HomePricePrediction

April 11, 2024

1 Toronto Housing Price Prediction Model

1.1 Introduction

This notebook details our efforts to develop a reliable housing price prediction model for the Toronto area. Given the absence of a robust model for this region, we utilized the Zillow API to gather data for training.

1.2 Data Collection

Training data was acquired using the Zillow API (see `get-data/zillow.py` script). The dataset includes various features of homes recently sold, or available for sale in GTA area.

1.3 Preprocessing

The dataset underwent significant preprocessing: - Removal of columns with excessive missing values or low predictive power. - Encoding of categorical variables such as `homeType` to numerical format suitable for modeling. - Removal of duplicate entries and entries with zero price.

1.4 Model Development

Two models were developed and evaluated: 1. **Linear Regression** 2. **Random Forest**

2 Loading the Dataset

[ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from scipy.stats import norm, skew

def load_dataset(filename):
    df = pd.read_csv(filename)
```

```python
    return df
df = load_dataset("zillow_homes_sold_toronto.csv")
```

[ ]:
```python
df['homeStatusForHDP'].value_counts()
```

[ ]:
```
homeStatusForHDP
RECENTLY_SOLD    5042
FOR_RENT           98
FOR_SALE           80
Name: count, dtype: int64
```

[ ]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5220 entries, 0 to 5219
Data columns (total 34 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   isShowcaseListing       5220 non-null   bool
 1   longitude               412 non-null    float64
 2   timeOnZillow            5100 non-null   float64
 3   zestimate               54 non-null     float64
 4   daysOnZillow            5220 non-null   int64
 5   zpid                    5220 non-null   int64
 6   taxAssessedValue        54 non-null     float64
 7   isUnmappable            5220 non-null   bool
 8   priceForHDP             5220 non-null   int64
 9   dateSold                5220 non-null   int64
 10  state                   5220 non-null   object
 11  isFeatured              5220 non-null   bool
 12  isPremierBuilder        5220 non-null   bool
 13  isPreforeclosureAuction 5220 non-null   bool
 14  lotAreaValue            1990 non-null   float64
 15  isNonOwnerOccupied      5220 non-null   bool
 16  homeStatus              5220 non-null   object
 17  latitude                412 non-null    float64
 18  lotAreaUnit             1990 non-null   object
 19  bedrooms                5220 non-null   int64
 20  zipcode                 5220 non-null   object
 21  homeStatusForHDP        5220 non-null   object
 22  isZillowOwned           5220 non-null   bool
 23  shouldHighlight         5220 non-null   bool
 24  homeType                5220 non-null   object
 25  bathrooms               5220 non-null   float64
 26  rentZestimate           55 non-null     float64
 27  price                   5220 non-null   int64
 28  city                    5220 non-null   object
 29  streetAddress           5220 non-null   object
```

```
 30   country                  5220 non-null   object
 31   currency                 5220 non-null   object
 32   listing_sub_type         5220 non-null   object
 33   livingArea               1251 non-null   float64
dtypes: bool(8), float64(9), int64(6), object(11)
memory usage: 1.1+ MB
```

[ ]:
```python
print(df.isnull().sum())
```

```
isShowcaseListing             0
longitude                  4808
timeOnZillow                120
zestimate                  5166
daysOnZillow                  0
zpid                          0
taxAssessedValue           5166
isUnmappable                  0
priceForHDP                   0
dateSold                      0
state                         0
isFeatured                    0
isPremierBuilder              0
isPreforeclosureAuction       0
lotAreaValue               3230
isNonOwnerOccupied            0
homeStatus                    0
latitude                   4808
lotAreaUnit                3230
bedrooms                      0
zipcode                       0
homeStatusForHDP              0
isZillowOwned                 0
shouldHighlight               0
homeType                      0
bathrooms                     0
rentZestimate              5165
price                         0
city                          0
streetAddress                 0
country                       0
currency                      0
listing_sub_type              0
livingArea                 3969
dtype: int64
```

3 Data Preprocessing and Clean up

3.1 Removing columns

Columns like state, homeStatus, lotAreaUnit, homeType, city, streetAddress, country, and currency are categorical and should be encoded to numerical values if you plan to use them in your model.

[ ]:
```python
def drop_columns(df):
    columns_with_many_missing = df.columns[df.isnull().mean() > .99]
    columns_with_low_predictive_power = [
        'zpid', 'streetAddress', 'listing_sub_type',
        'isUnmappable', 'isShowcaseListing', 'isFeatured',
        'isPremierBuilder', 'isPreforeclosureAuction',
        'isNonOwnerOccupied', 'isZillowOwned', 'shouldHighlight',
        'city', 'streetAddress', 'country', 'currency', 'zpid', 'lotAreaUnit',
        'homeStatusForHDP', 'homeStatus', 'state',
        'zestimate', 'taxAssessedValue', 'rentZestimate',
        'zipcode',
        'dateSold',
        'priceForHDP',
        # 'daysOnZillow'
    ]

    columns_to_drop = list(set(columns_with_many_missing.to_list() +␣
↪columns_with_low_predictive_power))

    df_dropped = df.drop(columns=columns_to_drop)

    print(f"Number of rows: {df.shape[0]}")
    print("columns_with_many_missing:", columns_with_many_missing)
    print("Columns dropped:", columns_to_drop)
    print("\nRemaining columns:", df_dropped.columns.tolist())

    df_dropped.head()
    return df_dropped

df = drop_columns(df)
```

```
Number of rows: 5220
columns_with_many_missing: Index([], dtype='object')
Columns dropped: ['isUnmappable', 'country', 'dateSold', 'rentZestimate',
'isNonOwnerOccupied', 'priceForHDP', 'homeStatus', 'isPreforeclosureAuction',
'zpid', 'zipcode', 'currency', 'city', 'listing_sub_type', 'isFeatured',
'state', 'shouldHighlight', 'isShowcaseListing', 'streetAddress', 'lotAreaUnit',
'taxAssessedValue', 'isPremierBuilder', 'isZillowOwned', 'zestimate',
'homeStatusForHDP']

Remaining columns: ['longitude', 'timeOnZillow', 'daysOnZillow', 'lotAreaValue',
'latitude', 'bedrooms', 'homeType', 'bathrooms', 'price', 'livingArea']
```

3.2 Categorical Data Encoding

Since homeType is a categorical variable, we'll need to encode it to use it in most machine learning models, which require numerical input.

One-Hot Encoding: This method converts the categorical variable into a series of binary columns, each representing a unique category. One-hot encoding is suitable when there is no ordinal relationship between the categories. Since homeType likely does not have a natural order, this method is appropriate.

[ ]:
```
df['homeType'].value_counts()
```

[ ]:
```
homeType
SINGLE_FAMILY    3577
CONDO             905
APARTMENT         461
TOWNHOUSE         222
MULTI_FAMILY       55
Name: count, dtype: int64
```

3.3 Handling Sparse Categories

Given the distribution of your homeType categories, it appears that MANUFACTURED, MULTI_FAMILY, and TOWNHOUSE categories are relatively sparse compared to the others. To clean up the homeType column and ensure your model remains robust, you could group these less frequent categories into an "Other" category. This approach simplifies your model while retaining the information these categories provide.

[ ]:
```
print(df.columns)
# print(type(df))
# df['homeType'] = df['homeType'].replace(['MANUFACTURED', 'MULTI_FAMILY',␣
 ↪'TOWNHOUSE'], 'Other')

# print("\nNew distribution of 'homeType':")
# print(df['homeType'].value_counts())
# print(type(df))
```

```
Index(['longitude', 'timeOnZillow', 'daysOnZillow', 'lotAreaValue', 'latitude',
       'bedrooms', 'homeType', 'bathrooms', 'price', 'livingArea'],
      dtype='object')
```

[ ]:
```
def encode_homeType(df):
    from sklearn.preprocessing import OneHotEncoder

    # Initializing the OneHotEncoder without specifying 'sparse'
    encoder = OneHotEncoder(drop='first')  # Assuming 'drop' argument is␣
 ↪supported in your version

    # Fit and transform 'homeType' column
```

```
    homeType_encoded_sparse = encoder.fit_transform(df[['homeType']])

    # Convert the sparse matrix to a dense matrix and then to a DataFrame
    homeType_encoded_df = pd.DataFrame(homeType_encoded_sparse.toarray(),␣
↪columns=encoder.get_feature_names_out(['homeType']))

    # Concatenate the encoded DataFrame back with the original DataFrame␣
↪(excluding the original 'homeType' column)
    df_final = pd.concat([df.drop('homeType', axis=1), homeType_encoded_df],␣
↪axis=1)

    print(f"Number of rows: {df_final.shape[0]}")
    df_final.head()
    return df_final

df = encode_homeType(df)
```

```
Number of rows: 5220
```

3.4 Removing duplicates

[ ]:
```
def remove_duplicates(df):
    # Check for duplicate rows
    print("Duplicates before:", df.duplicated().sum())

    # Remove duplicate rows
    df_removed = df.drop_duplicates()

    # Check again to confirm removal
    print("Duplicates after:", df_removed.duplicated().sum())

    print("Number of rows:", len(df_removed))
    df_removed.head()
    return df_removed

df = remove_duplicates(df)
```

```
Duplicates before: 857
Duplicates after: 0
Number of rows: 4363
```

3.5 Remove where price is 0

[ ]:
```
# print((df['price'] == 0).sum())
# print((df['priceForHDP'] == 0).sum())
# print(((df['price'] == 0) & (df['priceForHDP'] != 0)).sum())
# df = df[df['price'] != 0]
# print('------------')
```

```
# print((df['price'] == 0).sum())
# print((df['priceForHDP'] == 0).sum())
# print(((df['price'] == 0) & (df['priceForHDP'] != 0)).sum())

# num_rows = df.shape[0]
# print(f"Number of rows: {num_rows}")
```

4 Exploratory Data Analysis (EDA)

4.1 Describe

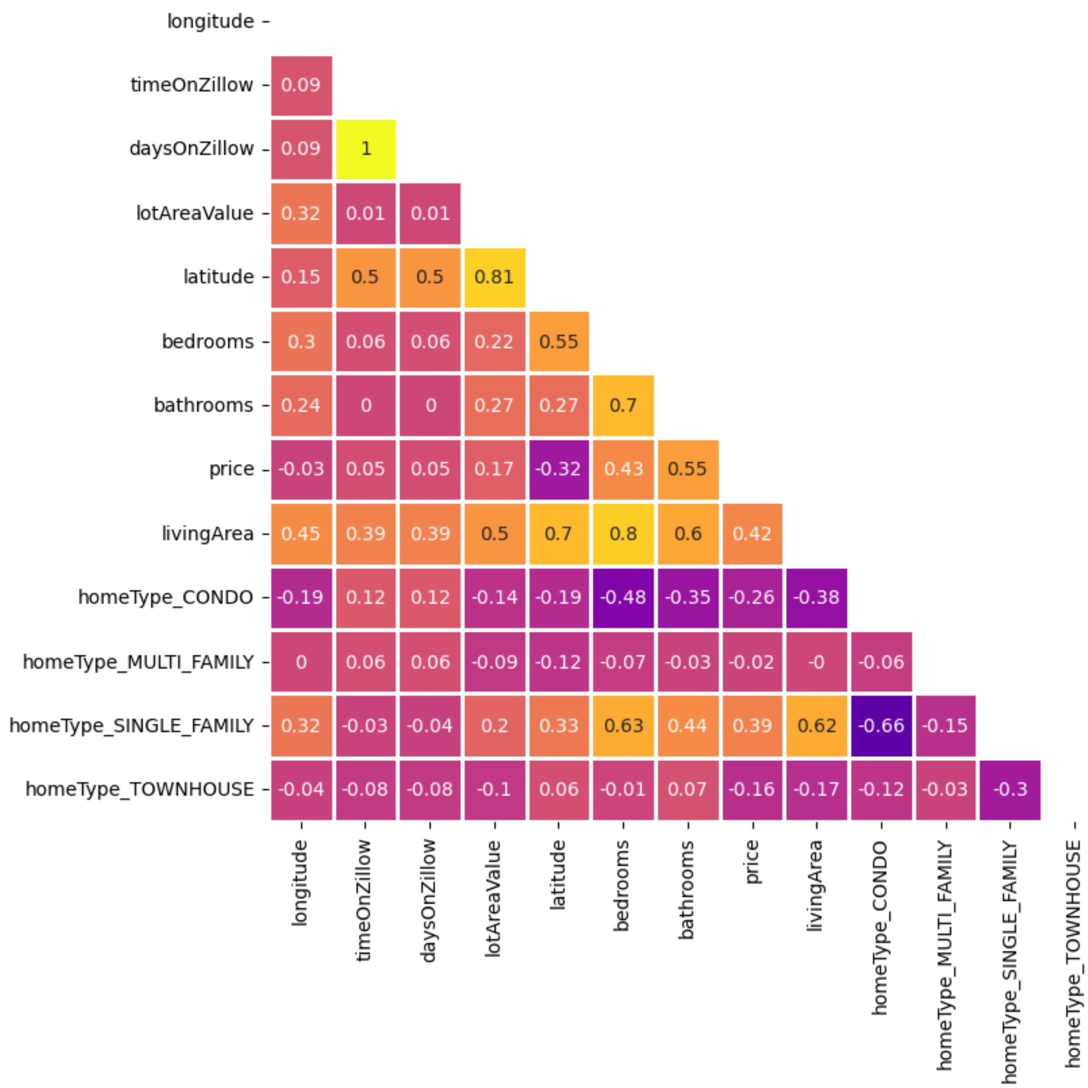
[ ]:
```
# display basic descriptive statistics for numerical columns
print(df.describe())
print(f"Number of rows: {df.shape[0]}")

df.head()
```

```
        longitude  timeOnZillow  daysOnZillow  lotAreaValue    latitude  \
count  355.000000  4.357000e+03   4363.000000   1496.000000  355.000000
mean   -79.400171  6.187088e+10    714.909008   3913.129733   43.919090
std      0.714517  1.791425e+10    208.894836   2488.899417    0.437308
min    -81.625465  2.004717e+09     -1.000000      0.000000   42.968273
25%    -79.784454  5.298072e+10    609.000000   2330.000000   43.667984
50%    -79.419914  6.222551e+10    720.000000   3234.000000   43.777637
75%    -79.351850  7.458072e+10    859.500000   5445.000000   44.275513
max    -77.059010  9.462557e+10   1095.000000  10419.000000   44.741947

          bedrooms    bathrooms         price   livingArea  homeType_CONDO  \
count  4363.000000  4363.000000  4.363000e+03   908.000000     4363.000000
mean      3.316755     2.675338  1.402629e+06  2190.943833        0.203071
std       1.664849     1.339355  8.614329e+05  1090.422818        0.402331
min       0.000000     0.000000  1.600000e+03   368.000000        0.000000
25%       2.000000     2.000000  8.311000e+05  1471.000000        0.000000
50%       3.000000     2.000000  1.250000e+06  1980.000000        0.000000
75%       5.000000     4.000000  1.650000e+06  2825.000000        0.000000
max       8.000000     8.000000  5.200000e+06  6534.000000        1.000000

       homeType_MULTI_FAMILY  homeType_SINGLE_FAMILY  homeType_TOWNHOUSE
count            4363.000000             4363.000000         4363.000000
mean                0.012606                0.627779            0.050882
std                 0.111579                0.483452            0.219783
min                 0.000000                0.000000            0.000000
25%                 0.000000                0.000000            0.000000
50%                 0.000000                1.000000            0.000000
75%                 0.000000                1.000000            0.000000
max                 1.000000                1.000000            1.000000
Number of rows: 4363
```

```
[ ]:    longitude  timeOnZillow  daysOnZillow  lotAreaValue   latitude  bedrooms  \
     0        NaN  6.205267e+10           718           NaN        NaN         0
     1 -79.419914  8.036947e+10           930           NaN  43.661144         0
     2        NaN  8.131987e+10           941           NaN        NaN         0
     3        NaN  8.563987e+10           991           NaN        NaN         0
     4        NaN  6.205267e+10           718           NaN        NaN         0

        bathrooms    price  livingArea  homeType_CONDO  homeType_MULTI_FAMILY  \
     0        1.0   430000         NaN             1.0                    0.0
     1        1.0  1452000       750.0             0.0                    0.0
     2        1.0   430000         NaN             1.0                    0.0
     3        4.0  1300000         NaN             0.0                    0.0
     4        1.0   430000         NaN             1.0                    0.0

        homeType_SINGLE_FAMILY  homeType_TOWNHOUSE
     0                     0.0                 0.0
     1                     1.0                 0.0
     2                     0.0                 0.0
     3                     1.0                 0.0
     4                     0.0                 0.0
```

4.2 Coorelation Matrix

```
[ ]: # this function is adopted from https://www.kaggle.com/code/shtrausslearning/
      ↪bayesian-regression-house-price-prediction/notebook

     def corrMat(df,id=False):
         corr_mat = df.corr().round(2)
         f, ax = plt.subplots(figsize=(8,8))
         # mask = np.zeros_like(corr_mat,dtype=np.bool)
         mask = np.zeros_like(corr_mat, dtype=bool)
         mask[np.triu_indices_from(mask)] = True
         sns.heatmap(corr_mat,mask=mask,vmin=-1,vmax=1,center=0,
                     cmap='plasma',square=False,lw=2,annot=True,cbar=False);plt.
      ↪show()

     # X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
      ↪random_state=42)
     corrMat(df)
```



bathrooms have a strong positive correlation with bedrooms (0.7), which is expected since more bedrooms often mean more bathrooms. based on this we create a 'Total Rooms' Feature: Since bathrooms and bedrooms are highly correlated, we create a new feature that combines them, such as total_rooms = bathrooms + bedrooms. This could help the model by consolidating two highly correlated features into one.

4.3 Add bedrooms+bathrooms

[ ]:
```python
def combine_and_drop(df):
    df['total_rooms'] = df['bedrooms'] + df['bathrooms']
    df = df.drop(['bedrooms', 'bathrooms'], axis=1)
    df.head()
    return df
```

```
df = combine_and_drop(df)
```

4.4 remove nan

[ ]:
```
def print_categorical_value_counts(df):
    categorical_columns = df.select_dtypes(include=['object', 'bool']).columns

    print("Categorical columns:"+ str(categorical_columns))

    for column in categorical_columns:
        print(f"Value counts for {column}:")
        print(df[column].value_counts())
        print("\n")

    print(f"Number of rows: {df.shape[0]}")
    df.head()
    return df

df = print_categorical_value_counts(df)
```

```
Categorical columns:Index([], dtype='object')
Number of rows: 4363
```

[ ]:
```
print(df.columns[df.isna().any()].tolist())

nan_cols = df.columns[df.isna().any()].tolist()

# df = df[df['price'].notna()]
# df = df[df['bedrooms'].notna() & df['bathrooms'].notna()]
# df = df[df['livingArea'].notna()]

for col in nan_cols:
    print(f"{col}: {df[col].isna().sum()}")

# Count columns with NaN values
nan_cols_count = df.columns[df.isna().any()].shape[0]
print(f"Number of columns with NaN values: {nan_cols_count}")

# Remove columns with NaN values
# df = df.dropna(axis=1)
print(f"Number of columns with NaN values: {nan_cols_count}")

print(f"Number of rows: {df.shape[0]}")
# print((df['bedrooms'] == 0).sum())
# df = df[df['bedrooms'] != 0]
```

```
print(f"Number of rows: {df.shape[0]}")
df.head()
```

```
['longitude', 'timeOnZillow', 'lotAreaValue', 'latitude', 'livingArea']
longitude: 4008
timeOnZillow: 6
lotAreaValue: 2867
latitude: 4008
livingArea: 3455
Number of columns with NaN values: 5
Number of columns with NaN values: 5
Number of rows: 4363
Number of rows: 4363
```

[ ]:

```
    longitude  timeOnZillow  daysOnZillow  lotAreaValue   latitude    price  \
0         NaN  6.205267e+10           718           NaN        NaN   430000
1  -79.419914  8.036947e+10           930           NaN  43.661144  1452000
2         NaN  8.131987e+10           941           NaN        NaN   430000
3         NaN  8.563987e+10           991           NaN        NaN  1300000
4         NaN  6.205267e+10           718           NaN        NaN   430000

   livingArea  homeType_CONDO  homeType_MULTI_FAMILY  homeType_SINGLE_FAMILY  \
0         NaN             1.0                    0.0                     0.0
1       750.0             0.0                    0.0                     1.0
2         NaN             1.0                    0.0                     0.0
3         NaN             0.0                    0.0                     1.0
4         NaN             1.0                    0.0                     0.0

   homeType_TOWNHOUSE  total_rooms
0                 0.0          1.0
1                 0.0          1.0
2                 0.0          1.0
3                 0.0          4.0
4                 0.0          1.0
```

4.5 Handle missing values

[ ]:

```
def handle_missing_values(df):
    df = df[df['price'].notna()]
    df = df.dropna(axis=1)
    return df

df = handle_missing_values(df)
```

5 Training the Model

Implementing a Linear Regression model and split the data into training and testing sets

[ ]:
```python
# Final columns before training
print(f"Number of rows: {df.shape[0]}")

df.head()
# df.to_csv('training_dataset_saved.csv', index=False)
```

```
Number of rows: 4363
```

[ ]:
```
   daysOnZillow    price  homeType_CONDO  homeType_MULTI_FAMILY  \
0           718   430000             1.0                    0.0
1           930  1452000             0.0                    0.0
2           941   430000             1.0                    0.0
3           991  1300000             0.0                    0.0
4           718   430000             1.0                    0.0

   homeType_SINGLE_FAMILY  homeType_TOWNHOUSE  total_rooms
0                     0.0                 0.0          1.0
1                     1.0                 0.0          1.0
2                     0.0                 0.0          1.0
3                     1.0                 0.0          4.0
4                     0.0                 0.0          1.0
```

5.1 Linear Regression

```
[ ]: from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
import numpy as np

def train_linear_regression(df, target):
    X = df.drop(target, axis=1)
    y = df[target]

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    scaler = StandardScaler()

    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    linear_model = LinearRegression()
    linear_model.fit(X_train_scaled, y_train)

    y_pred = linear_model.predict(X_test_scaled)
    return y_test, y_pred

y_test_lin, y_pred_lin = train_linear_regression(df, 'price')
```

5.1.1 Evaluation

```
[ ]: from sklearn.metrics import r2_score

def calculate_metrics(y_test, y_pred, model_name):
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r_squared = r2_score(y_test, y_pred)
    print(f"Root Mean Squared Error (RMSE) on Test Set for {model_name}:␣
 ↪{rmse}")
    print(f"R-squared (R2) on Test Set for {model_name}: {r_squared}")
  ␣
 ↪print("---------------------------------------------------------------------------\n")

calculate_metrics(y_test_lin, y_pred_lin, 'Linear Regression')
```

```
Root Mean Squared Error (RMSE) on Test Set for Linear Regression:
704751.6957342114
R-squared (R2) on Test Set for Linear Regression: 0.2774396916125408
---------------------------------------------------------------------------
```

5.2 Random Forest

[ ]:
```
from sklearn.ensemble import RandomForestRegressor

def train_random_forest(df, target):
    X = df.drop(target, axis=1)
    y = df[target]

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    scaler = StandardScaler()

    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    random_forest_model = RandomForestRegressor(n_estimators=100,␣
 ↪random_state=42)
    random_forest_model.fit(X_train_scaled, y_train)

    y_pred = random_forest_model.predict(X_test_scaled)
    return y_test, y_pred
y_test_rf, y_pred_rf = train_random_forest(df, 'price')
```

5.2.1 Evaluation

[ ]:
```
calculate_metrics(y_test_rf, y_pred_rf, "Random Forest")
```

```
Root Mean Squared Error (RMSE) on Test Set for Random Forest: 155818.39580078336
R-squared (R2) on Test Set for Random Forest: 0.9646784868734193
---------------------------------------------------------------------------
```

5.3 Update coordinates and rerun

We cannot use addresses directly in the model, and the original dataset had only a few records with both latitude and longitude set. However, addresses are crucial for home price prediction. To incorporate addresses into the model, we used a Python script (get-data/geoapify.py) to call the Geoapify API and fetch coordinates (latitude and longitude) for the addresses. These coordinates were then saved in a CSV file (zillow_homes_sold_toronto_coordinates_added.csv), and the dataset was updated with the retrieved coordinates. The models will be rerun using this new dataset.

[ ]:
```
df = load_dataset("zillow_homes_sold_toronto_cordinations_added.csv")
df = drop_columns(df)
df = encode_homeType(df)
df = remove_duplicates(df)
df = combine_and_drop(df)
df = handle_missing_values(df)
```

```python
df.head()
```

```
Number of rows: 5220
columns_with_many_missing: Index([], dtype='object')
Columns dropped: ['isUnmappable', 'country', 'dateSold', 'rentZestimate',
'isNonOwnerOccupied', 'priceForHDP', 'homeStatus', 'isPreforeclosureAuction',
'zpid', 'zipcode', 'currency', 'city', 'listing_sub_type', 'isFeatured',
'state', 'shouldHighlight', 'isShowcaseListing', 'streetAddress', 'lotAreaUnit',
'taxAssessedValue', 'isPremierBuilder', 'isZillowOwned', 'zestimate',
'homeStatusForHDP']

Remaining columns: ['longitude', 'timeOnZillow', 'daysOnZillow', 'lotAreaValue',
'latitude', 'bedrooms', 'homeType', 'bathrooms', 'price', 'livingArea']
Number of rows: 5220
Duplicates before: 857
Duplicates after: 0
Number of rows: 4363
```

[ ]:

```
   longitude  daysOnZillow   latitude    price  homeType_CONDO  \
0 -79.400083           718  43.636951   430000             1.0
1 -79.419914           930  43.661144  1452000             0.0
2 -79.400083           941  43.636951   430000             1.0
3 -80.344809           991  43.355007  1300000             0.0
4 -79.400083           718  43.636951   430000             1.0

   homeType_MULTI_FAMILY  homeType_SINGLE_FAMILY  homeType_TOWNHOUSE  \
0                    0.0                     0.0                 0.0
1                    0.0                     1.0                 0.0
2                    0.0                     0.0                 0.0
3                    0.0                     1.0                 0.0
4                    0.0                     0.0                 0.0

   total_rooms
0          1.0
1          1.0
2          1.0
3          4.0
4          1.0
```

[ ]:

```python
y_test_lin, y_pred_lin = train_linear_regression(df, 'price')
calculate_metrics(y_test_lin, y_pred_lin, "Linear Regression")

y_test_rf, y_pred_rf = train_random_forest(df, 'price')
calculate_metrics(y_test_rf, y_pred_rf, "Random Forest")
```

```
Root Mean Squared Error (RMSE) on Test Set for Linear Regression:
666110.8385600615
R-squared (R2) on Test Set for Linear Regression: 0.35450208646839176
```

```
--------------------------------------------------------------------------

Root Mean Squared Error (RMSE) on Test Set for Random Forest: 74350.92356764115
R-squared (R2) on Test Set for Random Forest: 0.991957805405347
--------------------------------------------------------------------------
```

6 RMSE Analysis

The RMSE of 74,350.92 is considered in the context of the dataset's statistics:

Average Price Context: The average price in the dataset is approximately $1,402,629. An RMSE of 74,350.92 represents about 5.3% of the average price, indicating a relatively small error relative to the average home value.

Compared to Standard Deviation: The standard deviation of the prices is $861,432.95, suggesting significant variability in house prices. An RMSE much smaller than the standard deviation indicates that the model predictions are reasonably accurate within the existing price variability.

Relative Error Scale: Considering the price range in the dataset, from a minimum of $1,600 to a maximum of $5,200,000, an RMSE of 74,350.92 demonstrates a moderate level of precision, especially effective for middle and high-end properties.

7 Conclusion

The Random Forest model significantly outperformed the Linear Regression model, indicating its better suitability for this application. The results validate the effectiveness of the Random Forest model in predicting housing prices with high accuracy. Future efforts might focus on integrating more dynamic features and improving data collection to refine predictions further. This is likely due to:

Non-linearity: Random Forest can capture complex, non-linear relationships between features that affect housing prices

Robust to Outliers: This model is less sensitive to extreme values, which are common in real estate markets, like the occasional luxury home sales that do not fit typical pricing patterns.

Feature Interactions: Random Forest automatically considers how combinations of features impact house prices. For example, the model can intuitively assess how the interaction between property size and its location (represented by latitude and longitude) affects its value, unlike Linear Regression, which requires explicitly crafted interaction terms to capture such complex relationships.

Prevention of Overfitting: Random Forest uses techniques like bootstrapping and feature randomness to build diverse trees that generalize better to new data, reducing the risk of overfitting seen in complex Linear Regression models.

Handles Various Data Types: It effortlessly processes different data types and does not require extensive preprocessing for categorical variables, streamlining the modeling process.

8 Model imporovements

Following potentially can be done to improve model's performance:

Feature Engineering: Develop new features that capture more nuances of the real estate market, such as proximity to amenities or property condition or walking score.

Data Enrichment: Integrate additional data sources, such as economic indicators or demographic statistics, to provide more context for predictions.

Advanced Modeling Techniques: Experiment with more complex models like Gradient Boosting Machines (GBM) or neural networks that might capture patterns better.

Parameter Tuning: Optimize hyperparameters of the Random Forest model through grid search or random search to enhance model accuracy.

Handling Outliers: Identify and manage outliers in the dataset that may skew the model predictions.