

Application of Machine Learning Models to Predict Property Prices

Pranav Pathak *

March 22, 2023

Abstract

The real estate market has always been filled with uncertainty. There are many players and economical factors that continuously affect property prices. For many home buyers, it is hard to estimate the house prices due to the numerous factors involved. The authors of this paper aim to develop an algorithm to help with the estimations. Thus, the research presented in this paper aims to develop and test machine learning models, then analyze the results of these models as applicable to the housing market. We studied and worked with two datasets, and developed numerous models to apply the best model for the data. We evaluated the results of the models and compared them to each other to draw inferences and conclusions. The results for one dataset were better compared to the results for the other dataset. Similarly, the predicted property prices were closer to the actual prices for one dataset than for the other dataset. We observed multiple linear and non-linear trends within these datasets by closely studying the results. Throughout our study, we have found that machine learning can be used to predict property prices with varying degrees of effectiveness. The results mainly depend on the quality of the data, and how well they reflect the values of the people.

1 Introduction

1.1 A Brief Introduction to Machine Learning

Machine Learning is a vast field and has varied applications. It is applied to solve problems where data analysis is not humanly possible. The problems that machine learning can solve are usually divided into three main types: supervised learning, unsupervised learning, and semi-supervised learning [BD17]. This paper focuses on using supervised learning. All three types of problems involve using data to teach a model to learn from the data to make future predictions. However, a key factor in supervised learning is that it involves features and labels. Labels are the results or outcomes that we want to predict. Features are

*Advised by: Dr. Guillermo Goldsztein

the characteristics or variables that are used to predict the label. Essentially, features are the factors influencing the label. The set of data containing the features and the labels used to teach the model is called the training set. Each training set contains examples. Examples are one set of features and labels. For instance, if we were predicting the number of people in a country based on certain features, one country would be one example.

1.2 Notation

This paper uses specific notation that is important to know to fully grasp the processes and methods that we used. Some notation used in this paper is: x_1, x_2, \dots, x_n , denotes the features, where n is the number of features. y_1, y_2, \dots, y_k , denote the labels, where k is the number of labels.

μ represents the average of a set of numbers

σ represents the standard deviation of a set of numbers

L represents the number of layers in a neural network $n^{[l]}$ denotes the number of nodes in layer l of a neural network

$$f^{[1]}(n^{[1]}), f^{[2]}(n^{[2]}), f^{[3]}(n^{[3]}), \dots, f^{[L-1]}(n^{[L-1]}) \quad (1)$$

denotes the architecture of a neural network

1.3 Thesis Statement

The real estate market has always been filled with uncertainty. There are many players and economic factors that continuously affect property prices. For many home buyers, it is hard to estimate the house prices due to the numerous factors involved. The authors of this paper aim to develop an algorithm to help with the estimations. Thus, the research presented in this paper aims to develop and test machine learning models, then analyze the results of these models as applicable to the housing market.

We applied supervised learning to two housing prices datasets. Throughout this paper, we will describe the processes that we used, and the outcomes of applying supervised learning to these datasets. The aim of this research is to compare the results within and between the regions. We believe that the research method is extendable to the larger housing market, as we have seen it work for two different geographies. We support the method of obtaining data from official real estate organizations, so as to get validated and current results. We also believe that a housing market expert or a real estate investor could be consulted in order to improve the data in the datasets.

2 Analyzing the Datasets

2.1 King County Dataset [Cha18]

The first dataset that we studied was the King County dataset. King County is located in Washington state. The dataset was obtained from Kaggle, and contained the sale prices of houses from May 2014 to May 2015. Over the last five years in King County, house prices have generally increased. The median home sale price has grown from around \$600,000 to around \$850,000, representing an average compounded growth of 7.2% per year. The number of homes sold has mostly stayed within the range of 4,500 homes sold and 1,500 homes sold over the past five years [Red]. Similarly, during 2012 - 2017, the median home sale price increased by 53%. This represents an annual compounded growth of 6.3% per year. King County is also a large area spanning many cities, including Seattle. Seattle is a hub for many major companies, which has contributed to accelerating house prices [Aff18].

First, we read the dataset into a pandas dataframe. This allowed us to manipulate and study the dataset easily. We did this by using the pandas `.read_csv()` method. The dataset contains the following features: `id`, `date`, `price`, `bedrooms`, `bathrooms`, `sqft_living`, `sqft_lot`, `floors`, `waterfront`, `view`, `condition`, `grade`, `sqft_above`, `sqft_basement`, `yr_built`, `yr_renovated`, `zipcode`, `lat`, `long`, `sqft_living15`, `sqft_lot15`. The features mean:

1. The `id` feature represents the identifier of the house [Spa20].
2. The `date` feature describes the date during which the house was sold.
3. The `price` gives the price for which the house was sold [Spa20].
4. The `bedrooms` and `bathrooms` features give the number of bedrooms and bathrooms in the property respectively.
5. The `sqft_living` and the `sqft_lot` features give the amount of living area in the property in square feet [Spa20] and the square footage of the land on which the house is built [Spa20].
6. The `floors` feature represents the number of floors in the property.
7. The `waterfront` feature indicates whether the property is a waterfront property or not.
8. The `view` feature measures the quality of the view.
9. The `condition` feature describes the current state of the property, while the `grade` feature describes the level of excellency of the build of the house [Spa20].
10. The `sqft_above` feature gives the amount of square feet above ground [Spa20], and the `sqft_basement` feature gives the size of the basement in square feet.
11. The `yr_built` feature gives the year that the property was built.
12. The `yr_renovated` feature gives the year that the property was renovated, if it ever was.
13. The `zipcode` feature gives the zip code of the property.
14. The `lat` and `long` features give the latitude and longitude of the property.

15. The `sqft_living15` and `sqft_lot15` features give the average of the properties' living areas and lot sizes of the nearest 15 houses in square feet [Spa20].

In this dataset, the label is the price, and the other columns are the features. We did not need all the features; we deleted some of them. We did not need the `id` feature, as we could not use it in any way to train the model. Using the `date`, `zipcode`, `lat` (latitude) and `long` (longitude) features in the model are part of the next step. However, for this paper, we deleted these features. We included the `floor` feature, as that does influence house prices. We also used the `condition` and `grade` features. The `yr_built` feature was used to train the machine learning models. Now, we will examine the other features as well. We deleted the `sqft_living15` and `sqft_lot15` features as we believed that these features might not be as helpful for the model. The `bedrooms` and `bathrooms` features are important factors affecting house prices. So, we kept these features. Lastly, the size of the lot of the property is also quite important, so we have included it.

Square feet living:

```
print((df['sqft_living'] <= 300).sum())  
1
```

Figure 1: King County living square feet feature

Only one of the values for the amount of living area was less than 300. A living area less than 300 ft^2 did not make much sense. However, the example whose `'sqft_living'` was less than 300 square feet did not have any bedrooms. So, we ignored this particular example, as we deleted it later.

Waterfront:

```
print(df['waterfront'].unique())  
[0 1]
```

Figure 2: King County waterfront feature

```
print((df['waterfront'] == 0).sum())  
21450
```

Figure 3: King County waterfront feature

This column had only zeros and ones (true or false). However, almost all of the properties were not waterfront properties, so it was quite hard to make the set “balanced” for this feature. A set is *balanced* for a feature if the values for the features are roughly evenly distributed. For this feature, around 99.25% of the values were zero. So, we deleted this feature. A similar problem occurred with the view feature, and we deleted the view feature as well.

Square feet basement:

There were some houses that had basements with sizes between 0 ft^2 and 300 ft^2 . So, we replaced these basement sizes with zero square feet, as their areas are negligible.

Year renovated:

```
print((df['yr_renovated'] != 0).sum())  
914
```

Figure 4: King County living square feet feature

The values for the year renovated feature are reasonable. Note that the year renovated is 0 if the house has not been renovated [Spa20]. However, only 914 houses were renovated. The year renovated can be an important factor in determining the price of a property. Therefore, we trained a few models with the year renovated feature, and a few models without the year renovated feature. This allowed us to determine if there is a relationship between house prices and the year in which a house was renovated.

2.2 The India Dataset [Tri22]

The second dataset that we worked with was the India dataset. This dataset was composed of housing data from the cities of Hyderabad, Pune, Kolkata and Gurgaon, and was also obtained through Kaggle. These four cities are major cities in India and have similar economies due to the booming software industries in these cities. This led to an increase in housing prices. In the past, these cities were considered as second tier cities. However, the software industry expansion led to a more attractive housing market. Before this, housing prices were still increasing, but at a slower rate. In the first three months of 2022, the average housing prices went up by 7% in Hyderabad [Ser22].

Now, we gathered the dataset and studied it to understand it. However, the dataset in Kaggle contained housing data for many cities. We chose four cities whose housing markets were similar in pricing. Then, we combined the housing data for these four cities in an excel sheet, which we read into a pandas

dataframe by using the `pandas.read_excel()` method.

The dataframe contained the following features:
Unnamed: 0, Type_of_plot, City, Total Price, Unit, Rupee_per_area, Type, Open_Date, Description, Area, Apartment, and Project Name. These features mean:

1. The Unnamed: 0 feature contains indices from the excel sheet.
2. The Type_of_plot feature describes the type of apartment.
3. The City feature describes the city in which each apartment is located.
4. The Total Price (the label) describes the selling price of each apartment.
5. The Unit feature describes the unit in which the price of each apartment is (lakhs or crores).
6. The Rupee_per_area feature describes the cost of each square feet in the apartment [Tri22].
7. The Type feature describes the status of the apartment (ready or under construction).
8. The Open_Date feature contains many different values.
9. The Description feature contains strings that describe the property.
10. The Area feature gives the size of the property in square feet [Tri22].
11. The Apartment feature contains the same values as the Type of Plot feature.
12. The Project Name feature describes the name under which the property was built.

Similar to the King County dataset, we deleted some of the features in this dataset. We deleted the 'Unnamed: 0' feature, as they are unusable indices from the excel sheet. We deleted the description feature, as for this paper, we decided not to include it. We used the area feature, as we believed that it was important. We deleted the apartment feature as it was the same as the type of plot feature. Finally, we deleted the project name feature because we believed that it would not be very important to include in the model. This feature might have also hindered the model, as a new project name for a new apartment would likely confuse the model. We deleted the city feature as we felt that it would be hard to use in order to achieve a more general model for property prices. However, we have explained this more in the observations and analysis section. Lastly, we deleted the unit feature, as the unit was already included in the label.

Type of plot:

```
df['Type_of_plot'].unique()
array(['2 BHK Apartment', '1 BHK Apartment', '3 BHK Apartment',
       '4 BHK Apartment', '1 RK Studio Apartment'], dtype=object)
```

Figure 5: King County living square feet feature

The type of property could be one to four BHK apartments, and 1 RK studio apartments. BHK means bedrooms, halls (living rooms), and kitchen. A 1 RK studio apartment is an apartment with one distinct bedroom and kitchen [Sai22]. For this feature, we decided to delete the ‘BHK Apartment’ or the ‘RK Studio Apartment’ part of each string. So, we considered 1 RK to be equivalent to 1 BHK. We converted this feature into the number of BHK.

Total Price:

```
print(df['Total Price'].unique())

[' 45.36 L' ' 87 L' ' 70.62 L' ' 51.04 L' ' 69 L' ' 46.8 L' ' 55.5 L'
 43.99 L' ' 30 L' ' 57.5 L' ' 44.75 L' ' 29.67 L' ' 68 L' ' 46.13 L'
 49.5 L' ' 1.7 Cr' ' 35.37 L' ' 86 L' ' 74.83 L' ' 1.19 Cr' ' 1.35 Cr'
 34.99 L' ' 25.38 L' ' 1.03 Cr' ' 81 L' ' 25 L' ' 28.75 L' ' 65.49 L'
 22.98 L' ' 45.98 L' ' 46.21 L' ' 50.05 L' ' 19.74 L' ' 26.98 L' ' 19 L'
 1.08 Cr' ' 65 L' ' 42.68 L' ' 27.97 L' ' 52.08 L' ' 1.09 Cr' ' 53.3 L'
 60.7 L' ' 1.4 Cr' ' 94.98 L' ' 64.47 L' ' 42.49 L' ' 60.68 L'
 82.65 L' ' 1.25 Cr' ' 82.42 L' ' 50.23 L' ' 1.57 Cr' ' 90.75 L'
 1.68 Cr' ' 3.34 Cr' ' 1.21 Cr' ' 83.92 L' ' 57.25 L' ' 89.22 L'
 81.83 L' ' 1.63 Cr' ' 93.72 L' ' 70.41 L' ' 98.35 L' ' 33 L' ' 75 L'
 50 L' ' 44 L' ' 24.5 L' ' 2.25 Cr' ' 26 L' ' 1.1 Cr']
```

Figure 6: King County living square feet feature

The label needed to be transformed. For the label, ‘L’ means lakhs, and ‘C’ means crores. These units were used as this dataset used the Indian place value system. The conversion is: 1 lakh = 1,00,000 (the equivalent of 100,000), and 1 crore = 1,00,00,000 (the equivalent of 10,000,000). So, we converted each of the values for the label into a number by multiplying it with the appropriate value.

Rupee per area:

```
print(df['Rupee_per_area'].unique())

['7,486 / sq ft' '11,725 / sq ft' '8,529 / sq ft' '7,452 / sq ft'
'8,834 / sq ft' '6,705 / sq ft' '8,823 / sq ft' '5,812 / sq ft'
'6,479 / sq ft' '7,646 / sq ft' '9,040 / sq ft' '8,778 / sq ft'
'7,709 / sq ft' '6,907 / sq ft' '6,599 / sq ft' '7,511 / sq ft'
'12,292 / sq ft' '7,417 / sq ft' '8,182 / sq ft' '8,940 / sq ft'
'9,844 / sq ft' '9,574 / sq ft' '3,804 / sq ft' '3,600 / sq ft'
'7,157 / sq ft' '3,133 / sq ft' '2,677 / sq ft' '4,125 / sq ft'
'8,911 / sq ft' '4,133 / sq ft' '5,300 / sq ft' '3,700 / sq ft'
'5,150 / sq ft' '5,500 / sq ft' '2,675 / sq ft' '3,500 / sq ft'
'2,960 / sq ft' '9,467 / sq ft' '4,955 / sq ft' '3,100 / sq ft'
'4,163 / sq ft' '4,999 / sq ft' '4,299 / sq ft' '5,705 / sq ft'
'7,800 / sq ft' '4,200 / sq ft' '6,965 / sq ft' '5,600 / sq ft'
'7,499 / sq ft' '5,200 / sq ft' '3,900 / sq ft' '5,999 / sq ft'
'9,980 / sq ft' '6,849 / sq ft' '5,800 / sq ft' '4,599 / sq ft'
'8,958 / sq ft' '5,617 / sq ft' '15,688 / sq ft' '9,277 / sq ft'
'9,064 / sq ft' '9,039 / sq ft' '5,076 / sq ft' '6,500 / sq ft'
'5,147 / sq ft' '5,000 / sq ft' '4,400 / sq ft' '4,285 / sq ft'
'4,298 / sq ft' '11,392 / sq ft' '4,727 / sq ft' '6,432 / sq ft'
'7,371 / sq ft' '6,359 / sq ft']
```

Figure 7: King County living square feet feature

Similar to the label, we had to transform this feature into numbers.

Type:

For this feature we performed a process *one-hot encoding*. *One-hot encoding* is a process of transforming categorical variables (variables that are not num-

```
print(df['Type'].unique())

['Under Construction' 'Ready to move']
```

Figure 8: King County living square feet feature

bers) into numerical variables (variables that are numbers).

Open date:

```
['Possession by Aug 2022' 'Possession by Nov 2023'
 'Possession by May 2024' 'Possession by Feb 2023'
 'Possession by Jun 2025' 'Possession by Dec 2025'
 'Possession by Sep 2024' 'Possession by Nov 2022' '5 - 6 years old'
 '0 - 1 year old' 'Possession by Nov 2025' 'Possession by Jan 2025'
 'Possession by Dec 2024' '2 Bathrooms' 'Possession by Dec 2026'
 'Possession by Sep 2025' 'Possession by Sep 2022' '8 - 9 years old'
 '1 - 2 years old' 'Possession by Nov 2024' 'Possession by Apr 2024'
 'Possession by May 2022' 'Possession by May 2026'
 'Possession by Oct 2022' 'Possession by Oct 2027'
 'Possession by Apr 2027' 'Possession by May 2023'
 'Possession by Apr 2023' 'Possession by Sep 2026' '2 - 3 years old'
 'Possession by May 2025' 'Possession by Mar 2026' '6 - 7 years old'
 '3 Bathrooms' '9 - 10 years old' '20 years old' '4 Bathrooms' 'Resale']
```

Figure 9: King County living square feet feature

This feature contained too many random values. So, we chose to change this feature to describe the age of each property. For example, if the open date was ‘Possession by . . .’, then we replaced it with 0, as the house was 0 years old.

3 Preparing the Datasets

3.1 The King County Dataset

After examining the King County dataset in detail, we made changes to the dataset for the model to work well with the dataset. We implemented the transformations that we described in the previous section. Note that we had to remove examples last, as if we removed the examples first, then the code would have returned an error when we tried to access a deleted element’s index.

```
df = df[df['bedrooms'] > 0]
df = df[df['bathrooms'] > 0]
print(df.shape)

(21597, 14)
```

Figure 12: Removing examples that had no bedrooms or bathrooms

These were all the modifications that we described previously. However, we


```
del df['zipcode']
del df['id']
del df['lat']
del df['long']
del df['date']
del df['waterfront']
del df['view']
print(df.columns.values)

['price' 'bedrooms' 'bathrooms' 'sqft_living' 'sqft_lot' 'floors'
 'condition' 'grade' 'sqft_above' 'sqft_basement' 'yr_built'
 'yr_renovated' 'sqft_living15' 'sqft_lot15']
```

Figure 10: Deleting unwanted features and printing the remaining variables

```
for x in range(len(df['sqft_basement'])):
    if df['sqft_basement'][x] < 300 and df['sqft_basement'][x] != 0:
        df['sqft_basement'][x] = 0
```

Figure 11: Replacing basements less than 300 square feet with 0

trained a few models with the year renovated feature, and a few models without the year renovated feature. In the case that we kept the year renovated feature, we needed to make the set balanced. We did this by collecting the houses that had been renovated and added them a number of times until the set became balanced. Before we did this, however, we split the dataframe into the label and the features. We assigned the label to the values in the ‘price’ column, and the feature to the values of the columns other than the ‘price’ column.

When we removed the year renovated feature, we did this using the same method [Yan] that we used when initially deleting features. Now, we were ready to split the data into the training and validation sets. We did this by using the `train_test_split` method from the `sklearn.model_selection` library. We chose to do this to prevent overfitting. Overfitting is a scenario in which the model becomes too adapted to the training set and cannot predict new examples well. So, we split the dataset in order to train the model with some examples and use the other examples to test how well the model does on examples it has not seen before. We train the model with the training set, and we test the model using the validation set. Then, we choose the model that produces the lowest error on the validation set.

```
f_train, f_val, l_train, l_val = train_test_split(fFeatures, fLabel, test_size = 0.25, random_state = 4) #with renovated houses
f_train, f_val, l_train, l_val = train_test_split(features, label, test_size = 0.25, random_state = 4) #without renovated houses
```

Figure 13: We split the dataset into the training and validation sets

We have also scaled the data. Scaling the data means that we reduce their size dramatically, allowing the model to work better with the data. The way in which we scaled is [Sk]:

$$(x_1, x_2, x_3, \dots, x_n) = \left(\frac{x_1 - \mu}{\sigma}, \frac{x_2 - \mu}{\sigma}, \frac{x_3 - \mu}{\sigma}, \dots, \frac{x_n - \mu}{\sigma} \right) \quad (2)$$

(with the same method of scaling the labels). As we have imported the preprocessing library, we scaled the data by using the methods included in this library.

```
scaler_f = preprocessing.StandardScaler()
scaler_l = preprocessing.StandardScaler()

scaler_f.fit(f_train)
f_train_scaled = scaler_f.transform(f_train)
scaler_f.fit(f_val)
f_val_scaled = scaler_f.transform(f_val)

scaler_l.fit(l_train.reshape(-1,1))
l_train_scaled = scaler_l.transform(l_train.reshape(-1,1))
scaler_l.fit(l_val.reshape(-1,1))
l_val_scaled = scaler_l.transform(l_val.reshape(-1,1))
```

Figure 14: Scaling the King County labels and features

3.2 The India Dataset

Certain transformations were performed as shown in Fig. 15, Fig. 16, Fig. 17, and Fig. 18.

```
del df['Unnamed: 0']
del df['City']
del df['Unit']
del df['Project Name']
del df['Description']
del df['Apartement']
```

Figure 15: Deleting the unwanted features

The next modification that we did was format the label. We replaced the string with a number in the corresponding unit (lakhs or crores).

```
for x in range(len(df['Total Price'])):
    df['Total Price'][x].strip()
    if df['Total Price'][x].replace(' L', '') != df['Total Price'][x]:
        df['Total Price'][x] = float(df['Total Price'][x].replace(' L', ''))*100000
    else:
        df['Total Price'][x] = float(df['Total Price'][x].replace(' Cr', ''))*10000000
```

Figure 16: Formatting the label by replacing each value with a number in the corresponding unit

```

for x in range(0, len(df['Type_of_plot'])):
    df['Type_of_plot'][x] = float(df['Type_of_plot'][x][0])
    df['Rupee_per_area'][x] = df['Rupee_per_area'][x].replace(',', '')
    df['Rupee_per_area'][x] = df['Rupee_per_area'][x].replace(' / sq ft', '')
    df['Rupee_per_area'][x] = float(df['Rupee_per_area'][x])
    if df['Open_Date'][x].replace('Possession by', '') != df['Open_Date'][x]: #contains "Possession by"
        df['Open_Date'][x] = 0
    elif df['Open_Date'][x].replace('year', '') != df['Open_Date'][x]: #contains 'year', so it is n years old
        df['Open_Date'][x] = float(df['Open_Date'][x][0]) + 0.5
    else:
        df['Open_Date'][x] = 5
print(df['Type_of_plot'].unique())

```

Figure 17: Modifying the Type of plot, Rupee per area, and the Open date features

As mentioned before, we needed to perform one-hot encoding on the 'Type' feature. We did this using the `.replace()` method for the dataframe.

```

df['Type'] = df['Type'].replace(['Under Construction', 'Ready to move'], [0,1])

```

Figure 18: Performing one-hot encoding on the Type feature (we replaced Under Construction with 0 and Ready to move with 1)

```

print(df['Rupee_per_area'].unique())
print(df['Type'].unique())
print(df['Type_of_plot'].unique())
print(df['Open_Date'].unique())

[7486.0 11725.0 8529.0 7452.0 8834.0 6705.0 8823.0 5812.0 6479.0 7646.0
 9040.0 8778.0 7709.0 6907.0 6599.0 7511.0 12292.0 7417.0 8182.0 8940.0
 9844.0 9574.0 3804.0 3600.0 7157.0 3813.0 2677.0 4125.0 8911.0 4133.0
 5300.0 3700.0 5150.0 5500.0 2675.0 3500.0 2960.0 9467.0 4955.0 3100.0
 4163.0 4999.0 4299.0 5705.0 7800.0 4200.0 6965.0 5600.0 7499.0 5200.0
 3900.0 5999.0 9980.0 6849.0 5800.0 4599.0 8958.0 5617.0 15688.0 9277.0
 9064.0 9039.0 5076.0 6500.0 5147.0 5000.0 4400.0 4285.0 4298.0 11392.0
 4727.0 6432.0 7371.0 6359.0]
[0 1]
[2.0 1.0 3.0 4.0]
[0 5.5 0.5 5 8.5 1.5 2.5 6.5 9.5]

```

Figure 19: Printing the end values of some features

4 King County Models and Results

4.1 With the Year Renovated Feature

4.1.1 Linear Regression

Now that we have prepared the dataset, we started to train the models. First, we trained the models with the dataset that includes the year renovated feature. We began with linear regression. Linear regression is one of the simplest techniques to model data, and works for numerical problems (problems in which the label is numerical). The formula for linear regression is:

$\hat{y} = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$, where \hat{y} is the predicted label.

In this formula, the w 's and the b are called parameters, and are chosen by the computer.

4.1.2 Regularization

We have also used regularization for all the models in this paper. Regularization is a technique to prevent overfitting. In regularization, we use a new variable to limit the sizes of the parameters. This variable is λ (lambda). λ is a hyperparameter, a variable whose value we can choose. So, we trained multiple models, each with different values of lambda. Then, we kept the model whose validation error was lowest. The equation for the mean square error function with regularization for linear regression is [Kha18]:

$loss = \frac{1}{x} \sum_{i=1}^x (y_i - \hat{y}_i)^2 + \frac{\lambda}{l} \sum_{i=1}^l w_i^2$, where x is the number of examples, and l is the number of parameters, excluding b .

We stored the training errors, validation errors, and the lambdas that we used in three lists.

```
J_train = []
J_val = []
lambdas = []
```

Figure 20: Creating three lists to store information

4.1.3 Neural Networks

Now, we started coding the models. We used the neural networks in tensorflow to implement these models. Neural networks are made up of layers and nodes. Neural networks have an input layer, and output layer, and most of the time, hidden layers. There can only be one input and output layer. Each layer, l , except the input layer, has an activation function, $f^{[l]}$. In this paper, we used fully connected neural networks, also known as dense neural networks. In fully connected neural networks, each node in one layer is connected by an edge to each node in the next layer. Each edge and node has its own parameter that is chosen by the computer. The parameter for the edge connecting the i th node in layer $l-1$ to the j th node in layer l in a neural network is $w_{ij}^{[l]}$. The parameter for the i th node in the l th layer of a neural network is $b_i^{[l]}$. Each node has data associated with it. In the input layer (the first layer), the data associated with the nodes are the features. The data for the i th node in layer l is $a_i^{[l]}$.

The formula used to calculate the data for nodes not in the input layer is:

$$a_i^{[l]} = f^{[l]}(\sum_{j=0}^{n^{[l-1]}} w_{ji}^{[l]} a_j^{[l-1]} + b_i^{[l]})$$

Similar to linear regression, the equation for the mean square error function with regularization for neural networks is:
 $loss = \frac{1}{x} \sum_{i=1}^x (y_i - \hat{y}_i)^2 + \frac{\lambda}{p} \sum_{c=0}^p (w_{ij}^{[l]})^2$, where p is the number of edges in the neural network. Note that $p = n^{[0]} \times n^{[1]} + n^{[1]} \times n^{[2]} + \dots + n^{[L-1]} \times n^{[L]}$. Note that the input layer is layer 0.

Each neural network has its own architecture, which refers to the arrangement of layers, activation functions for the layers, and the nodes in the neural network. Linear regression is a special form or architecture of a neural network. It only has one layer, and the activation function for the output layer is the identity function. There are multiple different possibilities for activation functions. Some are: the sigmoid function ($\sigma(x) = \frac{1}{1+e^{-x}}$) [Wei], the identity function ($id(x) = x$), the hyperbolic tangent function, and the Rectified Linear Unit [Sha17] (ReLU) activation function ($ReLU(x) = x$, if $x \geq 0$, or 0, if $x < 0$).

4.1.4 Linear Regression Models

Now, we started coding linear regression. The graphs below plot the steps (epochs) against the errors. The steps are represented on the x-axis, and the errors are displayed on the y-axis.

General Note: *All errors displayed and mentioned in this paper are subject to variation, and are not guaranteed to stay the same when training models again. We also used the stochastic gradient descent, and so the errors will vary.*

```
lambda = 0
model = 0 #restart model
model = Sequential() #gives the input layer
model.add(Dense(1, kernel_regularizer = regularizers.l2(lambda)))
model.compile(loss = 'MSE')
model.fit(f_train_scaled, l_train_scaled, epochs = 200, verbose = 0)
J_history = model.history.history['loss']
plt.plot(J_history)
```

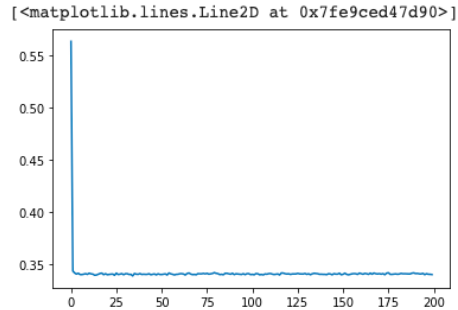


Figure 21: Linear regression code

In the code above, first we defined `lambda` as zero. Then, we restarted the model by essentially deleting the previous model, if any. When we assigned the model to `Sequential()`, a new model with an input layer was created. Then, we added the output layer, a layer with 1 node, no activation function (this uses the identity activation function), and used regularization. The next line told the model to minimize the mean square error on the training set. We fitted the training features and labels to the model. Then, we told the model to keep track of the loss (error), and then plot the errors. We viewed the exact errors by appending the errors to the arrays that we initialized before.

```
J_train.append(J_history[-1])
lambdas.append(lambdaA)
l_hat = model.predict(f_val_scaled)
J_val.append(mse(l_val_scaled, l_hat))
print(J_train)
print(J_val)
print(lambdas)

[0.3396799862384796]
[<tf.Tensor: shape=(), dtype=float32, numpy=0.34486264>]
[0]
```

Figure 22: Displaying errors for linear regression

Now, we increased `lambda` slightly, and ran linear regression on the dataset again. Then, we printed the errors.

```
lambdaA = 0.001
model = 0 #restart model
model = Sequential() #gives the input layer
model.add(Dense(1, kernel_regularizer = regularizers.l2(lambdaA)))
model.compile(loss = 'MSE')
model.fit(f_train_scaled, l_train_scaled, epochs = 250, verbose = 0)
J_history = model.history.history['loss']
plt.plot(J_history)
```

[<matplotlib.lines.Line2D at 0x7fea5ca99b50>]

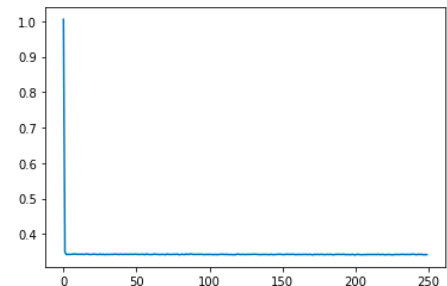


Figure 23: Linear regression code

We ran this process with different `lambdas`. We chose `lambdas` depending on the increase or decrease in validation errors. Note that at times, the lists might have different values, as we ran the models at different times. Sometimes,

```

J_train.append(J_history[-1])
l_hat = model.predict(f_val_scaled)
J_val.append(mse(l_val_scaled, l_hat))
lambdas.append(lambdaA)
print(J_train)
print(J_val)
print(lambdas)

[0.34182190895080566]
[<tf.Tensor: shape=(), dtype=float32, numpy=0.34690928>]
[0.001]

```

Figure 24: Displaying errors for linear regression

we ran the same model multiple times to ensure that it has found the global minimum for the error. Refer to Table 1 for more errors for linear regression.

Lambda	Number of Epochs	Train Error	Validation Error
0.0005	250	0.3426942229270935	0.3466163
0.0006	250	0.3416242003440857	0.34618902
0.0007	250	0.3430746793746948	0.3467944

Table 1: Training and validation errors for linear regression with different lambdas

The lowest validation error occurred when lambda was equal to 0.0006. This resulted in a training error of 0.3416242003440857 and a validation error of 0.34618902. Although the lowest validation error was produced when lambda was 0, we wanted to use regularization, and setting lambda to 0 did not use regularization. So, we ignored that model. Although we knew the errors, we could not discern what they mean. Hence, we calculated some more information for a randomly chosen subset of 100 examples from the validation set. Note that the information could be different as the timing of training these models and determining the information is different.

4.1.5 Evaluating the model

We determined three pieces of information: the unscaled label that the model predicts, the actual label, and the percentage error between the predicted label and the actual label. We calculated the information by making the model predict the scaled label using the scaled features for the validation set. Then, we unscaled the predicted labels. We went through the unscaled predicted labels, and made sure that they were all positive. If they were negative, we multiplied the predicted labels with -1. The information displayed below is for ten examples from the subset. Note that the following information was obtained by using the model that produced the least validation error, without taking into consideration if it had been trained again after producing the least validation error.

The leftmost column contains the unscaled predicted labels, the middle column displays the actual labels, and the rightmost column contains the per-

```

[911297.] 710000.0 [-28.35169]
[291198.2] 450000.0 [35.28929]
[544368.2] 565000.0 [3.6516483]
[455477.88] 509950.0 [10.681856]
[1111219.2] 1646000.0 [32.48972]
[759419.06] 780000.0 [2.6385818]
[574842.2] 539950.0 [-6.4621143]
[546252.9] 510000.0 [-7.108407]
[332000.78] 425000.0 [21.88217]
[1472452.9] 1476000.0 [0.24032012]

```

Figure 25: Predicted vs actual price, and percentage error

centage errors between the predicted and actual labels. The formula used to calculate the percentage error is:

$$\text{percent error (\%)} = \frac{y - \hat{y}}{y} \times 100\%$$

The mean error for the subset was -10.851832%. This meant that for this subset, the model generally predicted the label to be about 11% higher than the actual label.

Now, we trained one more model. We have trained many other models for these datasets, however, for the sake of the length of this paper, we have mainly talked about linear regression.

4.1.6 ReLU (6), ReLU (4), ReLU (5) models

The next model architecture that we used was ReLU (6), ReLU (4), ReLU (5). Note that the numbers in the brackets are the number of nodes in that layer. Refer to Table 2 for the errors that occurred when training models for this architecture.

Lambda	Number of Epochs	Train Error	Validation Error
0	300	0.18896162509918213	0.20669673
0.0005	300	0.20660923421382904	0.21101856
0.001	300	0.21743005514144897	0.22290714

Table 2: Training and validation errors for different lambdas for the architecture: ReLU (6), ReLU (4), ReLU (5)

The lowest validation error, with regularization, for this architecture is 0.21101856, when lambda is 0.0005.

4.1.7 Other models

As we mentioned before, we have trained many more models. Refer to Table 3 and Table 4 for the errors for some of these models.

Lambda	Number of Epochs	Train Error	Validation Error
0	100	0.2631775140762329	0.2822884
0.0005	200	0.3189285397529602	0.32071579
0.001	150	0.2674862742424011	0.28359616
0.0015	200	0.32559671998023987	0.32045856

Table 3: Training and validation errors for different lambdas for the architecture: ReLU (2)

Lambda	Number of Epochs	Train Error	Validation Error
0	500	0.24578668177127838	0.26487643
0.001	400	0.26277828216552734	0.25601012
0.0012	400	0.26819127798080444	0.26044652

Table 4: Training and validation errors for different lambdas for the architecture: ReLU (2), Tanh (2)

4.2 Without the Year Renovated Feature

4.2.1 Linear regression

Prior to training the following models, we deleted the year renovated feature. Now, we followed a similar process to train the models.

```
lambda = 0
model = 0 #restart model
model = Sequential() #gives the input layer
model.add(Dense(1, kernel_regularizer = regularizers.l2(lambda)))
model.compile(loss = 'MSE')
model.fit(f_train_scaled, l_train_scaled, epochs = 200, verbose = 0)
J_history = model.history.history['loss']
plt.plot(J_history)
```

[<matplotlib.lines.Line2D at 0x7fd5850e9150>]

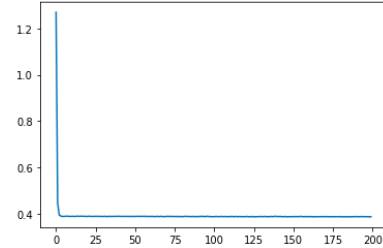


Figure 26: Linear regression code

```

J_train.append(J_history[-1])
l_hat = model.predict(f_val_scaled)
J_val.append(mse(l_val_scaled, l_hat))
lambdas.append(lambdaA)
print(J_train)
print(J_val)
print(lambdas)

[0.3872382938861847]
[<tf.Tensor: shape=(), dtype=float32, numpy=0.37459856>]
[0]

```

Figure 27: Displaying linear regression errors

Refer to Table 5 for more errors for linear regression.

Lambda	Number of Epochs	Train Error	Validation Error
0.001	200	0.3879319131374359	0.37372103
0.0012	200	0.3875928521156311	0.37174022
0.0015	200	0.38780951499938965	0.3721308

Table 5: Training and validation errors for linear regression with different lambdas

4.2.2 Evaluating the model

The least validation error for linear regression without the year renovated feature was 0.37174022, when lambda was 0.0012. Again, we calculated information for a different subset of 100 examples in the validation set by following the same process that we described previously. The information displayed below is for ten examples from the subset. Note that the following information was obtained by using the model that produced the least validation error, without taking into consideration if it had been trained again after producing the least validation error.

```

[684819.8] 730000.0 [6.189067]
[579913.75] 479950.0 [-20.827951]
[668118.3] 650000.0 [-2.7874327]
[848156.56] 420000.0 [-101.94204]
[259232.73] 427200.0 [39.31818]
[674734.4] 555000.0 [-21.573763]
[911637.44] 1165000.0 [21.74786]
[396018.1] 397500.0 [0.3728066]
[355051.25] 375000.0 [5.319667]
[314983.75] 299900.0 [-5.029593]

```

Figure 28: Predicted vs actual price, and percentage error

The leftmost column contains the unscaled predicted labels, the middle column displays the actual labels, and the rightmost column contains the percentage errors between the predicted and actual labels. The mean error for the subset was -4.152803%. This meant that for this subset, the model generally predicted the label to be 4% higher than the actual label.

4.2.3 ReLU (6), ReLU (4), ReLU (5) models

Now, we trained models for the next architecture: ReLU (6), ReLU (4), ReLU (5). Refer to Table 6 for the errors for this architecture.

Lambda	Number of Epochs	Train Error	Validation Error
0	400	0.29903194308280945	0.31010622
0.001	400	0.3164912462234497	0.30268273
0.0013	400	0.31477266550064087	0.29665732
0.0016	400	0.3172665536403656	0.29994994

Table 6: Training and validation error when lambda is 0.0015
The lowest validation error is 0.29665732, when lambda is 0.0013.

4.2.4 Other models

We have also trained other models for this dataset (without the year renovated column). Refer to Table 7 and Table 8 for the errors for some of these models.

Lambda	Number of Epochs	Train Error	Validation Error
0	200	0.32799747586250305	0.31698662
0.001	400	0.33125102519989014	0.31242096
0.0014	400	0.397177517414093	0.3719641

Table 7: Training and Validation errors for ReLU (2) with different lambdas

Lambda	Number of Epochs	Train Error	Validation Error
0	300	0.3295673727989197	0.31414253
0.0005	300	0.3347756564617157	0.31153727
0.001	300	0.3443007469177246	0.31559515

Table 8: Training and Validation errors for ReLU (2), Tanh (2) with different lambdas

5 India Models

5.1 Linear regression

We have finished training the models for the King County dataset. Now we trained models for the India dataset. First, we trained linear regression models.

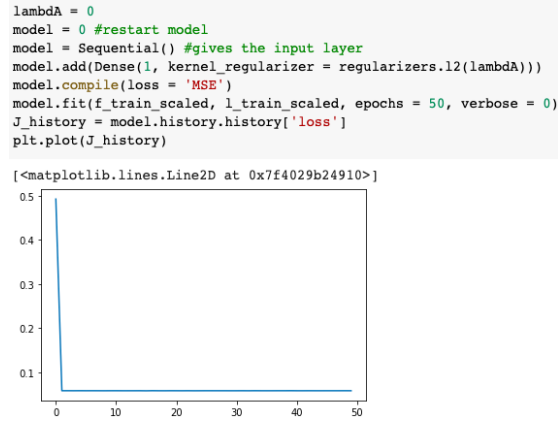


Figure 29: Linear regression code

```
J_train.append(J_history[-1])
lambdas.append(lambda)
l_hat = model.predict(f_val_scaled)
J_val.append(mse(l_val_scaled, l_hat))
print(J_train)
print(J_val)
print(lambdas)
```

[0.05849771574139595]
[<tf.Tensor: shape=(), dtype=float32, numpy=0.057862297>]
[0]

Figure 30: Displaying errors for linear regression

Refer to Table 9 for more errors for linear regression.

Lambda	Number of Epochs	Train Error	Validation Error
0.0005	50	0.0590645968914032	0.05804044
0.001	50	0.05953659117221832	0.057800516
0.0015	50	0.06008978188037872	0.05795654

Table 9: Training and validation errors for linear regression with different lambdas

5.2 Evaluating the model

The lowest validation error for linear regression for this dataset is 0.057800516, when lambda is 0.001. For that model, we again calculated the following information, using the same process that we used before. The information displayed below is for ten examples from the subset. Note that the following information was obtained by using the model that produced the least validation error, without taking into consideration if it had been trained again after producing the least validation error.

```
[12437577.] 12500000.0 [0.499384]
[3793109.] 4399000.0 [13.77338]
[4428232.] 5000000.0 [11.43536]
[15680326.] 17000000.0 [7.7627883]
[5807582.] 5750000.0 [-1.0014261]
[10296244.] 8700000.0 [-18.347631]
[7097041.5] 6548999.999999999 [-8.368323]
[5946779.] 6070000.0 [2.03]
[4639562.5] 5005000.0 [7.301449]
[8862932.] 8265000.000000001 [-7.234507]
```

Figure 31: Predicted vs actual price, and percentage error

The leftmost column contains the model's predictions, the middle column contains the actual prices, and the rightmost column displays the percentage errors between the model's predictions and the actual prices. The mean error for this subset is 3.7801437%. This meant that for this subset, the model generally predicted the label to be 4% lower than the actual label.

5.3 ReLU (6), ReLU (4), ReLU (5) models

Now, we trained models for the architecture: ReLU (6), ReLU (4), ReLU (5). Refer to Table 10 for the errors for these models.

Lambda	Number of Epochs	Train Error	Validation Error
0.0001	100	0.0012585741933435202	0.00016387845
0.0003	100	0.0030744753312319517	0.0008748942
0.001	100	0.007523918990045786	0.0006793593

Table 10: Training and validation errors for ReLU (6), ReLU (4), ReLU (5) with different lambdas.

The lowest validation error for this architecture is 0.00016387845, with a lambda of 0.0001.

5.4 Other models

We have also trained other models for this dataset. Refer to Table 11 and Table 12 for the errors for some of these models.

Lambda	Number of Epochs	Train Error	Validation Error
0	50	0.020178716629743576	0.019952854
0.0005	50	0.027878375723958015	0.023663329
0.001	50	0.0317220576107502	0.02660647

Table 11: Training and Validation errors for ReLU (2) with different lambdas.

Lambda	Number of Epochs	Train Error	Validation Error
0.0001	100	0.018649589270353317	0.016446289
0.001	100	0.025118308141827583	0.014619921
0.0012	100	0.015121044591069221	0.0028623657
0.0014	100	0.016498055309057236	0.0020094276
0.0017	100	0.03245490789413452	0.012285076

Table 12: Training and Validation errors for ReLU (2), Tanh (2) with different lambdas.

6 Observations and Analysis

6.1 King County Model Errors Observations and Analysis

Now that we have trained models for the King County housing dataset, we have been able to draw the following observations and inferences.

1. The lowest validation error for the dataset with the year renovated feature was 0.21101856. This was obtained using the architecture: ReLU (6), ReLU (4), ReLU (5). The lowest validation error for the dataset without the year renovated feature was 0.29162762. This was produced using ReLU (4), Tanh (2), ReLU (3) architecture. As both of these models are relatively complex, more complex models are able to best model the data.

2. We considered models that only used ReLU and models that also used other activation functions, for the dataset with the year renovated feature, and observed that models using only ReLU had varying errors, but had the lowest error. We also considered the models with only ReLU and models that also had other activation functions for the dataset without the year renovated feature, and found that models with only ReLU usually performed the same as the other models. These observations suggest that the King County dataset without the year renovated feature is modeled roughly equally well with any combination of activation functions, and the dataset with the year renovated feature is better modeled by the ReLU activation function. This could be because the ReLU

activation function is a powerful activation function that is able to model the data relatively well.

3. Models that were more complex than linear regression produced the lowest validation errors. In fact, the lowest validation error for linear regression was higher than the lowest validation errors for the other models. This could be because the dataset was not very linear, and the more complex models were able to better fit and adapt to the data.

4. The lowest validation error for the dataset with the year renovated feature was about 38.2% lower than the lowest validation error for the dataset without the year renovated feature. From this, we can infer that the year renovated feature does influence the price. So, the pricing trends in King County, and perhaps even the property buyers, consider the year renovated feature (as considering the year renovated feature allowed the model to better estimate the price of the property).

5. The best models for the King County dataset, with and without the year renovated feature, produced somewhat high percentage errors. The percentage errors were generally lower for the dataset with the year renovated feature than for the dataset without the year renovated feature. The average percentage error for the best model without the year renovated feature (around -16.25%) was much greater than the average percentage error for the best model with the year renovated feature (around -4.12%). This also shows that the year renovated feature is an important factor to consider when finding the price of a property.

6. For the subset (100 examples), the dataset with the year renovated feature had an average percentage error of around -5.77%. For a different subset (100 examples) in the dataset without the year renovated feature, the average percentage error of around -12.34%. However, we cannot generalize using this information since the subset does not represent the characteristics of the whole data. The subset was mainly used to compare the predicted prices to the actual prices to understand the model errors.

7. The features did not seem to be able to describe the pricing of the properties well. This is due to the relatively large error on both the training and validation sets for all of the models developed for this dataset. This suggests that either more features are required, or that the features do not accurately represent the concerns of the buyers, or that house prices are spread apart randomly.

8. The relatively large errors could also mean that the methods we used to prepare the datasets could be wrong. After all, the quality of the data influences the accuracy of the model.

9. Some ways in which we could improve the results of the models are: including the date feature, removing outliers, changing the year renovated value from 0 to the year built value for houses that have not been renovated, and using the square feet living 15 and square feet lot 15 features in the model.

6.2 India Model Errors Observation and Analysis

Now that we have trained the models for the India housing dataset, we have drawn the following observations and inferences:

1. The best model for this dataset was ReLU (6), ReLU (4), ReLU (5), with a lambda of 0.0001. ReLU (6), ReLU (4), ReLU (5) was able to model the data extremely well, while the other models were not able to do this as well. Therefore, the property prices in India follow a slightly complex trend, yet relatively linear trend (as linear regression has also done relatively well for this dataset).

2. As all of the models have worked quite well, the features have described the property prices very well. This means that the features have almost mirrored the values of property buyers in India.

3. It can be inferred that the method in which we prepared the dataset was successful in: changing all the values in the dataset into numbers, and doing so in a way that was consistent with the trends in the housing market in India.

4. Models that used only ReLU had varying errors, but also had the lowest errors. Models that also used other activation functions also had low and varying errors. However, the models with more layers (more complex models) generally had lower errors. This suggests that the more complex models were again able to identify and adapt to patterns within the dataset. There were also many examples in this dataset, and so more complex models were needed to fit the data better.

5. Linear regression produced the highest errors of all the models for this dataset. This suggests that although the data followed a relatively strong linear pattern, more layers and complexity was needed to properly and accurately model the data.

6. The average percentage error for the best model for the India dataset was around 0.51%. This means that the model is extremely accurate, predicting many property prices to the nearest lakh or thousand rupees. Since the average percentage error is positive, the predicted price was usually lower than the actual price.

7. The average percentage error for the best model for a subset of the India dataset was about 0.49%. Most of the percentage errors for the subset were between -1% and 2.5%. This suggests that the model could predict the prices of the apartments quite accurately. For the majority of the houses, the predicted price was lower than the actual price of the property.

8. The model could have been improved by using the city feature, or the project name feature, or even extracting key details from the description feature. We could have replaced the city feature with weightages to represent the strengths of the housing markets in the cities. We could have developed a ranking system for the project name. We could have used a loop to iterate through the descriptions, and find the details that we would want. Then, the code could return key words, and determine a number for these key words. Although doing these steps could have increased the error, this would help to include more important details into the model.

6.3 Comparing the Datasets' Models' Results

There are patterns that we can now observe within the results of the two datasets. These patterns are:

1. The India dataset's lowest mean percentage error was around 0.51%, whereas the lowest mean percentage errors for the King County dataset with and without the year renovated feature were -4.12% and -16.25% respectively. So, the best India model has done much better than the best King County models.
2. Complex models were better able to model the data for both datasets. This could be due to the complexities of the trends in these regions.
3. All of the models for the India dataset produced a lower error than the models developed for the King County dataset. This could mean that the Indian housing market follows a much stronger pattern. We can infer that there is a smaller percentage of outliers in the India dataset. However, it could also mean that the features included in the India dataset describe the Indian housing market better than the features included in the King County dataset. Of course, it is always possible that we were able to format the data in the India dataset better than we were able to do for the King County dataset.
4. The India dataset is more recent than the King County dataset, and this could have influenced the errors of the models. This could mean that more recent datasets tend to follow a more consistent pattern for modeling.

7 Conclusion and Further Analysis

The patterns identified in the previous parts of this paper are significant, because they can tell us about the housing markets of different regions of the world. They can also tell us about the values of people in specific areas, and what we should and should not look for when buying properties in those areas. The patterns allow us to realize the kinds of datasets, and the features that would be most important to develop the best model(s) in these regions.

From the observations we have made, we can conclude that for each region, a model specific for that region should be developed. This model should have enough complexity to model the region appropriately.

However, we have to also discuss the flaws that are included with this research method. Firstly, the geographies are quite different, and although we talked about the values of the people, they were not represented in the datasets themselves. Secondly, the comparisons cannot be generalized, and are subject to imperfections, due to the nature of the housing market. The housing market depends heavily on the values of the people, and on the history of the properties. In this paper, we have not considered the time difference between the two datasets. Thirdly, the two datasets describe different types of properties. One describes houses, while the other describes apartments. But this is also due to the preference of type of properties in various geographical areas. Lastly, we

have not considered or removed outliers. Removing outliers would have helped the models to perform better, as they would not have to adjust and increase the error in order to incorporate the outliers.

We would like to talk about future work in conclusion. One of the next steps is to consider the geographies of the properties. This is because the geography of the properties can significantly affect the prices of the properties. For example, in India, there is a much higher population to size ratio than in King County, so the prices are bound to be quite different. So, if we were to somehow test the models with the other datasets, we would get extremely high errors. An alternative next step is to replace the "year renovated" value with the "year built" value for houses that have not been renovated. This could help to reduce the gap between the year built and year renovated features for houses that have not been renovated, which would help to reduce the errors of the model. We believe that if enough examples are gathered, a solution to better solve this problem is deep learning. Deep learning is an even more powerful method to model and make predictions from data. However, we would need a much larger training set.

References

- [BD17] Roger Brooks and Karina Dahlke. *Understanding the 3 Categories of Machine Learning – AI vs. Machine Learning vs. Data Mining 101 (part 2)*. Oct. 2017. URL: <https://www.guavus.com/ai-vs-machine-learning-vs-data-mining-whats-big-difference-part-2/>.
- [Cha18] Shiva Chandel. *kc.house.data*. Kaggle, 2018.
- [Red] Redfin. *King County Housing Market*. URL: <https://www.redfin.com/county/118/WA/King-County/housing-market#trends>.
- [Aff18] Regional Affordable Housing Task Force. *Final Report and Recommendations for King County, WA*. Research report. Regional Affordable Housing Task Force, 2018.
- [Spa20] Center for Spatial Data Science. *2014-15 Home Sales in King County, WA*. Aug. 2020. URL: <https://geodacenter.github.io/data-and-lab/KingCounty-HouseSales2015/>.
- [Tri22] Devesh Tripathi. *Houses in Top Cities*. Kaggle, 2022.
- [Ser22] Express News Service. *At Rs 6000 per sqft, Hyderabad property rates second highest in India*. Apr. 2022. URL: <https://www.newindianexpress.com/states/telangana/2022/apr/14/at-rs-6000-per-sqft-hyderabad-property-rates-second-highest-in-india-2441818.html>.
- [Sai22] Nupur Saini. *What is BHK? – An Essential Guide With BHK Full Form Meaning*. Dec. 2022. URL: <https://www.magicbricks.com/blog/what-is-bhk-full-form/126037.html>.

- [Yan] Neko Yan. *How to delete a column in pandas*. URL: <https://www.educative.io/answers/how-to-delete-a-column-in-pandas>.
- [Sk] Sklearn. *sklearn.preprocessing.StandardScaler*. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>.
- [Kha18] Renu Khandelwal. *L1 and L2 Regularization*. Nov. 2018. URL: <https://medium.datadriveninvestor.com/l1-l2-regularization-7f1b4fe948f2>.
- [Wei] Eric Weisstein. *Sigmoid Function*. URL: <https://mathworld.wolfram.com/SigmoidFunction.html>.
- [Sha17] Sagar Sharma. *Activation Functions in Neural Networks*. Sept. 2017. URL: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>.