all the results as float numbers, so we used the round function.

```
# Init LinearRegression Regressor (bedrooms)
reg = LinearRegression().fit(data_train, target_train)
pred = reg.predict(data_test)
pred
```

```
array([2.93403509, 3.63574002, 3.16232577, ..., 3.54513049, 3.748252  ,
       4.55318949])
```

8. Get the R^2 and MSE scores of the model.

```
[45] # Round the prediction bedrooms number to match the original df values (int)
     pred = list(map(round, pred))
     reg_r2 = r2(target_test, pred)
     reg_mse = mse(target_test, pred)
     print("R2 : ", reg_r2)
     print("MSE : ", reg_mse)
```

```
R2 :  0.31818225710200665
MSE :  0.566736062919269
```

9. Initialize RandomForestRegressor and using a for loop, we tried to test if the number of estimators affects the R^2 and MSE of the model.

```
[ ] from sklearn.ensemble import RandomForestRegressor as RFR
    scores = {}

    for i in range(100, 1001, 100):
      rfr = RFR(n_estimators=i, random_state=23)
      rfr.fit(data_train, target_train)
      pred = rfr.predict(data_test)
      scores[i] = (r2(target_test, pred), mse(target_test, pred))

    for score in scores:
      print(f"{score} -> R2: {r2(target_test, pred)}, MSE: {mse(target_test, pred)}")
```

```
100 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
200 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
300 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
400 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
500 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
600 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
700 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
800 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
900 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
1000 -> R2: 0.5175256066967985, MSE: 0.40103919407818645
```

```python
# Random Forest Regressor - each iteration we drop another feature
scores = {}
rfr = RFR(n_estimators=200, random_state=23)
for col in data_train.columns:
  rfr.fit(data_train.drop(col,axis=1), target_train)
  pred = rfr.predict(data_test.drop(col,axis=1))
  scores[col] = (r2(target_test, pred), mse(target_test, pred))

for score in scores:
  print(f"{score} -> R2: {r2(target_test, pred)}, MSE: {mse(target_test, pred)}")
```

```
price -> R2: 0.514752496979205, MSE: 0.403344240111034
bathrooms -> R2: 0.514752496979205, MSE: 0.403344240111034
sqft_living -> R2: 0.514752496979205, MSE: 0.403344240111034
sqft_lot -> R2: 0.514752496979205, MSE: 0.403344240111034
floors -> R2: 0.514752496979205, MSE: 0.403344240111034
waterfront -> R2: 0.514752496979205, MSE: 0.403344240111034
view -> R2: 0.514752496979205, MSE: 0.403344240111034
condition -> R2: 0.514752496979205, MSE: 0.403344240111034
grade -> R2: 0.514752496979205, MSE: 0.403344240111034
sqft_above -> R2: 0.514752496979205, MSE: 0.403344240111034
sqft_basement -> R2: 0.514752496979205, MSE: 0.403344240111034
yr_built -> R2: 0.514752496979205, MSE: 0.403344240111034
yr_renovated -> R2: 0.514752496979205, MSE: 0.403344240111034
sqft_living15 -> R2: 0.514752496979205, MSE: 0.403344240111034
sqft_lot15 -> R2: 0.514752496979205, MSE: 0.403344240111034
```

11. According to the values distribution in the ‘bedrooms’ class, we chose to divide the data to 2 main intervals – under 3 (<=3) and over 3 (>3).
12. Now, apply the changes to the target class.

```python
# convert target class to 2 intervals categories (under <=3 and over >3)
target_class_categories = target_class.apply(lambda x: 'under' if x <= 3 else 'over')
print(set(target_class_categories))
```

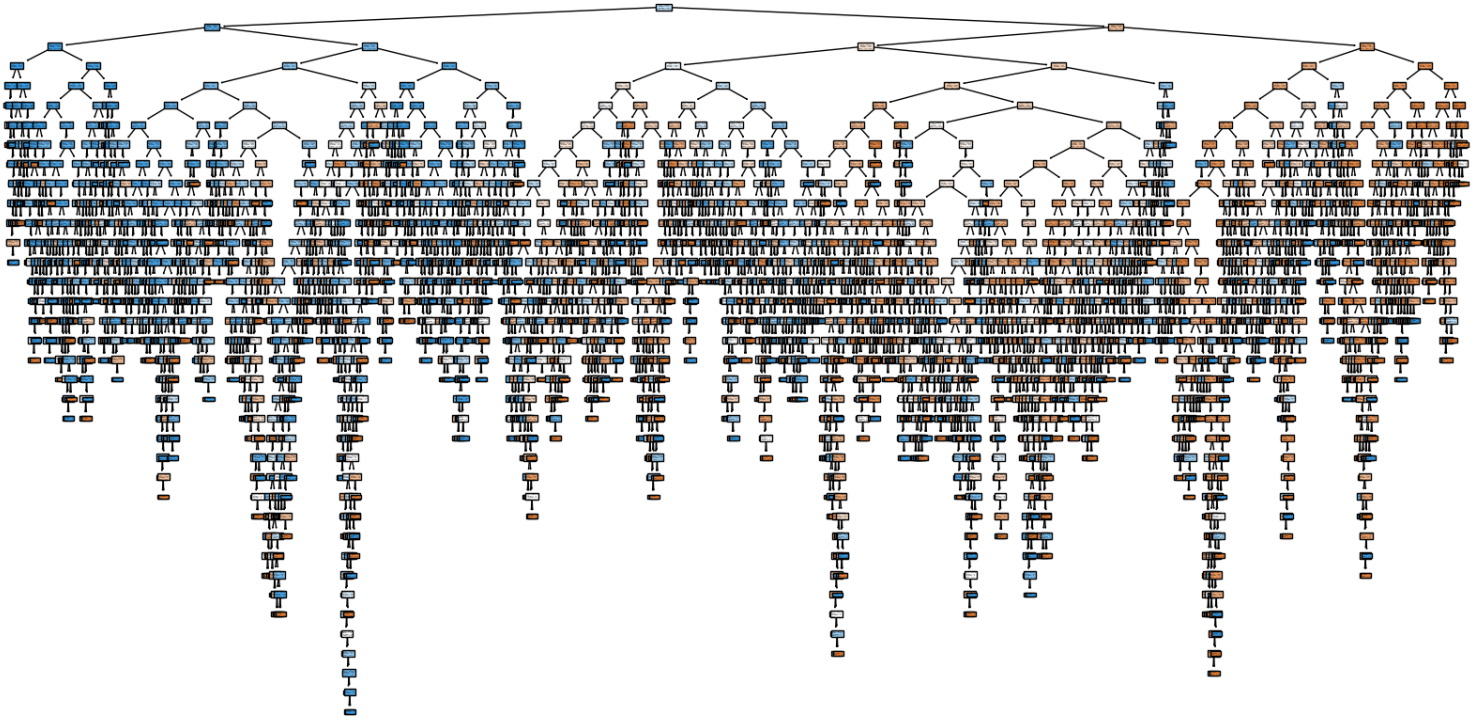
```
{'under', 'over'}
```

13. Split the data to train, test subsets.
14. Initialize the ID3 classifier and fit with the train set.

```python
# split the data and target_class_categories
data_train, data_test, target_train, target_test = train_test_split(data, target_class_categories, test_size=0.2, random_state=23)
```

```python
# Init ID3 classifier (bedrooms)
clf = DecisionTreeClassifier(criterion='entropy')
clf.fit(data_train, target_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(criterion='entropy')
```



15. Predict the train and test subset using ‘predict’ function.

```
# Predict on test set
id3_bedrooms_pred_test = clf.predict(data_test)

# Predict on train set
id3_bedrooms_pred_train = clf.predict(data_train)
```

16. Calculate the accuracy of the model and check whether the model suffers from over-fitting.

```
# Get ID3 train and test accuracy
id3_train_acc = accuracy_score(target_train, id3_bedrooms_pred_train)
id3_test_acc = accuracy_score(target_test, id3_bedrooms_pred_test)
print("Accuracy score (train): ", id3_train_acc)
print("Accuracy score (test): ", id3_test_acc)
if id3_train_acc - id3_test_acc > 0.1:
  print("The model suffers from Over-fitting")
else:
  print("No over-fitting :)")
id3_train_acc - id3_test_acc
```

```
Accuracy score (train):  1.0
Accuracy score (test):  0.7362942401110341
The model suffers from Over-fitting
0.26370575988896594
```

Conclusions –

Datasets are full of information. Sometimes, as in this case, we encountered a lot of unimportant data, which can lead to a poor-quality model, such as the zip code, ID, and date columns that we identified. Additionally, we attempted to extract more information from the “lat” and “long” columns because the model wouldn’t handle those numbers well, so we converted them to state and city. Then, we discovered that all the houses in the dataset were sold in Washington, so we dropped those columns to avoid introducing another feature that might confuse the models.

According to the tests conducted on several models with the data we received, we discovered that in some models, like the Random Forest Regressor, there is no difference between the number of estimators. As shown, we got the same results regardless of the number of estimators, ranging from 100 to 1000. Furthermore, it didn’t matter which feature we dropped; the results remained consistent, leading us to conclude that no single column had a significant influence over the others.

Eventually, we determined that the Random Forest Classifier was the best-performing model, providing the most reliable predictions despite the challenges posed by the dataset. This insight emphasizes the robustness of the Random Forest algorithm in handling complex datasets, making it a valuable tool for forecasting in real estate or similar domains.