Conception d'un modéle de prédiction des prix de maisons

Dataset : houses.csv

Par: Kamgaing Rodrigue Ulrich et Ngougoue Djeufa Emmanuella

Option: ICC

Télechargement des pré-requis

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://archive.apache.org/dist/spark/spark-3.1.1/spark-3.1.1-bin-hadoop
!tar xf spark-3.1.1-bin-hadoop3.2.tgz
!pip install -q findspark

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.1.1-bin-hadoop3.2"

import findspark
findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
spark.conf.set("spark.sql.repl.eagerEval.enabled", True) # Property used to form
spark
```

```
import pyspark
from pyspark.sql import SparkSession, functions

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline
from pyspark.sql.functions import col
from pyspark.sql import functions as F
from pyspark.sql.functions import when, col
```

Import des données Houses.csv

```
from google.colab import files
files.upload()
```

Browse... No files selected. Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable

Exploration des données Houses.csv

- Affichage du **nombre de datas total** à étudier dans le dataset ;
- Affichage du **nombre de lignes et colones** du dataset ;
- Affichage des **noms de colones** du dataset ;
- Affichage du **shéma de données** ;
- Affichage du **type de données** ;
- Affichage de la **description de chaque élément de colones** du dataset ;
- Affichage de **10 lignes du dataset house.csv**;

```
spark = SparkSession.builder.master('local[*]').appName('houses').getOrCreate()

# Read data from CSV file
houses = spark.read.csv('house.csv',inferSchema=True, header =True, nullValue='

# Count the number of rows
num_rows = houses.count()

# Count the number of columns
num_cols = len(houses.columns)

# Display basic information about the dataset
print("Size of records : ", houses.count())
print("Shape of the dataset: {} rows, {} columns".format(num_rows, num_cols))
print("\nColumns in the dataset:", houses.columns)
print("\nColumns in the dataset:\n", houses.printSchema())
print("\nData types of columns:\n", houses.dtypes)
print("\nSummary statistics of numerical columns:\n", houses.describe())

# View the first 10 five records
houses.show(10)
```

```
Size of records :  21613
Shape of the dataset: 21613 rows, 21 columns

Columns in the dataset: ['id', 'date', 'price', 'bedrooms', 'bathrooms', 's
root
 |-- id: long (nullable = true)
 |-- date: string (nullable = true)
```

```
 |-- price: double (nullable = true)
 |-- bedrooms: integer (nullable = true)
 |-- bathrooms: double (nullable = true)
 |-- sqft_living: integer (nullable = true)
 |-- sqft_lot: integer (nullable = true)
 |-- floors: double (nullable = true)
 |-- waterfront: integer (nullable = true)
 |-- view: integer (nullable = true)
 |-- condition: integer (nullable = true)
 |-- grade: integer (nullable = true)
 |-- sqft_above: integer (nullable = true)
 |-- sqft_basement: integer (nullable = true)
 |-- yr_built: integer (nullable = true)
 |-- yr_renovated: integer (nullable = true)
 |-- zipcode: integer (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
 |-- sqft_living15: integer (nullable = true)
 |-- sqft_lot15: integer (nullable = true)


Columns in the dataset:
 None

Data types of columns:
 [('id', 'bigint'), ('date', 'string'), ('price', 'double'), ('bedrooms', '

Summary statistics of numerical columns:
 +-------+--------------------+---------------+------------------+---------
|summary|                  id|           date|             price|        b
+-------+--------------------+---------------+------------------+----------
|  count|               21613|          21613|             21613|
|   mean| 4.580301520864988E9|           null| 540088.1417665294| 3.3708416
| stddev|2.8765655713120522E9|           null|367127.19648270035|0.93006183
|    min|             1000102|20140502T000000|           75000.0|
|    max|          9900000190|20150527T000000|         7700000.0|
+-------+--------------------+---------------+------------------+----------

+----------+---------------+---------+--------+---------+-----------+------
|        id|           date|    price|bedrooms|bathrooms|sqft_living|sqft_l
+----------+---------------+---------+--------+---------+-----------+------
|7129300520|20141013T000000| 221900.0|       3|      1.0|       1180|    56
|6414100192|20141209T000000| 538000.0|       3|     2.25|       2570|    72
|5631500400|20150225T000000| 180000.0|       2|      1.0|        770|   100
|2487200875|20141209T000000| 604000.0|       4|      3.0|       1960|    50
|1954400510|20150218T000000| 510000.0|       3|      2.0|       1680|    80
|7237550310|20140512T000000|1225000.0|       4|      4.5|       5420|  1019
|1321400060|20140627T000000| 257500.0|       3|     2.25|       1715|    68
|2008000270|20150115T000000| 291850.0|       3|      1.5|       1060|    97
|2414600126|20150415T000000| 229500.0|       3|      1.0|       1780|    74
|3793500160|20150312T000000| 323000.0|       3|      2.5|       1890|    65
```

Visualisation du dataset

- Création de geospatial_visualization.html pour éventuellement visualiser les éléments sur une carte.

- Visualisation à travers des graphes précédés d'une interprétation

```
# Geospatial Visualization
# Create maps because the dataset includes location information

import geopandas as gpd
import folium
from folium.plugins import MarkerCluster
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Assuming we have a column named 'lat' and 'long' in our PySpark DataFrame
latitude_column = "lat"
longitude_column = "long"

# Select relevant columns
location_data = houses.select("id", latitude_column, longitude_column)

# Convert PySpark DataFrame to Pandas DataFrame
location_df = location_data.toPandas()

# Create a GeoDataFrame using geopandas
gdf = gpd.GeoDataFrame(location_df, geometry=gpd.points_from_xy(location_df[lor

# Create a folium map centered around the mean coordinates
center_lat, center_lon = location_df[latitude_column].mean(), location_df[longi
mymap = folium.Map(location=[center_lat, center_lon], zoom_start=10)

# Add a marker cluster to the map
marker_cluster = MarkerCluster().add_to(mymap)

# Add markers to the marker cluster
for i in range(len(gdf)):
    folium.Marker([gdf.iloc[i][latitude_column], gdf.iloc[i][longitude_column]]

# Save the map as an HTML file or display it
mymap.save("geospatial_visualization.html")
```

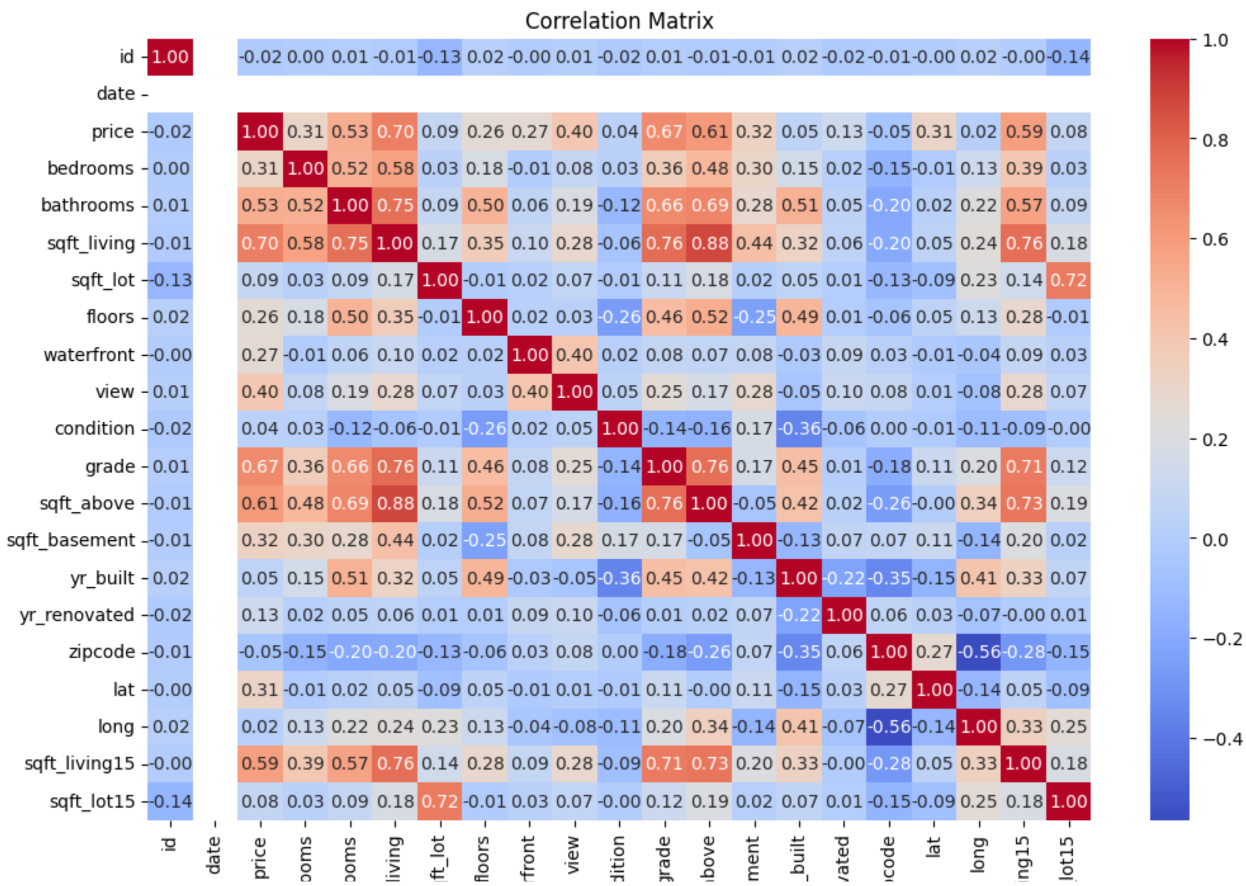
Une visualisation de l'ensemble des valeurs de prix nous permet de constater que les valeurs très élevées sont les moins présentes dans notre dataset. Le model d'entraînement pourrait donc avoir un peu plus de mal à les prédire avec exactitude.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming 'houses' is your DataFrame
spark = SparkSession.builder.master('local[*]').appName('houses').getOrCreate()
```

```
# Read data from CSV file
houses = spark.read.csv('house.csv',inferSchema=True, header =True, nullValue='

# Convert PySpark DataFrame to Pandas DataFrame
houses_pd = houses.toPandas()

# Histogram of the target variable (price) to visualize its distribution
plt.figure(figsize=(10, 6))
sns.histplot(houses_pd['price'], bins=30, kde=True)
plt.title('Distribution of Prices')
plt.show()
```

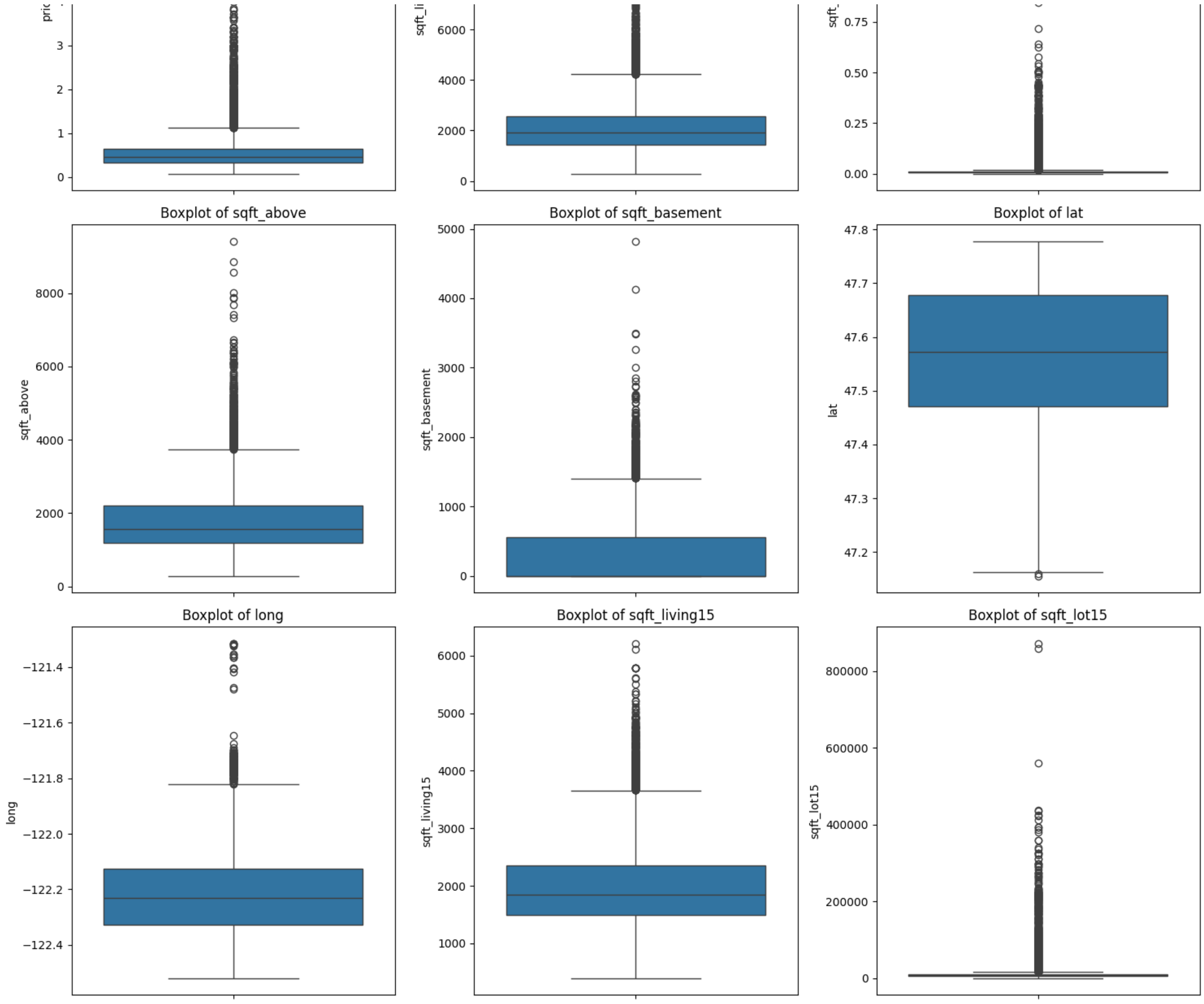
Néamoins, d'apres la matrice de corrélation, on peut constater que certaines variables par rapport à d'autres, sont assez fortement corrélées au prix comme (que l'on va considérer comme importtantes) : **bedrooms, bathrooms, sqft_living, floors, waterfront, view, grade, sqft_above, sqft_basement, lat, sqft_living15**

```
# Step 7: Correlation Matrix
correlation_matrix = houses.select([col(c).cast("float") for c in houses.columr
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix')
plt.show()
```



Cependant, les boxplots, montrent qu'il existe bien des valeurs abérantes.

```
# Define some others continuous variables
continuous_vars = ['price', 'sqft_living', 'sqft_lot', 'sqft_above', 'sqft_base
                     'lat', 'long', 'sqft_living15', 'sqft_lot15']

# Calculate grid dimensions
num_plots = len(continuous_vars)
grid_shape = (3, 3)

# Create subplots
fig, axes = plt.subplots(grid_shape[0], grid_shape[1], figsize=(15, 15))

# Flatten axes for easier indexing
axes = axes.flatten()

# Plot boxplots for continuous variables
for i, var in enumerate(continuous_vars):
    sns.boxplot(data=houses_pd, y=var, ax=axes[i])
    axes[i].set_title(f'Boxplot of {var}')
    axes[i].set_ylabel(var)

# Adjust layout
plt.tight_layout()
plt.show()
```


Boxplot of sqft_above
Boxplot of sqft_basement
Boxplot of lat
Boxplot of long
Boxplot of sqft_living15
Boxplot of sqft_lot15
sqft_above
sqft_basement
lat
long
sqft_living15
sqft_lot15

C'est la raison pour laquelle nous procédons à un PRÉ-TRAITEMENT des données

- On commence par verrifier qu'il n'ya pas de valeurs manquantes dans le dataset;
- Nous allons standardiser les données car la plage de valeurs minimales et maximales est assez large (démontré par le premier graphique)
- Nous allons ajouter des attributs supplémentaires. Notre variable dépendante étant également assez grande, nous allons ajuster légèrement les valeurs.

1 - Verrifier qu'il n'ya pas de valeurs manquantes dans le dataset

```
# Check the null values in dataset
# Count missing values in each column
missing_values = houses.toPandas().isna().sum()

# Print missing values
print("les valeurs manquantes:")
print(missing_values)
```

```
les valeurs manquantes:
id               0
date             0
price            0
bedrooms         0
bathrooms        0
```

```
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

OK Pas de vides !

2 - Continuons par le prix (price) , notre variable dépendante. Pour faciliter notre travail avec les valeurs cibles, nous exprimerons les valeurs de la maison en unités de 100 000. Cela signifie qu'un objectif tel que 452600,000000 devrait devenir 4,526 :

```
# Adjustement des valeurs de `price`
houses = houses.withColumn("price", col("price")/100000)

# Show the first 2 lines of `df`
houses.show(2)
```

```
+----------+---------------+-----+--------+---------+-----------+--------+-
|        id|           date|price|bedrooms|bathrooms|sqft_living|sqft_lot|f
+----------+---------------+-----+--------+---------+-----------+--------+-
|7129300520|20141013T000000|2.219|       3|      1.0|       1180|    5650|
|6414100192|20141209T000000| 5.38|       3|     2.25|       2570|    7242|
+----------+---------------+-----+--------+---------+-----------+--------+-
only showing top 2 rows
```

3 - Ajout de nouvelles colonnes

```
from datetime import datetime
```

```
# Get the current year
current_year = datetime.now().year

# Print the current year
print("Current Year:", current_year)


# Ajout de nouvelles colonnes dans le DataFrame

# Calculate total number of bathrooms
houses = houses.withColumn("total_bathrooms", houses["bathrooms"] + houses["bec

houses = houses.withColumn("house_age", current_year - houses["yr_built"])

# Add a column indicating whether the house has been renovated
houses = houses.withColumn("renovated", F.when(houses["yr_renovated"] > 0, 1).c

# Calculate the ratio of living space to lot space for the subject property
houses = houses.withColumn("living_to_lot_ratio", houses["sqft_living"] / house

# Calculate the average living space per neighbor
houses = houses.withColumn("avg_living_space_per_neighbor", houses["sqft_living

# Calculate the average lot space per neighbor
houses = houses.withColumn("avg_lot_space_per_neighbor", houses["sqft_lot15"] /

# Show the updated DataFrame with new columns
houses.show(5)
```

```
Current Year: 2024
+----------+---------------+-----+--------+---------+-----------+--------+-
|        id|           date|price|bedrooms|bathrooms|sqft_living|sqft_lot|f
+----------+---------------+-----+--------+---------+-----------+--------+-
|7129300520|20141013T000000|2.219|       3|      1.0|       1180|    5650|
|6414100192|20141209T000000| 5.38|       3|     2.25|       2570|    7242|
|5631500400|20150225T000000|  1.8|       2|      1.0|        770|   10000|
|2487200875|20141209T000000| 6.04|       4|      3.0|       1960|    5000|
|1954400510|20150218T000000|  5.1|       3|      2.0|       1680|    8080|
+----------+---------------+-----+--------+---------+-----------+--------+-
only showing top 5 rows
```

Sélection des colones ayant le plus haut degré de corrélation par rapport au prix.

```
# Implement correlation analysis to select relevant features
# correlation_matrix = X.corr()
# Select features with high correlation with the target variable ('price')
#features=["price", "sqft_living", "bedrooms", "bathrooms", "floors", "waterfrc
```

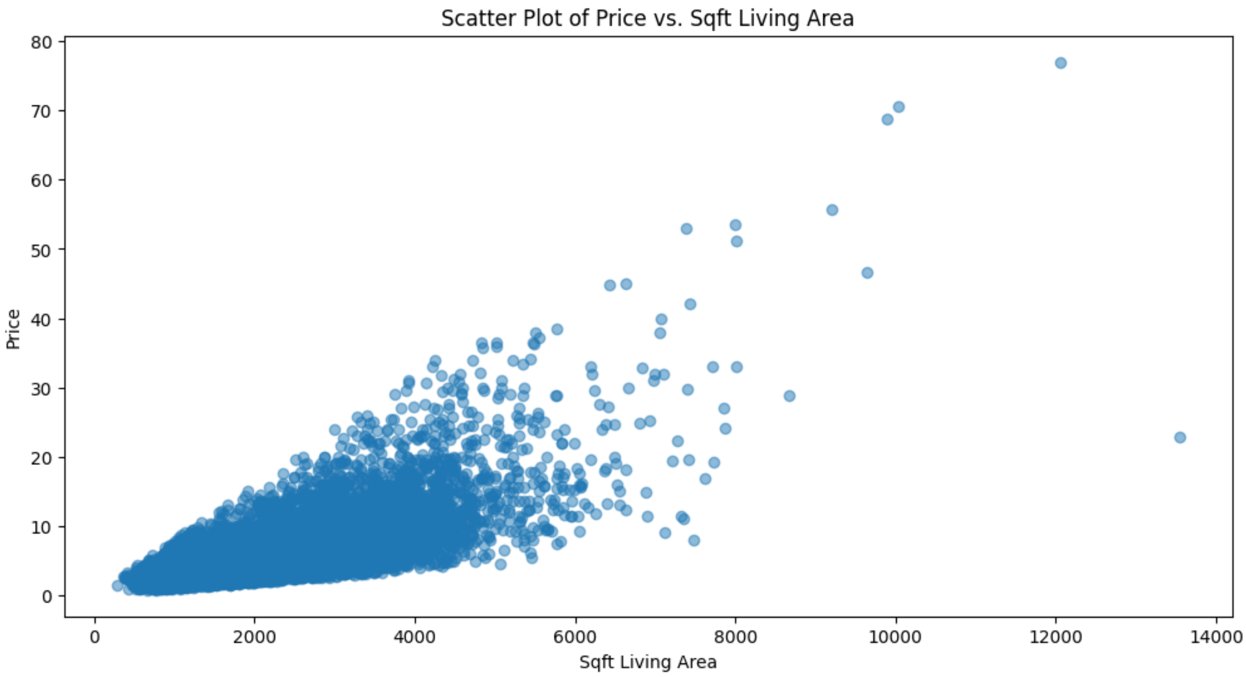
```
# Calculate correlation coefficients between 'price' and other features
correlation_with_price = houses.toPandas().corr(numeric_only=True)['price'].abs

# Select features with high correlation with 'price' (excluding 'price' itself)
selected_features = correlation_with_price[1:14].index.tolist()

# Print selected features
print("Selected Features with High Correlation with 'price':")
print(selected_features)
```

```
Selected Features with High Correlation with 'price':
['sqft_living', 'grade', 'sqft_above', 'avg_living_space_per_neighbor', 'sq
```

Avant la suppression des valeurs abérantes, on peut remarquer la différence entre les shémas représentant le nuage de point de la relation entre le prix et la région (sqft_linving)

Avant

```
# Scatter Plots
plt.figure(figsize=(12, 6))
plt.scatter(houses.select("sqft_living").toPandas(), houses.select("price").toF
plt.title('Scatter Plot of Price vs. Sqft Living Area')
plt.xlabel('Sqft Living Area')
plt.ylabel('Price')
plt.show()
```

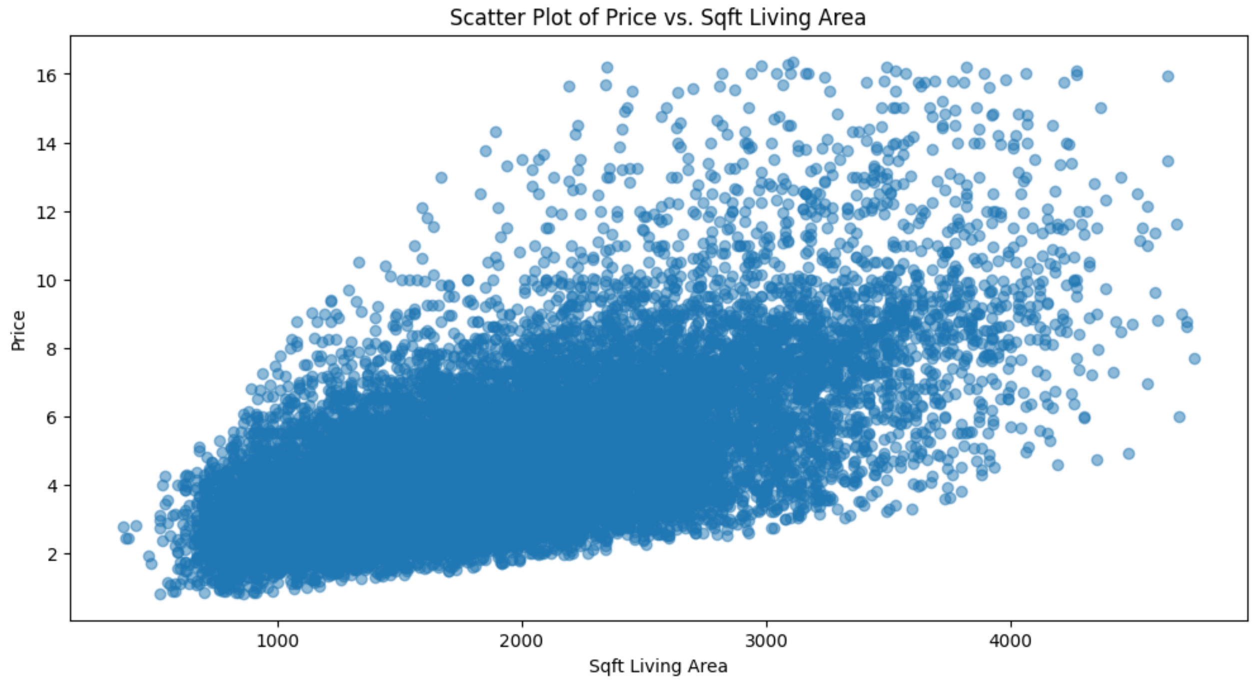


Après

```
from scipy import stats
# Calculate correlation matrix
corr_matrix = houses.toPandas().corr(numeric_only=True)
# Find features with high correlation
highly_correlated = (corr_matrix.abs() > 0.8) & (corr_matrix.abs() < 1)

# Create a set to store redundant features
redundant_features = set()

# Iterate through each feature
for feature in highly_correlated:
    # Find other features that are highly correlated with the current feature
    correlated_features = highly_correlated.index[highly_correlated[feature]].t
    for correlated_feature in correlated_features:
        # Add the correlated feature to the set of redundant features
        redundant_features.add(correlated_feature)


#print(redundant_features)
# Remove redundant features from the dataset
df_filtered = houses.toPandas()
# Assuming 'spark' is your SparkSession
spark = SparkSession.builder.master('local[*]').appName('houses').getOrCreate()

# Convert Pandas DataFrame to PySpark DataFrame
houses_no_outliers = spark.createDataFrame(df_filtered)
# Apply statistical methods for outlier detection
# Z-score method
z_scores = np.abs(stats.zscore(df_filtered.select_dtypes(include=np.number)))
threshold = 3
outliers = np.where(z_scores > threshold)

# Remove outliers from the dataset
houses_no_outliers = df_filtered[(z_scores < threshold).all(axis=1)]

plt.figure(figsize=(12, 6))
plt.scatter(houses_no_outliers["sqft_living"], houses_no_outliers["price"], alp
plt.title('Scatter Plot of Price vs. Sqft Living Area')
plt.xlabel('Sqft Living Area')
```

```
plt.xlabel('Sqft Living Area')
plt.ylabel('Price')
plt.show()
```

```
/content/spark-3.1.1-bin-hadoop3.2/python/pyspark/sql/pandas/conversion.py:
  for column, series in pdf.iteritems():
```



Le grahe précédent sur la matrice de corrélation vérifie bien que les features avec les plus grandes corrélations sont sqft_above'et 'sqft_living', ce qui peut traduire une certaines redondance. Nous n'allons pas les enlever pour pas biaiser la précision du modèle.

Assembler les caractéristiques en un vecteur en gérant les valeurs nulles

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline
from pyspark.sql.functions import col
from pyspark.sql import SparkSession
```

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.functions import when, col
from pyspark.ml.evaluation import RegressionEvaluator

# Assembler les caractéristiques en un vecteur en gérant les valeurs nulles
assembler = VectorAssembler(inputCols=selected_features, outputCol="features",

spark = SparkSession.builder \
    .appName("houses") \
    .getOrCreate()


# # Convert pandas DataFrame to PySpark DataFrame
houses_no_outliers = spark.createDataFrame(houses_no_outliers)

assembled_df = assembler.transform(houses_no_outliers)

assembled_df.show(10, truncate=False)
```

```
+----------+---------------+------+--------+---------+-----------+--------+
|id        |date           |price |bedrooms|bathrooms|sqft_living|sqft_lot|
+----------+---------------+------+--------+---------+-----------+--------+
|7129300520|20141013T000000|2.219 |3       |1.0      |1180       |5650    |
|5631500400|20150225T000000|1.8   |2       |1.0      |770        |10000   |
|2487200875|20141209T000000|6.04  |4       |3.0      |1960       |5000    |
|1954400510|20150218T000000|5.1   |3       |2.0      |1680       |8080    |
|1321400060|20140627T000000|2.575 |3       |2.25     |1715       |6819    |
|2008000270|20150115T000000|2.9185|3       |1.5      |1060       |9711    |
|2414600126|20150415T000000|2.295 |3       |1.0      |1780       |7470    |
|3793500160|20150312T000000|3.23  |3       |2.5      |1890       |6560    |
|9212900260|20140527T000000|4.68  |2       |1.0      |1160       |6000    |
|114101516 |20140528T000000|3.1   |3       |1.0      |1430       |19901   |
+----------+---------------+------+--------+---------+-----------+--------+
only showing top 10 rows
```

Standardisation des données

Nous pouvons enfin mettre à l'échelle les données à l'aide de StandardScaler. Les colonnes d'entrée sont les fonctionnalités, et la colonne de sortie avec le rescaled qui sera inclus dans scaled_df sera nommée "features_scaled" :

```
from pyspark.ml.feature import StandardScaler

# Initialize the `standardScaler`
standardScaler = StandardScaler(inputCol="features", outputCol="features_scalec

# Fit the DataFrame to the scaler
scaled_df = standardScaler.fit(assembled_df).transform(assembled_df)

# Inspect the result
```

```
scaled_df.select("features", "features_scaled").show(10, truncate=False)
```

```
+--------------------------------------------------------------------------------
|features
+--------------------------------------------------------------------------------
|[1180.0,7.0,1180.0,89.33333333333333,1340.0,1.0,4.0,0.0,0.0,3.0,47.5112,0.
|[770.0,6.0,770.0,181.33333333333334,2720.0,1.0,3.0,0.0,0.0,2.0,47.7379,0.0
|[1960.0,7.0,1050.0,90.66666666666667,1360.0,3.0,7.0,0.0,910.0,4.0,47.5208,
|[1680.0,8.0,1680.0,120.0,1800.0,2.0,5.0,0.0,0.0,3.0,47.6168,0.0,1.0]
|[1715.0,7.0,1715.0,149.2,2238.0,2.25,5.25,0.0,0.0,3.0,47.3097,0.0,2.0]
|[1060.0,7.0,1060.0,110.0,1650.0,1.5,4.5,0.0,0.0,3.0,47.4095,0.0,1.0]
|[1780.0,7.0,1050.0,118.66666666666667,1780.0,1.0,4.0,0.0,730.0,3.0,47.5123
|[1890.0,7.0,1890.0,159.33333333333334,2390.0,2.5,5.5,0.0,0.0,3.0,47.3684,0
|[1160.0,7.0,860.0,88.66666666666667,1330.0,1.0,3.0,0.0,300.0,2.0,47.69,0.0
|[1430.0,7.0,1430.0,118.66666666666667,1780.0,1.0,4.0,0.0,0.0,3.0,47.7558,0
+--------------------------------------------------------------------------------
only showing top 10 rows
```

Construction du modèle

1 - Création d'un modèle d'apprentissage automatique avec Spark ML

```
from pyspark.sql import SparkSession
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler

# Assuming you have created a Spark session named 'spark'

# Load your data into a DataFrame (replace 'your_data.csv' with your actual fil
df = spark.read.csv('house.csv', header=True, inferSchema=True)

# Assuming you have already performed the necessary data preparation steps

# Define your random seed
rnd_seed = 42

# Split the data into train and test sets
train_data, test_data = scaled_df.randomSplit([0.8, 0.2], seed=rnd_seed)

# Continue with the rest of your code
train_data.columns

# Initialize `lr`
lr = LinearRegression(featuresCol='features_scaled', labelCol='price', predicti
                      maxIter=10, regParam=0.3, elasticNetParam=0.8, standardiz
```

```
# Fit the data to the model
linearModel = lr.fit(train_data)

# Generate predictions
predictions = linearModel.transform(test_data)

# Extract the predictions and the "known" correct labels
predandlabels = predictions.select("predprice", "price")
predandlabels.show()
```

```
+------------------+-----+
|         predprice|price|
+------------------+-----+
| 4.806730280025931|6.475|
|1.8710783209747888| 2.81|
|  4.21849254259044|  5.2|
| 8.106797540550133| 7.15|
| 4.219881813322331| 2.54|
|2.4852198422946117| 1.89|
| 5.204958872231089|  5.9|
| 4.085059207540752|  3.8|
| 3.256174430346789| 3.98|
|3.9727311105970955| 2.55|
| 2.706680692505955|  1.9|
|  2.47611449324404|  1.7|
| 4.152159065356301| 3.93|
| 7.797939169030968|  7.2|
| 4.848014319180379| 2.75|
| 3.157640770734929| 2.68|
|3.3031056212044803|2.785|
|  4.57763234153623| 5.95|
| 4.248204542782986|  5.4|
| 5.383316444345411| 3.98|
+------------------+-----+
only showing top 20 rows
```

Inspection des métriques

Il s'agit d'examiner les valeurs prédites et certaines mesures pour avoir une meilleure idée de la qualité réelle de votre modèle.

```
# Utilisation de RegressionEvaluator du  package pyspark.ml :

evaluator = RegressionEvaluator(predictionCol="predprice", labelCol='price', me
print("RMSE: {0}".format(evaluator.evaluate(predandlabels)))

evaluator = RegressionEvaluator(predictionCol="predprice", labelCol='price', me
print("MAE: {0}".format(evaluator.evaluate(predandlabels)))

evaluator = RegressionEvaluator(predictionCol="predprice", labelCol='price', me
```

```
print("R2: {0}".format(evaluator.evaluate(predandlabels)))
```

```
RMSE: 1.4952875709354545
MAE: 1.06531280740458
R2: 0.598858095343904
```

Il y a certainement quelques améliorations à apporter à notre modèle ! Si nous voulons continuer avec ce modèle, nous pouvons jouer avec les paramètres que nous avons passés à notre modèle, les variables que nous avons incluses dans votre DataFrame d'origine.

Le modèle ci-dessus a une précision de 59,885 %. Il est donc utile d'en explorer d'autres.

- Méthode de Random Forest, Arbre de decision et Gradient
- Cross-validation
- La méthode ACP
- Méthode 1 en utilisant que les colones en forte corélation avec le prix

Random Forest - Arbre de décision - Gradient

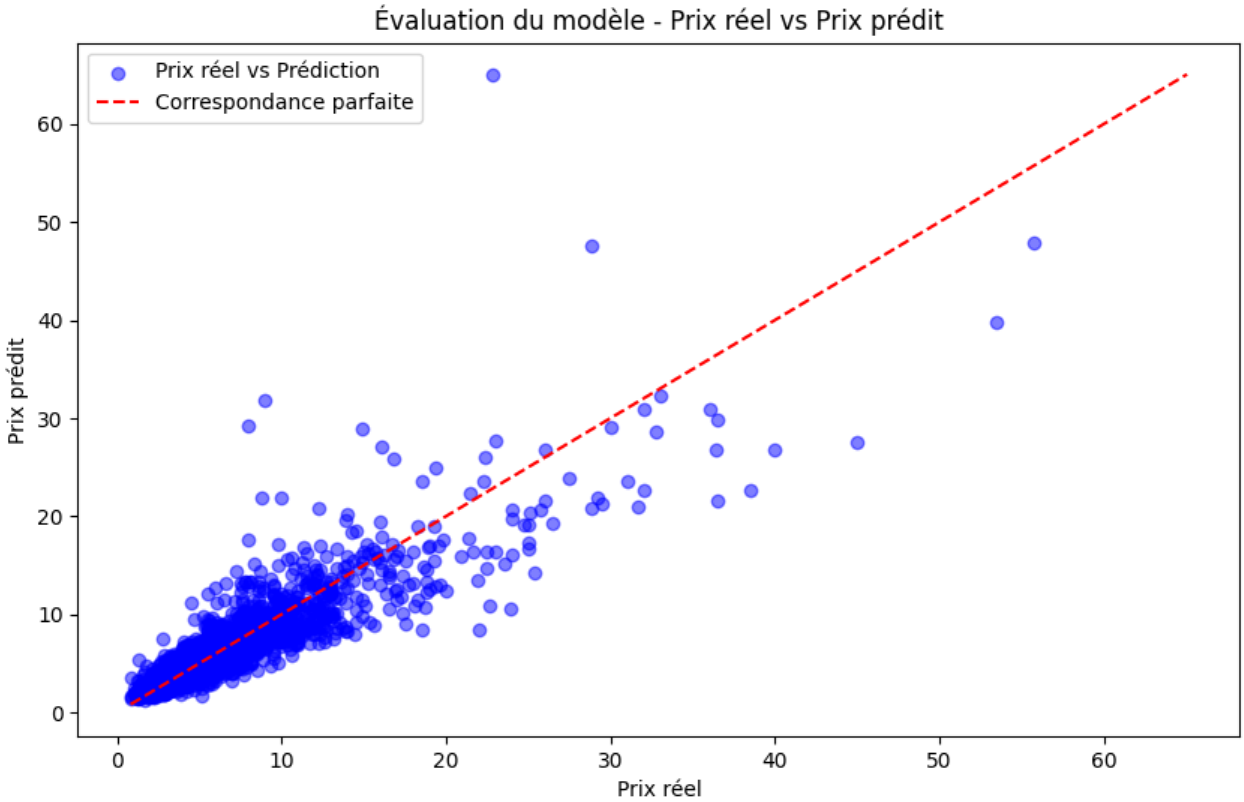
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

# # 1. Load the dataset and split into features (X) and target variable (y)
X = df_filtered[selected_features]  # Features
y = houses.toPandas()['price']  # Target variable

# # 1. Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random

# Initialize models
models = {
    'Linear Regression': LinearRegression(),
    'Decision Tree': DecisionTreeRegressor(),
    'Random Forest': RandomForestRegressor(),
```

```
    'Gradient Boosting': GradientBoostingRegressor()
}

# Initialize variables to track best model and its MAE
from sklearn.metrics import mean_squared_error, r2_score

# Initialize variables to track best model and its error metrics
best_model = None
best_mae = float('inf')  # Initialize with infinity
best_rmse = float('inf')
best_r2 = float('-inf')
r2_scores = []

# Iterate over models
for name, model in models.items():
    # Train the model
    model.fit(X_train, y_train)

    # Cross-validation for Mean Absolute Error (MAE)
    cv_mae_scores = cross_val_score(model, X_train, y_train, cv=5, scoring='neg
    mae_mean = -cv_mae_scores.mean()

    # Cross-validation for Root Mean Squared Error (RMSE)
    cv_rmse_scores = cross_val_score(model, X_train, y_train, cv=5, scoring='ne
    rmse_mean = -cv_rmse_scores.mean()

    # Cross-validation for R2
    cv_r2_scores = cross_val_score(model, X_train, y_train, cv=5, scoring='r2')
    r2_mean = cv_r2_scores.mean()

    r2_scores.append(r2_mean)

    # Update best model if current model has lower MAE
    if mae_mean < best_mae:
        best_model_name = name
        best_model = model
        best_mae = mae_mean
        best_rmse = rmse_mean
        best_r2 = r2_mean
        best_cv_r2_scores = cv_r2_scores

# Print the best model and its error metrics
print("Best Model: ", best_model_name," donc : ", best_model)
# print(best_model)
print("\nMean Absolute Error (MAE):", best_mae)
print("\nRoot Mean Squared Error (RMSE):", best_rmse)
print("\nR2 Score:", best_r2)
print("\n\n")


import matplotlib.pyplot as plt
import numpy as np

# Convertir les résultats de PySpark DataFrame en Pandas DataFrame
```

```
# predictions_pd = predictions.select("price", "prediction").toPandas()

# Make predictions using the best model
predictions = best_model.predict(X_test)

# Create a DataFrame to store actual prices and predictions
predictions_df = pd.DataFrame({'Actual Prices': y_test.values, 'Predicted Price

# Display the first 50 predictions and actual prices
print(predictions_df.head(20))


# Extraire les valeurs réelles et prédites
prices_actual = predictions_df["Actual Prices"]
prices_predicted = predictions_df["Predicted Prices"]

# Créer un diagramme de dispersion
plt.figure(figsize=(10, 6))
plt.scatter(prices_actual, prices_predicted, alpha=0.5, color='blue', label='Pr

# Ajouter une ligne de référence pour la correspondance parfaite
min_val = min(min(prices_actual), min(prices_predicted))
max_val = max(max(prices_actual), max(prices_predicted))
plt.plot([min_val, max_val], [min_val, max_val], linestyle='--', color='red', l

# Ajouter des étiquettes et un titre
plt.xlabel('Prix réel')
plt.ylabel('Prix prédit')
plt.title('Évaluation du modèle - Prix réel vs Prix prédit')

# Ajouter une légende
plt.legend()

# Afficher le diagramme
plt.show()
```

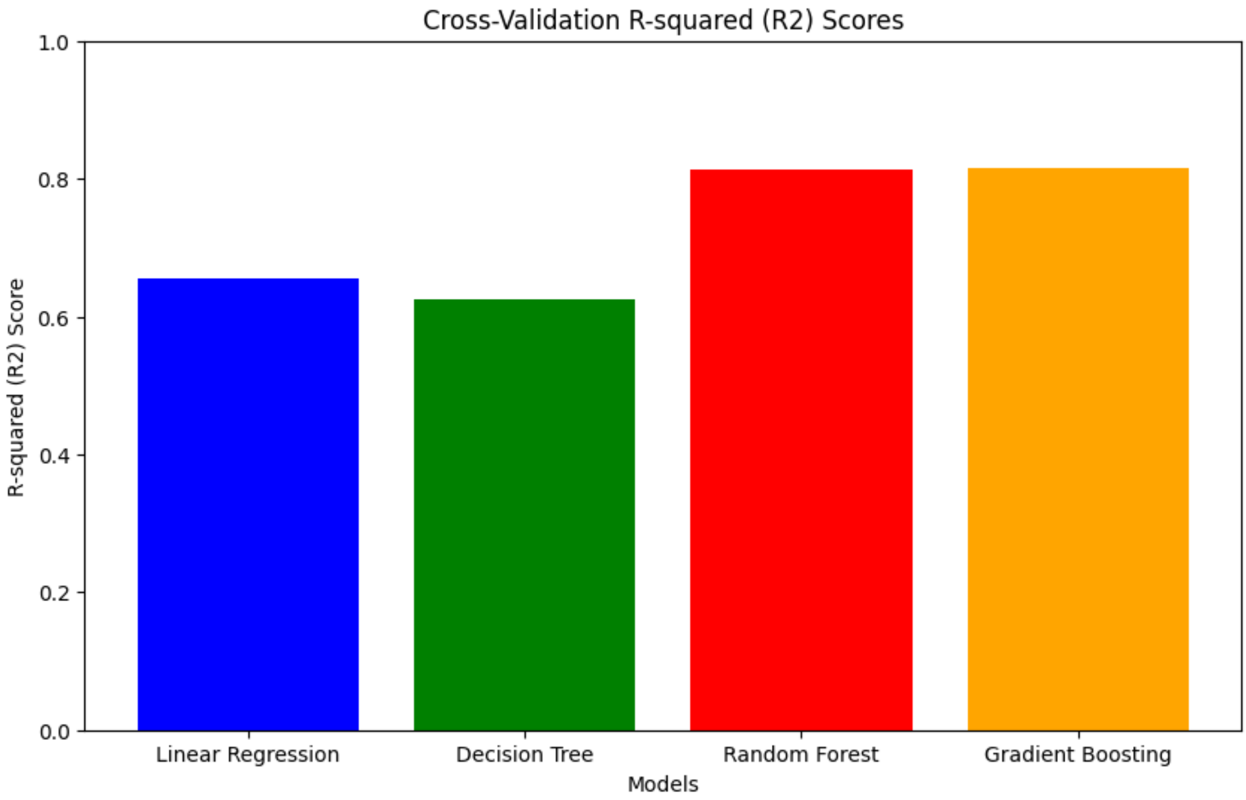
```
Best Model:  Random Forest  donc :  RandomForestRegressor()

Mean Absolute Error (MAE): 0.8765662856272136

Root Mean Squared Error (RMSE): 1.558818074355648

R2 Score: 0.8135485147744624



    Actual Prices  Predicted Prices
0         3.65000          3.925227
1         8.65000          8.053467
2        10.38000         10.942299
3        14.90000         15.236410
4         7.11000          7.495565
5         2.11000          2.525264
6         7.90000          8.903999
7         6.80000          6.046280
8         3.84500          4.086396
```

```
9         6.05000         5.475412
10        6.38000         6.483589
11        3.85000         3.950305
12        1.75000         2.670060
13        3.65000         3.224840
14        1.60000         3.352902
15       10.70000        11.330400
16        8.00000         6.350275
17        7.95127        17.643855
18        3.55000         3.674970
19        4.74000         4.088030
```




```
import matplotlib.pyplot as plt

# Extract R2 scores for each model
r2_scores2 = [cv_r2_scores.mean() for cv_r2_scores in r2_scores]
models_names = list(models.keys())

# Plot the R2 scores
plt.figure(figsize=(10, 6))
plt.bar(models_names, r2_scores2, color=['blue', 'green', 'red', 'orange'])
plt.title('Cross-Validation R-squared (R2) Scores')
plt.xlabel('Models')
plt.ylabel('R-squared (R2) Score')
plt.ylim(0, 1)  # Set y-axis limits between 0 and 1 for R2 score
plt.show()
```

```
plt.show()
```



Cross-Validation

La méthode ACP

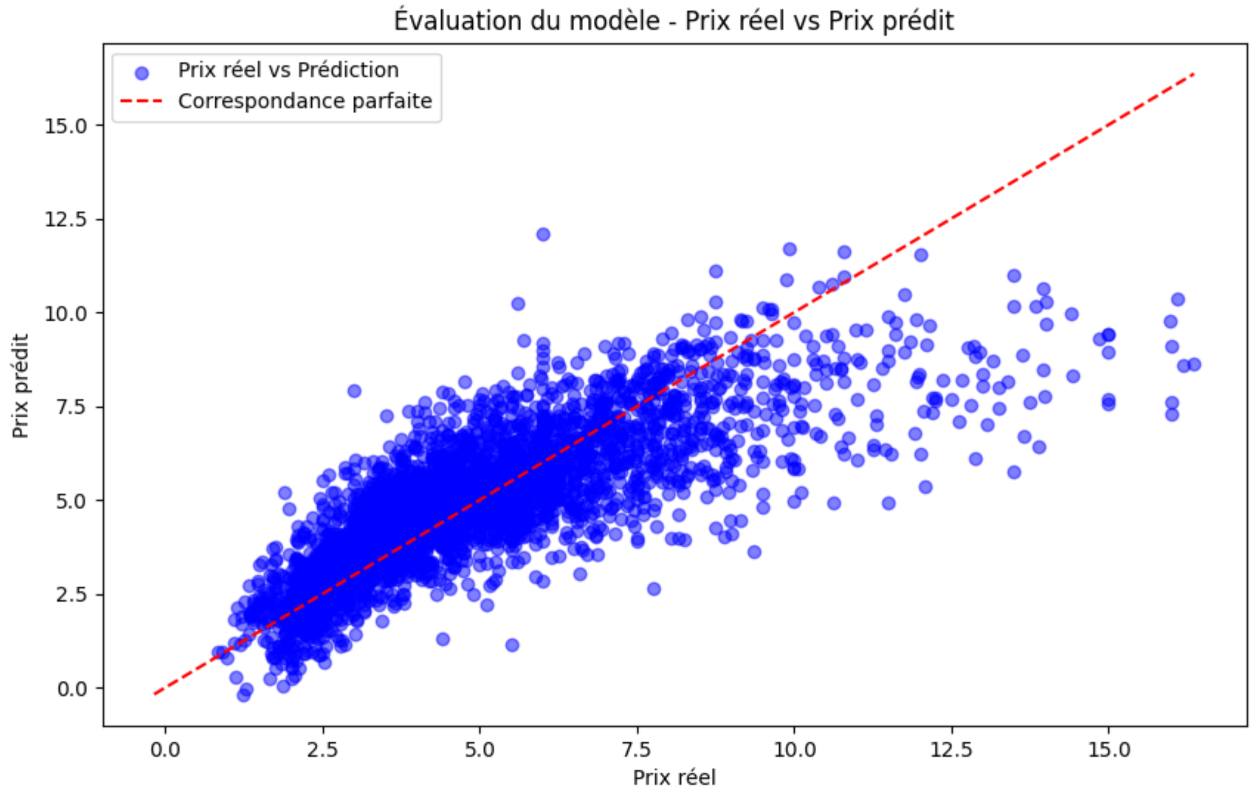
```
from pyspark.ml.feature import PCA
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import LinearRegression

# # Calculate correlation coefficients between 'price' and other features
# correlation_with_price = houses_no_outliers.corr(numeric_only=True)['price'].

# # Select features with high correlation with 'price' (excluding 'price' itsel
# selected_features = correlation_with_price[1:11].index.tolist()

# Assemble the selected features into a vector column
assembler = VectorAssembler(inputCols=selected_features, outputCol="features2",

# Define the PCA transformer
```

```
num_pca_components = 3  # You can adjust this value
pca = PCA(k=num_pca_components, inputCol="features2", outputCol="pca_features")

# Define the Linear Regression model
lr = LinearRegression(featuresCol="features2", labelCol="price")

# Create a pipeline with feature assembling, PCA, and linear regression
pipeline = Pipeline(stages=[assembler, pca, lr])

# Define parameter grid for cross-validation
paramGrid = ParamGridBuilder().addGrid(pca.k, [3, 5, 7]).build()

# Define evaluator
evaluator = RegressionEvaluator(labelCol="price", predictionCol="prediction", m

# Create CrossValidator
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator,
                          numFolds=5)

# Train the model using CrossValidator
cv_model = crossval.fit(train_data)

# Make predictions on the test data
predictions = cv_model.transform(test_data)

# Show predictions
predictions.select("price", "prediction").show(50)

# Evaluate the model and get the RMSE
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE):", rmse)

# Create a RegressionEvaluator
evaluator = RegressionEvaluator(labelCol="price", predictionCol="prediction", m

# Evaluate the model and get the R-squared value
r2 = evaluator.evaluate(predictions)

# Display the R-squared value or coefficient of determination
print("R2: %.3f" %r2)

eval = RegressionEvaluator(labelCol="price", predictionCol="prediction", metric
# Root Mean Square Error
rmse = eval.evaluate(predictions)
print("RMSE: %.3f" % rmse)

# Mean Square Error
mse = eval.evaluate(predictions,  {eval.metricName: "mse"})
print("MSE: %.3f" % mse)

# Mean Absolute Error
mae = eval.evaluate(predictions,  {eval.metricName: "mae"})
```

```
print("MAE: %.3f" % mae)

# Convertir les résultats de PySpark DataFrame en Pandas DataFrame
predictions_pd = predictions.select("price", "prediction").toPandas()

# Extraire les valeurs réelles et prédites
prices_actual = predictions_pd["price"]
prices_predicted = predictions_pd["prediction"]

# Créer un diagramme de dispersion
plt.figure(figsize=(10, 6))
plt.scatter(prices_actual, prices_predicted, alpha=0.5, color='blue', label='Pr

# Ajouter une ligne de référence pour la correspondance parfaite
min_val = min(min(prices_actual), min(prices_predicted))
max_val = max(max(prices_actual), max(prices_predicted))
plt.plot([min_val, max_val], [min_val, max_val], linestyle='--', color='red', l

# Ajouter des étiquettes et un titre
plt.xlabel('Prix réel')
plt.ylabel('Prix prédit')
plt.title('Évaluation du modèle - Prix réel vs Prix prédit')

# Ajouter une légende
plt.legend()

# Afficher le diagramme
plt.show()
```

```
+-------+------------------+
|  price|        prediction|
+-------+------------------+
|  6.475| 4.796037909742779|
|   2.81|1.2569231046385312|
|    5.2|3.9994883171071933|
|   7.15| 8.806835283401995|
|   2.54|3.8276822704322058|
|   1.89|1.9044491016077814|
|    5.9| 6.038437924051834|
|    3.8| 3.710210288351732|
|   3.98|3.0040758027651577|
|   2.55| 3.777805270927388|
|    1.9|2.1201063900715553|
|    1.7| 1.961673856851121|
|   3.93| 4.088582604244209|
|    7.2| 8.115831630859418|
|   2.75|4.9809307221803465|
|   2.68|2.5536202983001886|
|  2.785|2.6816573518498785|
|   5.95|  4.51016343267662|
|    5.4| 4.085090377912707|
|   3.98|5.5889369201640875|
|    3.8| 5.998356137940732|
|    4.5| 5.795466083292524|
|   9.45| 6.326138561257096|
|   8.56| 5.921994305566045|
|  5.599| 8.391675060503701|
```

```
|   4.85|   6.1712737199245|
|   6.41|7.4508085610825106|
|    6.2| 6.904316753879073|
|3.94999| 5.317903233705351|
|    6.5| 5.561700388576298|
|  7.265| 5.946354963155841|
|   4.69| 6.075183319139114|
|   2.25| 4.301748494585183|
|    4.1|3.1873251758181596|
|   6.24| 8.671878771340573|
|   5.05|  4.20939116819244|
|    4.0| 5.210176790488674|
|   6.08| 5.610707505564676|
|   8.75| 6.360102369755452|
|   7.51| 5.158554754742283|
|  2.399|2.6162277230152426|
|  3.295| 4.959108035369638|
|  2.451| 2.069433588035338|
|  2.695| 2.537294760567022|
|   2.15|2.2710612284514013|
|   3.75|2.3595034177515117|
|    2.6|2.8613833457450255|
|   2.59|2.0393038065856786|
|  6.675| 4.899692650362454|
|   15.0| 9.420895556847313|
+-------+------------------+
only showing top 50 rows

Root Mean Squared Error (RMSE): 1.4294954502236994
R2: 0.633
RMSE: 1.429
MSE: 2.043
MAE: 1.018
```



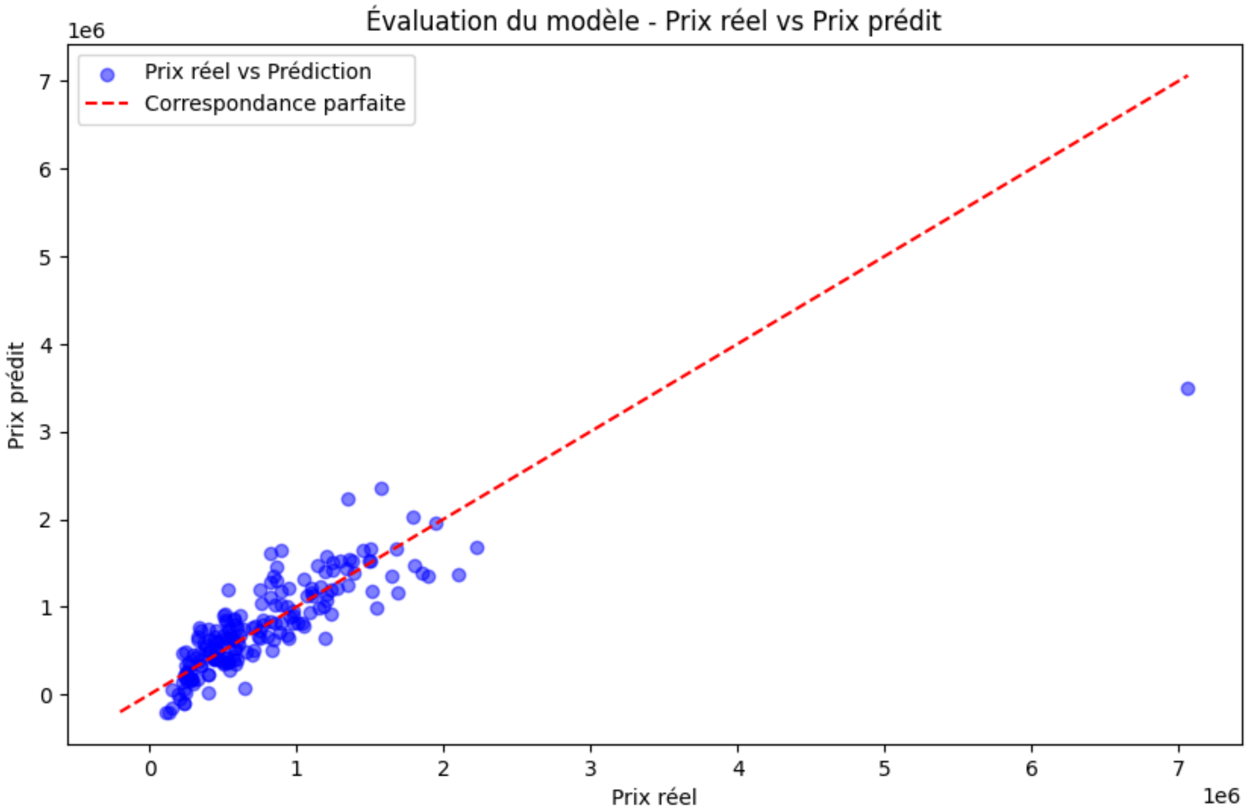
Méthode 1 en utilisant que les colones en forte corélation avec le prix

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline
from pyspark.sql.functions import col, when
from pyspark.ml.evaluation import RegressionEvaluator
import matplotlib.pyplot as plt
import numpy as np


# Initialiser SparkSession
spark = SparkSession.builder.master('local[*]').appName('houses').getOrCreate()

# Lire les données depuis le fichier CSV
houses = spark.read.csv('house.csv', inferSchema=True, header=True, nullValue='

# Afficher le nombre d'enregistrements
print("Nombre d'enregistrements : ", houses.count())

# Sélectionner uniquement les colonnes spécifiques
selected_columns = [ "bathrooms", "sqft_living", "view", "grade", "sqft_above",
houses = houses.select(selected_columns + ["price"])

# Vérifier si la colonne "yr_renovated" existe avant de la traiter
if "yr_renovated" in houses.columns:
    # Remplacer les zéros par None dans la colonne yr_renovated
    houses = houses.withColumn("yr_renovated", when(col("yr_renovated") == 0, N

# Diviser les données en ensembles d'entraînement et de test
train_data, test_data = houses.randomSplit([0.8, 0.2], seed=42)

# Définir les colonnes de caractéristiques
feature_cols = houses.columns[:-1]  # Exclure la colonne "price"

# Assembler les caractéristiques en un vecteur en gérant les valeurs nulles
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features", handl

# Initialiser le modèle de régression linéaire
lr = LinearRegression(featuresCol="features", labelCol="price")

# Créer le pipeline avec l'assemblage des caractéristiques et le modèle
```

```
pipeline = Pipeline(stages=[assembler, lr])

# Entraîner le modèle sur les données d'entraînement
model = pipeline.fit(train_data)

# Faire des prédictions sur les données de test
predictions = model.transform(test_data)

# Afficher les résultats des prédictions
print("Voici une comparaison du prix réel et celui qui a été prédit :")
predictions.select("price", "prediction").show(5)



# Create a RegressionEvaluator
evaluator = RegressionEvaluator(labelCol="price", predictionCol="prediction", m

# Evaluate the model and get the R-squared value
r2 = evaluator.evaluate(predictions)

# Display the R-squared value
print("R-squared:", r2)

# Convertir les résultats de PySpark DataFrame en Pandas DataFrame
predictions_pd = predictions.select("price", "prediction").toPandas()

# Extraire les valeurs réelles et prédites
prices_actual = predictions_pd["price"]
prices_predicted = predictions_pd["prediction"]

# Créer un diagramme de dispersion
plt.figure(figsize=(10, 6))
plt.scatter(prices_actual, prices_predicted, alpha=0.5, color='blue', label='Pr

# Ajouter une ligne de référence pour la correspondance parfaite
min_val = min(min(prices_actual), min(prices_predicted))
max_val = max(max(prices_actual), max(prices_predicted))
plt.plot([min_val, max_val], [min_val, max_val], linestyle='--', color='red', l

# Ajouter des étiquettes et un titre
plt.xlabel('Prix réel')
plt.ylabel('Prix prédit')
plt.title('Évaluation du modèle - Prix réel vs Prix prédit')

# Ajouter une légende
plt.legend()

# Afficher le diagramme
plt.show()
```

```
Nombre d'enregistrements :  21613
Voici une comparaison du prix réel et celui qui a été prédit :
+--------+------------------+
```

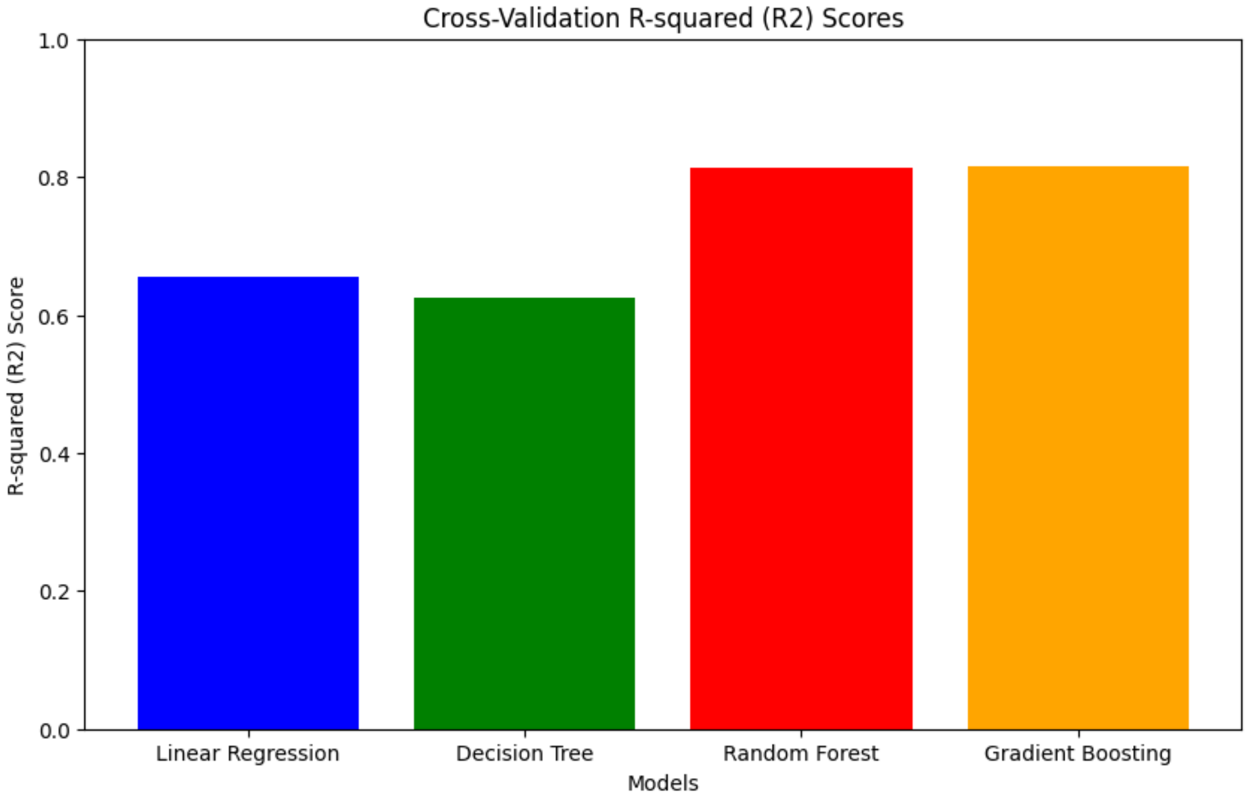
```
|   price|        prediction|
+--------+------------------+
|247500.0|  487907.3095747754|
|330600.0| 180387.62489681318|
|135000.0|-197067.03201676905|
|252000.0| 198669.75718412548|
|110000.0| -196389.4547709301|
+--------+------------------+
only showing top 5 rows

R-squared: 0.6655729534207662
```



Visualisation de l'erreur

```
import matplotlib.pyplot as plt

# Extract R2 scores for each model
r2_scores2 = [cv_r2_scores.mean() for cv_r2_scores in r2_scores]
models_names = list(models.keys())

# Plot the R2 scores
plt.figure(figsize=(10, 6))
plt.bar(models_names, r2_scores2, color=['blue', 'green', 'red', 'orange'])
plt.title('Cross Validation R-squared (R2) Scores')
```

```
plt.title('Cross-Validation R-squared (R2) Scores')
plt.xlabel('Models')
plt.ylabel('R-squared (R2) Score')
plt.ylim(0, 1)  # Set y-axis limits between 0 and 1 for R2 score
plt.show()
```



On obtient dans les résultats:

- Methode 1 : 59,885%
- Méthode de Random Forest : 81.395%
- Cross-validation : 61.4%
- La méthode ACP : 61.4%
- Méthode 1 mais en utilisant que les colones en forte corélation avec le prix : 66.5%**

Ce résultat est bien vérifié par le graphe ci-dessus. La méthode de **Random Forest** reste donc la meilleur pour ce problème de regression.