# HOUSE PRICE PREDICTION IN SEATTLE

## FALL 22 BIG DATA CS-GY 6513


### Team Members:

- Gayathri Prerepa - gp2254
- Akhil kumar Devada - ad5695
- Prudhvi Raj - nk3185

TANDON SCHOOL OF ENGINEERING

NYU

# Contents

# 1. Introduction.

In recent years, buying a property has gotten more challenging since buyers are getting more frugal with their money and market approach. They continuously strive to get the most out of the budget while still meeting their **demands** and **expectations**. A property that meets the needs of the buyer and is within their budget is great. So pricing estimates are essential for budget planning. This **vast** amount of data contains information and unobserved patterns. One useful application of big data is house price forecasting. Various tools are utilized to determine the **factors** for feature selection in order to extract the significant qualities that predict the ultimate sale price of a house due to the large number of variables. Understanding the **relationship** between housing attributes and how these features affect pricing is the aim of this statistical analysis. Machine learning will be used to predict the price of homes using these features and correlations.

## Why is this a big data problem?

About 210,613 entries and 21 columns make up the Seattle home sales prices dataset. Given that our goal is to extract meaningful **insights** from the dataset, which includes useful data about id, date, price, bedrooms, bathrooms, sqft living, sqft lot, floors, waterfront, view, condition, grade, sqft above, sqft basement, yr built, yr renovated, zip code, lat (latitude), long (longitude), sqft_living15, and sqft_lot15, visualizations and other quantitative analysis are performed

Calculating enormous inputs on a single machine takes a very long time. Keep in mind that these are records on home prices reported between 2014 and 2015. This indicates that data is **continuously generated** at regular intervals of time. If this concept is implemented in further regions, there will likely be a vast volume of data that will need to be analyzed and processed using the appropriate **big data tool.**. Furthermore, a single machine might not be able to handle various processes. Therefore, there are two approaches

to solving this issue: **vertical scaling** and **horizontal scaling**. The price to performance ratio makes horizontal scaling the superior strategy between the two. Instead of purchasing a single strong machine and conducting computations, we might spend our money on purchasing multiple commodity devices. Additionally, since the performance is now **divided** throughout **numerous** machines, we must also disperse the data and make the systems work together in order to produce correct results. Big data is therefore necessary in this situation. Using computer clusters, it is possible to implement Big Data ideas like MapReduce and run several data chunks concurrently and in real time. This task would also be advanced thanks to **Apache Spark,** a very capable tool that can complete the same task much more quickly.

# The Dataset:

We are using the Seattle home sales dataset (2014-2015). Excellent learning opportunities are provided by this dataset. Predicting the price of a property sale using the dataset's many attributes will be the objective for this dataset. There are about 210613 rows and 21 columns in this dataset (features). The full dataset can be viewed: **click here!**

**Id:** Unique Identification number allotted to the house
**Date :** Date house was sold
**Price:** Price of the house
**Bedrooms:** No. of Bedrooms
**Bathrooms:** Number of bathrooms in the house
**Sqft_living:** square footage of the living space
**Sqft_lot:** square footage of the lot
**Floors**: Floor in which the house is located
**Waterfront**: House which has a view to a waterfront (represented by 0/1)
**View**: Has been viewed
**Condition**: How good the condition is ( Overall )
**Grade**: Grade given to the housing unit, based on King County grading system

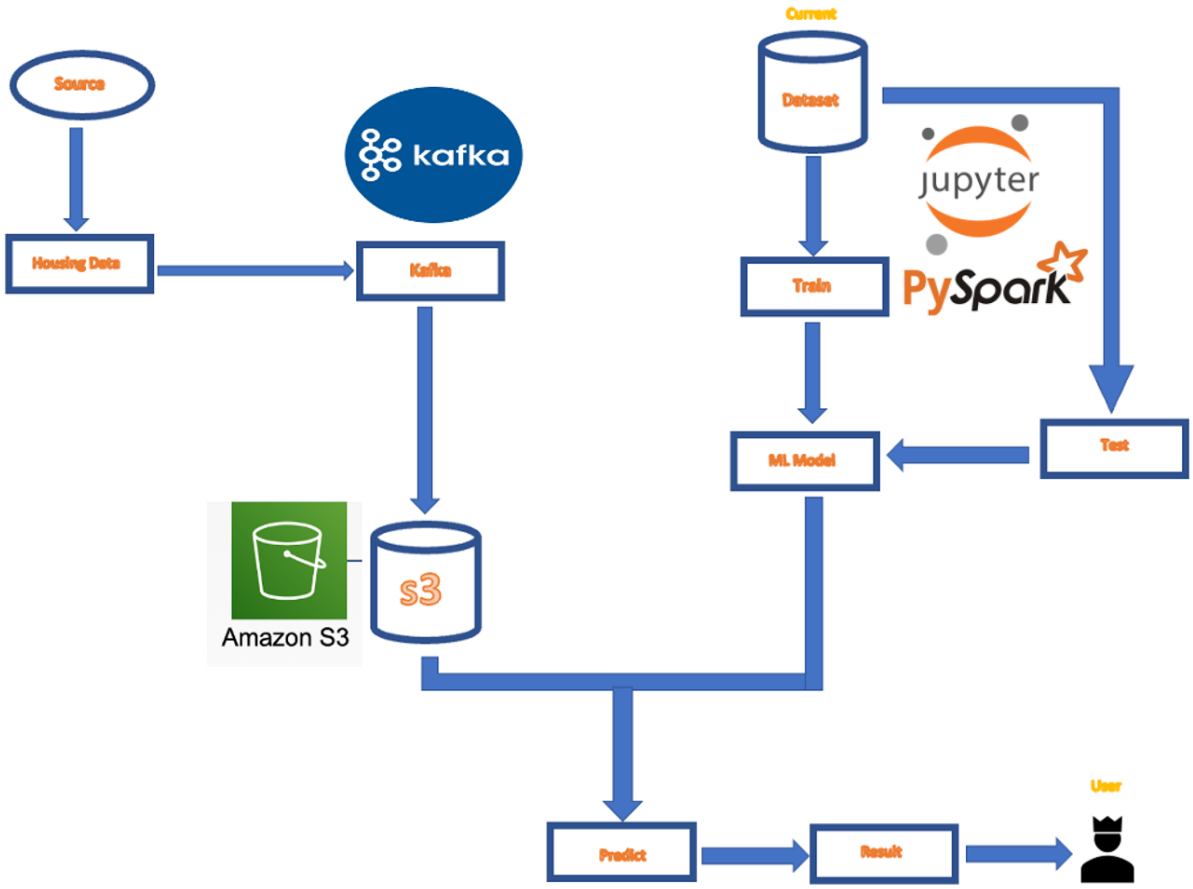
**Sqft_above**: Square footage of house apart from basement
**Sqft_basement**: Square footage of the basement
**Yr_built**: Year when the house was built
**Yr_renovated**: Year when the house was renovated
**Zip-code:** Zip Code
**Lat (latitude):** The latitude of the location.
**Long (longitude):** The longitude of the location.
**Sqft_living15:** Living room area in 2015 (implies — some renovations)
**Sqft_lot15:** lotSize area in 2015 (implies — some renovations)

A picture of dataset that we are working with is shown below:

| id | date | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | view | condition | grade | s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7129300520 | 20141013T000000 | 221900 | 3 | 1.0 | 1180 | 5650 | 1.0 | 0 | 0 | 3 | 7 | |
| 6414100192 | 20141209T000000 | 538000 | 3 | 2.25 | 2570 | 7242 | 2.0 | 0 | 0 | 3 | 7 | |
| 5631500400 | 20150225T000000 | 180000 | 2 | 1.0 | 770 | 10000 | 1.0 | 0 | 0 | 3 | 6 | |
| 2487200875 | 20141209T000000 | 604000 | 4 | 3.0 | 1960 | 5000 | 1.0 | 0 | 0 | 5 | 7 | |
| 1954400510 | 20150218T000000 | 510000 | 3 | 2.0 | 1680 | 8080 | 1.0 | 0 | 0 | 3 | 8 | |
| 7237550310 | 20140512T000000 | 1225000 | 4 | 4.5 | 5420 | 101930 | 1.0 | 0 | 0 | 3 | 11 | |
| 1321400060 | 20140627T000000 | 257500 | 3 | 2.25 | 1715 | 6819 | 2.0 | 0 | 0 | 3 | 7 | |
| 2008000270 | 20150115T000000 | 291850 | 3 | 1.5 | 1060 | 9711 | 1.0 | 0 | 0 | 3 | 7 | |
| 2414600126 | 20150415T000000 | 229500 | 3 | 1.0 | 1780 | 7470 | 1.0 | 0 | 0 | 3 | 7 | |
| 3793500160 | 20150312T000000 | 323000 | 3 | 2.5 | 1890 | 6560 | 2.0 | 0 | 0 | 3 | 7 | |
| 1736800520 | 20150403T000000 | 662500 | 3 | 2.5 | 3560 | 9796 | 1.0 | 0 | 0 | 3 | 8 | |
| 9212900260 | 20140527T000000 | 468000 | 2 | 1.0 | 1160 | 6000 | 1.0 | 0 | 0 | 4 | 7 | |
| 114101516 | 20140528T000000 | 310000 | 3 | 1.0 | 1430 | 19901 | 1.5 | 0 | 0 | 4 | 7 | |
| 6054650070 | 20141007T000000 | 400000 | 3 | 1.75 | 1370 | 9680 | 1.0 | 0 | 0 | 4 | 7 | |
| 1175000570 | 20150312T000000 | 530000 | 5 | 2.0 | 1810 | 4850 | 1.5 | 0 | 0 | 3 | 7 | |
| 9297300055 | 20150124T000000 | 650000 | 4 | 3.0 | 2950 | 5000 | 2.0 | 0 | 3 | 3 | 9 | |
| 1875500060 | 20140731T000000 | 395000 | 3 | 2.0 | 1890 | 14040 | 2.0 | 0 | 0 | 3 | 7 | |
| 6865200140 | 20140529T000000 | 485000 | 4 | 1.0 | 1600 | 4300 | 1.5 | 0 | 0 | 4 | 7 | |
| 16000397 | 20141205T000000 | 189000 | 2 | 1.0 | 1200 | 9850 | 1.0 | 0 | 0 | 4 | 7 | |
| 7983200060 | 20150424T000000 | 230000 | 3 | 1.0 | 1250 | 9774 | 1.0 | 0 | 0 | 4 | 7 | |

| sqft_above | sqft_basement | yr_built | yr_renovated | zipcode | lat | long | sqft_living15 | sqft_lot15 |
|---|---|---|---|---|---|---|---|---|
| 1180 | 0 | 1955 | 0 | 98178 | 47.5112 | -122.257 | 1340 | 5650 |
| 2170 | 400 | 1951 | 1991 | 98125 | 47.721 | -122.319 | 1690 | 7639 |
| 770 | 0 | 1933 | 0 | 98028 | 47.7379 | -122.233 | 2720 | 8062 |
| 1050 | 910 | 1965 | 0 | 98136 | 47.5208 | -122.393 | 1360 | 5000 |
| 1680 | 0 | 1987 | 0 | 98074 | 47.6168 | -122.045 | 1800 | 7503 |
| 3890 | 1530 | 2001 | 0 | 98053 | 47.6561 | -122.005 | 4760 | 101930 |
| 1715 | 0 | 1995 | 0 | 98003 | 47.3097 | -122.327 | 2238 | 6819 |
| 1060 | 0 | 1963 | 0 | 98198 | 47.4095 | -122.315 | 1650 | 9711 |
| 1050 | 730 | 1960 | 0 | 98146 | 47.5123 | -122.337 | 1780 | 8113 |
| 1890 | 0 | 2003 | 0 | 98038 | 47.3684 | -122.031 | 2390 | 7570 |
| 1860 | 1700 | 1965 | 0 | 98007 | 47.6007 | -122.145 | 2210 | 8925 |
| 860 | 300 | 1942 | 0 | 98115 | 47.69 | -122.292 | 1330 | 6000 |
| 1430 | 0 | 1927 | 0 | 98028 | 47.7558 | -122.229 | 1780 | 12697 |
| 1370 | 0 | 1977 | 0 | 98074 | 47.6127 | -122.045 | 1370 | 10208 |
| 1810 | 0 | 1900 | 0 | 98107 | 47.67 | -122.394 | 1360 | 4850 |
| 1980 | 970 | 1979 | 0 | 98126 | 47.5714 | -122.375 | 2140 | 4000 |
| 1890 | 0 | 1994 | 0 | 98019 | 47.7277 | -121.962 | 1890 | 14018 |
| 1600 | 0 | 1916 | 0 | 98103 | 47.6648 | -122.343 | 1610 | 4300 |
| 1200 | 0 | 1921 | 0 | 98002 | 47.3089 | -122.21 | 1060 | 5095 |
| 1250 | 0 | 1969 | 0 | 98003 | 47.3343 | -122.306 | 1280 | 8850 |

# Architecture Diagram:




The following is a description of the components utilized in the above architecture and how they are used:

## Kafka

Real-time streaming data pipelines and applications that can adapt to the data streams are typically built using Apache Kafka. To enable the storage and analysis of both historical and **real-time data**, it mixes messaging, storage, and **stream processing**. The architecture diagram depicts how Apache Kafka receives our input data source. We will submit the data to the Apache Kafka Broker as end users (Queue). Kafka will serve as middleware and can handle a

lot of real-time data. Whenever we have a website that **frequently updates** data in batches. We recommend utilizing Kafka as we expand our data sources outside of Seattle home sales, such as kaggle and other data-generating websites. Kafka is not used for the current application because we are not having access to continuous data generation.

# PySpark

Spark is at the center of this architecture's processing. Spark's in-memory **processing** allows for quick computation of big datasets. We can expect thousands or millions of records per day in the **scalable** environment, especially in the case of a house sales data, because it is a continuous market capable of generating vast **volumes** of data on a daily basis and Spark can handle all processes from pre-processing through publishing this data. For reporting and monitoring, data that has been filtered Spark processing is used. We are using **PySpark API** with **JupyterHub** as the python **IDE**.

# Spark.ML

With the help of a new package called Spark.ml, which was included in Spark 1.2, users may build and fine-tune useful machine learning **pipelines**. Machine learning involves a lot of computationally intensive operations. The most straightforward, quickest, and effective method is to distribute these operations using **PySpark**. Industrial applications require an engine that can process data in-memory and in batch mode, as well as one that is powerful enough to process data in real-time. With a quick and easy interface, PySpark offers real-time streaming, interactive processing, graph processing, in-memory processing, and batch processing.

# Spark.SQL

Structured Query Language: language to access databases and manipulate data. **SQL** is an uncomplicated language and therefore faster to learn than Java or PHP. Probably the most popular variant. When linked with hadoop, It enables unmodified queries to run up to **100x faster** on existing deployments and data.

## Target Data Store(s3 bucket):

To store data into the databases, Target Datastores will subscribe for the Kafka broker. Any **relational database**, including PostgreSQL, NoSQL, or a distributed storage system, can be used as the data store. These may be owned by various organizations from various cities who are interested in subscription and **data collection**.

# 2. Analysis.

## Exploratory Data Analysis:

Exploratory Data Analysis (EDA) is the crucial process of using summary statistics and graphical representations to perform preliminary investigations on data in order to uncover patterns, detect anomalies and verify assumptions. **EDA** is a data exploration technique to understand the various aspects of the data. EDA is often used to see what data may disclose outside of formal modeling as it has many **hidden patterns** and to learn more about the variables in a data collection and how they interact. It could also help us figure out if the statistical procedures we are considering for data analysis are appropriate. Before modeling the data, it gives insight into all of the data and the numerous interactions between the data elements.

### Computing Age of the property:

We know the year the house was built from the yr_built column in the dataframe, so we can compute the age, today, by subtracting each entry in this column with 2022.

```
[ ]  df["age"] = df["yr_built"].subtract(2022).multiply(-1)
```

```
[ ]  spark.createDataFrame(df).show()
```

| price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | view | condition | grade | sqft_basement | yr_built | zipcode | lat | long | age |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 221900 | 3 | 1.0 | 1180 | 5650 | 1.0 | 0 | 0 | 3 | 7 | 0 | 1955 | 98178 | 47.5112 | -122.257 | 67 |
| 538000 | 3 | 2.25 | 2570 | 7242 | 2.0 | 0 | 0 | 3 | 7 | 400 | 1951 | 98125 | 47.721 | -122.319 | 71 |
| 180000 | 2 | 1.0 | 770 | 10000 | 1.0 | 0 | 0 | 3 | 6 | 0 | 1933 | 98028 | 47.7379 | -122.233 | 89 |
| 604000 | 4 | 3.0 | 1960 | 5000 | 1.0 | 0 | 0 | 5 | 7 | 910 | 1965 | 98136 | 47.5208 | -122.393 | 57 |
| 510000 | 3 | 2.0 | 1680 | 8080 | 1.0 | 0 | 0 | 3 | 8 | 0 | 1987 | 98074 | 47.6168 | -122.045 | 35 |
| 1225000 | 4 | 4.5 | 5420 | 101930 | 1.0 | 0 | 0 | 3 | 11 | 1530 | 2001 | 98053 | 47.6561 | -122.005 | 21 |
| 257500 | 3 | 2.25 | 1715 | 6819 | 2.0 | 0 | 0 | 3 | 7 | 0 | 1995 | 98003 | 47.3097 | -122.327 | 27 |
| 291850 | 3 | 1.5 | 1060 | 9711 | 1.0 | 0 | 0 | 3 | 7 | 0 | 1963 | 98198 | 47.4095 | -122.315 | 59 |
| 229500 | 3 | 1.0 | 1780 | 7470 | 1.0 | 0 | 0 | 3 | 7 | 730 | 1960 | 98146 | 47.5123 | -122.337 | 62 |
| 323000 | 3 | 2.5 | 1890 | 6560 | 2.0 | 0 | 0 | 3 | 7 | 0 | 2003 | 98038 | 47.3684 | -122.031 | 19 |
| 662500 | 3 | 2.5 | 3560 | 9796 | 1.0 | 0 | 0 | 3 | 8 | 1700 | 1965 | 98007 | 47.6007 | -122.145 | 57 |
| 468000 | 2 | 1.0 | 1160 | 6000 | 1.0 | 0 | 0 | 4 | 7 | 300 | 1942 | 98115 | 47.69 | -122.292 | 80 |
| 310000 | 3 | 1.0 | 1430 | 19901 | 1.5 | 0 | 0 | 4 | 7 | 0 | 1927 | 98028 | 47.7558 | -122.229 | 95 |
| 400000 | 3 | 1.75 | 1370 | 9680 | 1.0 | 0 | 0 | 4 | 7 | 0 | 1977 | 98074 | 47.6127 | -122.045 | 45 |
| 530000 | 5 | 2.0 | 1810 | 4850 | 1.5 | 0 | 0 | 3 | 7 | 0 | 1900 | 98107 | 47.67 | -122.394 | 122 |
| 650000 | 4 | 3.0 | 2950 | 5000 | 2.0 | 0 | 3 | 3 | 9 | 970 | 1979 | 98126 | 47.5714 | -122.375 | 43 |
| 395000 | 3 | 2.0 | 1890 | 14040 | 2.0 | 0 | 0 | 3 | 7 | 0 | 1994 | 98019 | 47.7277 | -121.962 | 28 |
| 485000 | 4 | 1.0 | 1600 | 4300 | 1.5 | 0 | 0 | 4 | 7 | 0 | 1916 | 98103 | 47.6648 | -122.343 | 106 |
| 189000 | 2 | 1.0 | 1200 | 9850 | 1.0 | 0 | 0 | 4 | 7 | 0 | 1921 | 98002 | 47.3089 | -122.21 | 101 |
| 230000 | 3 | 1.0 | 1250 | 9774 | 1.0 | 0 | 0 | 4 | 7 | 0 | 1969 | 98003 | 47.3343 | -122.306 | 53 |

## Dataset Schema:

Spark DataFrames schemas are defined as the structure of the DataFrame and the collection of typed columns.

```
[ ] data=spark.createDataFrame(df)
```

```
data.printSchema()
```

```
+--------------------+---------+
|         Column Name|Data type|
+--------------------+---------+
|               price|   bigint|
|            bedrooms|   bigint|
|           bathrooms|   double|
|         sqft_living|   bigint|
|            sqft_lot|   bigint|
|              floors|   double|
|                view|   bigint|
|           condition|   bigint|
|               grade|   bigint|
|       sqft_basement|   bigint|
|            yr_built|   bigint|
|             zipcode|   bigint|
|                 lat|   double|
|                long|   double|
|                 age|   bigint|
|ocean_proximity_i...|   double|
+--------------------+---------+
```

```
root
 |-- price: long (nullable = true)
 |-- bedrooms: long (nullable = true)
 |-- bathrooms: double (nullable = true)
 |-- sqft_living: long (nullable = true)
 |-- sqft_lot: long (nullable = true)
 |-- floors: double (nullable = true)
 |-- waterfront: long (nullable = true)
 |-- view: long (nullable = true)
 |-- condition: long (nullable = true)
 |-- grade: long (nullable = true)
 |-- sqft_basement: long (nullable = true)
 |-- yr_built: long (nullable = true)
 |-- zipcode: long (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
 |-- age: long (nullable = true)
```

```
data.select(['price','bedrooms','bathrooms','sqft_living','age']).describe().show()
```

```
+-------+-----------------+------------------+------------------+------------------+------------------+
|summary|            price|          bedrooms|         bathrooms|       sqft_living|               age|
+-------+-----------------+------------------+------------------+------------------+------------------+
|  count|            21613|             21613|             21613|             21613|             21613|
|   mean|540088.1419053348|  3.37084162309721|2.1147573219821405|2079.8997362698374| 50.99486420210059|
| stddev|367127.1959683635|0.9300618311474523| 0.770163157217741| 918.4408970468107|29.373410802386562|
|    min|            75000|                 0|               0.0|               290|                 7|
|    max|          7700000|                33|               8.0|             13540|               122|
+-------+-----------------+------------------+------------------+------------------+------------------+
```

- The most **spacious** house in this data, covering an area of 13540 sqft and the smallest house in this data, covering an area of 290 sq ft
- The **highest** number of bedrooms is 33 and the **lowest** number of bedrooms (studio apartments) is 0
- The **Oldest** house in the dataset is 122 years old and 7 years is the most newest house.

### Checking for null/ NaN values:

```
#Check any missing value
for column in data.columns:
    print(column, data.filter(col(column).cast("float").isin([None,np.nan])).count())
```

```
price 0
bedrooms 0
bathrooms 0
sqft_living 0
sqft_lot 0
floors 0
waterfront 0
view 0
condition 0
grade 0
sqft_basement 0
yr_built 0
zipcode 0
lat 0
long 0
age 0
```

There are no null values in each column of data. So, there is no need to assign or drop any values. But, for future implementation, it is possible to have missing values, in case of any rows containing NaN values, they are dropped using **dropna()**.

## DataFrame analysis:

Data filtering is the process of examining a dataset using **sparkSQL** to exclude, rearrange, or apportion data according to certain criteria/conditions.

```
print("No of houses that cost greater than $350000 are: %i" % data.filter(data["price"] > 350000).count())
data.where(data.price > "350000").show()
```

```
No of houses that cost greater than $350000 are: 14816
+-------+--------+---------+-----------+--------+------+----------+----+---------+-----+-------------+--------+-------+-------+--------+---+
|  price|bedrooms|bathrooms|sqft_living|sqft_lot|floors|waterfront|view|condition|grade|sqft_basement|yr_built|zipcode|    lat|    long|age|
+-------+--------+---------+-----------+--------+------+----------+----+---------+-----+-------------+--------+-------+-------+--------+---+
| 538000|       3|     2.25|       2570|    7242|   2.0|         0|   0|        3|    7|          400|    1951|  98125| 47.721|-122.319| 71|
| 604000|       4|      3.0|       1960|    5000|   1.0|         0|   0|        5|    7|          910|    1965|  98136|47.5208|-122.393| 57|
| 510000|       3|      2.0|       1680|    8080|   1.0|         0|   0|        3|    8|            0|    1987|  98074|47.6168|-122.045| 35|
|1225000|       4|      4.5|       5420|  101930|   1.0|         0|   0|        3|   11|         1530|    2001|  98053|47.6561|-122.005| 21|
| 662500|       3|      2.5|       3560|    9796|   1.0|         0|   0|        3|    8|         1700|    1965|  98007|47.6007|-122.145| 57|
| 468000|       2|      1.0|       1160|    6000|   1.0|         0|   0|        4|    7|          300|    1942|  98115|  47.69|-122.292| 80|
| 400000|       3|     1.75|       1370|    9680|   1.0|         0|   0|        4|    7|            0|    1977|  98074|47.6127|-122.045| 45|
| 530000|       5|      2.0|       1810|    4850|   1.5|         0|   0|        3|    7|            0|    1900|  98107|  47.67|-122.394|122|
| 650000|       4|      3.0|       2950|    5000|   2.0|         0|   3|        3|    9|          970|    1979|  98126|47.5714|-122.375| 43|
| 395000|       3|      2.0|       1890|   14040|   2.0|         0|   0|        3|    7|            0|    1994|  98019|47.7277|-121.962| 28|
| 485000|       4|      1.0|       1600|    4300|   1.5|         0|   0|        4|    7|            0|    1916|  98103|47.6648|-122.343|106|
| 385000|       4|     1.75|       1620|    4980|   1.0|         0|   0|        4|    7|          760|    1947|  98133|47.7025|-122.341| 75|
|2000000|       3|     2.75|       3050|   44867|   1.0|         0|   4|        3|    9|          720|    1968|  98040|47.5316|-122.233| 54|
| 937000|       3|     1.75|       2450|    2691|   2.0|         0|   0|        3|    8|          700|    1915|  98119|47.6386| -122.36|107|
| 667000|       3|      1.0|       1400|    1581|   1.5|         0|   0|        5|    8|            0|    1909|  98112|47.6221|-122.314|113|
| 438000|       3|     1.75|       1520|    6380|   1.0|         0|   0|        3|    7|          730|    1948|  98115| 47.695|-122.304| 74|
| 719000|       4|      2.5|       2570|    7173|   2.0|         0|   0|        3|    8|            0|    2005|  98052|47.7073| -122.11| 17|
| 580500|       3|      2.5|       2320|    3980|   2.0|         0|   0|        3|    8|            0|    2003|  98027|47.5391| -122.07| 19|
| 687500|       4|     1.75|       2330|    5000|   1.5|         0|   0|        4|    7|          820|    1929|  98117|47.6823|-122.368| 93|
| 535000|       3|      1.0|       1090|    3000|   1.5|         0|   0|        4|    8|            0|    1929|  98117|47.6889|-122.375| 93|
+-------+--------+---------+-----------+--------+------+----------+----+---------+-----+-------------+--------+-------+-------+--------+---+
only showing top 20 rows
```

The houses are shown according to their price below, Houses greater than $350,000 are shown in this data. In this Data there are 14,816 Houses which are greater than $350,000.

```
[126] data.createOrReplaceTempView("hou")
      summ = spark.sql(""" SELECT bedrooms, count(*)
                             as count_bedrooms
                           FROM hou GROUP BY bedrooms """)
      summ.show()
```

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

The **count** of bedrooms is displayed for understanding how many of each type we have. We can see a significant decrease in the number of after 6 bedrooms.

```
[ ] s1 = spark.sql(""" SELECT price, bedrooms, bathrooms, lat, long, age FROM hou WHERE age>20 and price<1000000""")
    print("No of houses meeting requirements", s1.count())
    s1.show()
```

```
No of houses meeting requirements 16283
+------+--------+---------+-------+--------+---+
| price|bedrooms|bathrooms|    lat|    long|age|
+------+--------+---------+-------+--------+---+
|221900|       3|      1.0|47.5112|-122.257| 67|
|538000|       3|     2.25| 47.721|-122.319| 71|
|180000|       2|      1.0|47.7379|-122.233| 89|
|604000|       4|      3.0|47.5208|-122.393| 57|
|510000|       3|      2.0|47.6168|-122.045| 35|
|257500|       3|     2.25|47.3097|-122.327| 27|
|291850|       3|      1.5|47.4095|-122.315| 59|
|229500|       3|      1.0|47.5123|-122.337| 62|
|662500|       3|      2.5|47.6007|-122.145| 57|
|468000|       2|      1.0|  47.69|-122.292| 80|
|310000|       3|      1.0|47.7558|-122.229| 95|
|400000|       3|     1.75|47.6127|-122.045| 45|
|530000|       5|      2.0|  47.67|-122.394|122|
|650000|       4|      3.0|47.5714|-122.375| 43|
|395000|       3|      2.0|47.7277|-121.962| 28|
|485000|       4|      1.0|47.6648|-122.343|106|
|189000|       2|      1.0|47.3089| -122.21|101|
|230000|       3|      1.0|47.3343|-122.306| 53|
|385000|       4|     1.75|47.7025|-122.341| 75|
|285000|       5|      2.5|47.3266|-122.169| 27|
+------+--------+---------+-------+--------+---+
```

This query was used to **filter the houses by their age where age is greater than 20 years and the price of the house is less than $1,000,000.** There are approximately 16,000 such houses satisfying the requirements.

```
[ ] s2 = spark.sql(""" SELECT price, bedrooms, bathrooms, lat, long, age FROM hou WHERE bedrooms between 2 and 3 and price<90000""")
    print("No of houses meeting requirements", s2.count())
    s2.show()
```

```
No of houses meeting requirements 10
+-----+--------+---------+-------+--------+---+
|price|bedrooms|bathrooms|    lat|    long|age|
+-----+--------+---------+-------+--------+---+
|82500|       2|      1.0|47.4799|-122.296| 71|
|84000|       2|      1.0|47.4752|-122.271| 73|
|89000|       3|      1.0|47.3026|-122.363| 53|
|82000|       3|      1.0|47.4987|-122.341| 68|
|85000|       2|      1.0|47.3813|-122.243| 83|
|86500|       3|      1.0|47.3277|-122.341| 62|
|78000|       2|      1.0|47.4739| -122.28| 80|
|81000|       2|      1.0|47.4808|-122.315| 79|
|85000|       2|      1.0|47.3897|-122.236| 75|
|83000|       2|      1.0|47.4727| -122.27|104|
+-----+--------+---------+-------+--------+---+
```

This query was used to filter the data to show **2 and 3 bed houses which cost no more than $90,000**. There are 10 houses satisfying the requirement, all of which are 2 beds and 1 bath. We can also observe that these houses are aged between 53 and 104. There are no 3 bed houses matching our requirement.

```
[ ] s3 = spark.sql(""" SELECT price, bedrooms, bathrooms, lat, long, age
    FROM hou WHERE age between 20 and 25 and price between 800000 and 900000
    and bedrooms = 3 and bathrooms = 2.5""")
    print("No of houses meeting requirements", s3.count())
    s3.show()
```

```
No of houses meeting requirements 16
+------+--------+---------+-------+--------+---+
| price|bedrooms|bathrooms|    lat|    long|age|
+------+--------+---------+-------+--------+---+
|900000|       3|      2.5|47.6012|-122.023| 22|
|822500|       3|      2.5|47.6868|-122.108| 23|
|843500|       3|      2.5|47.7126|-122.104| 25|
|900000|       3|      2.5|47.5709|-122.408| 25|
|859000|       3|      2.5|47.3809| -122.13| 23|
|817250|       3|      2.5|47.6769|-122.177| 23|
|807000|       3|      2.5| 47.659|-122.398| 22|
|865000|       3|      2.5|47.6549|-122.089| 25|
|820000|       3|      2.5| 47.664|-122.041| 25|
|825000|       3|      2.5|47.5489|-122.007| 20|
|815000|       3|      2.5|47.5611|-122.032| 21|
|805000|       3|      2.5| 47.654|-122.088| 25|
|898000|       3|      2.5|47.5539|-122.103| 25|
|819000|       3|      2.5|47.4376|-122.456| 22|
|839000|       3|      2.5|47.7614|-122.015| 22|
|812000|       3|      2.5|47.6156|-122.295| 24|
+------+--------+---------+-------+--------+---+
```

This query was used to filter the data to show the **3 bed, 2.5 bath houses that are aged between 20 and 25, and that cost more than $800,000 but no more than $900,000.** There are 16 such houses available

```
s4 = spark.sql(""" SELECT price, bedrooms, bathrooms, lat, long, age
FROM hou WHERE bedrooms = 0""")
print("No of Studio Apartments (0) bedrooms):", s4.count())
s4.show()
```

```
No of Studio Apartments (0) bedrooms): 13
+-------+--------+---------+-------+--------+---+
|  price|bedrooms|bathrooms|    lat|    long|age|
+-------+--------+---------+-------+--------+---+
|1095000|       0|      0.0|47.6362|-122.322| 32|
| 380000|       0|      0.0|47.7145|-122.356| 16|
| 288000|       0|      1.5|47.7222| -122.29| 23|
| 228000|       0|      1.0| 47.526|-122.261| 69|
|1295648|       0|      0.0|47.6642|-122.069| 32|
| 339950|       0|      2.5|47.3473|-122.151| 37|
| 240000|       0|      2.5|47.3493|-122.053| 19|
| 355000|       0|      0.0|47.4095|-122.168| 32|
| 235000|       0|      0.0|47.5265|-121.828| 26|
| 320000|       0|      2.5|47.5261|-121.826| 23|
| 139950|       0|      0.0|47.2781| -122.25|109|
| 265000|       0|     0.75|47.4177|-122.491| 19|
| 142000|       0|      0.0|47.5308|-121.888| 59|
+-------+--------+---------+-------+--------+---+
```

This query was used to filter the houses to show the **studio apartments** in the Seattle dataset. There are 13 houses with 0 bedrooms, the oldest being 109 years and least priced at $139,000.

# Data Visualization :

In our project, data **monitoring** and visualization are crucial. The end-user can be provided with pertinent information using tools like Plotly and seaborn . When deciding how to strategically allocate resources or aid, this depiction is helpful. They can keep an eye on the data and view it, which will provide them with useful information. Data has so many hidden patterns and information.

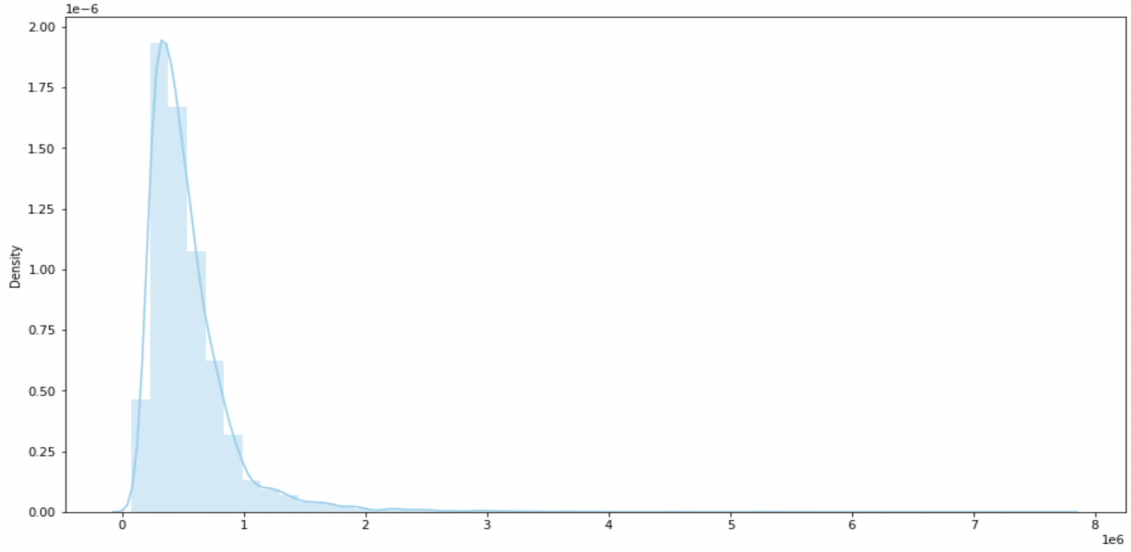
Data visualization can help by delivering data in the most efficient way possible. As one of the essential steps in the business intelligence process, data visualization takes the **raw data**, models it, and delivers the data so that conclusions can be reached. In advanced analytics, data scientists are creating machine learning algorithms to better compile essential data into visualizations that are easier to understand and interpret.

### Skewness analysis:

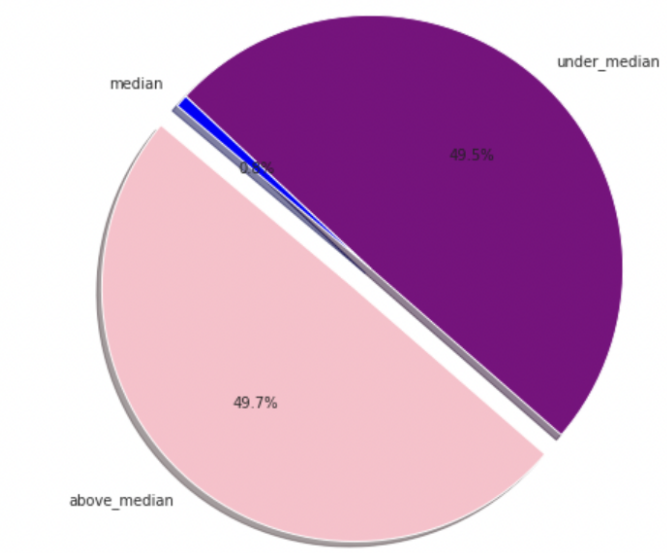
```
[ ]  #Checking if the prices are normally distributed
     data.select(F.skewness('price'), F.kurtosis('price')).show()
```

```
+------------------+------------------+
|  skewness(price)|  kurtosis(price)|
+------------------+------------------+
|4.023789835939301|34.57726217115535|
+------------------+------------------+
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f19e884a460>
```

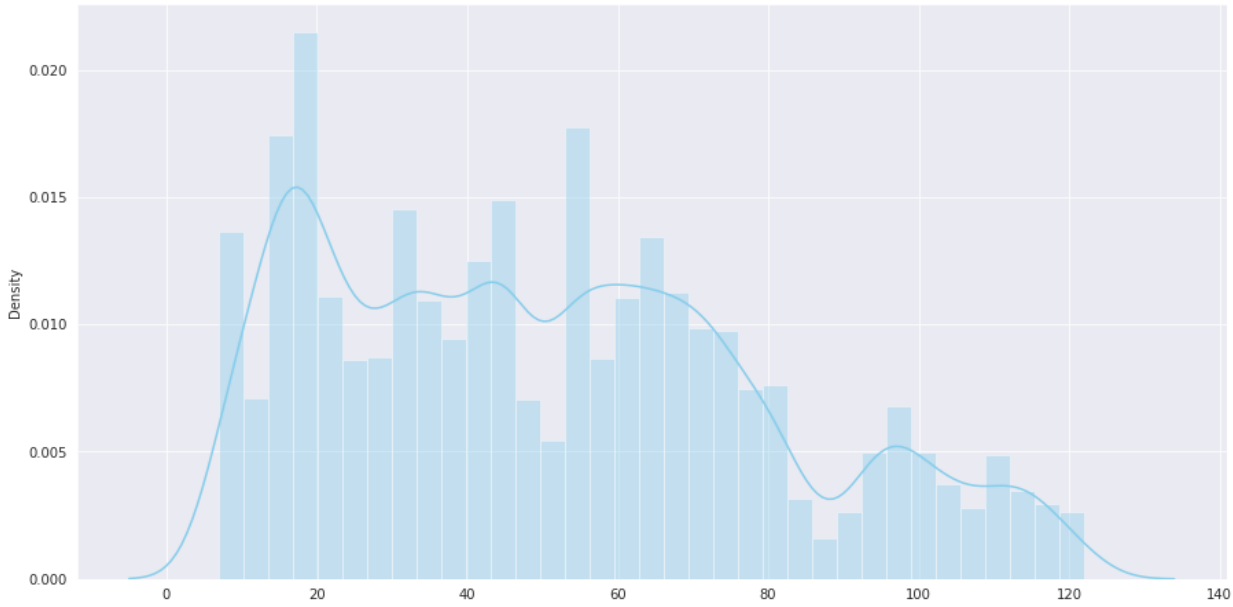



Checking for **skewness** of the Price values using distplot to visualize the distribution graph. This shows that there are **peaks** in the distribution.



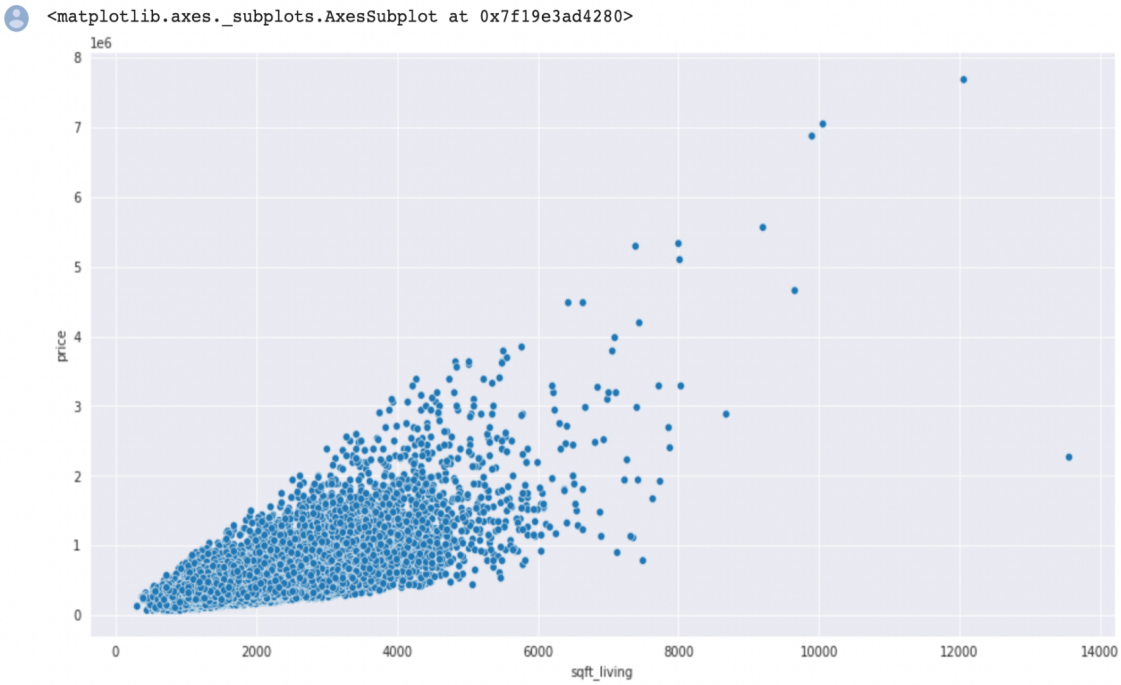

From the above pie chart we can see that:

- Houses with price above_median - 10749 (49.7%)
- Houses with price under_median - 10692 (49.5%)
- Houses with price equal to median - 172 (0.80%)

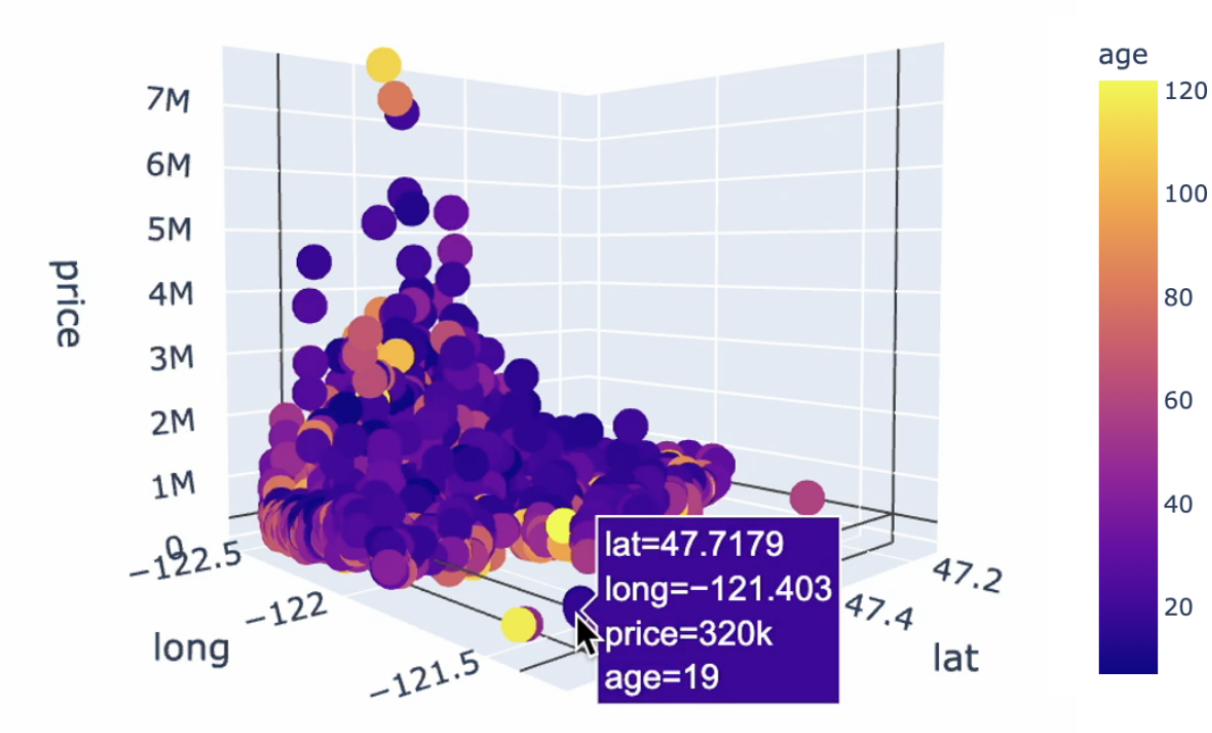



Checking for **skewness** of the age values using distplot to visualize the distribution graph. This shows that there are no sudden changes/**no peaks** in the distribution, so we have fairly even distributed data.

## Scatter Plots analysis:

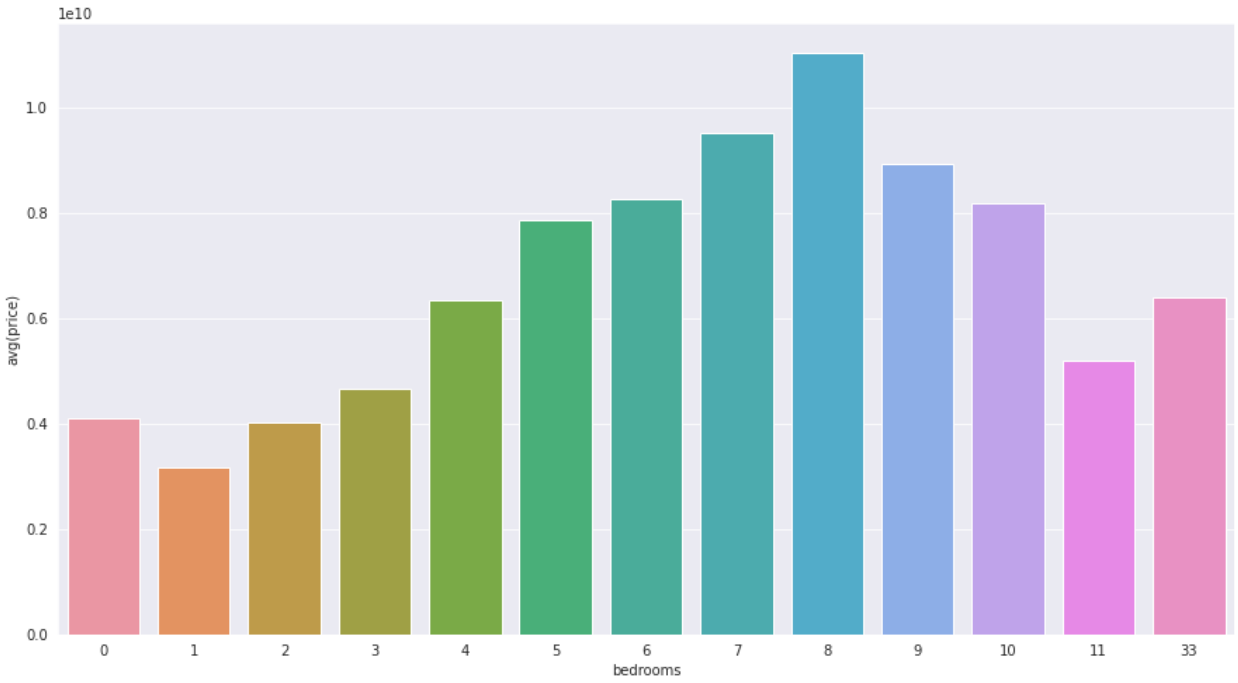



Scatter plot between the Living area (Sgft_living) and the price of the house



The 3D scatter plot is plotted between latitude and longitude with price as the Z dimension, showing variation by color. Dots that are in the purple -ish hues have ages between 0-60 years, as the **hues** change to shades of orange and yellow, the age ranges **beyond** 60 to 120. The 3D plot is dominated by purple shades, ie, the houses that are aged mostly **below** the 50.

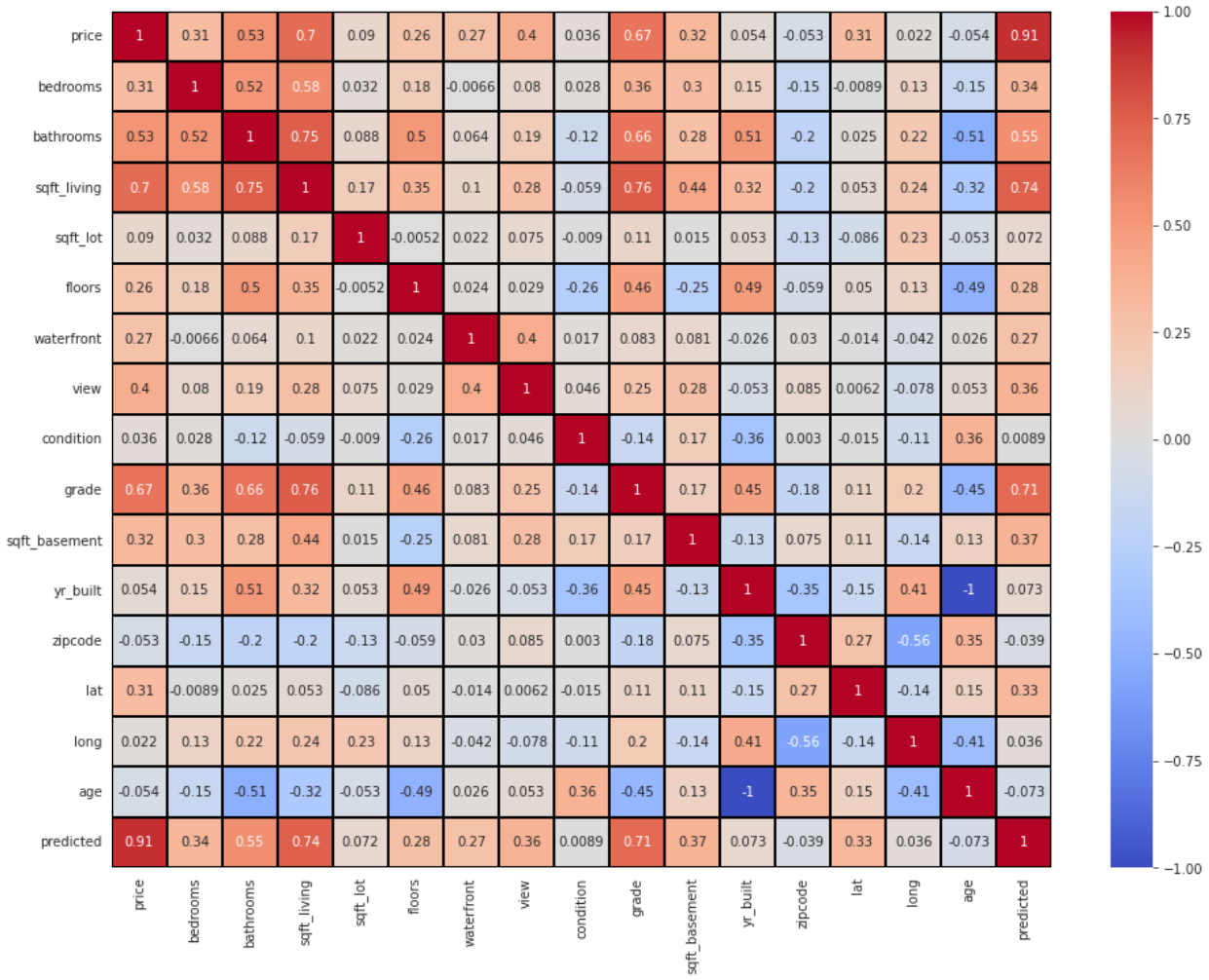
### Box Plots analysis:




**Average** price of the houses arranged by the number of bedrooms they have. The average price of 8 bedrooms is the **highest**, approximately $1,000,000. The least is 1 bedroom houses with average price at around $300,000. As we go further we will address which factors/parameters affect the pricing.

### Correlation map analysis:

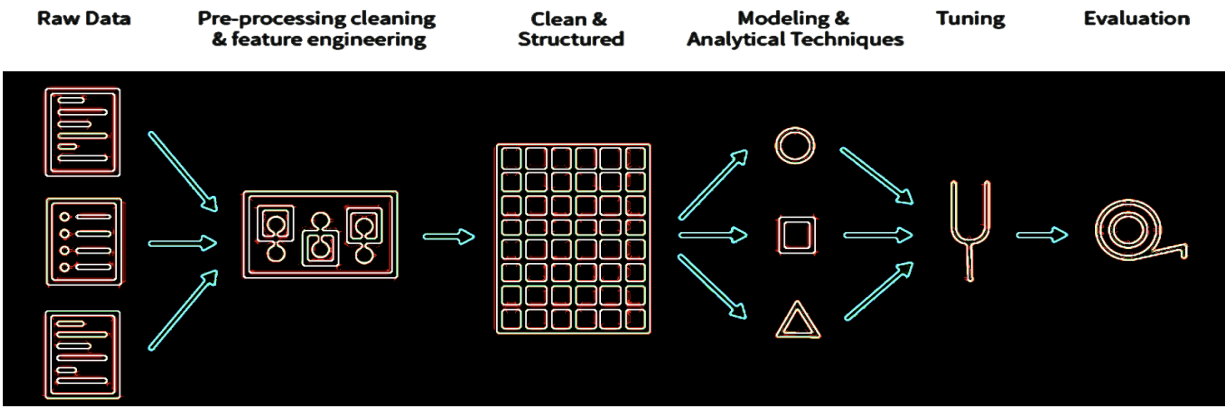
A correlation matrix is a table containing correlation coefficients between variables. Each cell in the table represents the correlation between two variables. The value lies between -1 and 1. Key **takeaways** from the above values: larger the magnitude, stronger the correlation.




If the sign is positive, there is a regular correlation. Negative sign means there is an inverse correlation. Diagonal elements (the feature correlated with itself) will always have the value of +1.

# Preprocessing:

Data preprocessing involves transforming raw data to well-formed data sets so that data **mining** analytics can be applied. Preprocessing is done here in order to remove the unwanted columns in the data. It's a crucial process that can affect the success of data mining and machine learning projects. It makes knowledge **discovery** from data sets faster and can ultimately affect the performance of machine learning models.




Further, some of the data in the columns needed to be **standardized** and **indexed**, so that the computation process was made smoother. Some of the required columns also had unnecessary data present and some of the columns are unnecessary for analysis so we can remove these. These entire steps account for data **cleaning**. All these actions were performed using **Pyspark.** All the predictive machine learning modeling done in the next section will be done in spark as well, using **Spark.ML** libraries and functions

We have also added the age column instead of using yr_built for simplicity. The columns that are not in use will either be dropped or ignored in feature selection.

### Dataset after the Preprocessing:

| price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | view | condition | grade | sqft_basement | yr_built | zipcode | lat | long | age |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 221900 | 3 | 1.0 | 1180 | 5650 | 1.0 | 0 | 0 | 3 | 7 | 0 | 1955 | 98178 | 47.5112 | -122.257 | 67 |
| 538000 | 3 | 2.25 | 2570 | 7242 | 2.0 | 0 | 0 | 3 | 7 | 400 | 1951 | 98125 | 47.721 | -122.319 | 71 |
| 180000 | 2 | 1.0 | 770 | 10000 | 1.0 | 0 | 0 | 3 | 6 | 0 | 1933 | 98028 | 47.7379 | -122.233 | 89 |
| 604000 | 4 | 3.0 | 1960 | 5000 | 1.0 | 0 | 0 | 5 | 7 | 910 | 1965 | 98136 | 47.5208 | -122.393 | 57 |
| 510000 | 3 | 2.0 | 1680 | 8080 | 1.0 | 0 | 0 | 3 | 8 | 0 | 1987 | 98074 | 47.6168 | -122.045 | 35 |
| 1225000 | 4 | 4.5 | 5420 | 101930 | 1.0 | 0 | 0 | 3 | 11 | 1530 | 2001 | 98053 | 47.6561 | -122.005 | 21 |
| 257500 | 3 | 2.25 | 1715 | 6819 | 2.0 | 0 | 0 | 3 | 7 | 0 | 1995 | 98003 | 47.3097 | -122.327 | 27 |
| 291850 | 3 | 1.5 | 1060 | 9711 | 1.0 | 0 | 0 | 3 | 7 | 0 | 1963 | 98198 | 47.4095 | -122.315 | 59 |
| 229500 | 3 | 1.0 | 1780 | 7470 | 1.0 | 0 | 0 | 3 | 7 | 730 | 1960 | 98146 | 47.5123 | -122.337 | 62 |
| 323000 | 3 | 2.5 | 1890 | 6560 | 2.0 | 0 | 0 | 3 | 7 | 0 | 2003 | 98038 | 47.3684 | -122.031 | 19 |
| 662500 | 3 | 2.5 | 3560 | 9796 | 1.0 | 0 | 0 | 3 | 8 | 1700 | 1965 | 98007 | 47.6007 | -122.145 | 57 |
| 468000 | 2 | 1.0 | 1160 | 6000 | 1.0 | 0 | 0 | 4 | 7 | 300 | 1942 | 98115 | 47.69 | -122.292 | 80 |
| 310000 | 3 | 1.0 | 1430 | 19901 | 1.5 | 0 | 0 | 4 | 7 | 0 | 1927 | 98028 | 47.7558 | -122.229 | 95 |
| 400000 | 3 | 1.75 | 1370 | 9680 | 1.0 | 0 | 0 | 4 | 7 | 0 | 1977 | 98074 | 47.6127 | -122.045 | 45 |
| 530000 | 5 | 2.0 | 1810 | 4850 | 1.5 | 0 | 0 | 3 | 7 | 0 | 1900 | 98107 | 47.67 | -122.394 | 122 |
| 650000 | 4 | 3.0 | 2950 | 5000 | 2.0 | 0 | 3 | 3 | 9 | 970 | 1979 | 98126 | 47.5714 | -122.375 | 43 |
| 395000 | 3 | 2.0 | 1890 | 14040 | 2.0 | 0 | 0 | 3 | 7 | 0 | 1994 | 98019 | 47.7277 | -121.962 | 28 |
| 485000 | 4 | 1.0 | 1600 | 4300 | 1.5 | 0 | 0 | 4 | 7 | 0 | 1916 | 98103 | 47.6648 | -122.343 | 106 |
| 189000 | 2 | 1.0 | 1200 | 9850 | 1.0 | 0 | 0 | 4 | 7 | 0 | 1921 | 98002 | 47.3089 | -122.21 | 101 |
| 230000 | 3 | 1.0 | 1250 | 9774 | 1.0 | 0 | 0 | 4 | 7 | 0 | 1969 | 98003 | 47.3343 | -122.306 | 53 |

# 3. Predictive Modeling.

## StringIndexer:

A string column of labels is **encoded** into a column of label indices using the StringIndexer. The indices are in [0, numLabels], and there are four ordering options available: "alphabetDesc" for descending alphabetical order, "alphabetAsc" for ascending alphabetical order, "frequencyDesc" for descending order by **label frequency** (most frequent label assigned 0), and "frequencyDesc" for ascending order by label frequency (default = "frequencyDesc"). You must set the input column of the component to the name of the string-indexed column when using this string-indexed label in downstream pipeline components like Estimator or Transformer.

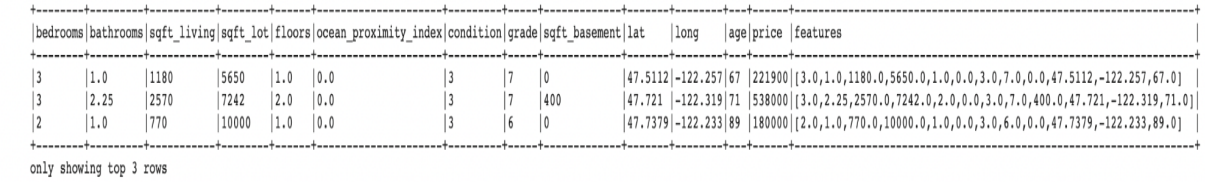
```
#Label-encoding for the "ocean_proximity" column

indexer = StringIndexer(inputCol="waterfront", outputCol="ocean_proximity_index")
data = indexer.fit(data).transform(data)
data = data.drop('waterfront')
```

## VectorAssembler

A transformer called VectorAssembler creates a single vector column from a provided list of columns. In order to train ML models like logistic regression and decision trees, it is important for integrating **raw features** and **features** produced by various feature converters into a single feature vector. All integer types, Boolean types, and vector types are all acceptable input column types

```
[ ] #Assembling features
    feature_assembly = VectorAssembler(inputCols = ['bedrooms',
     'bathrooms',
     'sqft_living',
     'sqft_lot',
     'floors',
     'ocean_proximity_index',
     'condition',
     'grade',
     'sqft_basement',
     'lat',
     'long',
     'age'], outputCol = 'features')
    output = feature_assembly.transform(housing_model)
    output.show(3, truncate = False)
```

for VectorAssembler. The input column values in each row will be concatenated into a vector in the prescribed order.



| bedrooms | bathrooms | sqft_living | sqft_lot | floors | ocean_proximity_index | condition | grade | sqft_basement | lat | long | age | price | features |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1.0 | 1180 | 5650 | 1.0 | 0.0 | 3 | 7 | 0 | 47.5112 | -122.257 | 67 | 221900 | [3.0,1.0,1180.0,5650.0,1.0,0.0,0.0,3.0,7.0,0.0,0.0,47.5112,-122.257,67.0] |
| 3 | 2.25 | 2570 | 7242 | 2.0 | 0.0 | 3 | 7 | 400 | 47.721 | -122.319 | 71 | 538000 | [3.0,2.25,2570.0,7242.0,2.0,0.0,0.0,3.0,7.0,400.0,47.721,-122.319,71.0] |
| 2 | 1.0 | 770 | 10000 | 1.0 | 0.0 | 3 | 6 | 0 | 47.7379 | -122.233 | 89 | 180000 | [2.0,1.0,770.0,10000.0,1.0,0.0,0.0,3.0,6.0,0.0,0.0,47.7379,-122.233,89.0] |

only showing top 3 rows

Bedroom, bathroom, sqft living, sqft lot, floors, ocean proximity index, condition, grade, sqft basement, lat, long, age) have all been integrated into a single vector; price has not been included because the target variable will be avoided in this case.

# StandardScaler

Changes a dataset of Vector rows, bringing each feature's standard deviation and/or mean values to **one unit SD**. It needs these inputs: withStd: By default, true. the data is scaled to one standard deviation. withMean: By default, False. Before scaling, the data is **centered** with the mean. When applying to sparse input, be careful because it will create a dense output. To compute summary statistics, a dataset can be fitted with a StandardScaler Estimator to create a StandardScalerModel.

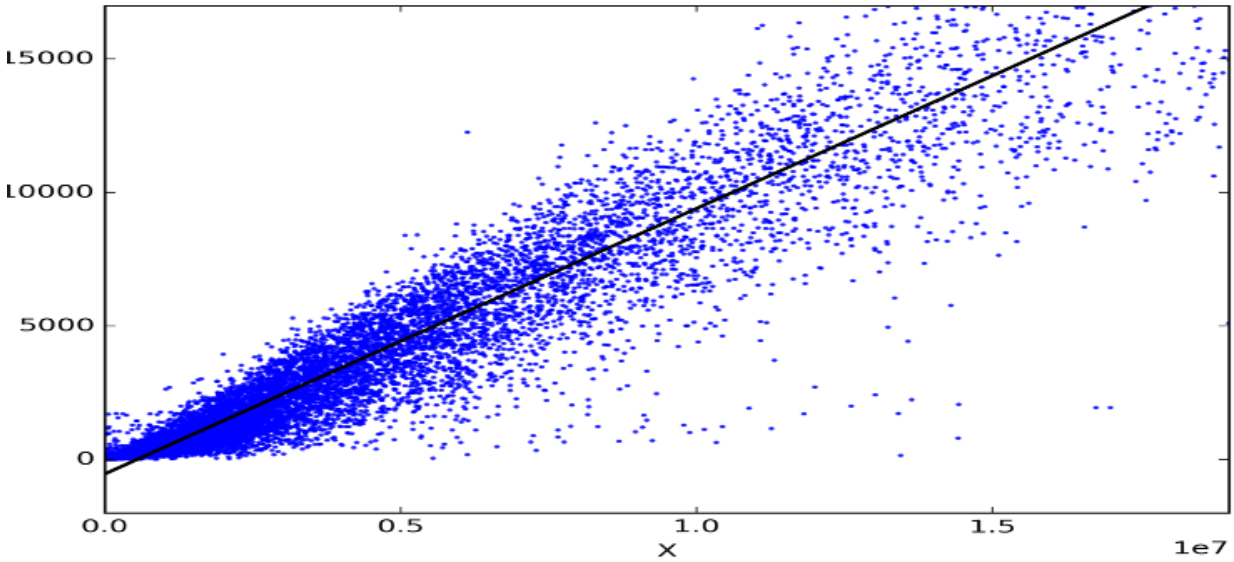
```
#Normalizing the features

scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",
                        withStd=True, withMean=False)

# Compute summary statistics by fitting the StandardScaler
scalerModel = scaler.fit(output)

# Normalize each feature to have unit standard deviation.
scaledOutput = scalerModel.transform(output)
scaledOutput.show(3, truncate = False)
```

The scaledfeatures column in our dataset has the modified Vector column that has features with unit standard deviation and/or zero mean.

| bedrooms | bathrooms | sqft_living | sqft_lot | floors | ocean_proximity_index | condition | grade | sqft_basement | lat | long | age | price |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1.0 | 1180 | 5650 | 1.0 | 0.0 | 3 | 7 | 0 | 47.5112 | -122.257 | 67 | 221900 |
| 3 | 2.25 | 2570 | 7242 | 2.0 | 0.0 | 3 | 7 | 400 | 47.721 | -122.319 | 71 | 538000 |
| 2 | 1.0 | 770 | 10000 | 1.0 | 0.0 | 3 | 6 | 0 | 47.7379 | -122.233 | 89 | 180000 |

| features | scaledFeatures |
|---|---|
| [3.0,1.0,1180.0,5650.0,1.0,0.0,0.0,3.0,7.0,0.0,0.0,47.5112,-122.257,67.0] | [2.150394665193954,1.2984261719458985,0.7621611823371609,0.4485911011153082,1.8518899351372 |
| [3.0,2.25,2570.0,7242.0,2.0,0.0,0.0,3.0,7.0,400.0,47.721,-122.319,71.0] | [1.0751973325969773,1.2984261719458985,0.6206169627602597,0.0985019221336541,1.8518899351372 |
| [2.0,1.0,770.0,10000.0,1.0,0.0,0.0,3.0,6.0,0.0,0.0,47.7379,-122.233,89.0] | [2.150394665193954,1.2984261719458985,0.8274892836032461,0.13278463859664616,2.777834902705 |
| | [2.150394665193954,1.2984261719458985,0.7294971316655683,0.11395320403697241,1.8518899351372 |
| | [2.150394665193954,1.2984261719458985,0.7403851485569092,0.19468615159181815,1.8518899351372 |

# Linear regression:



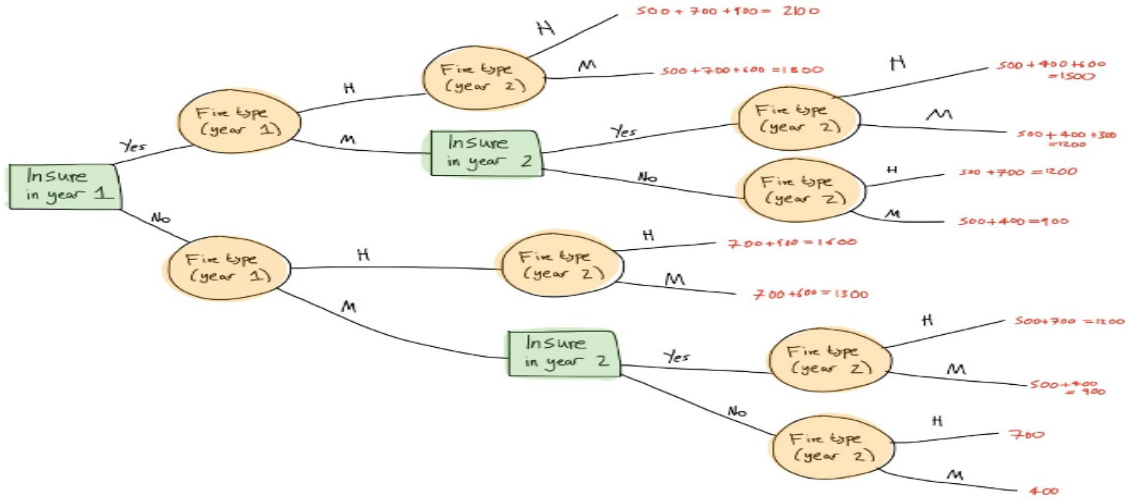

Predictive modeling most frequently used is linear regression. The line equation Y = mX + c is employed. X is the independent variable, Y is the dependent variable, and m is the slope/inclination of the straight line that forms the graph of the equation. Similar to the linear regression equation The intercept is given by Yo. The line's slope or coefficient is m, and the error term or residual is. In a linear regression study, the **best fit line** with the least amount of error must be obtained. To determine if an independent variable adequately explains the dependent variable is the primary goal of the linear regression study.

```
[ ] lr = LinearRegression(featuresCol = 'scaledFeatures', labelCol='price', maxIter=10, regParam=0.01, elasticNetParam=0.8)
    lr_model = lr.fit(df_train)

    lr_predictions = lr_model.transform(df_test)
    lr_predictions.select("prediction","price","scaledFeatures").show(5, truncate = False)
```

```
[ ]  lr_evaluator2 = RegressionEvaluator(predictionCol="prediction", \
                     labelCol="price",metricName="r2")
     lr_evaluator3 = RegressionEvaluator(predictionCol="prediction", \
                     labelCol="price",metricName="mae")

     print("R Squared (R2) on test data = %g" % lr_evaluator2.evaluate(lr_predictions))
     print("MAE on test data = %g" % lr_evaluator3.evaluate(lr_predictions))

     R Squared (R2) on test data = 0.673179
     MAE on test data = 130441
```

Model performance: R2 of 0.67 and mse tells us that the prediction has an offset of $130,000 by using linear regression.

# Decision Tree:




Decision tree is one of the well known and powerful supervised machine learning algorithms that can be used for classification as well as regression problems. The model is based on **decision rules** extracted from the training data. In a regression problem, the model uses the value of output; r squared and mean absolute error is used for a decision accuracy. Decision tree model is not good in generalization and **sensitive** to the changes in training data. A small change in a training dataset affects the model predictive performance. This can be observed by tweaking the value of the maxIter parameter.

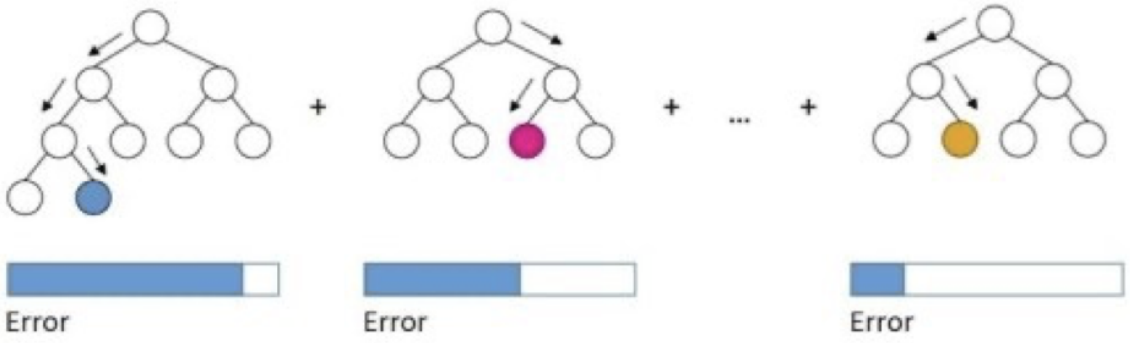
```
dt = DecisionTreeRegressor(featuresCol ='scaledFeatures', labelCol = 'price')
dt_model = dt.fit(df_train)
dt_predictions = dt_model.transform(df_test)


dt_evaluator = RegressionEvaluator(
    labelCol="price", predictionCol="prediction", metricName="r2")
r22 = dt_evaluator.evaluate(dt_predictions)

dt_evaluator = RegressionEvaluator(
    labelCol="price", predictionCol="prediction", metricName="mae")
mae = dt_evaluator.evaluate(dt_predictions)
```

```
[ ] print("R Squared (R2) on test data = %g" % r22)
    print("MAE on test data = %g" % mae)

    R Squared (R2) on test data = 0.660329
    MAE on test data = 113451
```

The model will predict with R2 of 0.729583 and mae tells us that the prediction has an offset of $111329 by using Decision Tree. The performance we got is better as compared to the linear regression. Thus, the Decision Tree shows the best performance so far.

# Gradient Boosting Tree:

Gradient boosting tree which sequentially combines many numbers of decision trees, called as **weak learners**, so that overall, it reduces the error that each individual tree faces. It follows the process of an **ensemble** technique called boosting to reduce overfitting. That is, each new tree corrects the errors of the previous one by adjusting the dataset.




Boosting algorithms play a crucial role in dealing with bias variance trade-off. Unlike bagging algorithms, which only control for high variance in a model, boosting controls both the aspects (bias & variance), and is considered to be more effective. A sincere understanding of GBM here should give you much needed confidence to deal with such critical issues.

```
[ ]  gbt = GBTRegressor(featuresCol = 'scaledFeatures', labelCol = 'price', maxIter=10)
     gbt_model = gbt.fit(df_train)
     gbt_predictions = gbt_model.transform(df_test)
```

```
[ ]  gbt_evaluator = RegressionEvaluator(
         labelCol="price", predictionCol="prediction", metricName="r2")
     r222 = gbt_evaluator.evaluate(gbt_predictions)

     gbt_evaluator = RegressionEvaluator(
         labelCol="price", predictionCol="prediction", metricName="mae")
     mae = gbt_evaluator.evaluate(gbt_predictions)


     print("R Squared (R2) on test data = %g" % r222)
     print("MAE on test data = %g" % mae)
```

```
R Squared (R2) on test data = 0.804318
MAE on test data = 95514.4
```

The model will predict with R2 of 0.729583 and mae tells us that the prediction has an offset of $111329 by using Decision Tree. The Performance we got is better as compared to both linear regression and Decision tree. Thus, Gradient boosting Tree shows the best performance.

# Gradient boosting tree with cross validation for hyperparameter tuning

Cross validation technique is used to identify how well our model performed and there is always a need to test the **performance** of our model to verify that our model is well trained with data without any overfitting and underfitting. This process of validation is performed only after training the model with data.

The general idea behind cross validation is that it is used to **resample** the data for training. **K fold** cross validation method (k = 5) was performed on the best of the 3 models used. **Hyperparameter Tuning** is hugely important in getting good performance with models. In order to understand this process, we first need to understand the difference between a model parameter and a model hyperparameter. Hyperparameters are model-specific properties that are 'fixed' before you even train and test your model on data.

```
[ ]  # Create 5-fold CrossValidator
     gbcv = CrossValidator(estimator = gbt,
                           estimatorParamMaps = gbparamGrid,
                           evaluator = gbevaluator,
                           numFolds = 5)
```

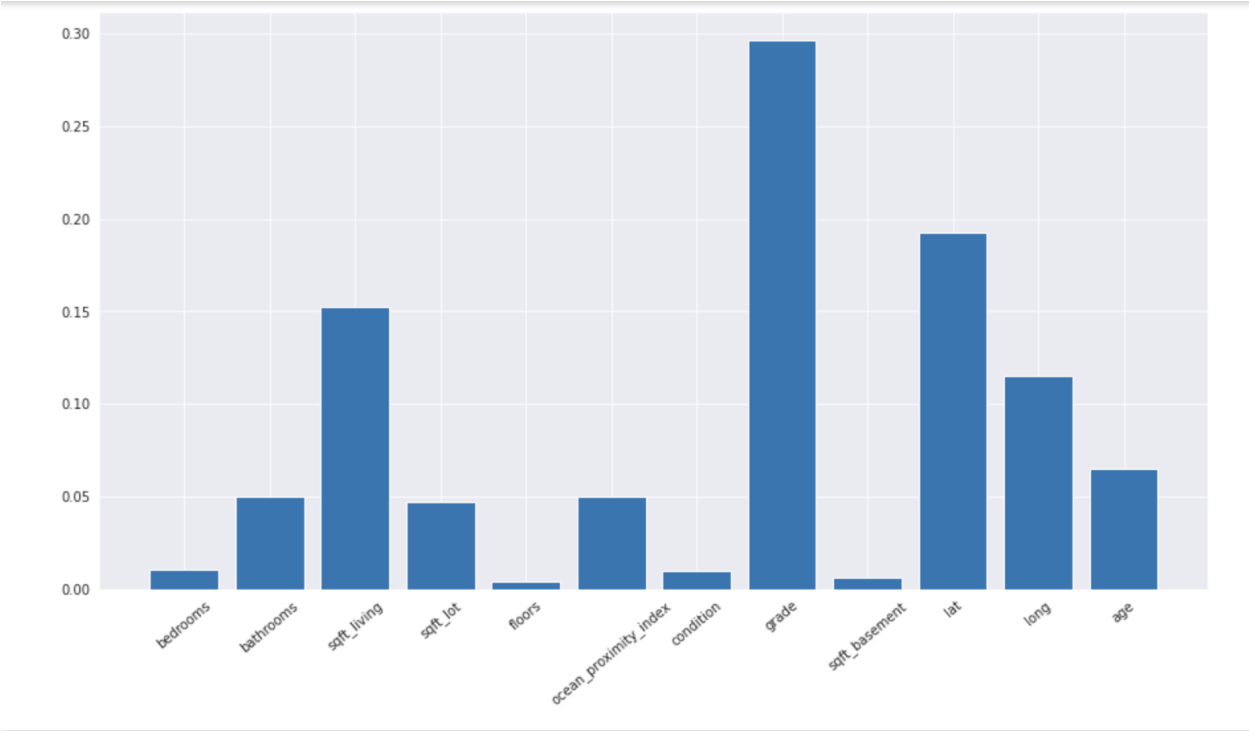
```
[ ]  gbcv_model = gbcv.fit(df_train)
     gbcv_predictions = gbcv_model.transform(df_test)
```

```
[ ]  gbt_predictions1 = gbt_model.transform(df_model_final)

     gbt_evaluator = RegressionEvaluator(
         labelCol="price", predictionCol="prediction", metricName="r2")
     r222 = gbt_evaluator.evaluate(gbt_predictions1)

     gbt_evaluator = RegressionEvaluator(
         labelCol="price", predictionCol="prediction", metricName="mae")
     mae = gbt_evaluator.evaluate(gbt_predictions1)

     print("R Squared (R2) on test data = %g" % r222)
     print("MAE on test data = %g" % mae)

     R Squared (R2) on test data = 0.820881
     MAE on test data = 94315.9
```

Gradient boosting tree with **5 fold cross validation** for hyperparameter tuning model will predict R2 and MAE of 0.874039 and 78828.1.

# Results:

Linear regression showed r2 and MAE of 0.7 and 126,000$. We need to **increase** the model fit and reduce error. We tried decision trees and gradient boosting trees as well for computation. Where the decision tree gave us an r2 and MAE of 0.73 and 110,000$. while gradient boosting trees gave us an r2 and MAE of 0.8 and 95,000 $. We can see a significant increase in fit and decrease in error. Since **Gradient boosting tree** performed the **best** of the 3 trained models we chose this for cross validation. Which finally gave us r2 and MAE 0.82 and 91,000$ offset.




```
[ ]  print("number of trees:",pi.getNumTrees)
     print("maxDepth:",pi.getOrDefault("maxDepth"))

     number of trees: 20
     maxDepth: 5
```

As discussed earlier, we were able to find the **feature importance**, ie, the features that influence the pricing values of the houses. We can see that Grade, Location coordinates (Lat and long) followed by living area and age affect the pricing the most.

# 4. Applications.

### Business Applications:

1. Cost savings:
   - ❖ Helping businesses **(Brokers, realtors)** to identify their target customers or recommend houses to them based on requirements, to carry out the business more efficiently.
   - ❖ Helps **buyers** to find the right fit that satisfies their needs and budget.
   - ❖ Helps a **seller** list his property based on the **market trends** and analysis. This way he can sell the house faster for maximum value.

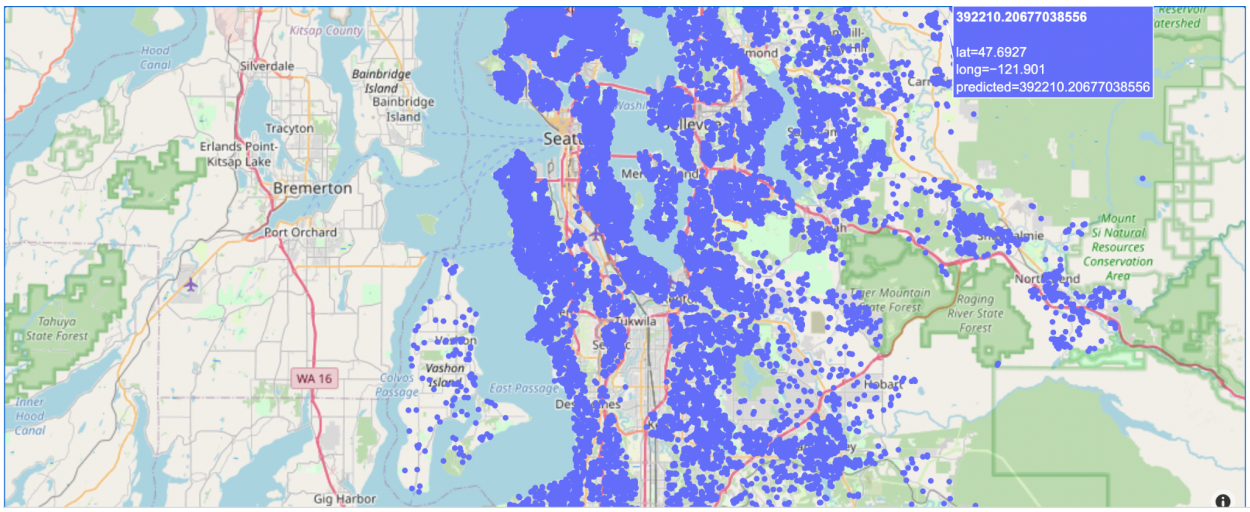
2. Product development: Providing a better understanding of customer needs with user friendly interface applications.

3. Market insights: Tracking past purchase behavior of customers and analyzing the market trends affected by various factors like no of. bedrooms, baths, size of property.

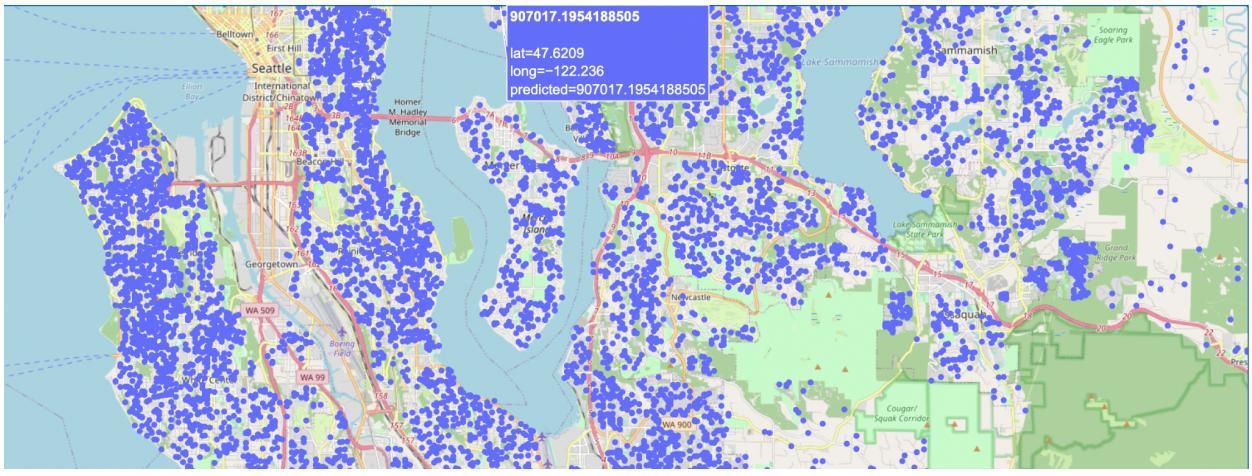
### Interactive map:

The houses that are for sale or listed in Seattle are displayed on an interactive map. A User friendly Interactive Map was developed to show users the trends in property prices and to display longitude and latitude when they hover over individual properties. While the first graph showed all housing data, denoted by **blue dots**. In the future, this might be expanded to show sale trends in which homes that are no longer available and homes that are up for rent might have distinct colored dots.

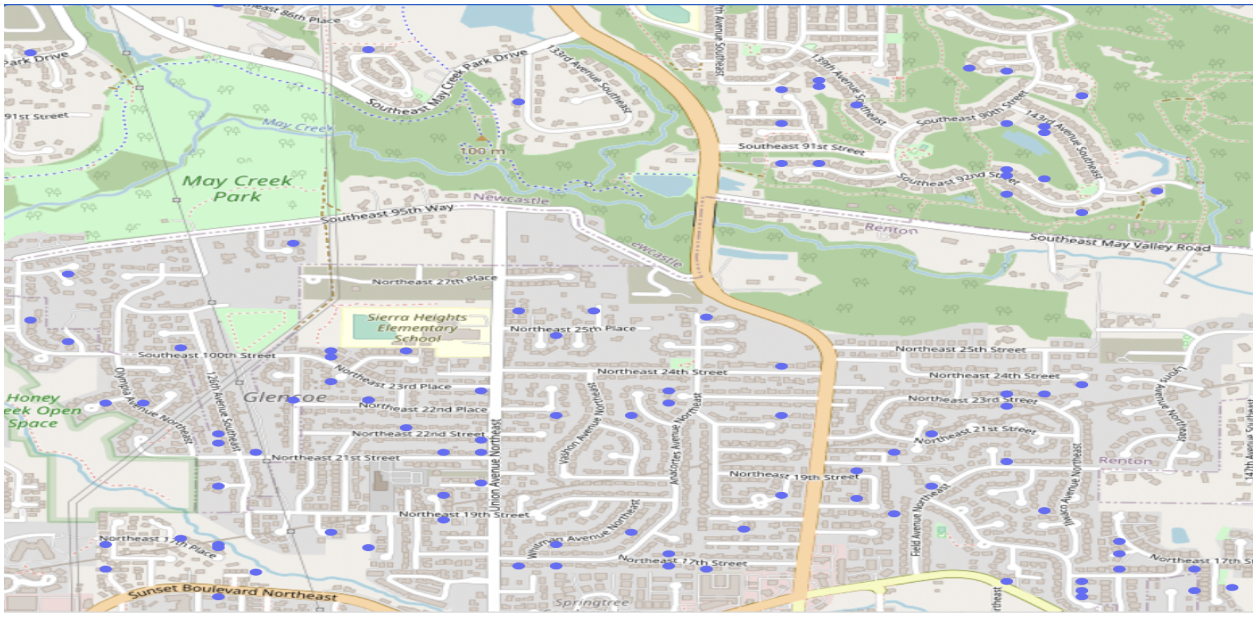
The predicted price, longitude, and latitude were highlighted on the **hover** to define the properties. Users may simply view the estimated pricing from the map as a result. It wouldn't be user-friendly to display our model's predictions in a table or dataframe. So that it is simpler to understand, we can share the results using this map.




There are fewer homes available for purchase near **Raging State Forest** and more homes for sale in Greenwood City in the areas where there is a higher demand for home purchases, as indicated by the high density of dots in graph 2. Using characteristics, prices will be projected based on location and neighborhood. If buyers are looking for a house on the outskirts of the city, they might select houses in **Vashon island** because price estimates near **Capital hill city** are high in comparison to the neighborhood.




Map shows the clear view of the houses which are in specific locations in **street view** also indicating their longitude, latitude and the predicted price.




This map can also be used to display houses that we saw in the dataframe manipulation section. This way, users will be able to see the location and features all in one place, thus selecting their best choice.

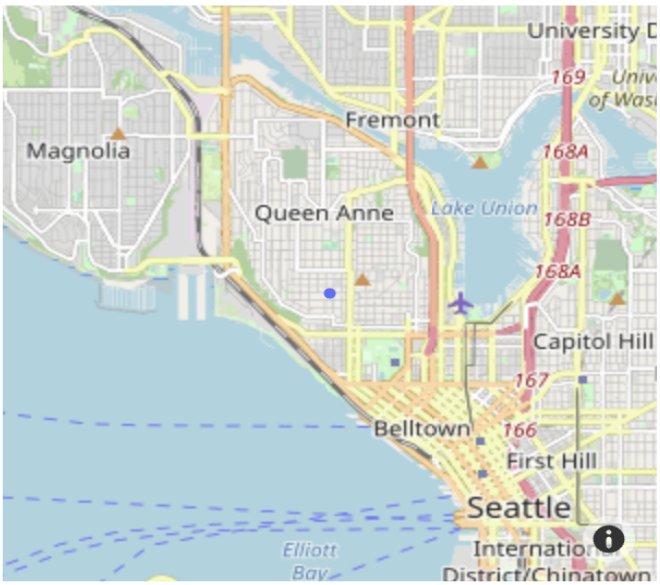
# 5. Conclusions and Future work.

```
def predicting(location, sqft_living, bedrooms, bath):

        url = 'https://nominatim.openstreetmap.org/search/' + urllib.parse.quote(location) +'?format=json'

        response = requests.get(url).json()

        x = np.zeros(5)
        x[0] = bedrooms
        x[1] = bath
        x[2] = sqft_living
        x[3] = response[0]["lat"]
        x[4] = response[0]["lon"]

        value = gb_clf.predict([x])[0]

        value1 = pd.DataFrame([value], columns = ['predicted'], index=[0])

        xx = pd.DataFrame([x], columns = ["bed", "bath", 'sqft', "lat", "long"], index=[0])

        final = pd.concat([value1, xx], axis=1)

        Interactive_map(final, zm=11, ht=300, wdt=300)


        return print("House price for your requirements:  $", value)
```

We have written a **function** to give us the predicted price of the house when we take a set of inputs from the user. Additionally, this function also plots the

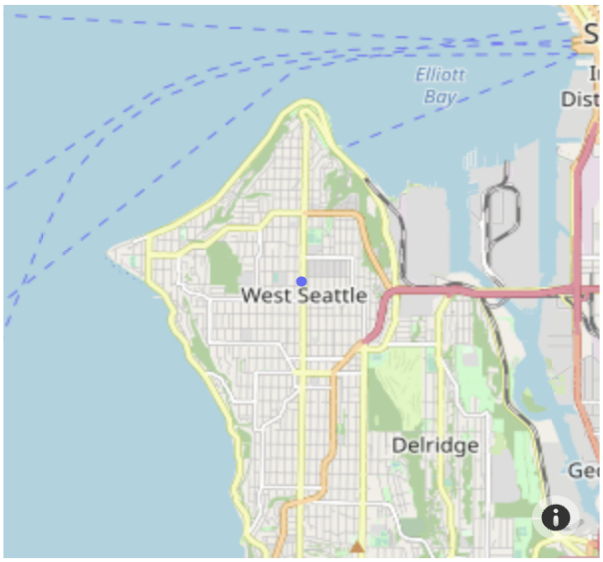
location on the map showing us where the house is exactly located. We are making predictions below for a 4 bed, 2 bath and 1000 sq ft at 221 W Highland Dr  and a 3 bed, 1 bath with 1000 sqft at 3435 California Ave.

```
predicting("211 W Highland Dr, Seattle, WA 98119", 1000, 4, 2)
```




```
House price for your requirements:  $ 491640.22605140164
```

```
predicting("3435 California Ave SW, Seattle, WA 98116", 1000, 3, 1)
```




```
House price for your requirements:  $ 433958.9025598331
```

Initially we wanted to merge various datasets available online, of different cities and states to create a larger dataset. But due to the difference in the features available in each dataset it was not easy to find the suitable datasets to carry forward this idea, given the time constraints. By utilizing **real-time data** and **online scraping**, we can increase the scope of the data sources. We may use a variety of other big data tools to examine **enormous volumes** of data in a manner similar to what we have learned and done in this project.

Given that population and housing are directly proportional entities, the dataset would continue to grow **exponentially** over the ensuing years. The operations that can be carried out on this dataset by a single machine become less and less viable. A **CI/CD** (continuous integration/continuous deployment) strategy can be used to ensure that the ML model is continually updated with the most recent data, taking things up a notch. This can improve the model's training and address potential future clashes..

# 6. References.


- https://www.velotio.com/engineering-blog/real-time-text-classification-using-kafka-and-scikit-learn
- https://www.kaggle.com/datasets/sameersmahajan/seattle-house-sales-prices?select=house_sales.csv
- Lecture slides: - ML tools
- https://stackoverflow.com/questions
- Articles from: https://medium.com/tag/big-data
- https://spark.apache.org/docs/2.3.1/api/python/pyspark.ml.html