In [1]:
```python
# Importacion de las bibliotecas de trabajo
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
sns.set()
```

Paso 1: Lectura del dataset

In [2]:
```python
df = pd.read_csv("https://raw.githubusercontent.com/levraines/Portfolio/master/Master_Artificial%20Intelligence/Data/act4_kc_house_data.csv",
                 delimiter = ',')

# Creando copia del dataset
data = df.copy()

# Visualizando dataset
data.head()
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sq
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...	7	1180	
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...	7	2170	
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	...	6	770	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	...	7	1050	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	...	8	1680	

5 rows × 21 columns

In [3]:
```python
# Comprobando el tamaño del dataset
data.shape
```

Out[3]: (21613, 21)

Se puede ver que el dataset cuenta con un total de 21613 observaciones y un total de 21 características.

In [4]:
```python
# Viendo tipo de las variables
data.dtypes
```

Out[4]:
```
id                 int64
date              object
price            float64
bedrooms           int64
bathrooms        float64
sqft_living        int64
sqft_lot           int64
floors           float64
waterfront         int64
view               int64
condition          int64
grade              int64
sqft_above         int64
sqft_basement      int64
yr_built           int64
yr_renovated       int64
zipcode            int64
lat              float64
long             float64
sqft_living15      int64
sqft_lot15         int64
dtype: object
```

Se puede observar que todas las variables son de tipo integer o flotante, excepto la variable date que corresponde a string (objeto).

Paso 2: Preprocesado

Lo siguiente que necesitamos hacer es convertir las áreas a metros cuadrados y además convertir la fecha de venta en string "YYYYMMDD".

In [5]:
```python
# Convirtiendo la variable "date" a formato "YYYYMMDD".
data['date'] = data['date'].str.slice(0, 8)
```

In [6]:
```python
# Confirmando que cambio fue efectivo en la variable
data['date'].value_counts()
```

Out[6]:
```
20140623    142
20140626    131
20140625    131
20140708    127
20150427    126
           ...
20141102      1
20150131      1
20150524      1
20140517      1
20140727      1
Name: date, Length: 372, dtype: int64
```

Conversión de variables:

Para este caso un metro cuadrado corresponde a 10,764 pies cuadrados, entonces se va a proceder con una división.

In [7]:
```python
# Convirtiendo las variables que tienen áreas a metros cuadrados
data['sqft_living'] = data['sqft_living'] / 10.764
data['sqft_lot'] = data['sqft_lot'] / 10.764
data['sqft_above'] = data['sqft_above'] / 10.764
data['sqft_basement'] = data['sqft_basement'] / 10.764
data['sqft_living15'] = data['sqft_living15'] / 10.764
data['sqft_lot15'] = data['sqft_lot15'] / 10.764
```

In [8]:
```python
# Confirmando que cambios fueron efectivos en las variables
data.head()
```

Out[8]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_b
0	7129300520	20141013	221900.0	3	1.00	109.624675	524.897808	1.0	0	0	...	7	109.624675	
1	6414100192	20141209	538000.0	3	2.25	238.758826	672.798216	2.0	0	0	...	7	201.597919	
2	5631500400	20150225	180000.0	2	1.00	71.534745	929.022668	1.0	0	0	...	6	71.534745	
3	2487200875	20141209	604000.0	4	3.00	182.088443	464.511334	1.0	0	0	...	7	97.547380	
4	1954400510	20150218	510000.0	3	2.00	156.075808	750.650316	1.0	0	0	...	8	156.075808	

5 rows × 21 columns

Ahora que ya se hizo el proprocesado de las variables indicadas, se obtiene que la variable "date" en formato estándar de fecha, tal y como se solicita: "YYYYMMDD" está en formato object por ende se puede proceder de dos formas:

1. Se podría convertir cada una de las observaciones a formato numérico para poder utilizarla en el modelo, ya que el formato de "object" no puede ser leído por el algoritmo, entonces podríamos utilizar el Label Encoder para poder pasar dicha variable a tipo numérica, sin embargo caemos en el problema que habíamos mencionado anteriormente; ya que los algoritmos podrían pensar que hay un orden establecido y sí que lo hay, ya que son fechas, pero al ser más de 21000 datos se estarían creando más categorías de las necesarias y esto podría hacer que el algoritmo primero sea deficiente y segundo que no encuentre ninguna asociación clara. Otra solución sería utilizar el One Hot Encoder, pero este crearía matrices de 1s y 0s por la cantidad distinta de valores que contiene dicha variable y crearía demasiadas columnas, esto tampoco es eficiente.
2. Lo que se podría hacer en este caso para poder utilizar dicha variable, ya que, el año, mes y día son importantes porque reflejan una característica específica en un tiempo determinado, sería en lugar de tratar toda la fecha, podríamos hacer una ingeniería de variable y separarla por año, mes y día y así podemos encontrar diferentes tipos de relaciones ya que el algoritmo podría agrupar meses o días específicos y podría encontrar relaciones para la predicción y con ello hacer una mejor estimación de los precios. Dicho lo anterior se procederá a realizar tres nuevas variables artificiales: "Año", "Mes" y "Día".

In [9]:
```python
# Creando variable de año y convirtiendola a formato de "integer"
data['year'] = data['date'].str.slice(0, 4)
data['year'] = data['year'].astype(int)
```

In [10]:
```python
# Creando variable de mes y convirtiendola a formato de "integer"
data['month'] = data['date'].str.slice(4, -2)
data['month'] = data['month'].astype(int)
```

In [11]:
```python
# Creando variable de dia y convirtiendola a formato de "integer"
data['day'] = data['date'].str.slice(6)
data['day'] = data['day'].astype(int)
```

In [12]:
```python
# Visualizando resultados
data.head()
```

Out[12]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	yr_built	yr_renovated	zi
0	7129300520	20141013	221900.0	3	1.00	109.624675	524.897808	1.0	0	0	...	1955	0	
1	6414100192	20141209	538000.0	3	2.25	238.758826	672.798216	2.0	0	0	...	1951	1991	
2	5631500400	20150225	180000.0	2	1.00	71.534745	929.022668	1.0	0	0	...	1933	0	
3	2487200875	20141209	604000.0	4	3.00	182.088443	464.511334	1.0	0	0	...	1965	0	
4	1954400510	20150218	510000.0	3	2.00	156.075808	750.650316	1.0	0	0	...	1987	0	

5 rows × 24 columns

In [13]:
```python
# Consultando los formatos de las variables
data.dtypes
```

Out[13]:
```
id                 int64
date              object
price            float64
bedrooms           int64
bathrooms        float64
sqft_living      float64
sqft_lot         float64
floors           float64
waterfront         int64
view               int64
condition          int64
grade              int64
sqft_above       float64
sqft_basement    float64
yr_built           int64
yr_renovated       int64
zipcode            int64
lat              float64
long             float64
sqft_living15    float64
sqft_lot15       float64
year               int64
month              int64
day                int64
dtype: object
```

Podemos observar como todas las variables que creamos ya estan en formato de número y pueden ser usadas por nuestro algoritmo.

In [14]:
```python
# Comprobando variables nulas o registros faltantes
np.sum(data.isnull())
```

Out[14]:
```
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
year             0
month            0
day              0
dtype: int64
```

Se puede comprobar que no existen registros o datos nulos en el dataset.

In [15]:
```python
# Reorganizando el orden de las variables para dejar la variable a predecir de ultima en el dataset
cols = list(data.columns.values)
cols
```

Out[15]:
```
['id',
 'date',
 'price',
 'bedrooms',
 'bathrooms',
 'sqft_living',
 'sqft_lot',
 'floors',
 'waterfront',
 'view',
 'condition',
 'grade',
 'sqft_above',
 'sqft_basement',
 'yr_built',
 'yr_renovated',
 'zipcode',
 'lat',
 'long',
 'sqft_living15',
 'sqft_lot15',
 'year',
 'month',
 'day']
```

In [16]:
```python
# Reorganizando el dataset
data = data[['id',
 'date',
 'bedrooms',
 'bathrooms',
 'sqft_living',
 'sqft_lot',
 'floors',
 'waterfront',
 'view',
 'condition',
 'grade',
 'sqft_above',
 'sqft_basement',
 'yr_built',
 'yr_renovated',
 'zipcode',
 'lat',
 'long',
 'sqft_living15',
 'sqft_lot15',
 'year',
 'month',
 'day',
 'price']]
```

In [17]:
```python
# Consultando resultado
data.head()
```

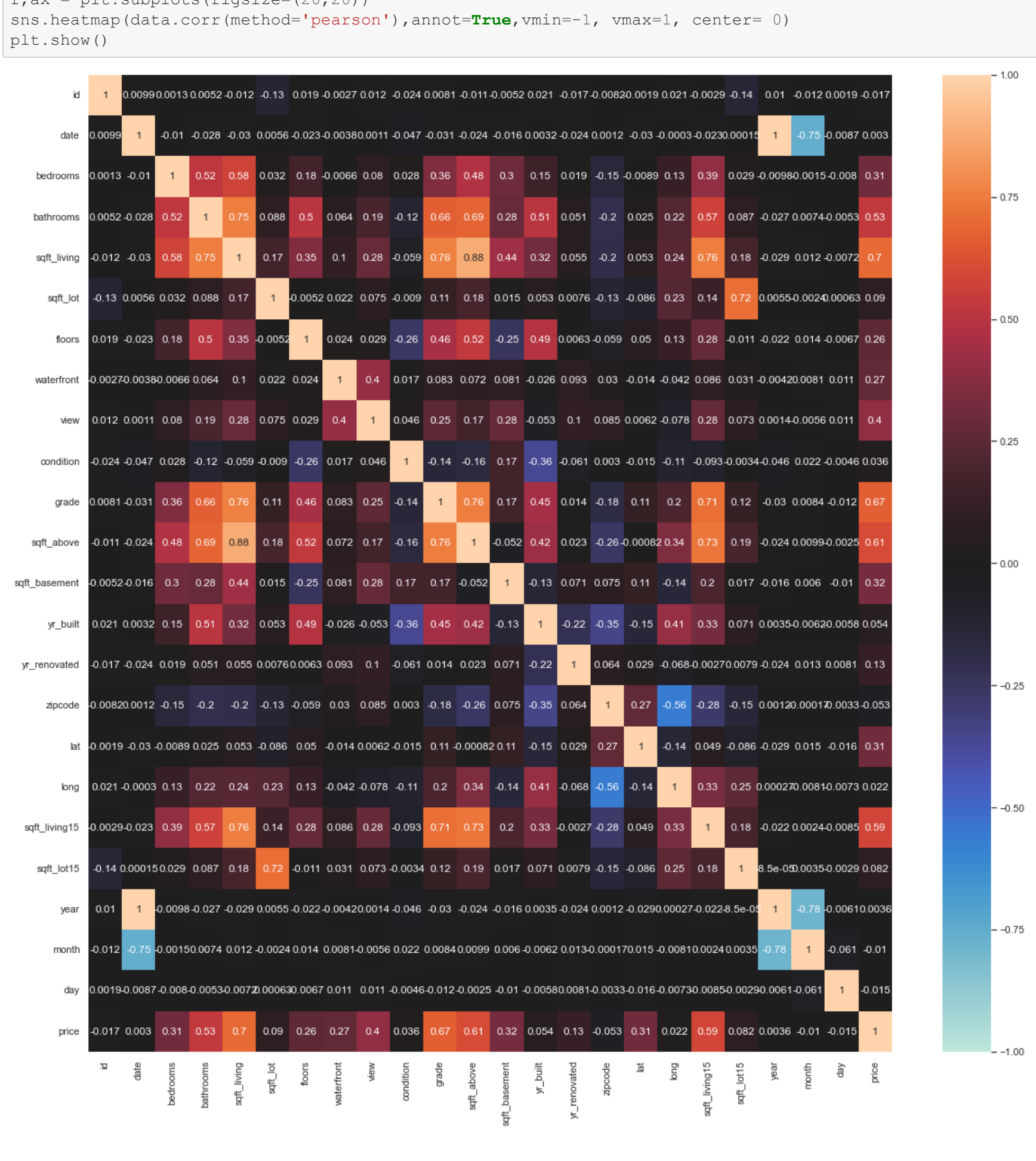
Out[17]:

	id	date	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	...	yr_renovated	zipcode
0	7129300520	20141013	3	1.00	109.624675	524.897808	1.0	0	0	3	...	0	98178
1	6414100192	20141209	3	2.25	238.758826	672.798216	2.0	0	0	3	...	1991	98125
2	5631500400	20150225	2	1.00	71.534745	929.022668	1.0	0	0	3	...	0	98028
3	2487200875	20141209	4	3.00	182.088443	464.511334	1.0	0	0	5	...	0	98136
4	1954400510	20150218	3	2.00	156.075808	750.650316	1.0	0	0	3	...	0	98074

5 rows × 24 columns

In [18]:
```python
# Convirtiendo variable "date" a integer
data['date'] = data['date'].astype(int)
```

In [19]:
```python
# visualizando correlaciones
import matplotlib.pyplot as plt
import seaborn as sns

f,ax = plt.subplots(figsize=(20,20))
sns.heatmap(data.corr(method='pearson'),annot=True,vmin=-1, vmax=1, center= 0)
plt.show()
```



Ahora bien, la variable "date" la convertimos a formato integer y procedimos a ver si tiene correlación con las nuevas variables creadas (año, mes y día) y sí que hay correlación (como era de esperarse) más que todo con año y mes. Por ende, en nuestro análisis vamos a eliminar la variable "date" para que no le añada ruedo al modelo.

Hay otras correlaciones fuertes como grade y bathrooms, bedrooms y bathrooms, yr_built y bathrooms, y otro puñado más. Al ser una regresión lo que estamos tratando de hacer, lo ideal sería dejar solo aquellas variables significativas en el modelo, como esto es un proceso académico vamos a trabajar con todas las variables, eso sí vamos a eliminar la variable "id" ya que esta variable es el número de llave del registro y no añade mucha más información al modelo, solo es un identificador.

In [20]:
```python
# Retornando columnas
data.columns
```

Out[20]:
```
Index(['id', 'date', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot',
       'floors', 'waterfront', 'view', 'condition', 'grade', 'sqft_above',
       'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat', 'long',
       'sqft_living15', 'sqft_lot15', 'year', 'month', 'day', 'price'],
      dtype='object')
```

In [21]:
```python
# Creando nuevo subset
df = data[['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot',
       'floors', 'waterfront', 'view', 'condition', 'grade', 'sqft_above',
       'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat', 'long',
       'sqft_living15', 'sqft_lot15', 'year', 'month', 'day', 'price']]
```

In [22]:
```python
# Consultando nuevo subset
df.head()
```

Out[22]:

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	...	yr_renovated	zipcode	
0	3	1.00	109.624675	524.897808	1.0	0	0	3	7	109.624675	...	0	98178	47.51
1	3	2.25	238.758826	672.798216	2.0	0	0	3	7	201.597919	...	1991	98125	47.72
2	2	1.00	71.534745	929.022668	1.0	0	0	3	6	71.534745	...	0	98028	47.73
3	4	3.00	182.088443	464.511334	1.0	0	0	5	7	97.547380	...	0	98136	47.52
4	3	2.00	156.075808	750.650316	1.0	0	0	3	8	156.075808	...	0	98074	47.61

5 rows × 22 columns

Se puede observar como el nuevo subset fue creado de forma correcta y ahora estamos listos para modelizar.

Generación del modelo de Árboles de Decisión

Seguidamente vamos a proceder a realizar la primera regresión, para ello vamos a apoyarnos de la gráfica para seleccionar cuál podría ser el max_depth idóneo y además vamos a utilizar la biblioteca GridSearch, la que nos va a ayudar a determinar cuáles son los mejores parámetros a utilizar en el hipertuneo, así podremos garantizar que nuestro algoritmo utilizará los mejores parámetros posibles.

Al ser un proceso computacional costoso, vamos a hacerlo con solo aquellos paramétros que son más populares y no utilizaremos todos de forma exhaustiva, ya que esto puede llevar a que se dure bastante determinando los mejores parámetros, por ello vamos a utilizar solamente:

1. Splitter
2. Min_samples_leaf
3. Max_features
4. Max_depth
5. Criterion

In [23]:
```python
# División en Train/Test
X = df.iloc[:,0:-1]
y = df.iloc[:,-1]
```

In [24]:
```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X.values, y.values,
                                                    random_state=100, test_size=0.3)
print (X_train.shape, y_train.shape)
print (X_test.shape, y_test.shape)
```

```
(15129, 21) (15129,)
(6484, 21) (6484,)
```

In [25]:
```python
# Importando biblioteca
from sklearn.tree import DecisionTreeRegressor
```

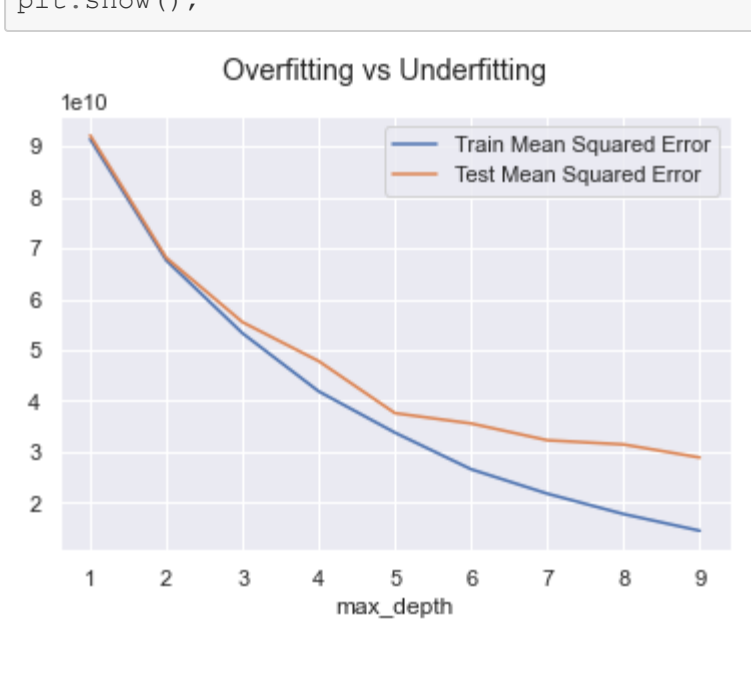
Bucle para la selección de "max_depth"

In [26]:
```python
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

In [27]:
```python
md_list = []
score_train_list = []
score_test_list = []
mse_train_list = []
mse_test_list = []


for maxdepth in list(range(1,10)):
  tree_model = DecisionTreeRegressor(random_state=0, max_depth=maxdepth)
  tree_model.fit(X_train, y_train)

  pred_train = tree_model.predict(X_train)
  pred_test = tree_model.predict(X_test)

  ## Generación de listas
  md_list.append(maxdepth)
  score_train_list.append(100.0*tree_model.score(X_train, y_train))
  score_test_list.append(100.0*tree_model.score(X_test, y_test))
  mse_train_list.append(mean_squared_error(y_train, pred_train))
  mse_test_list.append(mean_squared_error(y_test, pred_test))

df = pd.DataFrame({"max_depth":md_list,
                   "score_train":score_train_list,
                   "score_test":score_test_list,
                   "mse_train":mse_train_list,
                   "mse_test":mse_test_list
                   })
```

In [28]:
```python
df.set_index("max_depth", inplace=True)
```

In [29]:
```python
df["mse_train"].plot(label="Train Mean Squared Error")
df["mse_test"].plot(label="Test Mean Squared Error")
plt.suptitle("Overfitting vs Underfitting")
plt.legend()
plt.show();
```



Se puede ver como en el train (línea de color azul) entre mayor es la cantidad del max depth el error cuadrático medio disminuye notablemente. Sin embargo en cuando al test que es el que verdaderamente nos interesa (color naranja), después de un max_depth de 5, este error cuadrático medio empieza a disminuir considerablemente, pero para mí punto de vista no vale la pena complejizar el modelo agregando más de 5, ya que si nos ponemos a ver con 5 o más de 5 da resultados muy parecidos.

In [30]:
```python
df["score_train"].plot(label="Train Score")
df["score_test"].plot(label="Test Score")
plt.suptitle("R2: Overfitting vs Underfitting")
plt.legend()
plt.show();
```

Si ahora nos centramos en ver el gráfico del R2 (coeficiente de determinación) podemos observar que en el testing cuando estamos en un max_depth de 5, es cuando podemos tener un R2 > 70%, ya que después de este valor más bien el R2 empieza a subir pero muy levemente y se empieza a estabilizar, por ende podríamos utilizar 5 en el hiperparámetro.

In [31]:
```python
# Creando el modelo Decision Tree para utilizar con GridSearch
tree_1= DecisionTreeRegressor(random_state=0)
```

In [32]:
```python
# Creando estimador de parametros
param_grid = {
    'splitter':['best','random'],
    'min_samples_leaf':[1,2,3],
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth' : [1,3,5,7],
    'criterion' :['squared_error', 'friedman_mse', 'absolute_error', 'poisson']
}
```

In [33]:
```python
# Realizando Cross Validation con 3 y probando parametros
CV_1 = GridSearchCV(estimator = tree_1, param_grid=param_grid, cv= 3)
CV_1.fit(X_train, y_train)
```

Out[33]:
```
GridSearchCV(cv=3, estimator=DecisionTreeRegressor(random_state=0),
             param_grid={'criterion': ['squared_error', 'friedman_mse',
                                       'absolute_error', 'poisson'],
                         'max_depth': [1, 3, 5, 7],
                         'max_features': ['auto', 'sqrt', 'log2'],
                         'min_samples_leaf': [1, 2, 3],
                         'splitter': ['best', 'random']})
```

In [34]:
```python
# Comprobando cuales son los mejores parametros a utilizar
CV_1.best_params_
```

Out[34]:
```
{'criterion': 'squared_error',
 'max_depth': 7,
 'max_features': 'auto',
 'min_samples_leaf': 3,
 'splitter': 'best'}
```

Cuando hicimos la prueba para ver cuál podría ser el mejor max_depth determinamos que para arriba de 5 daba buenos resultados y aquí podemos ver que el Grid Search recomienda utilizar 7, así que vamos a utilizar un intermedio, es decir 6 para evitar el overfitting.

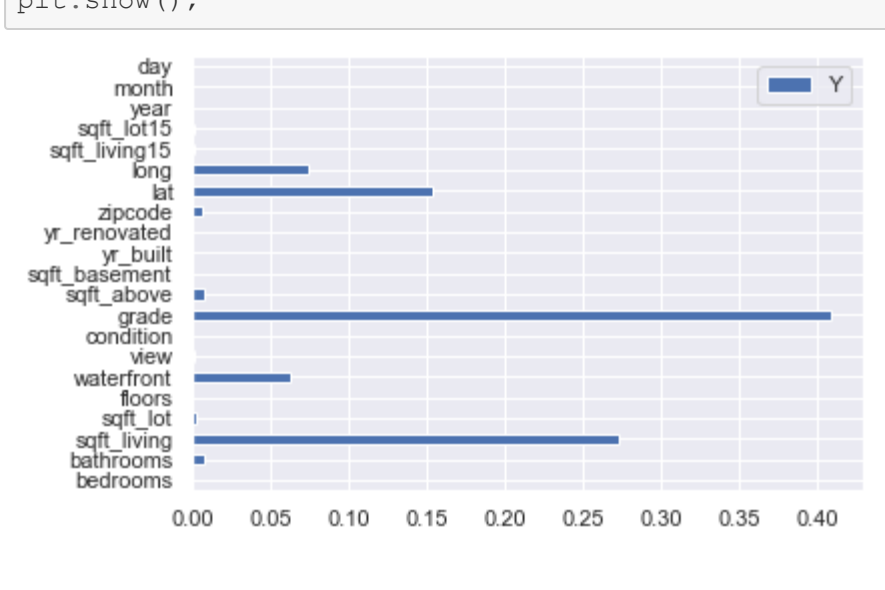
In [35]:
```python
# Entrenamiento y evaluacion del modelo
tree_model = DecisionTreeRegressor(random_state=0, max_depth=6, criterion = 'squared_error', max_featur
es= 'auto', min_samples_leaf= 3, splitter= 'best')
tree_model.fit(X_train, y_train)
pred_train = tree_model.predict(X_train)
print (r2_score(y_train, pred_train))
```

```
0.7988501164948076
```

Podemos observar que obtenemos casi un 80%, este R2 nos da un resultado decente, ahora solo que hay compararlo con el R2 que nos den los otros dos.

Pero este modelo por sí solo es capaz de explicar la variable a predecir.

In [36]:
```python
# Calculo de la importancia de las variables
df_imp = pd.DataFrame(tree_model.feature_importances_.reshape(1,-1), columns=X.columns, index=["Y"])
df_imp.T.plot.barh()
plt.legend()
plt.show();
```



Se puede observar como el modelo le está dando demasiada importancia a la variable "grade" (calificación dada al inmueble, basada en un sistema propio del Condado de King (cuanto mayor es el valor, mejor es el inmueble)) la importancia de dicha variable para el modelo es de más de un 40%.

La segunda más importante es "sqrt_living" (área media de los 15 inmuebles vecinos más cercanos) con más de un 25% de importancia.

Resulta interesante que el modelo esté utilizando variables que tienen que ver con la valoración del inmueble en el mercado, ya que al final el mercado es el que determina el precio de un inmueble o cuánto puede costar.

Generación del modelo de Random Forest

Ahora vamos a proceder a realizar la segunda regresión, para ello vamos a apoyarnos de la gráfica para seleccionar cuál podría ser el max_depth idóneo y además vamos a utilizar la biblioteca GridSearch, la que nos va a ayudar a determinar cuáles son los mejores parámetros a utilizar en el hipertuneo, así podremos garantizar que nuestro algoritmo utilizará los mejores parámetros posibles.

Al ser un proceso computacional costoso, vamos a hacerlo con solo aquellos paramétros que son más populares y no utilizaremos todos de forma exhaustiva, ya que esto puede llevar a que se dure bastante determinando los mejores parámetros, por ello vamos a utilizar solamente:

1. Min_samples_leaf
2. Max_features
3. Max_depth
4. Criterion
5. Min_samples_split

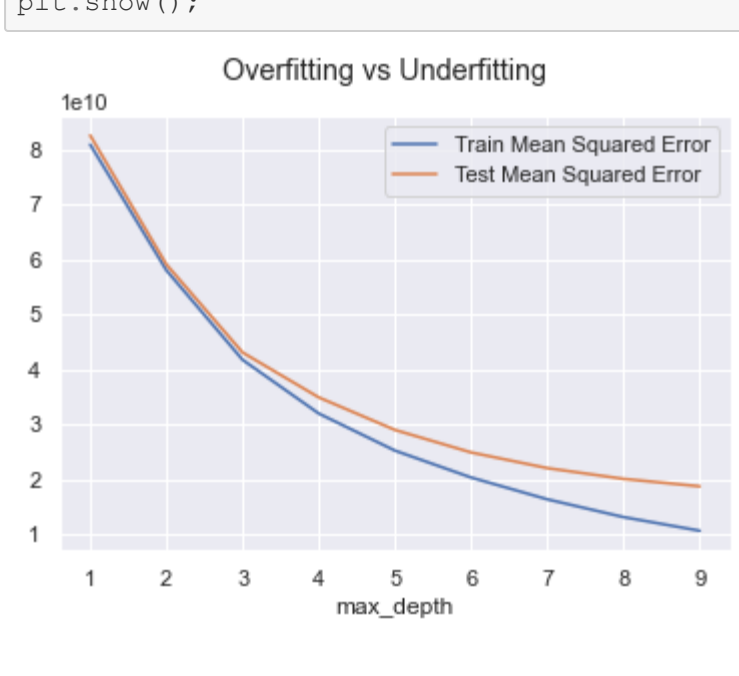
In [37]:
```python
# Importando biblioteca
from sklearn.ensemble import RandomForestRegressor
```

Bucle para la selección de "max_depth"

In [38]:
```python
md_list = []
score_train_list = []
score_test_list = []
mse_train_list = []
mse_test_list = []


for maxdepth in list(range(1,10)):
  random_model = RandomForestRegressor(random_state=0, max_depth=maxdepth)
  random_model.fit(X_train, y_train)

  pred_train = random_model.predict(X_train)
  pred_test = random_model.predict(X_test)

  ## Generación de listas
  md_list.append(maxdepth)
  score_train_list.append(100.0*random_model.score(X_train, y_train))
  score_test_list.append(100.0*random_model.score(X_test, y_test))
  mse_train_list.append(mean_squared_error(y_train, pred_train))
  mse_test_list.append(mean_squared_error(y_test, pred_test))

df = pd.DataFrame({"max_depth":md_list,
                   "score_train":score_train_list,
                   "score_test":score_test_list,
                   "mse_train":mse_train_list,
                   "mse_test":mse_test_list
                   })
```

In [39]:
```python
df.set_index("max_depth", inplace=True)
```

In [40]:
```python
df["mse_train"].plot(label="Train Mean Squared Error")
df["mse_test"].plot(label="Test Mean Squared Error")
plt.suptitle("Overfitting vs Underfitting")
plt.legend()
plt.show();
```



Se puede ver como en el train (línea de color azul) entre mayor es la cantidad del max depth el error cuadrático medio disminuye notablemente (muy parecido al modelo anterior). Sin embargo en cuando al test que es el que verdaderamente nos interesa (color naranja), después de un max_depth de 5, este error cuadrático medio empieza a disminuir considerablemente, pero para mi punto de vista no vale la pena complejizar el modelo agregando más de 5, ya que si nos ponemos a ver con 5 o más de 5 da resultados muy parecidos.

In [41]:
```python
df["score_train"].plot(label="Train Score")
df["score_test"].plot(label="Test Score")
plt.suptitle("R2: Overfitting vs Underfitting")
plt.legend()
plt.show();
```

En cunto al gráfico del R2 (coeficiente de determinación) podemos observar que en el testing cuando estamos en un max_depth de 5, es cuando podemos tener un R2 > 80%, ya que después de este valor más bien el R2 empieza a subir pero muy levemente y se empieza a estabilizar, por ende podríamos utilizar 5 en el hiperparámetro.

In [42]:
```python
# Creando el modelo Random Forest para utilizar con GridSearch
random_tree = RandomForestRegressor(random_state=0)
```

In [43]:
```python
# Creando estimador de parametros
param_grid = {
    'min_samples_leaf':[1,2],
    'min_samples_split': [2, 3],
    'max_features': ['sqrt', 'log2'],
    'max_depth' : [3,5],
    'criterion':['squared_error', 'absolute_error']
}
```

In [44]:
```python
# Realizando Cross Validation con 2 (para que no tome mucho tiempo) y probando parametros
CV_2 = GridSearchCV(estimator = random_tree, param_grid = param_grid, cv= 2)
CV_2.fit(X_train, y_train)
```

Out[44]:
```
GridSearchCV(cv=2, estimator=RandomForestRegressor(random_state=0),
             param_grid={'criterion': ['squared_error', 'absolute_error'],
                         'max_depth': [3, 5], 'max_features': ['sqrt', 'log2'],
                         'min_samples_leaf': [1, 2],
                         'min_samples_split': [2, 3]})
```

In [45]:
```python
# Comprobando cuales son los mejores parametros a utilizar
CV_2.best_params_
```

Out[45]:
```
{'criterion': 'squared_error',
 'max_depth': 5,
 'max_features': 'sqrt',
 'min_samples_leaf': 2,
 'min_samples_split': 2}
```

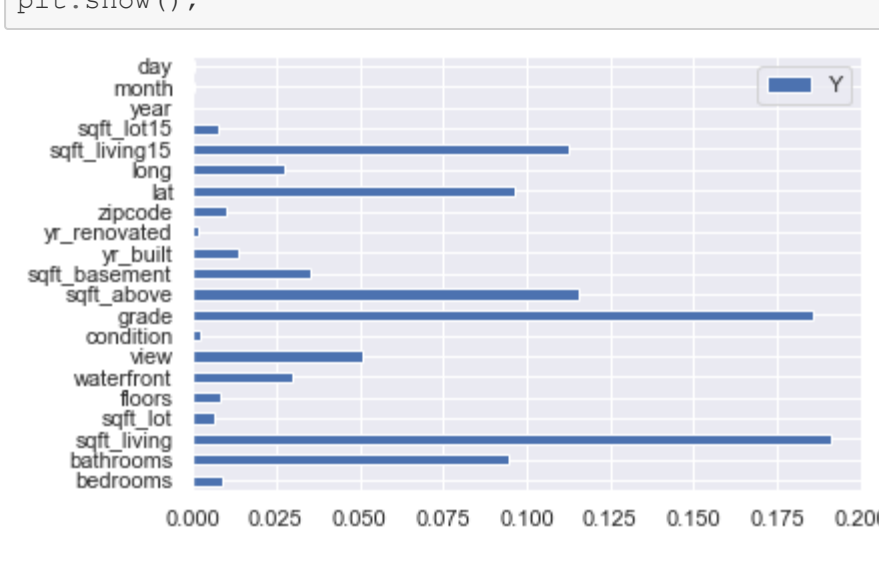
Tal y como habíamos podido calcular anteriormente, el modelo está tomando un max_depth de 5, por ende también vamos a utilizar dicho parámetro en nuestro modelo final.

In [46]:
```python
# Entrenamiento y evaluacion del modelo
random_model = RandomForestRegressor(random_state=0, max_depth=5, criterion='squared_error',
                                     min_samples_split=2, min_samples_leaf =2, max_features= 'sqrt')
random_model.fit(X_train, y_train)
pred_train = random_model.predict(X_train)
print (r2_score(y_train, pred_train))
```

```
0.7662712939860395
```

Podemos observar que obtenemos casi un 77% en cuanto a nuestro R2, lo que lo hace bastante similar al modelo anterior, este también podría decirse que es un R2 aceptable para predecir.

In [47]:
```python
# Calculo de la importancia de las variables
df_imp = pd.DataFrame(random_model.feature_importances_.reshape(1,-1), columns=X.columns, index=["Y"])
df_imp.T.plot.barh()
plt.legend()
plt.show();
```



Se puede observar como el modelo le está dando casi la misma importancia a la variable "grade", ya que la importancia de dicha variable para el modelo es de más de 17.5%, pero ahora le dá un poco más de importancia a la variable "sqrt_living" con casi un 20%.

Nuevamente el modelo está utilizando variables que tienen que ver con la valoración del inmueble en el mercado.

Generación del modelo de Gradient Boosting

Ahora vamos a proceder a realizar la tercera regresión, para ello vamos a apoyarnos de la gráfica para seleccionar cuál podría ser el max_depth idóneo y además vamos a utilizar la biblioteca GridSearch, la que nos va a ayudar a determinar cuáles son los mejores parámetros a utilizar en el hipertuneo, así podremos garantizar que nuestro algoritmo utilizará los mejores parámetros posibles.

Al ser un proceso computacional costoso, vamos a hacerlo con solo aquellos paramétros que son más populares y no utilizaremos todos de forma exhaustiva, ya que esto puede llevar a que se dure bastante determinando los mejores parámetros, por ello vamos a utilizar solamente:

1. Learning_rate
2. Subsample
3. Loss
4. N_estimators
5. Max_Depth

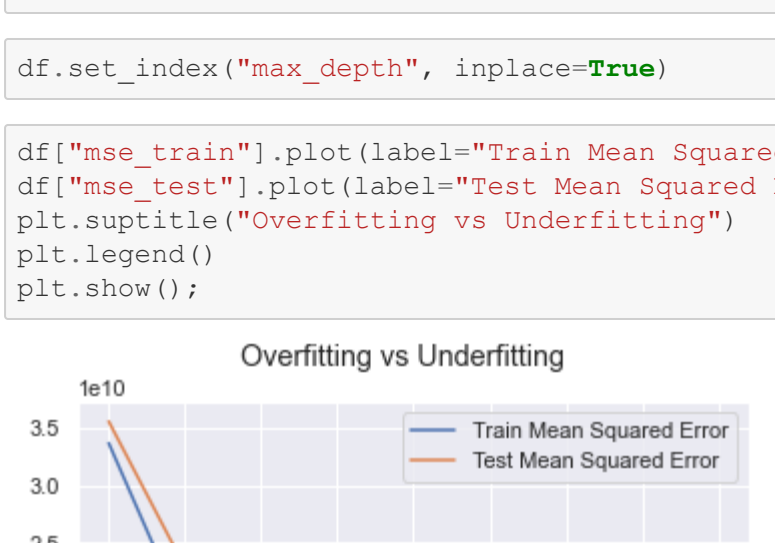
In [48]:
```
# Importando la biblioteca
from sklearn.ensemble import GradientBoostingRegressor
```

Bucle para la selección de "max_depth"

In [49]:
```
md_list = []
score_train_list = []
score_test_list = []
mse_train_list = []
mse_test_list = []


for maxdepth in list(range(1,10)):
  gradient_model = GradientBoostingRegressor(random_state=0, max_depth=maxdepth)
  gradient_model.fit(X_train, y_train)

  pred_train = gradient_model.predict(X_train)
  pred_test = gradient_model.predict(X_test)

  ## Generación de listas
  md_list.append(maxdepth)
  score_train_list.append(100.0*gradient_model.score(X_train, y_train))
  score_test_list.append(100.0*gradient_model.score(X_test, y_test))
  mse_train_list.append(mean_squared_error(y_train, pred_train))
  mse_test_list.append(mean_squared_error(y_test, pred_test))

df = pd.DataFrame({"max_depth":md_list,
                   "score_train":score_train_list,
                   "score_test":score_test_list,
                   "mse_train":mse_train_list,
                   "mse_test":mse_test_list
                   })
```

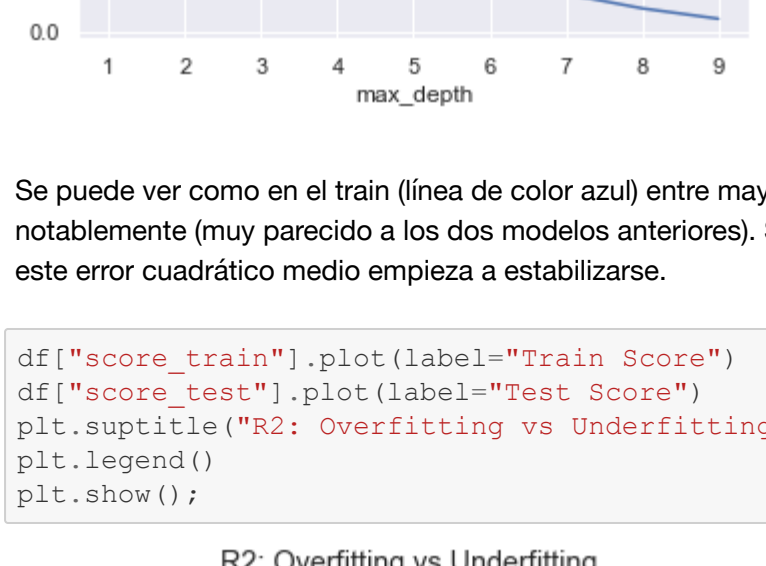
In [50]:
```
df.set_index("max_depth", inplace=True)
```

In [51]:
```
df["mse_train"].plot(label="Train Mean Squared Error")
df["mse_test"].plot(label="Test Mean Squared Error")
plt.suptitle("Overfitting vs Underfitting")
plt.legend()
plt.show();
```



Se puede ver como en el train (línea de color azul) entre mayor es la cantidad del max depth el error cuadrático medio disminuye notablemente (muy parecido a los dos modelos anteriores). Sin embargo en cuando al test (color naranja), después de un max_depth de 5, este error cuadrático medio empieza a estabilizarse.

In [52]:
```
df["score_train"].plot(label="Train Score")
df["score_test"].plot(label="Test Score")
plt.suptitle("R2: Overfitting vs Underfitting")
plt.legend()
plt.show();
```



En cunto al gráfico del R2 (coeficiente de determinación) podemos observar que en el testing cuando estamos en un max_depth entre 3 y 5, es cuando podemos tener un R2 > 90%, ya que después de este valor más bien el R2 empieza a subir pero muy levemente y se estabiliza, por ende podríamos usar un max_depth = 5.

In [53]:
```
# Creando el modelo Gradient Boosting para utilizar con GridSearch
Gradient = GradientBoostingRegressor(random_state=0)
```

In [54]:
```
# Creando estimador de parametros
param_grid = {
    'learning_rate': [0.01, 0.04],
    'subsample'    : [1.0, 0.1],
    'n_estimators' : [50,75],
    'max_depth'    : [5,7],
    'loss': ['squared_error', 'absolute_error', 'huber', 'quantile']
}
```

In [55]:
```
# Realizando Cross Validation con 2 y probando parametros
CV_3 = GridSearchCV(estimator = Gradient, param_grid = param_grid, cv= 2)
CV_3.fit(X_train, y_train)
```

Out[55]:
```
GridSearchCV(cv=2, estimator=GradientBoostingRegressor(random_state=0),
             param_grid={'learning_rate': [0.01, 0.04],
                         'loss': ['squared_error', 'absolute_error', 'huber',
                                  'quantile'],
                         'max_depth': [5, 7], 'n_estimators': [50, 75],
                         'subsample': [1.0, 0.1]})
```

In [ ]:
```
# Comprobando cuales son los mejores parametros a utilizar
CV_3.best_params_
```

Out[ ]:
```
{'learning_rate': 0.04,
 'loss': 'squared_error',
 'max_depth': 7,
 'n_estimators': 75,
 'subsample': 0.1}
```

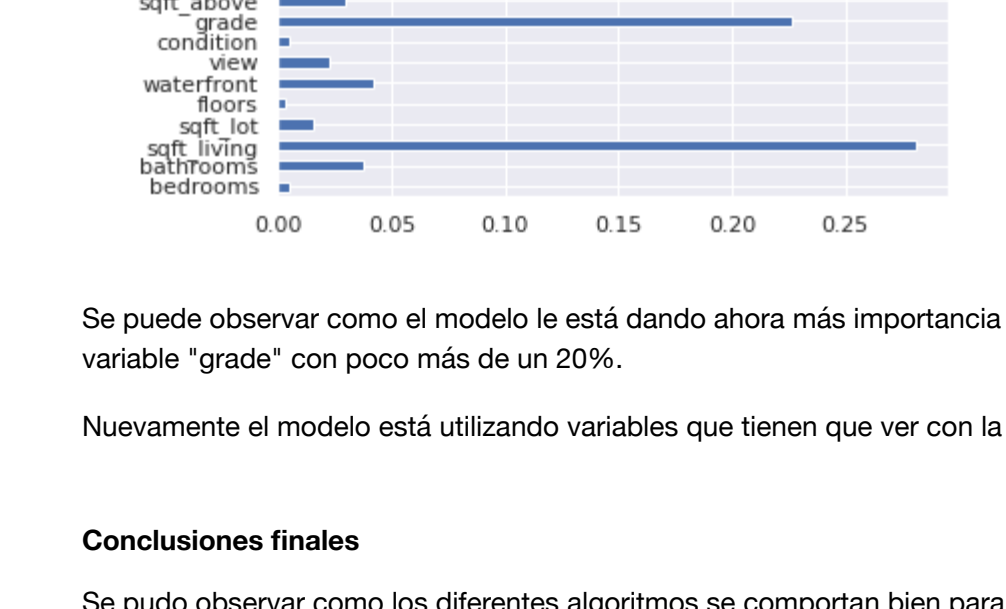
Podemos observar como obtenemos un max_depth en este caso de 7, mientras que en el análisis anterior fue de 5, entonces vamos a utilizar un intermedio, es decir, 6 para evitar el overfitting.

In [ ]:
```
# Entrenamiento y evaluacion del modelo
gradient_model = GradientBoostingRegressor(random_state=0, n_estimators=75, subsample = 0.1, loss ='squ
ared_error', max_depth=6, learning_rate=0.04)
gradient_model.fit(X_train, y_train)
pred_train = gradient_model.predict(X_train)
print (r2_score(y_train, pred_train))
```

```
0.8819661028897025
```

Se puede observar como este modelo nos da bastante buenos resultados ya que obtenemos un 88% en cuanto al R2 lo que lo convierte en el mejor modelo de los 3 realizados.

In [ ]:
```
# Calculo de la importancia de las variables
df_imp = pd.DataFrame(gradient_model.feature_importances_.reshape(1,-1), columns=X.columns, index=["Y"
])
df_imp.T.plot.barh()
plt.legend()
plt.show();
```



Se puede observar como el modelo le está dando ahora más importancia a la variable "sqrt_living" con más de un 25% y se segunda a la variable "grade" con poco más de un 20%.

Nuevamente el modelo está utilizando variables que tienen que ver con la valoración del inmueble en el mercado.

Conclusiones finales

Se pudo observar como los diferentes algoritmos se comportan bien para predecir el precio de venta final de los inmuebles. Para este caso notamos que hay una mejoría bastante considerable cuando se usa el algoritmo de Gradient Boosting vs el Árbol de Decisión tradicional, ya que entre uno y otro hay casi una diferencia de un 10%.

El Random Forest también provee resultados decentes para el tipo de tarea de pedicción que se está ejecutando pero no tan buenos como el Gradient.

También podemos observar como el uso de la biblioteca de GridSearch nos puede ayudar en la tarea de predicción pero aumenta considerablemente el tiempo de la ejecución.