VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

BIG DATA ANALYTICS AND BUSINESS INTELLIGENCE

House Price Analysis with PySpark

Instructor:	Nguyen Thanh Binh, PhD	
Student:	Ho Tri Khang	1952069
	Tran Tien Phat	1952386
	Vo Ngoc Sang	1952638

HO CHI MINH CITY, MAY 31, 2023

Contents

1 Introduction **2**

2 Dataset **2**

3 Big Data Analytics with PySpark **3**
3.1 Create session and read data . . . 3
3.2 Data cleaning . . . 4
3.2.1 Processing NULL values . . . 4
3.2.2 Removing outliers . . . 5
3.2.3 Processing categorical data . . . 6
3.3 Exploratory data analysis . . . 9
3.4 PySpark MLlib . . . 14
3.4.1 Feature extraction . . . 15
3.4.2 Regression models . . . 17
3.4.3 Classification models . . . 18
3.4.4 Clustering . . . 20
3.5 PySpark MySQL with MySQL database . . . 26

4 Further Study: Data Stack with Hadoop and Spark **27**
4.1 Our System . . . 28
4.2 Running our Data Stack . . . 29

5 Conclusion **33**

1 Introduction

In today's world, *Big Data*[2] is a crucial factor for the development of businesses and organizations. Big Data allows organizations to establish baselines, benchmarks, and goals to keep moving forward. With the massive amount of data generated every minutes, *Big Data Analytics* is super important to gain valuable information that helps organization to grow up. Fortunately, developers nowadays do not need to spend several hours writing code just to implement a single function anymore. The appearance of Big Data Analytics tools, such as *Hadoop Mapreduce* and *Apache Spark*[4, 1] makes life easier by providing useful APIs to work with Big Data, from Data Cleaning, Data Processing, Data Analytics and the functionality to work with data in distributed storage systems. Therefore, equipping and mastering the skill of using Big Data Analytics tools are extremely essential for organizations to save their efforts and simplify the processes.

In this project, we applied *PySpark*[5], an interface of Apache Spark for Python programming language to process and analyze a public dataset in order to learn the skill of using Big Data Analytics tool. Apache Spark is an open source analytics engine for large-scale data processing. Spark provides useful APIs to work with Big Data and an interface for programming clusters with implicit data parallelism and fault tolerance. We mainly investigated two components of PySpark: *SparkSQL*, which provides the interface to work with a data abstraction called Dataframes, and *Spark MLib*, a distributed Machine Learning framework built on top of *Spark Core* - the core engine of Spark.

We also conducted a further study in which we created a Datastack with PySpark and Hadoop.

2 Dataset

The dataset we worked with is a public dataset on Kaggle called India House Price. The link to this dataset can be found **here** (last accessed on 13/11/2022). There are three datasets corresponding to three regions in India found on the main Kaggle page, we chose to work with the Delhi house data in this project.

Summary of the dataset columns is shown in Table 1:

Column	**Description**
Area	Area of the house in square feet
BHK	No. of Bedrooms along with 1 Hall and 1 kitchen
Bathroom	No. of Bathrooms
Furnishing	Whether the house is furnished or not
Locality	Location of the house
Status	House status as 'Ready to move' or not
Transaction	Whether the house is New or Being re-sold
Type	Type of the house (Apartment or Builder Floor)
Price	House price in INR (India Rupee)

Table 1: Description table for the dataset

3 Big Data Analytics with PySpark

We used PySpark, an interface of Apache Spark for Python Programming, PySpark comes with many of the functionalities in Spark.
PySpark can be easily installed with pip using the command:

```
pip install pyspark
```

Normally, PySpark requires setting up clusters to run in distributed environments. In this work, we only ran PySpark on local mode, which means our local machine acted as both a master node and a worker node in a single-node cluster. Our work will be demonstrated in the following sections.

3.1 Create session and read data

First we need to create a Spark session. Spark session acts as an entry point for any Spark applications.

```python
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('mysparkapp').getOrCreate()
spark
```

SparkSession - in-memory
SparkContext

Spark UI
Version
`v3.3.1`
Master
`local[*]`
AppName
`mysparkapp`

After spark session has been created, we can read data from csv file:

```python
dataset_path = "Delhi house data.csv"
df = spark.read.csv(dataset_path, header=True, inferSchema=True)
df.show(10)
```

Area	BHK	Bathroom	Furnishing	Locality	Parking	Price	Status	Transaction	Type	Per_Sqft
800.0	3	2	Semi-Furnished	Rohini Sector 25	1	6500000	Ready_to_move	New_Property	Builder_Floor	null
750.0	2	2	Semi-Furnished	J R Designers Flo...	1	5000000	Ready_to_move	New_Property	Apartment	6667
950.0	2	2	Furnished	Citizen Apartment...	1	15500000	Ready_to_move	Resale	Apartment	6667
600.0	2	2	Semi-Furnished	Rohini Sector 24	1	4200000	Ready_to_move	Resale	Builder_Floor	6667
650.0	2	2	Semi-Furnished	Rohini Sector 24 ...	1	6200000	Ready_to_move	New_Property	Builder_Floor	6667
1300.0	4	3	Semi-Furnished	Rohini Sector 24	1	15500000	Ready_to_move	New_Property	Builder_Floor	6667
1350.0	4	3	Semi-Furnished	Rohini Sector 24	1	10000000	Ready_to_move	Resale	Builder_Floor	6667
650.0	2	2	Semi-Furnished	Delhi Homes, Rohi...	1	4000000	Ready_to_move	New_Property	Apartment	6154
985.0	3	3	Unfurnished	Rohini Sector 21	1	6800000	Almost_ready	New_Property	Builder_Floor	6154
1300.0	4	4	Semi-Furnished	Rohini Sector 22	1	15000000	Ready_to_move	New_Property	Builder_Floor	6154

only showing top 10 rows

After reading the file, Spark will create a Dataframe object. Each Dataframe is a distributed collection of data, which is organized into named columns.
df.count() and *df.columns* are used to show the number of rows and the list of columns of the Dataframe, There are 1259 rows and 11 columns in total.

```python
print('Number of rows: ', df.count())
print('Columns: ', df.columns)
print('Number of columns: ', len(df.columns))
```

```
Number of rows:  1259
Columns:  ['Area', 'BHK', 'Bathroom', 'Furnishing', 'Locality', 'Parking', 'Price', 'Status', 'Transaction', 'Type', 'Per_Sqft']
Number of columns:  11
```

We can also print the schema of our dataframe. When a csv is read by Spark, its schema will be inferred based on the values of each column.

```python
df.printSchema()
```

```
root
 |-- Area: double (nullable = true)
 |-- BHK: integer (nullable = true)
 |-- Bathroom: integer (nullable = true)
 |-- Furnishing: string (nullable = true)
 |-- Locality: string (nullable = true)
 |-- Parking: integer (nullable = true)
 |-- Price: integer (nullable = true)
 |-- Status: string (nullable = true)
 |-- Transaction: string (nullable = true)
 |-- Type: string (nullable = true)
 |-- Per_Sqft: integer (nullable = true)
```

3.2 Data cleaning

Data cleaning is the process of fixing or removing incorrect, corrupted or duplicate data within a dataset. First we need to import some functions from *pyspark.sql* module that are useful for data cleaning:

```python
from pyspark.sql.functions import col,isnan,when,count,udf,mean
from pyspark.sql.types import StringType, IntegerType, FloatType
```

3.2.1 Processing NULL values

First we count the number of NULL values for each column in the dataset.

```python
df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in
df.columns]).show()
```

```
+----+---+--------+----------+--------+-------+-----+------+-----------+----+--------+
|Area|BHK|Bathroom|Furnishing|Locality|Parking|Price|Status|Transaction|Type|Per_Sqft|
+----+---+--------+----------+--------+-------+-----+------+-----------+----+--------+
|   0|  0|       2|         5|       0|     33|    0|     0|          0|   5|     241|
+----+---+--------+----------+--------+-------+-----+------+-----------+----+--------+
```

As there are too many NULL values in the column Per_sqft (241 rows compared to 1259 total sample size), also the meaning of this column is unclear, we will remove this column.

```python
df = df.drop('Per_sqft')
```

The number rows with NULL values for columns Bathroom, Furnishing and Type is insignificant, we can remove them:

```python
df = df.dropna('any', subset=['Bathroom', 'Furnishing', 'Type'])
```

The column *Parking* has 33 NULL values, removing all the NULL rows can lead to considerable data loss, a better solution is to replace the NULL cells with the median value of all other *Parking* values.

```python
median_parking = round(df.approxQuantile("Parking", [0.5], 0)[0])
df = df.fillna(median_parking, 'Parking')
print(median_parking)
```

```
1
```

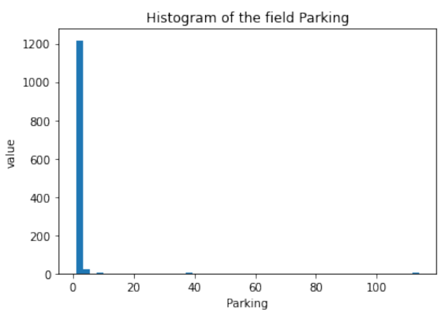
Now the dataframe is left with no NULL values:

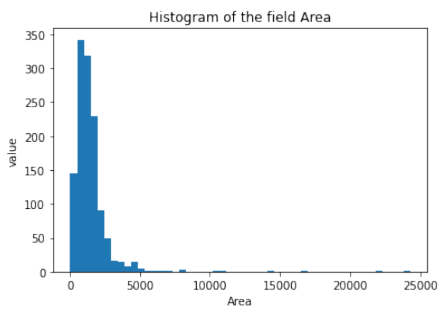
Area	BHK	Bathroom	Furnishing	Locality	Parking	Price	Status	Transaction	Type
0	0	0	0	0	0	0	0	0	0

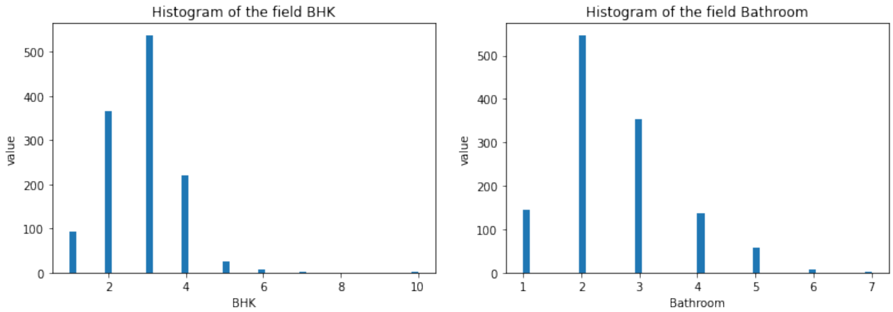
3.2.2 Removing outliers

For numeric values, we first have to plot the histogram to examine the distribution of these fields.

```python
from pyspark.pandas import  DataFrame, set_option
import matplotlib.pyplot as plt
set_option('plotting.backend', 'matplotlib')
for field in ['Parking','Area','BHK','Bathroom']:
  ppdf = DataFrame(df)
  ax = ppdf[field].plot.hist(bins=50)
  ax.set_xlabel(field)
  ax.set_ylabel('value')
  ax.set_title(f'Histogram of the field {field}')
  plt.show()
```



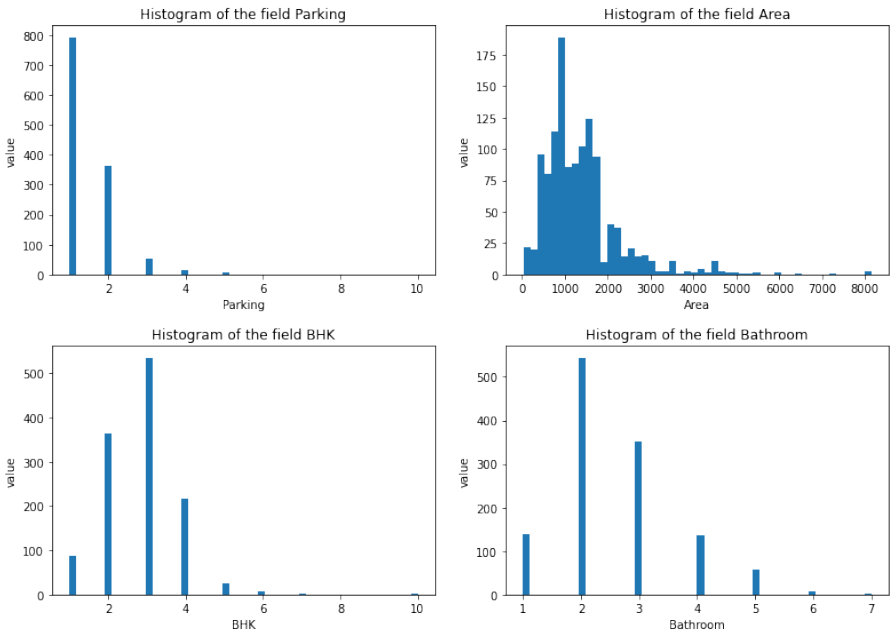




After examining the distribution of the numerical value, we then cut off some of the outliers, using the filter function.

```
df = df.filter((df['Parking']<=10) & (df['Area']<=10000))
```

We then plot the histogram to examine the distribution of the numerical fields again, we can now see the distribution is much more like the normal distribution.



3.2.3 Processing categorical data

Categorical data is often recorded strings having different values or formats. We need to process the categorical data to make sure the number of samples for each categories are balance and there should not be too many categories, as it can affect further analysis processes.
First we will use the PySpark Dataframe *groupby* function to see the counts of each categories for all categorical variables in our dataframe:

```python
df.groupby('Furnishing').count().show()
df.groupby('Status').count().show()
df.groupby('Transaction').count().show()
df.groupby('Type').count().show()
```

```
+-------------+-----+
|   Furnishing|count|
+-------------+-----+
|Semi-Furnished|  706|
|    Furnished|  183|
|  Unfurnished|  353|
+-------------+-----+

+------------+-----+
|      Status|count|
+------------+-----+
|Ready_to_move| 1167|
| Almost_ready|   75|
+------------+-----+

+-----------+-----+
| Transaction|count|
+-----------+-----+
|      Resale|  766|
|New_Property|  476|
+-----------+-----+

+------------+-----+
|        Type|count|
+------------+-----+
|   Apartment|  581|
|Builder_Floor|  661|
+------------+-----+
```

The four categorical columns *Furnishing*, *Status*, *Transaction*, *Type* are quite clean and don't need further processing.

```python
grouped_locality = df.groupby('Locality').count().orderBy('Locality')
print('Number of categories: ', len(grouped_locality.collect()) )
grouped_locality.show(truncate=False)
```

```
Number of categories:  361
+--------------------+-----+
|            Locality|count|
+--------------------+-----+
|APL Builder Floor...|    2|
|Aashirwaad Chowk,...|    6|
|Abhimanyu Apartme...|    2|
|Abul Fazal Enclav...|    3|
|Abul Fazal Enclav...|    3|
|Adarsh Homes, Dwa...|    3|
|Ahinsha Vatika, R...|    1|
|           Alaknanda|   20|
|Amar Colony, Lajp...|    2|
|Andheria Mor, Meh...|    3|
|Anekant Apartment...|    1|
|Anupam Enclave, S...|    1|
|Apna Apartments, ...|    1|
|Aravali Apartment...|    4|
|Aravali Tower, Ch...|    1|
```

However, the column *Locality* has too many categories (363 categories), some of them has only 1-2 samples in the whole dataframe. This can can affect the analysis process as we move to the next stages.

If we pay attention to every single location strings, some of the localities belong to the same larger district/area, we will group each group of similar localities to a larger category in order

to have a smaller number of categories.
First we will try to extract the District/Region name from each detailed location. For example, the two locations:

- Abul Fazal Enclave Part 1, Okhla
- Abul Fazal Enclave Part-II, Okhla

Both of them belong to Okhla, we will replace their values with 'Okhla'. By doing so, we come up with a new column with significantly less number of categories.

Some of the long strings contain redundant information. We will try to find and extract the location string from those long strings, using the list of filtered locations that we have achieved in the above step.

```python
def filterLocation(location):
    if len(location) > 100:
        return location

    filtered_area = location.rsplit(',',1)[-1].strip()

    # filter out unnecessary information for classifying location
    tokens = filtered_area.rsplit(' ', 1)
    if len(tokens) > 1 and tokens[-1].isdigit():
        filtered_area = tokens[0]

    pos = filtered_area.find('Sector')
    pos = filtered_area.find('Phase') if pos == -1 else pos
    pos = filtered_area.find('Block') if pos == -1 else pos
    pos = filtered_area.find('Pocket') if pos == -1 else pos

    if pos >= 0:
        filtered_area = filtered_area[:pos]

    return filtered_area.strip()

locality_convert = udf(filterLocation, StringType())
df = df.withColumn('filtered_locality', locality_convert(df.Locality))
```

```python
filtered_locations = set()
for x in df.collect():
    if len(x['filtered_locality']) < 100:
        filtered_locations.add(x['filtered_locality'])

def filterLocation2(locations):
    def filterLongLocation(x):
        if len(x) < 100:
            return x
        for location in locations:
```

```
                if x.find(location) != -1:
                    return location
            return x
        return udf(filterLongLocation)

df = df.withColumn('filtered_locality',
filterLocation2(filtered_locations)('filtered_locality'))
```

For categories with less than 5 occurrences, we group them to a group "others".

```
grouped_locality = df.groupby('Locality').count().orderBy('Locality')
print('Number of categories: ', len(grouped_locality.collect()) )
grouped_locality.show(truncate=False)
```

The final *Locality* column has 42 categories, a huge optimization compared to 363 categories initially.

```
Number of categories:  42
+--------------------+-----+
|            Locality|count|
+--------------------+-----+
|           Alaknanda|   56|
|          Budh Vihar|   18|
|         Chhattarpur|   22|
| Chhattarpur Enclave|    8|
|   Chittaranjan Park|   28|
|Commonwealth Game...|   28|
|      Dilshad Garden|   30|
|              Dwarka|   74|
|          Dwarka Mor|   13|
|        Geeta Colony|    5|
|     Greater Kailash|   42|
```

3.3 Exploratory data analysis

For categorical data, first, we have to see the number of distinct values in a specific field. To do this, we use the following lines of code:

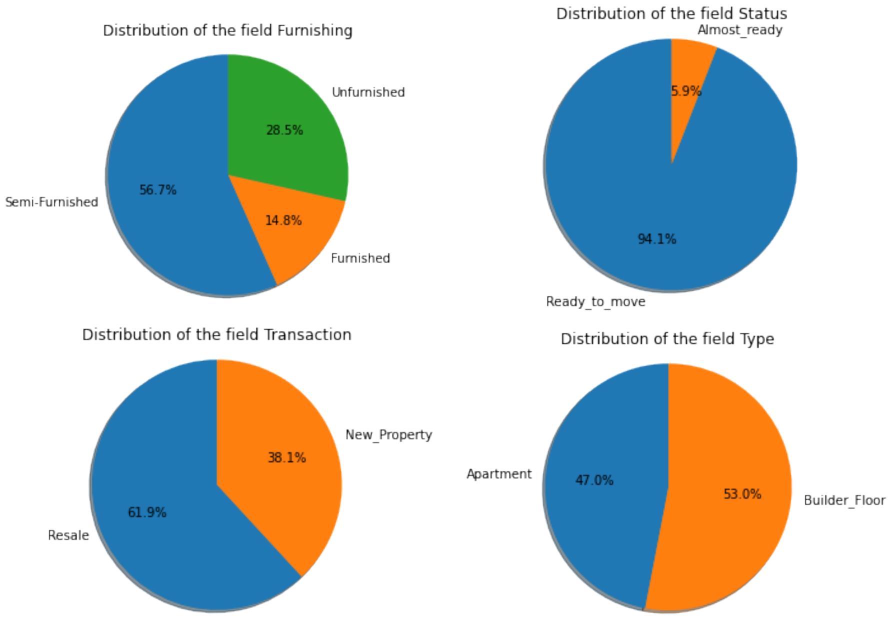
```
for field in ['Furnishing', 'Status', 'Transaction', 'Type', 'Locality']:
  n = df.select(field).distinct().count()
  print(f'Field: {field} \nNumber of unique values: {n}\n\n')
```

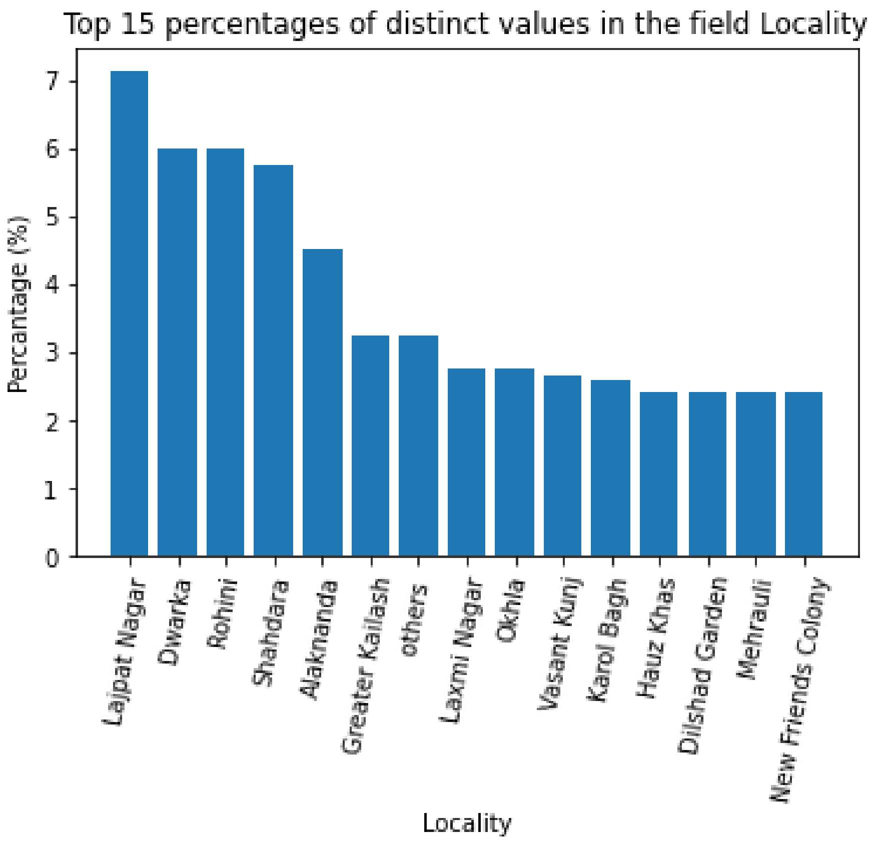
```
Field: Furnishing
Number of unique values: 3

Field: Status
Number of unique values: 2

Field: Transaction
Number of unique values: 2

Field: Type
Number of unique values: 2

Field: Locality
```

```
Number of unique values: 42
```

We then visualize the distribution of these categorical fields, for *Furnishing*, *Status*, *Transaction*, *Type*, because of the low number of distinct values (from 2 to 3), we can draw pie chart for these fields. *Locality*, however, has a relatively high number of distinct values (42), so using bar chart would be more appropriate.

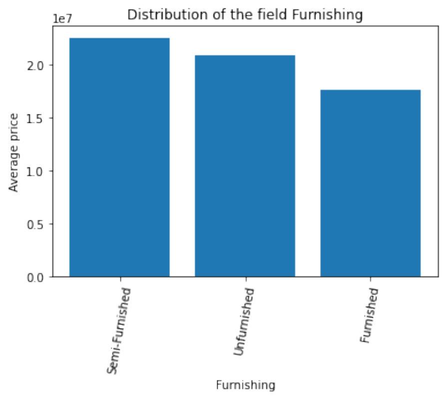
```python
import matplotlib.pyplot as plt
# first, we draw pie chart for the two field area_type, availability
for field in ['Furnishing', 'Status', 'Transaction', 'Type']:
  labels = df.groupBy(field).count().rdd.map(lambda x: x[0]).collect()
  sizes = df.groupBy(field).count().rdd.map(lambda x: x[1]).collect()

  fig1, ax1 = plt.subplots()
  ax1.pie(sizes, labels=labels, autopct='%1.1f%%',
          shadow=True, startangle=90)
  ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

  plt.title(f'Distribution of the field {field}')
  plt.show()

# as the field location has too many distinct value (42 distinct value), it is more appropriate to draw a barchar here:
labels = df.groupBy('Locality').count().sort(col('count').desc()).rdd.map(lambda x: x[0]).collect()
sizes = df.groupBy('Locality').count().sort(col('count').desc()).rdd.map(lambda x: x[1]).collect()
n_cols = df.count()
sizes = [ x/n_cols*100 for x in sizes]

plt.bar(labels[:15], sizes[:15])
plt.ylabel('Percantage (%)')
plt.xlabel('Locality')
plt.title('Top 15 percentages of distinct values in the field Locality')
plt.xticks(rotation=80)
plt.show()
```

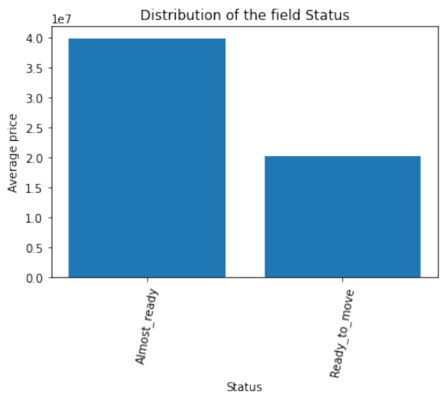


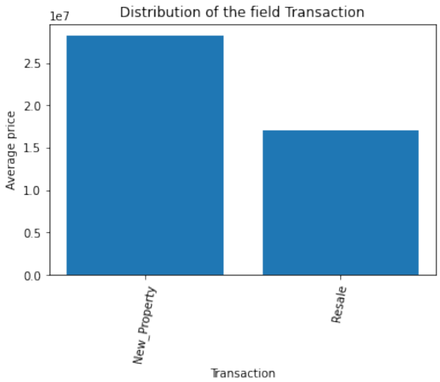


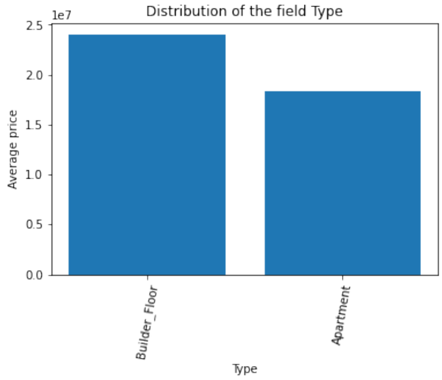
To make the exploratory data analysis more interesting, we can even compare the average house price between categories using bar charts.

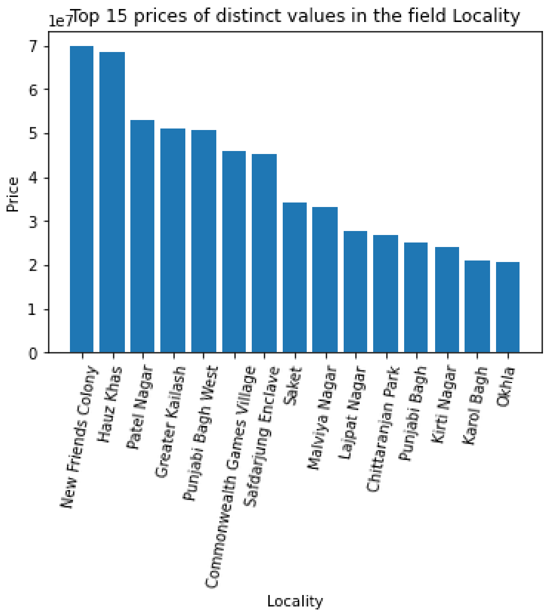
```python
for field in ['Furnishing', 'Status', 'Transaction', 'Type']:
  labels = df.groupBy(field).mean().sort(col('avg(price)').desc()).rdd.map(lambda x: x[0]).collect()
  sizes = df.groupBy(field).mean().sort(col('avg(price)').desc()).rdd.map(lambda x: x[5]).collect()

  plt.bar(labels, sizes)
  plt.ylabel('Average price')
  plt.xlabel(field)
  plt.title(f'Average prices between distinct values in the field {field}')
  plt.xticks(rotation=80)
  plt.title(f'Distribution of the field {field}')
  plt.show()

labels = df.groupBy('Locality').mean().sort(col('avg(price)').desc()).rdd.map(lambda x: x[0]).collect()
sizes = df.groupBy('Locality').mean().sort(col('avg(price)').desc()).rdd.map(lambda x: x[5]).collect()

plt.bar(labels[:15], sizes[:15])
plt.ylabel('Price')
plt.xlabel('Locality')
plt.title('Top 15 prices of distinct values in the field Locality')
plt.xticks(rotation=80)
plt.show()
```







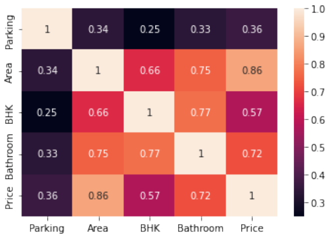




For numerical fields, as we have seen the distribution of each numeric column in the previous section, we now will examine the correlation coefficient between numeric fields to each other.

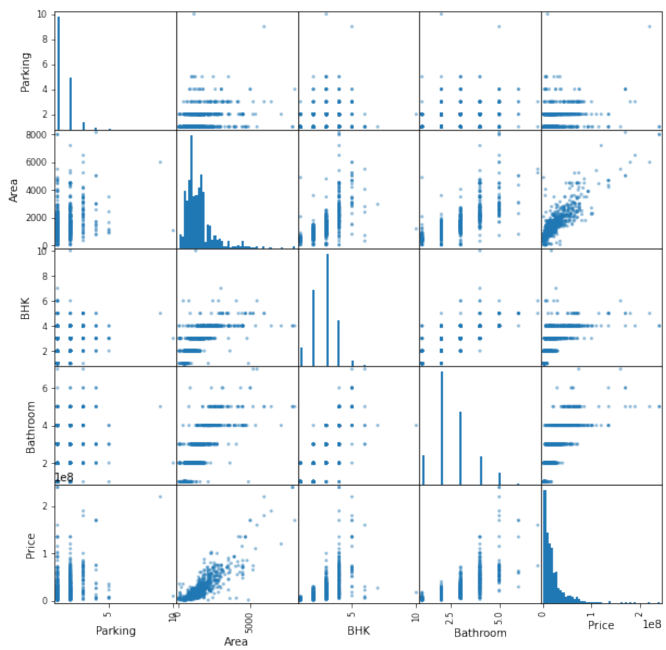
```
import seaborn as sns
import numpy as np
confusion_matrix = []
for field1 in ['Parking','Area','BHK','Bathroom', 'Price']:
  confusion_matrix_row = []
  for field2 in ['Parking','Area','BHK','Bathroom', 'Price']:
    x = df.select(field2).rdd.flatMap(lambda x: x).collect()
    y = df.select(field1).rdd.flatMap(lambda x: x).collect()
    confusion_matrix_row.append(np.corrcoef(x, y)[0][1])
  confusion_matrix.append(confusion_matrix_row)
confusion_matrix = np.array(confusion_matrix)
```

```python
ax = sns.heatmap(confusion_matrix, annot=True, xticklabels =
['Parking','Area','BHK','Bathroom', 'Price'], yticklabels =
['Parking','Area','BHK','Bathroom', 'Price'])
```



To be more clear about how relevant these numerical fields are to each other, we need to plot the scatter matrix, using the library pandas:

```python
import pandas as pd
# to be more clear,
pd.plotting.scatter_matrix(df.toPandas()[['Parking','Area','BHK','Bathroom',
'Price']], hist_kwds={'bins':50}, figsize = (10,10))
```



3.4 PySpark MLlib

In this section we will investigate PySpark MLlib, a distributed Machine Learning library to work with PySpark Dataframes. In particular, We will play with some models to predict the house price value from the other values in our dataframe.

3.4.1 Feature extraction

Before we can fit our dataset to PySpark models, an important step is to extract features from our columns. we first import some necessary libraries from PySpark MLlib:

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
StandardScaler, MinMaxScaler, VectorAssembler
```

String values are not a good option for categorical data to fit to prediction models. We need to convert them to One-hot-encoded vectors. There are 2 steps to process categorical data:

1. Transform strings to indexes.
2. One hot encode the indexed values. The result is a set of one-hot-encoded vectors

We will use the two classes, *StringIndexer* and *OneHotEncoder* for this two steps.

```
string_indexer = StringIndexer(
    inputCols=['Furnishing', 'Status', 'Transaction', 'Type', 'Locality'],
    outputCols=['indexed_furnishing', 'indexed_status',
    'indexed_transaction', 'indexed_type', 'indexed_locality']
)

# demonstrate the output of StringIndexer
show_df = string_indexer.fit(df).transform(df)
show_df.select(['indexed_furnishing', 'indexed_status',
'indexed_transaction', 'indexed_type', 'indexed_locality']).show(5)
```

```
+------------------+--------------+-------------------+------------+----------------+
|indexed_furnishing|indexed_status|indexed_transaction|indexed_type|indexed_locality|
+------------------+--------------+-------------------+------------+----------------+
|               0.0|           0.0|                1.0|         0.0|             2.0|
|               0.0|           0.0|                1.0|         1.0|             2.0|
|               2.0|           0.0|                0.0|         1.0|             2.0|
|               0.0|           0.0|                0.0|         0.0|             2.0|
|               0.0|           0.0|                1.0|         0.0|             2.0|
+------------------+--------------+-------------------+------------+----------------+
only showing top 5 rows
```

The output of *stringIndexer* for each string is an index value ranging from 0 to total number of categories.

```
one_hot_encoder = OneHotEncoder(
    inputCols=['indexed_furnishing', 'indexed_status',
    'indexed_transaction', 'indexed_type', 'indexed_locality'],
    outputCols=['encoded_furnishing', 'encoded_status',
    'encoded_transaction', 'encoded_type', 'encoded_locality'],
    dropLast=False
)
# demonstrate the output of OneHotEncoder
show_df = one_hot_encoder.fit(show_df).transform(show_df)
show_df.select(['encoded_furnishing', 'encoded_status',
'encoded_transaction', 'encoded_type', 'encoded_locality']).show(5)
```

```
+------------------+--------------+--------------------+-------------+----------------+
|encoded_furnishing|encoded_status|encoded_transaction| encoded_type|encoded_locality|
+------------------+--------------+--------------------+-------------+----------------+
|     (3,[0],[1.0])| (2,[0],[1.0])|       (2,[1],[1.0])|(2,[0],[1.0])|  (42,[2],[1.0])|
|     (3,[0],[1.0])| (2,[0],[1.0])|       (2,[1],[1.0])|(2,[1],[1.0])|  (42,[2],[1.0])|
|     (3,[2],[1.0])| (2,[0],[1.0])|       (2,[0],[1.0])|(2,[1],[1.0])|  (42,[2],[1.0])|
|     (3,[0],[1.0])| (2,[0],[1.0])|       (2,[0],[1.0])|(2,[0],[1.0])|  (42,[2],[1.0])|
|     (3,[0],[1.0])| (2,[0],[1.0])|       (2,[1],[1.0])|(2,[0],[1.0])|  (42,[2],[1.0])|
+------------------+--------------+--------------------+-------------+----------------+
only showing top 5 rows
```

OneHotEncoder will convert index values to vectors. The length of a vector is the total number of categories.

For numeric variables, values will be normalized to a smaller range foe the ease of computation and optimization. In order to do so, We use *VectorAssembler* to assemble all numeric values to a single vector, then *MinMaxScaler* is used to normalized that vector.

```
numeric_assembler = VectorAssembler(inputCols=['Area','BHK','Bathroom','Parking'], outputCol=f'numeric_vec')
# scaler = StandardScaler(inputCol='numeric_vec', outputCol='scaled_numeric')
scaler = MinMaxScaler(inputCol='numeric_vec', outputCol='scaled_numeric')
show_df = numeric_assembler.transform(show_df)
show_df = scaler.fit(show_df).transform(show_df)

show_df.select(['scaled_numeric']).show(5, truncate=False)
```

```
+--------------------------------------------------------------+
|scaled_numeric                                                |
+--------------------------------------------------------------+
|[0.03180619644034278,0.2222222222222222,0.16666666666666666,0.0] |
|[0.0297462096242584,0.1111111111111111,0.16666666666666666,0.0]  |
|[0.03798615688859591,0.1111111111111111,0.16666666666666666,0.0] |
|[0.02356624917600527,0.1111111111111111,0.16666666666666666,0.0] |
|[0.025626235992089647,0.1111111111111111,0.16666666666666666,0.0]|
+--------------------------------------------------------------+
only showing top 5 rows
```

Next we concatenate all the columns, including categorical and numeric columns into one single vector column. This features column will be used to fit the Machine Learning model.

```
assembler = VectorAssembler(inputCols=['encoded_furnishing', 'encoded_status', 'encoded_transaction',
                                       'encoded_type',
                                       'encoded_locality','scaled_numeric'],
                                       outputCol='features')
show_df = assembler.transform(show_df)
show_df.select('features').show(5, truncate=False)
```

```
+------------------------------------------------------------------------------------------------------------+
|features                                                                                                    |
+------------------------------------------------------------------------------------------------------------+
|(55,[0,3,6,7,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.03180619644034278,0.2222222222222222,0.16666666666666666]) |
|(55,[0,3,6,8,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.0297462096242584,0.1111111111111111,0.16666666666666666])  |
|(55,[2,3,5,8,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.03798615688859591,0.1111111111111111,0.16666666666666666]) |
|(55,[0,3,5,7,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.02356624917600527,0.1111111111111111,0.16666666666666666]) |
|(55,[0,3,6,7,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.025626235992089647,0.1111111111111111,0.16666666666666666])|
+------------------------------------------------------------------------------------------------------------+
only showing top 5 rows
```

We have defined all the necessary steps for feature extraction. Now let's create a Pipeline object that wraps all the feature extraction modules so that we can run the whole process end-to-end.

```
from pyspark.ml import Pipeline
features_pipeline = Pipeline(stages=[string_indexer, one_hot_encoder,
numeric_assembler, scaler, assembler])
features_extractor = features_pipeline.fit(df)

features_extractor.transform(df).select('features').show(5, truncate=False)
```

```
+-----------------------------------------------------------------------------------------------------------+
|features                                                                                                   |
+-----------------------------------------------------------------------------------------------------------+
|(55,[0,3,6,7,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.03180619644034278,0.2222222222222222,0.1666666666666666]) |
|(55,[0,3,6,8,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.0297462096242584,0.1111111111111111,0.1666666666666666])  |
|(55,[2,3,5,8,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.03798615688859591,0.1111111111111111,0.1666666666666666]) |
|(55,[0,3,5,7,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.02356624917600527,0.1111111111111111,0.1666666666666666]) |
|(55,[0,3,6,7,11,51,52,53],[1.0,1.0,1.0,1.0,1.0,0.025626235992089647,0.1111111111111111,0.1666666666666666])|
+-----------------------------------------------------------------------------------------------------------+
only showing top 5 rows
```

As can be seen, when we call *pipeline.fit*() with our dataframe *df*, it will create a pipeline instance which can be interpreted as a fitted model. Whenever we want to extract features from a dataset, we can simply call $features_extractor.transform()$, all the steps involved in feature extracting processes will be invoked sequentially.

3.4.2 Regression models

Now we are ready to build the model. We will build a Linear Regression model that receives the feature vector as input and predicts the house price.
First we create a Linear Regression object. We need to specify the *inputCol* argument, *outputCol* argument, leave other arguments with default values. Also, argument *standardization* is set to False because we have already normalized the data at feature extraction step.

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol='features', labelCol='Price',
predictionCol='prediction', standardization=False)
```

Next we split the dataframe into train set and test set with the ratio of 80:20:

```
train_df,test_df = df.randomSplit([0.8,0.2], seed=1234)
```

Now we train the model with *train_df* by calling *.fit*() function. This takes around some seconds:

```
lr_model = lr.fit(features_extractor.transform(train_df))
```

The function call *.fit*() returns a *LinearRegressionModel* object which is a fitted model. Now let's make house price predictions on test set.

```
lr_predictions = lr_model.transform(features_extractor.transform(test_df))
lr_predictions.select([col('Price').alias('Label'), 'prediction']).tail(10)
```

```
[Row(Label=65000000, prediction=70148124.0861545),
 Row(Label=75000000, prediction=66375174.60910439),
 Row(Label=60000000, prediction=66927229.47267903),
 Row(Label=70000000, prediction=71767909.68023005),
 Row(Label=22500000, prediction=70931740.53313239),
 Row(Label=58000000, prediction=79664587.48944594),
 Row(Label=79000000, prediction=86105578.58576645),
 Row(Label=115000000, prediction=84958451.79087318),
 Row(Label=135000000, prediction=103715003.59321827),
 Row(Label=170000000, prediction=164680586.85060284)]
```

As can be seen, the predictions results are not quite good. Perhaps Linear Regression model is too simple to find out complex patterns in our dataset.
Let's use the *RegressionEvaluator* class to evaluate the test set predictions with the ground truth labels. We choose 'r2' as the evaluation metric. The result is 0.8418.

```python
from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator(predictionCol='prediction',
labelCol='Price', metricName='r2')
r2_score = evaluator.evaluate(lr_predictions, {evaluator.metricName: "r2"})
print("Model r2 score on test set: ", r2_score)
```

```
Model r2 score on test set:  0.8418146788328705
```

Now we try to fit the dataset to *RandomForestRegressor*, using the similar steps:

```python
# define the model
from pyspark.ml.regression import RandomForestRegressor
rf = RandomForestRegressor(featuresCol='features', labelCol='Price',
predictionCol='prediction')

# train the model on train set
rf_model = rf.fit(features_extractor.transform(train_df))

# make predictions on test set
rf_predictions = rf_model.transform(features_extractor.transform(test_df))

# Evaluate
rf_evaluator = RegressionEvaluator(predictionCol='prediction',
labelCol='Price', metricName='r2')
r2_score = rf_evaluator.evaluate(rf_predictions, {evaluator.metricName:
"r2"})
print("Model r2 score on test set: ", r2_score)
```

```
Model r2 score on test set:  0.8658623345025791
```

Random Forest algorithm provides a better test set performance with $r2 = 0.8659$

3.4.3 Classification models

To setup our experiment for classification models, we split house prices into three categories: Low, Medium and High based on Q1 and Q3 values:

```
from pyspark.sql.functions import mean
# mean_price = df.select(mean(df['Price'])).collect()[0][0]
q1, q3 = df.approxQuantile("Price", [0.25, 0.75], 0)
print("Q1 value: ", q1)
print("Q3 value: ", q3)
```

```
Q1 value:  5800000.0
Q3 value:  25900000.0
```

From the calculated Quantiles, Low house price is from 0 to 5.800.000, medium house price is from 5.800.000 to 25.900.000 and high house price is above 25.900.000.
We create a new column *price_category* and assign value 0 for houses with low price, 1 for medium price and 2 for high price.

```
def price_segmentation(x):
    #Low
    if x < 5800000:
        return 0.0
    #Medium
    elif 5800000 <= x <= 25900000:
        return 1.0
    #High
    else:
        return 2.0
df = df.withColumn('price_category', udf(price_segmentation,
FloatType())(df['Price']))
```

```
+--------------+-----+
|price_category|count|
+--------------+-----+
|           2.0|  308|
|           1.0|  619|
|           0.0|  308|
+--------------+-----+
```

Next we split the data to train set and test set, and fit a features extractor as we did for Regression.
Let's now build a Logistic Regression model to classify house price:

```
# Define the model
from pyspark.ml.classification import LogisticRegression
log_r = LogisticRegression(featuresCol='features',
labelCol='price_category', predictionCol='prediction')

# fit the model with train set
log_r_model = log_r.fit(features_extractor.transform(train_df))

# make predictions on test set
log_r_predictions =
log_r_model.transform(features_extractor.transform(test_df))
```

```
log_r_predictions.select(col('price_category').alias('label'),
'prediction').show(10)
```

```
+-----+----------+
|label|prediction|
+-----+----------+
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  1.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       1.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
+-----+----------+
only showing top 10 rows
```

Then we evaluate the model performance on test set, the default metric is F1 score:

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
log_r_evaluator =
BinaryClassificationEvaluator(rawPredictionCol='prediction',
labelCol='price_category')
f1_score = log_r_evaluator.evaluate(log_r_predictions)
print("Model F1 score on test set: ", f1_score)
```

```
Model F1 score on test set:  0.8965960179833012
```

The F1 score of Logistic Regression model is 0.897.

3.4.4 Clustering

First group numeric values in a row into a numeric vector and then scale the data.

```
# Group numeric values in a row into a numeric vector
numeric_assembler =
VectorAssembler(inputCols=['Area','BHK','Bathroom','Parking','Price'],
outputCol=f'numeric_vec')
assembled = numeric_assembler.transform(df)

# Scale and standardize data by using StandardScalerModel
scaler = StandardScaler(inputCol='numeric_vec', outputCol='features')
data = scaler.fit(assembled).transform(assembled)
```

```
+--------------------------------------------------------------------------------------------+
|features                                                                                    |
+--------------------------------------------------------------------------------------------+
|[0.8364949736732389,3.172635307548206,1.9377970190850902,1.3115518730635414,0.25545190924496297] |
|[0.7842140378186614,2.1150902050321374,1.9377970190850902,1.3115518730635414,0.1965014686499715] |
|[0.9933377812369711,2.1150902050321374,1.9377970190850902,1.3115518730635414,0.6091545528149117] |
|[0.6273712302549291,2.1150902050321374,1.9377970190850902,1.3115518730635414,0.16506123366597605]|
|[0.6796521661095066,2.1150902050321374,1.9377970190850902,1.3115518730635414,0.24366182112596466]|
+--------------------------------------------------------------------------------------------+
only showing top 5 rows
```

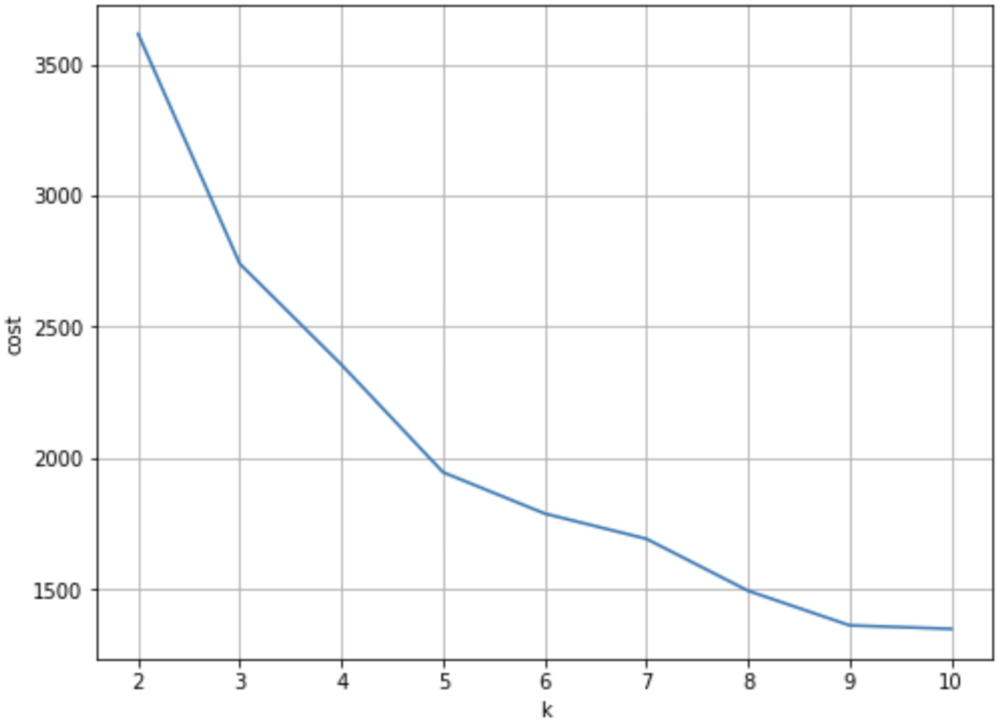
To get the number of clusters, we use two popular algorithms, Elbow Method and Silhouette Coefficient. Start with the Elbow method, we run k from 2 to 10:

```
from pyspark.ml.clustering import KMeans

no_cluster = 10;
cost = [0] * (no_cluster + 1)
for k in range(2, no_cluster + 1):
    kmeans = KMeans()\
            .setK(k)\
            .setSeed(1) \
            .setFeaturesCol("features")\
            .setPredictionCol("cluster")

    model = kmeans.fit(data)
    cost[k] = model.summary.trainingCost
```

```
Cost k=2: 3615.398839547335
Cost k=3: 2739.4321740042615
Cost k=4: 2354.9072178305487
Cost k=5: 1944.586573608512
Cost k=6: 1787.273487911017
Cost k=7: 1691.150433884
Cost k=8: 1493.0380225139263
Cost k=9: 1361.165413357026
Cost k=10: 1347.7228750929855
```

We plot a diagram to for visualization:

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

fig, ax = plt.subplots(1,1, figsize =(8,6))
ax.plot(range(2, no_cluster + 1), cost[2:no_cluster + 1])
ax.set_xlabel('k')
ax.set_ylabel('cost')
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.grid()
plt.show()
```



From the above diagram, we can actually take k = 3, k = 5 or k = 8.
For Silhouette Score, we implement an `optimal_k` function, which is defined as follows:

- Input: dataset, indexed feature column name, min number k, max number k, number of runs.
- Output: k and its Silhouette score respectively.

```
import time
import numpy as np
from pyspark.ml.evaluation import ClusteringEvaluator

def optimal_k(df_in,index_col,k_min, k_max,num_runs):
    start = time.time()
    silh_lst = []
    k_lst = range(k_min, k_max + 1)

    for k in k_lst:
        silh_val = []
        for run in range(1, num_runs+1):

            # Trains a k-means model.
            kmeans = KMeans()\
                    .setK(k)\
```

```
                      .setFeaturesCol(index_col)\
                      .setSeed(123)
              model = kmeans.fit(df_in)

              # Make predictions
              predictions = model.transform(df_in)

              # Evaluate clustering by computing Silhouette score
              evaluator = ClusteringEvaluator()
              silhouette = evaluator.evaluate(predictions)
              silh_val.append(silhouette)

          # Take average
          silh_array=np.asanyarray(silh_val)
#           print("k =", k, silh_array.mean())
          silh_lst.append(float(silh_array.mean()))

      elapsed =  time.time() - start

      silhouette = spark.createDataFrame(list(zip(k_lst, silh_lst))).toDF('k', 'silhouette')

      print('+-----------------------------------------------------------+')
      print("|         The finding optimal k phase took %8.0f s.       |" %(elapsed))
      print('+-----------------------------------------------------------+')

      return silhouette
```

We run k from 2 to 10 and draw a graph:

```
# Getting the optimal number of clusters by finding silhouette coefficients
# Input: dataset, indexed feature column name, min k, max k, number of runs
# Output: a list of silhouette scores from k = 2 to k = 10 by running each k value 1 time.

silh_lst = optimal_k(data, 'features', 2, 10, 1)
```

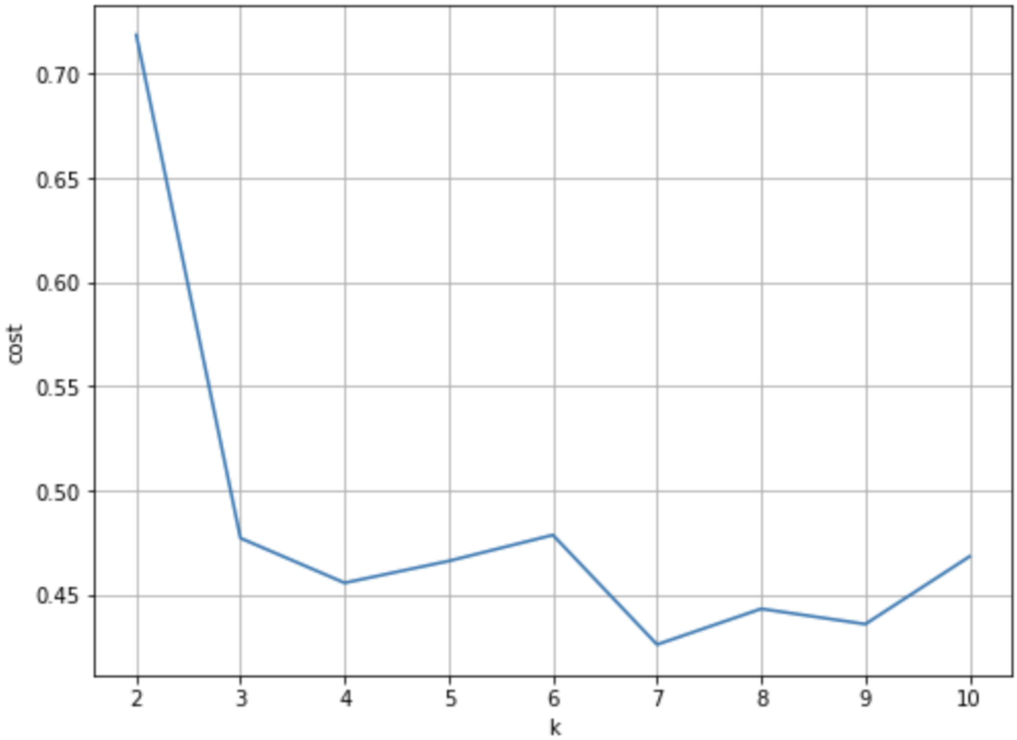
```
+-----------------------------------------------------------+
|         The finding optimal k phase took       15 s.       |
+-----------------------------------------------------------+
```

```
silh_lst.show()
```

```
+---+-------------------+
|  k|         silhouette|
+---+-------------------+
|  2| 0.7185841189832898|
|  3|0.47715336269377456|
|  4|   0.45568933596496|
|  5| 0.4661745636762831|
|  6|0.47865964109478376|
|  7| 0.4260958911337515|
|  8|0.44324351309237287|
|  9|0.43589965375885587|
| 10|0.46832343647036445|
+---+-------------------+
```

Plotting for visualization:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1,1, figsize =(8,6))
ax.plot(range(2, no_cluster + 1), [x['silhouette'] for x in
silh_lst.collect()])

ax.set_xlabel('k')
ax.set_ylabel('cost')
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.grid()
plt.show()
```



We decided based on the Silhouette score plot and chose the number of clusters k = 6 as the optimal k. The reason is that at there is a peak in Silhouette score at k = 6. Notice that here we create a Pipeline and select the region from the Vector transform and the K-means model.

```
kmeans = KMeans() \
          .setK(8) \
          .setFeaturesCol("features")\
          .setPredictionCol("cluster")

# Chain indexer and tree in a Pipeline
# Get stage from transformer and K-Means
pipeline = Pipeline(stages=[numeric_assembler , kmeans])
pre_clustered_data = data.drop('numeric_vec')
model = pipeline.fit(pre_clustered_data)
clustered_data = model.transform(pre_clustered_data)
```

```
clustered_data.show(10)
```

Area	BHK	Bathroom	Furnishing	Parking	Price	Status	Transaction	Type	Locality	price_category	features	numeric_vec	cluster
800.0	3	2	Semi-Furnished	1	6500000	Ready_to_move	New_Property	Builder_Floor	Rohini	1.0	[0.83649497367323...	[800.0,3.0,2.0,1....	1
750.0	2	2	Semi-Furnished	1	5000000	Ready_to_move	New_Property	Apartment	Rohini	0.0	[0.78421403781866...	[750.0,2.0,2.0,1....	1
950.0	2	2	Furnished	1	15500000	Ready_to_move	Resale	Apartment	Rohini	1.0	[0.99333778123697...	[950.0,2.0,2.0,1....	1
600.0	2	2	Semi-Furnished	1	4200000	Ready_to_move	Resale	Builder_Floor	Rohini	0.0	[0.62737123025492...	[600.0,2.0,2.0,1....	1
650.0	2	2	Semi-Furnished	1	6200000	Ready_to_move	New_Property	Builder_Floor	Rohini	1.0	[0.67965216610950...	[650.0,2.0,2.0,1....	1
1300.0	4	3	Semi-Furnished	1	15500000	Ready_to_move	New_Property	Builder_Floor	Rohini	1.0	[1.35930433221901...	[1300.0,4.0,3.0,1...	4
1350.0	4	3	Semi-Furnished	1	10000000	Ready_to_move	Resale	Builder_Floor	Rohini	1.0	[1.41158526807359...	[1350.0,4.0,3.0,1...	4
650.0	2	2	Semi-Furnished	1	4000000	Ready_to_move	New_Property	Apartment	Rohini	0.0	[0.67965216610950...	[650.0,2.0,2.0,1....	1
985.0	3	3	Unfurnished	1	6800000	Almost_ready	New_Property	Builder_Floor	Rohini	1.0	[1.02993443633517...	[985.0,3.0,3.0,1....	6
1300.0	4	4	Semi-Furnished	1	15000000	Ready_to_move	New_Property	Builder_Floor	Rohini	1.0	[1.35930433221901...	[1300.0,4.0,4.0,1...	4

only showing top 10 rows

3.5 PySpark MySQL with MySQL database

In this subsection we play with some functionalities of PySpark MySQL module. In particular, we use PySpark to write data from a json file stored on Hadoop to a MySQL database. To do that, first we need to create the corresponding database and table, named `bigdata` and `user` respectively.

```
~/Doc/B/22/P/data-stack  main  docker exec -it db bash          system  22:59:58
root@docker-desktop:/# mysql -u root -plocal
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.22 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE IF NOT EXISTS bigdata;
;
CREATE TABLE IF NOT EXISTS user (
    name VARCHAR(255),
    age INT
);
Query OK, 1 row affected, 1 warning (0.02 sec)

mysql> USE bigdata;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> CREATE TABLE IF NOT EXISTS user (
    ->     name VARCHAR(255),
    ->     age INT
    -> );
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> 
```

`test.json`:

```json
[
    {
        "name": "Tran Van A",
        "age": 99
    },
    {
        "name": "Nguyen Thi B",
        "age": 66
    },
    {
        "name": "Le Van C",
```

```
        "age": 16
    },
    {
        "name": "Ly Thi D",
        "age": 19
    }
]
```

After we read and write the data, we can check if the data has been saved to the MySQL database:

```
jdbcDF.show()
```

```
+------------+---+
|        name|age|
+------------+---+
|  Tran Van A| 99|
|Nguyen Thi B| 66|
|    Le Van C| 16|
|    Ly Thi D| 19|
|  Tran Van A| 99|
|Nguyen Thi B| 66|
|    Le Van C| 16|
|    Ly Thi D| 19|
|  Tran Van A| 99|
|Nguyen Thi B| 66|
|    Le Van C| 16|
|    Ly Thi D| 19|
+------------+---+
```

```
mysql> USE bigdata;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> CREATE TABLE IF NOT EXISTS user (
    ->     name VARCHAR(255),
    ->     age INT
    -> );
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SELECT * from user;
+--------------+------+
| name         | age  |
+--------------+------+
| Tran Van A   |   99 |
| Nguyen Thi B |   66 |
| Le Van C     |   16 |
| Ly Thi D     |   19 |
| Tran Van A   |   99 |
| Nguyen Thi B |   66 |
| Le Van C     |   16 |
| Ly Thi D     |   19 |
| Tran Van A   |   99 |
| Nguyen Thi B |   66 |
| Le Van C     |   16 |
| Ly Thi D     |   19 |
+--------------+------+
12 rows in set (0.01 sec)

mysql>
```

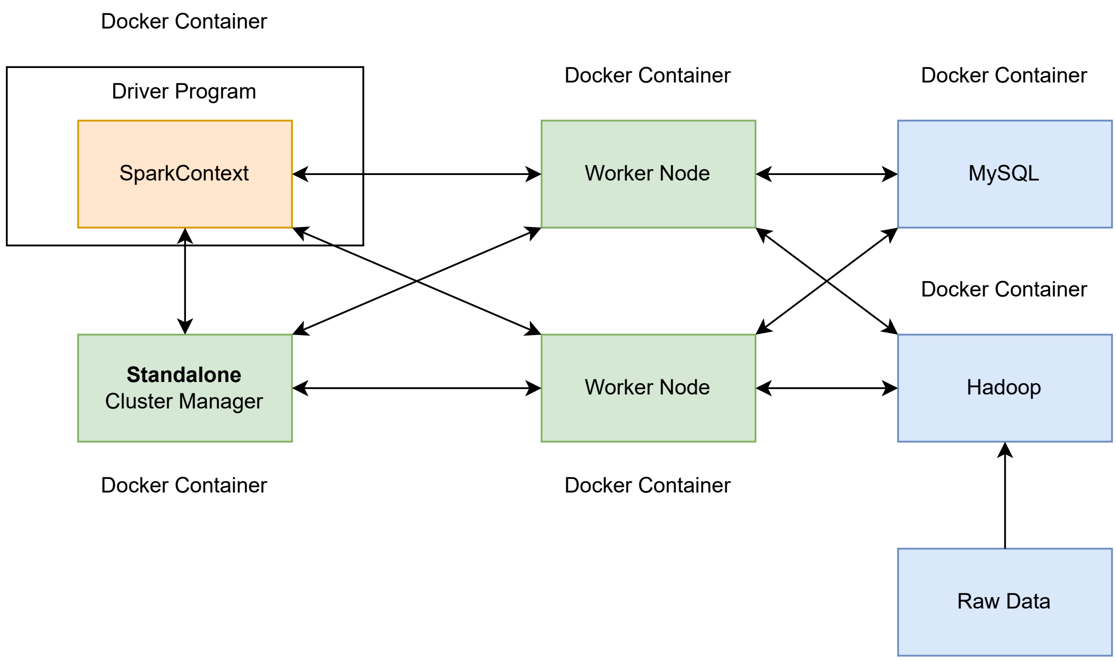
4 Further Study: Data Stack with Hadoop and Spark

In this subsection, we set up a data stack using Apache Spark standalone cluster with Docker containers. This is a further study from our work on PySpark Data Analytics. For more details, you can find this work here.

4.1 Our System

In this study, we use *Docker*[3] because we want to give this system the ability to run independently. That means it can run across any operating system platform, thanks to Docker's capabilities. Docker containers provide a way to package applications with everything needed to run them, including base operating system images, databases, libraries, and binaries. By running a Docker engine on a host machine, Docker containers interact solely with the kernel of the host OS, meaning all containerized apps function the same regardless of the underlying infrastructure. Furthermore, running multiple apps on a single host machine, which leads to impressive cost savings by letting enterprises run more apps on existing hardware. Docker ties into and helps to handle Big Data sets which are fast-moving, voluminous, and contain a huge variety of information from disparate sources and in different formats.

We have set up a data stack whose architecture is shown as shown below:



Explain the components included in the architecture:

- **Driver Program**
  This is where the main program receives requests from the user (client). For ease of visualization, we run through a Jupyter notebook which can connect via http://localhost:8888.
- **Cluster Manager - Spark Master**
  This container is used to run a Spark Master node which is used to configure and manage Spark Workers. Spark Master UI can be accessed at http://localhost:8080/.
- **Spark Worker**
  We set up 2 containers to hold 2 Spark Worker nodes which are used to execute jobs or tasks sent from `SparkContext`. We configure each Spark Worker has `SPARK_WORKER_CORES=1` and `SPARK_WORKER_MEMORY=512m`. Spark Worker 1 UI can be accessed at http://localhost:8081 and Spark Worker 2 UI at http://localhost:8082.

- **Hadoop**
  This is the main raw data (data lake) container in our system. The UI can be accessed a at http://localhost:50070.

- **MySQL**
  We use a container to build an SQL database, the reason is that PySpark is very powerful in interacting and handling structured data. After processing the data, users can save the results by writing data to MySQL, note that reading and writing data is only performed in the subnet of Docker containers.

- **Raw Data**
  This is where data is gathered at many different data sources before being transferred to the Hadoop file management system, in this mini project, for simplicity we consider this to be the local computer running this system.

4.2 Running our Data Stack

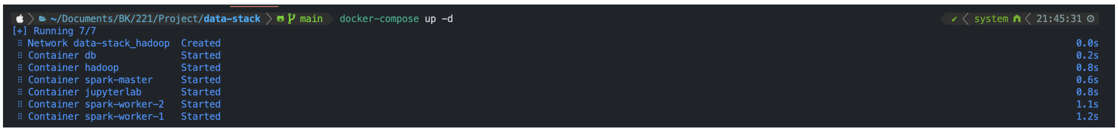


Figure 1: Run our system through Docker

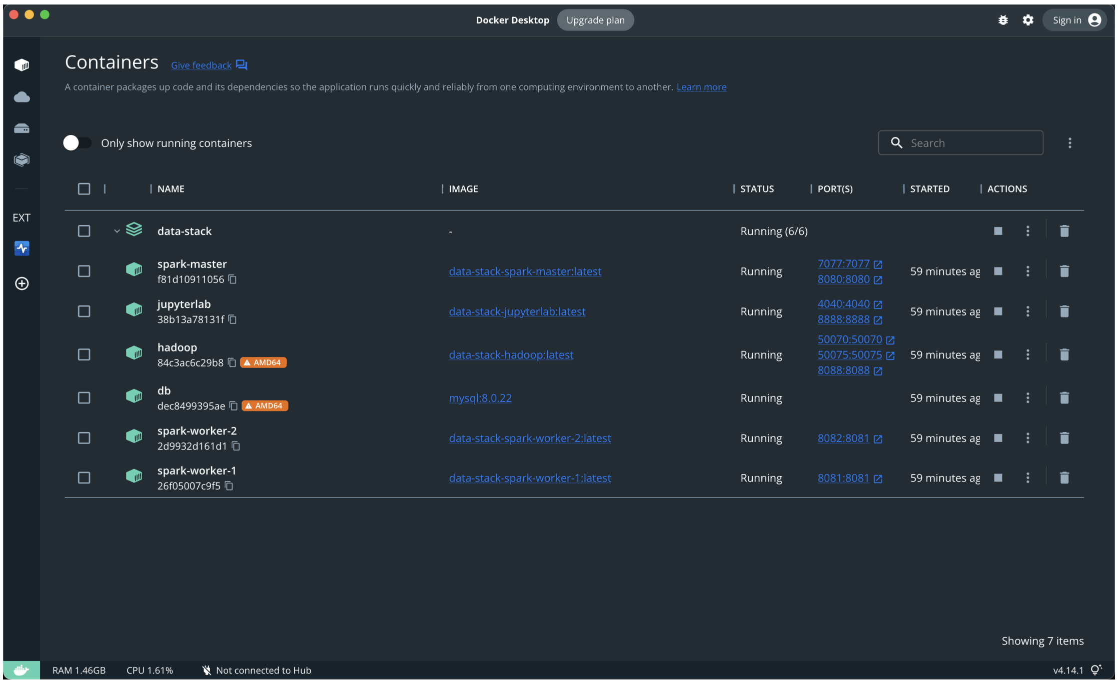


Figure 2: Docker UI

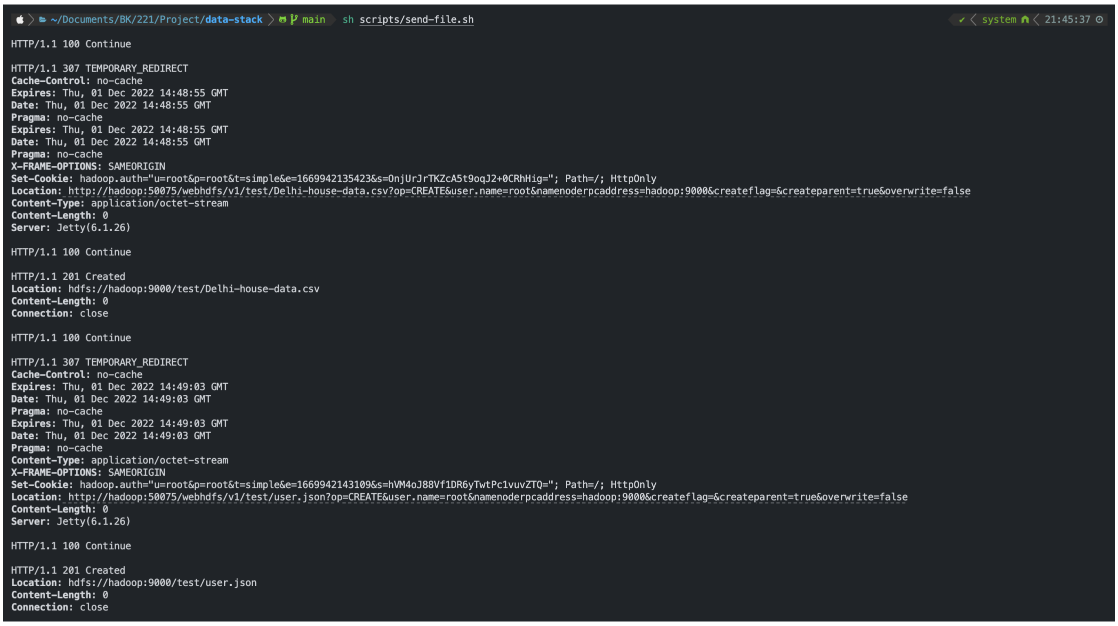


Figure 3: Send raw data to Hadoop

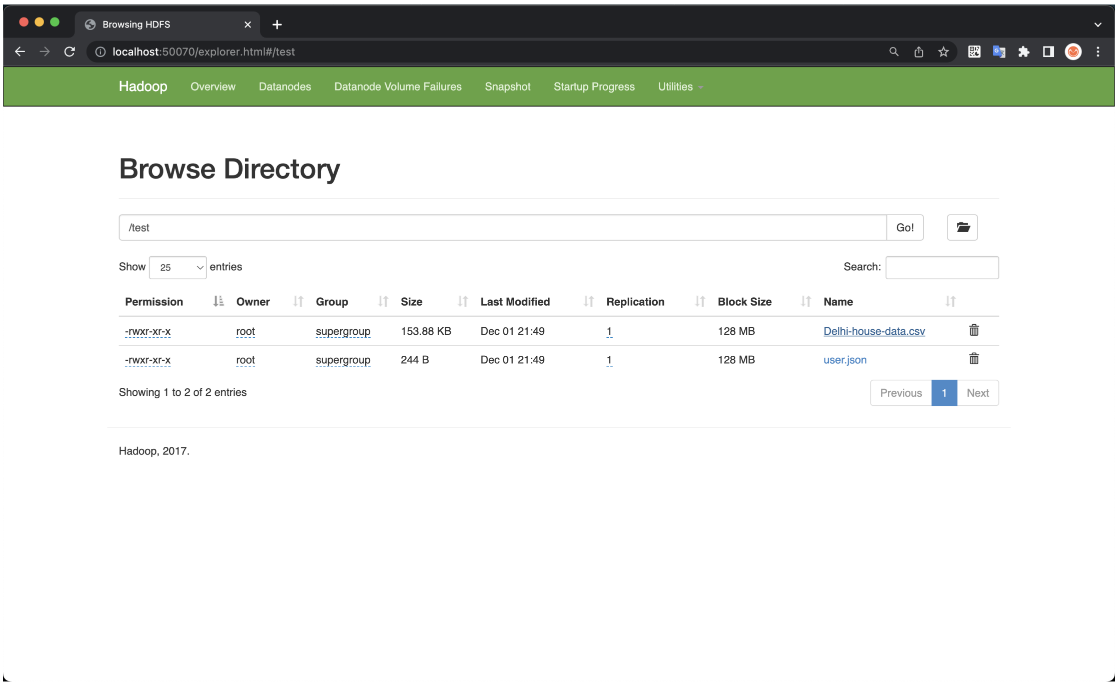


Figure 4: Check data file on Hadoop UI

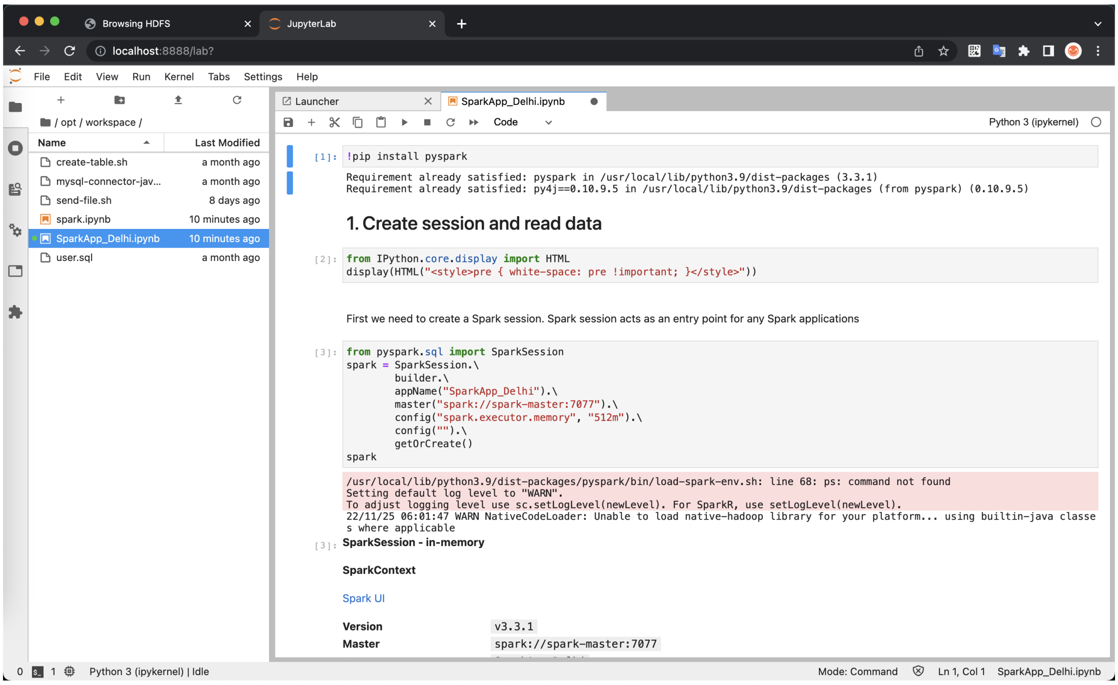


Figure 5: Client UI - Jupyter notebook

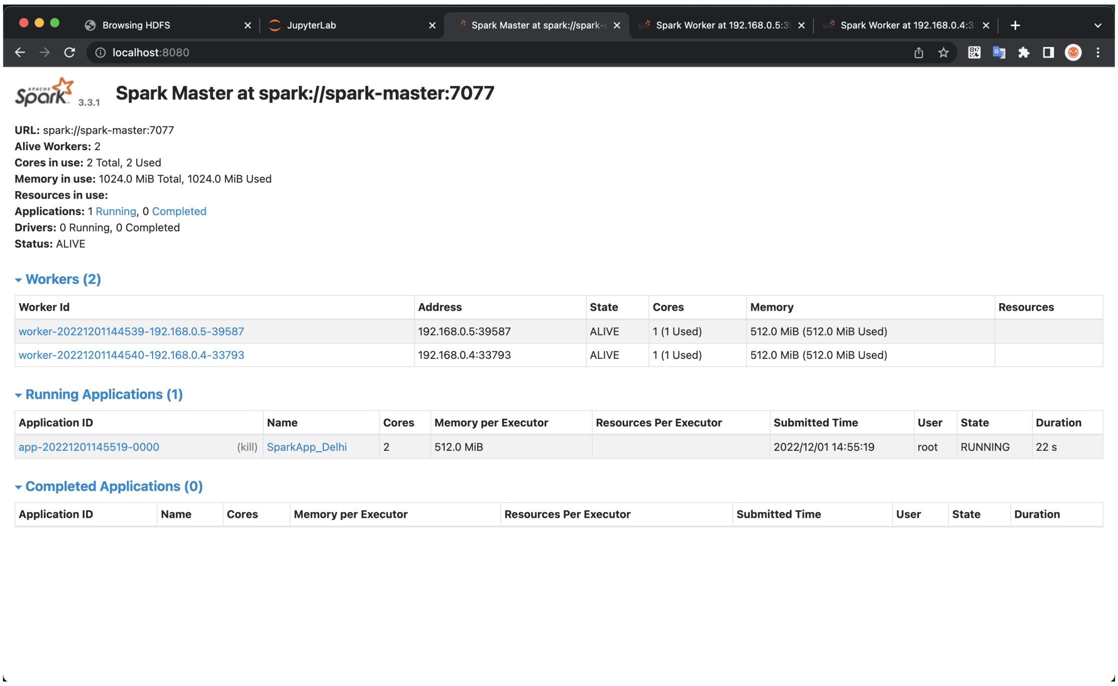


Figure 6: Spark Master UI with 2 Worker nodes

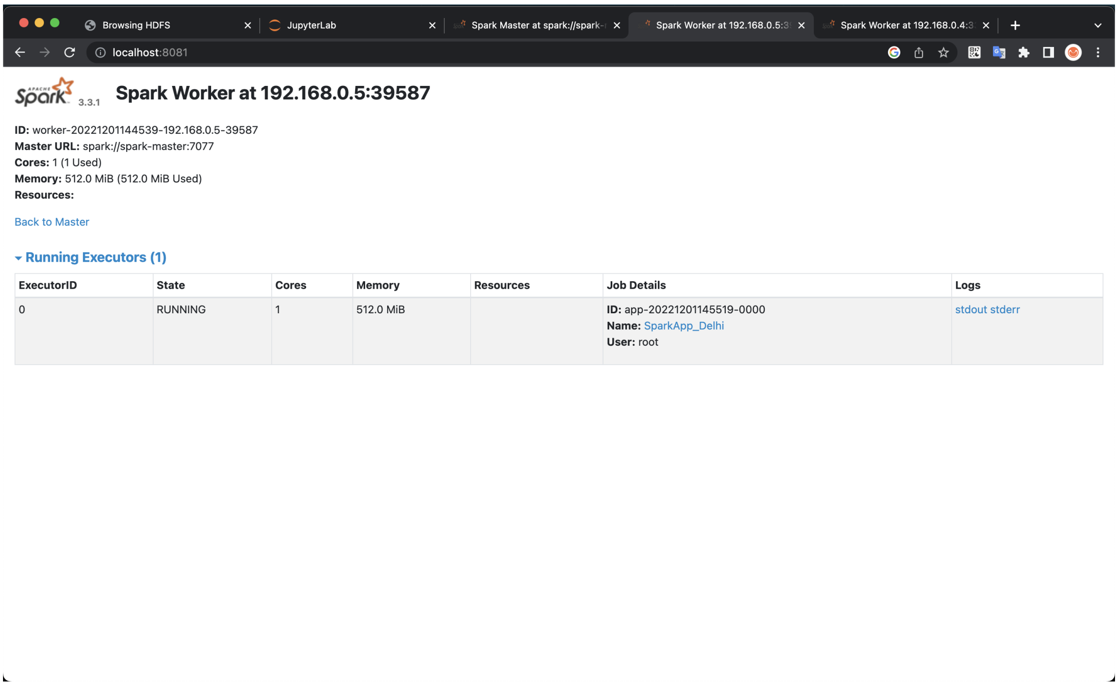


Figure 7: Spark Worker 1 UI

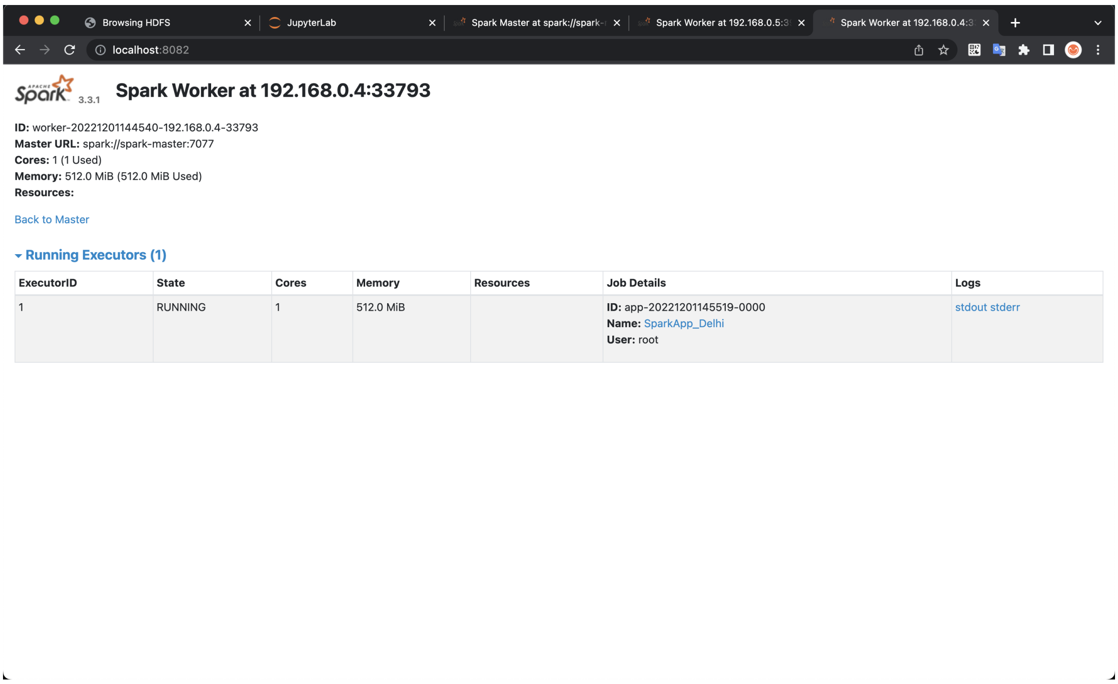


Figure 8: Spark Worker 2 UI

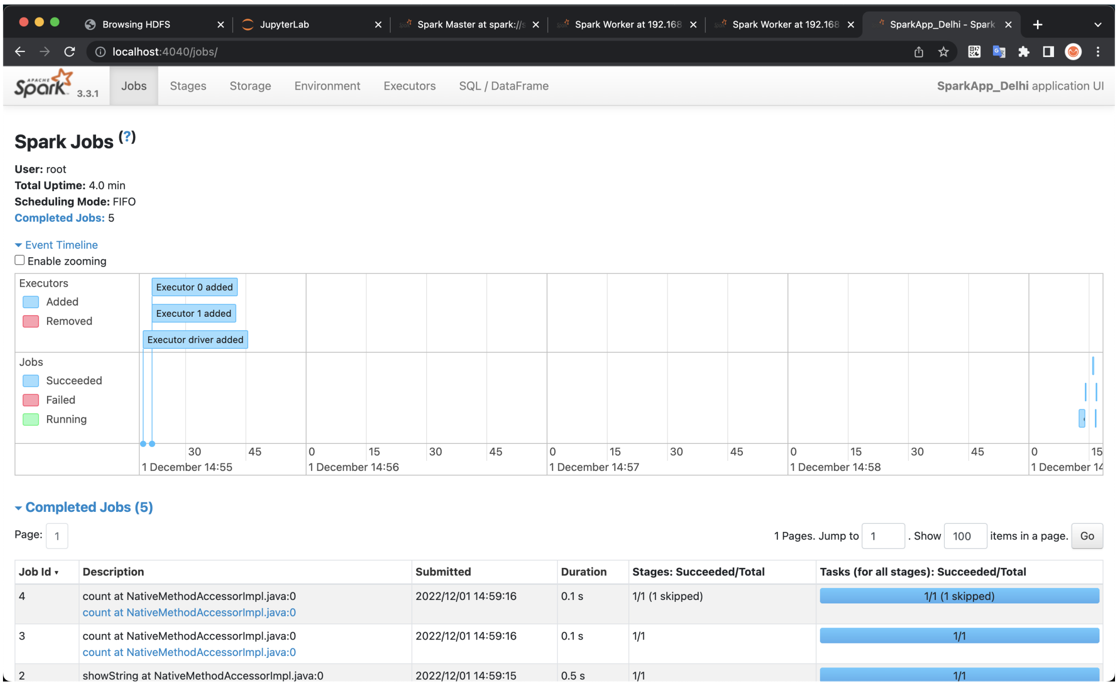


Figure 9: Spark UI

5 Conclusion

In this project, we have experimented with Apache Spark functionalities and applied them to work on the Indian House Price dataset. We believed Apache Spark is a great tool and should be a prior choice for businesses for distributed data analytics. We also conducted a further study on creating a small Data Stack that runs on docker.

References

[1] *Apache Spark*. URL: https://spark.apache.org/ (visited on 12/09/2022).

[2] *Big data*. URL: https://en.wikipedia.org/wiki/Big_data (visited on 12/09/2022).

[3] *Docker*. URL: https://www.docker.com/ (visited on 12/09/2022).

[4] *MapReduce*. URL: https://en.wikipedia.org/wiki/MapReduce (visited on 12/09/2022).

[5] *PySpark*. URL: https://spark.apache.org/docs/latest/api/python/ (visited on 12/09/2022).