ACTIVIDAD 1 - ÁRBOLES Y RANDOM FOREST PARA REGRESIÓN Y CLASIFICACIÓN

Integrantes del grupo de trabajo:

- Javier Blasco
- Daniel Rodríguez
- Gregorio Ferrer
- Albert Marquillas

1. Preparación del entorno

Importación de librerias

In [1]:
```
import os
import warnings
warnings.filterwarnings("ignore")

import pandas as pd
import numpy as np

# Representacion y graficos
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn import tree
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.model_selection import KFold, train_test_split, StratifiedKFold, GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, mean_squared_log_error, confusion_
accuracy_score, roc_auc_score, f1_score, recall_score
```

Cargar el dataframe

In [2]:
```
df = pd.read_csv("housing_train.csv").drop(columns=["Id"])
```

2. Análisis desctiptivo de los datos

Variables categoricas y numéricas

Separamos primero las diferentes variables del dataset en categóricas y numéricas observando su tipo y las asignamos a dos nuevas listas de datos, después ambas listas se muestran por pantalla.

In [3]:
```
numericos = [f for f in df.columns if df.dtypes[f] != 'object']
numericos.remove('SalePrice')
categoricos = [f for f in df.columns if df.dtypes[f] == 'object']

print("Datos categoricos: {}".format(categoricos))
print()
print("Datos numericos: {}".format(numericos))
```

```
Datos categoricos: ['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfi
g', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle', 'RoofStyle', 'R
oofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQua
l', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'Heating', 'HeatingQC', 'CentralAir',
'Electrical', 'KitchenQual', 'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature', 'SaleType', 'SaleCondition']

Datos numericos: ['MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond', 'YearBuilt',
'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2nd
FlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath', 'BedroomA
bvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodD
eckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal', 'MoSold',
'YrSold']
```

Datos estadísticos de las variables numéricas

De las variables numéricas se muestran los distintos datos de media, mediana, minimo, máximo y cuartiles.

In [4]:
```
df.describe()
```

Out[4]:

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF
count	1460.000000	1201.000000	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000	1452.000000	1460.00000
mean	56.897260	70.049958	10516.828082	6.099315	5.575342	1971.267808	1984.865753	103.685262	443.63972
std	42.300571	24.284752	9981.264932	1.382997	1.112799	30.202904	20.645407	181.066207	456.09809
min	20.000000	21.000000	1300.000000	1.000000	1.000000	1872.000000	1950.000000	0.000000	0.00000
25%	20.000000	59.000000	7553.500000	5.000000	5.000000	1954.000000	1967.000000	0.000000	0.00000
50%	50.000000	69.000000	9478.500000	6.000000	5.000000	1973.000000	1994.000000	0.000000	383.50000
75%	70.000000	80.000000	11601.500000	7.000000	6.000000	2000.000000	2004.000000	166.000000	712.25000
max	190.000000	313.000000	215245.000000	10.000000	9.000000	2010.000000	2010.000000	1600.000000	5644.00000

8 rows × 37 columns

Datos de las variables categóricas

Para las variables categóricas se listan a continuación la frecuencia de cada clase.

In [5]:
```
for categoria in categoricos:
    print(categoria)
    print(pd.value_counts(df[categoria]))
    print()
```

```
MSZoning
RL         1151
RM          218
FV           65
RH           16
C (all)      10
Name: MSZoning, dtype: int64

Street
Pave    1454
Grvl       6
Name: Street, dtype: int64

Alley
Grvl    50
Pave    41
Name: Alley, dtype: int64

LotShape
Reg    925
IR1    484
IR2     41
IR3     10
Name: LotShape, dtype: int64

LandContour
Lvl    1311
Bnk      63
HLS      50
Low      36
Name: LandContour, dtype: int64

Utilities
AllPub    1459
NoSeWa       1
Name: Utilities, dtype: int64

LotConfig
Inside     1052
Corner      263
CulDSac      94
FR2          47
FR3           4
Name: LotConfig, dtype: int64

LandSlope
Gtl    1382
Mod      65
Sev      13
Name: LandSlope, dtype: int64

Neighborhood
NAmes      225
CollgCr    150
```

```
OldTown    113
Edwards    100
Somerst     86
Gilbert     79
NridgHt     77
Sawyer      74
NWAmes      73
SawyerW     59
BrkSide     58
Crawfor     51
Mitchel     49
NoRidge     41
Timber      38
IDOTRR      37
ClearCr     28
SWISU       25
StoneBr     25
Blmngtn     17
MeadowV     17
BrDale      16
Veenker     11
NPkVill      9
Blueste      2
Name: Neighborhood, dtype: int64

Condition1
Norm      1260
Feedr       81
Artery      48
RRAn        26
PosN        19
RRAe        11
PosA         8
RRNn         5
RRNe         2
Name: Condition1, dtype: int64

Condition2
Norm      1445
Feedr        6
PosN         2
Artery       2
RRNn         2
RRAn         1
RRAe         1
PosA         1
Name: Condition2, dtype: int64

BldgType
1Fam      1220
TwnhsE     114
Duplex      52
Twnhs       43
2fmCon      31
Name: BldgType, dtype: int64

HouseStyle
1Story    726
2Story    445
1.5Fin    154
SLvl       65
SFoyer     37
1.5Unf     14
2.5Unf     11
2.5Fin      8
Name: HouseStyle, dtype: int64

RoofStyle
Gable      1141
Hip         286
Flat         13
Gambrel      11
Mansard       7
Shed          2
Name: RoofStyle, dtype: int64

RoofMatl
CompShg    1434
Tar&Grv      11
WdShngl       6
WdShake       5
Metal         1
Membran       1
Roll          1
ClyTile       1
```

```
Name: RoofMatl, dtype: int64

Exterior1st
VinylSd    515
HdBoard    222
MetalSd    220
Wd Sdng    206
Plywood    108
CemntBd     61
BrkFace     50
WdShing     26
Stucco      25
AsbShng     20
Stone        2
BrkComm      2
CBlock       1
AsphShn      1
ImStucc      1
Name: Exterior1st, dtype: int64

Exterior2nd
VinylSd    504
MetalSd    214
HdBoard    207
Wd Sdng    197
Plywood    142
CmentBd     60
Wd Shng     38
Stucco      26
BrkFace     25
AsbShng     20
ImStucc     10
Brk Cmn      7
Stone        5
AsphShn      3
Other        1
CBlock       1
Name: Exterior2nd, dtype: int64

MasVnrType
None       864
BrkFace    445
Stone      128
BrkCmn      15
Name: MasVnrType, dtype: int64

ExterQual
TA    906
Gd    488
Ex     52
Fa     14
Name: ExterQual, dtype: int64

ExterCond
TA    1282
Gd     146
Fa      28
Ex       3
Po       1
Name: ExterCond, dtype: int64

Foundation
PConc     647
CBlock    634
BrkTil    146
Slab       24
Stone       6
Wood        3
Name: Foundation, dtype: int64

BsmtQual
TA    649
Gd    618
Ex    121
Fa     35
Name: BsmtQual, dtype: int64

BsmtCond
TA    1311
Gd      65
Fa      45
Po       2
Name: BsmtCond, dtype: int64

BsmtExposure
```

```
No    953
Av    221
Gd    134
Mn    114
Name: BsmtExposure, dtype: int64

BsmtFinType1
Unf    430
GLQ    418
ALQ    220
BLQ    148
Rec    133
LwQ     74
Name: BsmtFinType1, dtype: int64

BsmtFinType2
Unf    1256
Rec      54
LwQ      46
BLQ      33
ALQ      19
GLQ      14
Name: BsmtFinType2, dtype: int64

Heating
GasA     1428
GasW       18
Grav        7
Wall        4
OthW        2
Floor       1
Name: Heating, dtype: int64

HeatingQC
Ex    741
TA    428
Gd    241
Fa     49
Po      1
Name: HeatingQC, dtype: int64

CentralAir
Y    1365
N      95
Name: CentralAir, dtype: int64

Electrical
SBrkr    1334
FuseA      94
FuseF      27
FuseP       3
Mix         1
Name: Electrical, dtype: int64

KitchenQual
TA    735
Gd    586
Ex    100
Fa     39
Name: KitchenQual, dtype: int64

Functional
Typ     1360
Min2      34
Min1      31
Mod       15
Maj1      14
Maj2       5
Sev        1
Name: Functional, dtype: int64

FireplaceQu
Gd    380
TA    313
Fa     33
Ex     24
Po     20
Name: FireplaceQu, dtype: int64

GarageType
Attchd     870
Detchd     387
BuiltIn     88
Basment     19
CarPort      9
```

```
2Types       6
Name: GarageType, dtype: int64

GarageFinish
Unf    605
RFn    422
Fin    352
Name: GarageFinish, dtype: int64

GarageQual
TA    1311
Fa      48
Gd      14
Ex       3
Po       3
Name: GarageQual, dtype: int64

GarageCond
TA    1326
Fa      35
Gd       9
Po       7
Ex       2
Name: GarageCond, dtype: int64

PavedDrive
Y    1340
N      90
P      30
Name: PavedDrive, dtype: int64

PoolQC
Gd    3
Ex    2
Fa    2
Name: PoolQC, dtype: int64

Fence
MnPrv    157
GdPrv     59
GdWo      54
MnWw      11
Name: Fence, dtype: int64

MiscFeature
Shed    49
Othr     2
Gar2     2
TenC     1
Name: MiscFeature, dtype: int64

SaleType
WD       1267
New       122
COD        43
ConLD       9
ConLI       5
ConLw       5
CWD         4
Oth         3
Con         2
Name: SaleType, dtype: int64

SaleCondition
Normal     1198
Partial     125
Abnorml     101
Family       20
Alloca       12
AdjLand       4
Name: SaleCondition, dtype: int64
```

Correlaciones de los datos

Primero de todo encontramos la matriz de correlación con todas las variables.

In [6]:
```
df.corr(method="pearson")
```

Out[6]:

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinS
MSSubClass	1.000000	-0.386347	-0.139781	0.032628	-0.059316	0.027850	0.040581	0.022936	-0.0698
LotFrontage	-0.386347	1.000000	0.426095	0.251646	-0.059213	0.123349	0.088866	0.193458	0.2336

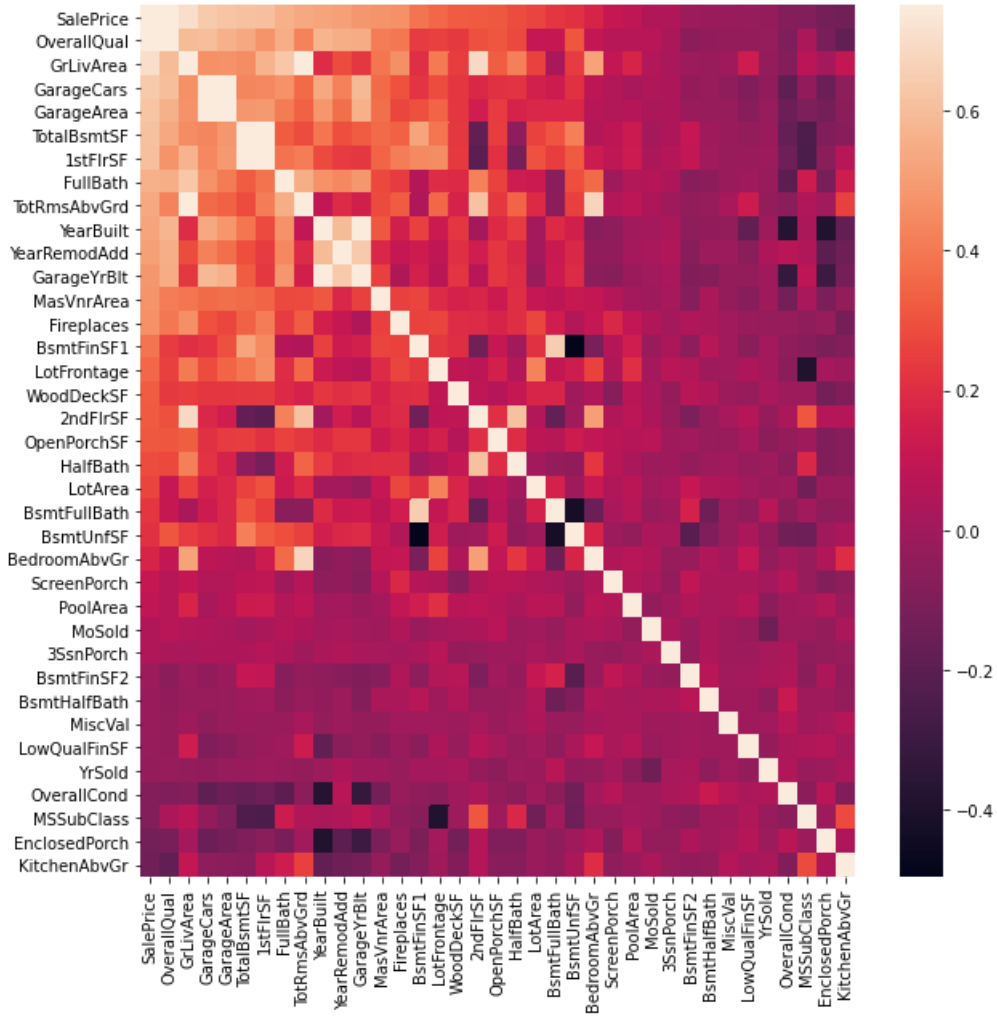
	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinS
LotArea	-0.139781	0.426095	1.000000	0.105806	-0.005636	0.014228	0.013788	0.104160	0.214
OverallQual	0.032628	0.251646	0.105806	1.000000	-0.091932	0.572323	0.550684	0.411876	0.239
OverallCond	-0.059316	-0.059213	-0.005636	-0.091932	1.000000	-0.375983	0.073741	-0.128101	-0.046
YearBuilt	0.027850	0.123349	0.014228	0.572323	-0.375983	1.000000	0.592855	0.315707	0.249
YearRemodAdd	0.040581	0.088866	0.013788	0.550684	0.073741	0.592855	1.000000	0.179618	0.128
MasVnrArea	0.022936	0.193458	0.104160	0.411876	-0.128101	0.315707	0.179618	1.000000	0.264
BsmtFinSF1	-0.069836	0.233633	0.214103	0.239666	-0.046231	0.249503	0.128451	0.264736	1.000
BsmtFinSF2	-0.065649	0.049900	0.111170	-0.059119	0.040229	-0.049107	-0.067759	-0.072319	-0.050
BsmtUnfSF	-0.140759	0.132644	-0.002618	0.308159	-0.136841	0.149040	0.181133	0.114442	-0.495
TotalBsmtSF	-0.238518	0.392075	0.260833	0.537808	-0.171098	0.391452	0.291066	0.363936	0.522
1stFlrSF	-0.251758	0.457181	0.299475	0.476224	-0.144203	0.281986	0.240379	0.344501	0.445
2ndFlrSF	0.307886	0.080177	0.050986	0.295493	0.028942	0.010308	0.140024	0.174561	-0.137
LowQualFinSF	0.046474	0.038469	0.004779	-0.030429	0.025494	-0.183784	-0.062419	-0.069071	-0.064
GrLivArea	0.074853	0.402797	0.263116	0.593007	-0.079686	0.199010	0.287389	0.390857	0.208
BsmtFullBath	0.003491	0.100949	0.158155	0.111098	-0.054942	0.187599	0.119470	0.085310	0.649
BsmtHalfBath	-0.002333	-0.007234	0.048046	-0.040150	0.117821	-0.038162	-0.012337	0.026673	0.067
FullBath	0.131608	0.198769	0.126031	0.550600	-0.194149	0.468271	0.439046	0.276833	0.058
HalfBath	0.177354	0.053532	0.014259	0.273458	-0.060769	0.242656	0.183331	0.201444	0.004
BedroomAbvGr	-0.023438	0.263170	0.119690	0.101676	0.012980	-0.070651	-0.040581	0.102821	-0.107
KitchenAbvGr	0.281721	-0.006069	-0.017784	-0.183882	-0.087001	-0.174800	-0.149598	-0.037610	-0.081
TotRmsAbvGrd	0.040380	0.352096	0.190015	0.427452	-0.057583	0.095589	0.191740	0.280682	0.044
Fireplaces	-0.045569	0.266639	0.271364	0.396765	-0.023820	0.147716	0.112581	0.249070	0.260
GarageYrBlt	0.085072	0.070250	-0.024947	0.547766	-0.324297	0.825667	0.642277	0.252691	0.153
GarageCars	-0.040110	0.285691	0.154871	0.600671	-0.185758	0.537850	0.420622	0.364204	0.224
GarageArea	-0.098672	0.344997	0.180403	0.562022	-0.151521	0.478954	0.371600	0.373066	0.296
WoodDeckSF	-0.012579	0.088521	0.171698	0.238923	-0.003334	0.224880	0.205726	0.159718	0.204
OpenPorchSF	-0.006100	0.151972	0.084774	0.308819	-0.032589	0.188686	0.226298	0.125703	0.111
EnclosedPorch	-0.012037	0.010700	-0.018340	-0.113937	0.070356	-0.387268	-0.193919	-0.110204	-0.102
3SsnPorch	-0.043825	0.070029	0.020423	0.030371	0.025504	0.031355	0.045286	0.018796	0.026
ScreenPorch	-0.026030	0.041383	0.043160	0.064886	0.054811	-0.050364	-0.038740	0.061466	0.062
PoolArea	0.008283	0.206167	0.077672	0.065166	-0.001985	0.004950	0.005829	0.011723	0.140
MiscVal	-0.007683	0.003368	0.038068	-0.031406	0.068777	-0.034383	-0.010286	-0.029815	0.003
MoSold	-0.013585	0.011200	0.001205	0.070815	-0.003511	0.012398	0.021490	-0.005965	-0.015
YrSold	-0.021407	0.007450	-0.014261	-0.027347	0.043950	-0.013618	0.035743	-0.008201	0.014
SalePrice	-0.084284	0.351799	0.263843	0.790982	-0.077856	0.522897	0.507101	0.477493	0.386

37 rows × 37 columns

Para poder identificar mejor las variables correlacionadas entre si, se ha realizado un mapa de calor, donde los valores de correlación mayores a 0.75 son mostrados en blanco. Como el valor de SalePrice es la variable de interés para el problema, se ha puesto como primera variable, y se han ordenado las otras a partir de esta.

Columnas con porcentaje de valores nulos mas elevados

In [7]:
```
train = df
columns = df.corr(method="pearson")["SalePrice"].sort_values(ascending=False)[:].index #:15 o dejamos

corr = train[columns].corr(method="pearson")

plt.figure(figsize=(10,10))
sns.heatmap(corr, vmax=0.75);
```



Para tener más seguridad antes de eliminar los datos, tambien se muestran los 3 valores máximos para cada variable de correlacion, ignorando la correlación de valor 1 de la misma variable.

In [8]:
```
for column in columns:
  temp = df.corr(method="pearson")[column].sort_values(ascending=False)[1:4]
  print("{}: {}".format(column,temp))
  print("##########################################")
```

```
SalePrice: OverallQual    0.790982
GrLivArea      0.708624
GarageCars     0.640409
Name: SalePrice, dtype: float64
##########################################
OverallQual: SalePrice     0.790982
GarageCars    0.600671
GrLivArea     0.593007
Name: OverallQual, dtype: float64
##########################################
GrLivArea: TotRmsAbvGrd    0.825489
SalePrice       0.708624
2ndFlrSF        0.687501
Name: GrLivArea, dtype: float64
##########################################
GarageCars: GarageArea     0.882475
SalePrice      0.640409
OverallQual    0.600671
Name: GarageCars, dtype: float64
##########################################
GarageArea: GarageCars     0.882475
SalePrice      0.623431
GarageYrBlt    0.564567
Name: GarageArea, dtype: float64
##########################################
TotalBsmtSF: 1stFlrSF       0.819530
SalePrice      0.613581
OverallQual    0.537808
Name: TotalBsmtSF, dtype: float64
##########################################
1stFlrSF: TotalBsmtSF    0.819530
SalePrice      0.605852
```

```
GrLivArea      0.566024
Name: 1stFlrSF, dtype: float64
##############################################
FullBath: GrLivArea       0.630012
SalePrice       0.560664
TotRmsAbvGrd    0.554784
Name: FullBath, dtype: float64
##############################################
TotRmsAbvGrd: GrLivArea       0.825489
BedroomAbvGr    0.676620
2ndFlrSF        0.616423
Name: TotRmsAbvGrd, dtype: float64
##############################################
YearBuilt: GarageYrBlt     0.825667
YearRemodAdd    0.592855
OverallQual     0.572323
Name: YearBuilt, dtype: float64
##############################################
YearRemodAdd: GarageYrBlt    0.642277
YearBuilt      0.592855
OverallQual    0.550684
Name: YearRemodAdd, dtype: float64
##############################################
GarageYrBlt: YearBuilt       0.825667
YearRemodAdd    0.642277
GarageCars      0.588920
Name: GarageYrBlt, dtype: float64
##############################################
MasVnrArea: SalePrice      0.477493
OverallQual    0.411876
GrLivArea      0.390857
Name: MasVnrArea, dtype: float64
##############################################
Fireplaces: SalePrice    0.466929
GrLivArea    0.461679
1stFlrSF     0.410531
Name: Fireplaces, dtype: float64
##############################################
BsmtFinSF1: BsmtFullBath    0.649212
TotalBsmtSF     0.522396
1stFlrSF        0.445863
Name: BsmtFinSF1, dtype: float64
##############################################
LotFrontage: 1stFlrSF     0.457181
LotArea      0.426095
GrLivArea    0.402797
Name: LotFrontage, dtype: float64
##############################################
WoodDeckSF: SalePrice      0.324413
GrLivArea      0.247433
OverallQual    0.238923
Name: WoodDeckSF, dtype: float64
##############################################
2ndFlrSF: GrLivArea       0.687501
TotRmsAbvGrd    0.616423
HalfBath        0.609707
Name: 2ndFlrSF, dtype: float64
##############################################
OpenPorchSF: GrLivArea      0.330224
SalePrice      0.315856
OverallQual    0.308819
Name: OpenPorchSF, dtype: float64
##############################################
HalfBath: 2ndFlrSF        0.609707
GrLivArea       0.415772
TotRmsAbvGrd    0.343415
Name: HalfBath, dtype: float64
##############################################
LotArea: LotFrontage    0.426095
1stFlrSF       0.299475
Fireplaces     0.271364
Name: LotArea, dtype: float64
##############################################
BsmtFullBath: BsmtFinSF1     0.649212
TotalBsmtSF    0.307351
1stFlrSF       0.244671
Name: BsmtFullBath, dtype: float64
##############################################
BsmtUnfSF: TotalBsmtSF    0.415360
1stFlrSF       0.317987
OverallQual    0.308159
Name: BsmtUnfSF, dtype: float64
##############################################
BedroomAbvGr: TotRmsAbvGrd    0.676620
GrLivArea       0.521270
```

```
2ndFlrSF        0.502901
Name: BedroomAbvGr, dtype: float64
##########################################
ScreenPorch: Fireplaces    0.184530
SalePrice     0.111447
GrLivArea     0.101510
Name: ScreenPorch, dtype: float64
##########################################
PoolArea: LotFrontage    0.206167
GrLivArea      0.170205
BsmtFinSF1     0.140491
Name: PoolArea, dtype: float64
##########################################
MoSold: OpenPorchSF    0.071255
OverallQual    0.070815
FullBath       0.055872
Name: MoSold, dtype: float64
##########################################
3SsnPorch: LotFrontage     0.070029
1stFlrSF        0.056104
YearRemodAdd    0.045286
Name: 3SsnPorch, dtype: float64
##########################################
BsmtFinSF2: BsmtFullBath    0.158678
LotArea         0.111170
TotalBsmtSF     0.104810
Name: BsmtFinSF2, dtype: float64
##########################################
BsmtHalfBath: OverallCond    0.117821
BsmtFinSF2     0.070948
BsmtFinSF1     0.067418
Name: BsmtHalfBath, dtype: float64
##########################################
MiscVal: OverallCond     0.068777
KitchenAbvGr    0.062341
LotArea         0.038068
Name: MiscVal, dtype: float64
##########################################
LowQualFinSF: GrLivArea       0.134683
TotRmsAbvGrd    0.131185
BedroomAbvGr    0.105607
Name: LowQualFinSF, dtype: float64
##########################################
YrSold: BsmtFullBath    0.067049
OverallCond     0.043950
YearRemodAdd    0.035743
Name: YrSold, dtype: float64
##########################################
OverallCond: BsmtHalfBath     0.117821
YearRemodAdd     0.073741
EnclosedPorch    0.070356
Name: OverallCond, dtype: float64
##########################################
MSSubClass: 2ndFlrSF        0.307886
KitchenAbvGr    0.281721
HalfBath        0.177354
Name: MSSubClass, dtype: float64
##########################################
EnclosedPorch: OverallCond     0.070356
2ndFlrSF        0.061989
LowQualFinSF    0.061081
Name: EnclosedPorch, dtype: float64
##########################################
KitchenAbvGr: MSSubClass      0.281721
TotRmsAbvGrd    0.256045
BedroomAbvGr    0.198597
Name: KitchenAbvGr, dtype: float64
##########################################
```

Juntado el mapa de calor con los datos obtenidos en este punto, podemos eliminar ciertas variables por su correlación.

La primera que hemos decidido eliminar es la de 'GarageArea', esta tiene alta correlación con la variable de 'GarageCars', además lógicamente el área del garage está muy ligada al número de coches que pueden haber, y se elimina la de área porque son valores mucho mayores que la del número de coches que son valores más acotados.

Por otro lado, la variable de 'OverallQual' tiene una alta correlación con la de 'SalePrice' que es la variable a predecir, y está con ciertas operaciones puede llevar a encontrar directamente el valor esperado del precio, cosa que no es interesante, ya que un modelo entrenado de esta forma, daría malas predicciones si faltara la calidad general de la casa, por este motivo también se elimina.

In [9]:
```
#Eliminamos variables con alta correlacion y con sentido
```

```
df.drop(["GarageArea","OverallQual"],axis=1,inplace=True)
```

3. Tratamiento de missing

El primer paso realizado ha sido identificar el porcentaje de valores faltantes en las distintas columnas y mostrar los más altos.

In [10]:
```
(((df.isnull().sum()/ len(df)).sort_values(ascending=False)[:10])*100)
```

Out[10]:
```
PoolQC          99.520548
MiscFeature     96.301370
Alley           93.767123
Fence           80.753425
FireplaceQu     47.260274
LotFrontage     17.739726
GarageCond       5.547945
GarageYrBlt      5.547945
GarageFinish     5.547945
GarageQual       5.547945
dtype: float64
```

Como se puede observar, existen un total de 6 variables con valores faltantes mayores al 10%, como se considera que es suficientemente grande la falta de datos, estas columnas se eliminan para evitar añadir error en las predicciones debido a añadir demasiados datos no reales.

In [11]:
```
df.drop(["PoolQC","MiscFeature","Alley","Fence","FireplaceQu","LotFrontage","LotFrontage"],axis=1,inp
```

El siguiente paso realizado en el tratamiento de missing trata de analizar las variables que tienen información sobre el garage.

Con los porcentajes de error de los atributos relacionados con el garaje se puede ver como todas las variables de garaje tienen el mismo valor de nulos y además observando los datos estos coinciden, en este caso se considera que no existe garaje y por lo tanto se reemplazan los valores NaN con una categoría adicional denominada "None".

Para comprobar que esta suposición es cierta solo es necesario comparar si el número de garages con capacidad para 0 coches equivale a el número de garajes con las variables faltantes.

In [12]:
```
df['GarageType'].isnull().sum()==(df['GarageCars']==0).sum() #Comparacion
```

Out[12]:
```
True
```

Como es cierta la comparación se puede proceder a aplicar el reemplazamiento mencionado.

In [13]:
```
# NaN en garage implica que no hay garage
df["GarageYrBlt"].fillna("None",inplace=True)
df["GarageType"].fillna("None",inplace=True)
df["GarageFinish"].fillna("None",inplace=True)
df["GarageCond"].fillna("None",inplace=True)
df["GarageQual"].fillna("None",inplace=True)
```

Se van a tratar también con cierto detalle las otras variables que tienen valores missing para poder obtener una mejor predicción. A continuación se muestran las variables que aún tienen valores missing:

In [14]:
```
(((df.isnull().sum()/ len(df)).sort_values(ascending=False)[:10])*100)
```

Out[14]:
```
BsmtFinType2    2.602740
BsmtExposure    2.602740
BsmtQual        2.534247
BsmtFinType1    2.534247
BsmtCond        2.534247
MasVnrType      0.547945
MasVnrArea      0.547945
Electrical      0.068493
ExterQual       0.000000
Exterior1st     0.000000
dtype: float64
```

Un tipo de variables que son muy parecidas a las de garaje son las del sótano. Aunque para estas se hará un análisis, ya que las de BsmtFinType2 y BsmtExposure tienen más variables nulas que las otras correspondientes a sótano. Para ello se muestran todas las filas con valores NaN correspondientes a columnas relacionadas con el sótano.

In [15]:
```
df_basement=df[["BsmtQual","BsmtCond","BsmtExposure","BsmtFinType1","BsmtFinSF1","BsmtFinType2","BsmtF

df_basement[df_basement.isnull().any(axis=1)]
```

Out[15]:

	BsmtQual	BsmtCond	BsmtExposure	BsmtFinType1	BsmtFinSF1	BsmtFinType2	BsmtFinSF2	BsmtUnfSF	BsmtFullBath	B

	BsmtQual	BsmtCond	BsmtExposure	BsmtFinType1	BsmtFinSF1	BsmtFinType2	BsmtFinSF2	BsmtUnfSF	BsmtFullBath	B
17	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
39	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
90	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
102	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
156	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
182	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
259	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
332	Gd	TA	No	GLQ	1124	NaN	479	1603	1	
342	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
362	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
371	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
392	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
520	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
532	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
533	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
553	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
646	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
705	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
736	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
749	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
778	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
868	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
894	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
897	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
948	Gd	TA	NaN	Unf	0	Unf	0	936	0	
984	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1000	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1011	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1035	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1045	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1048	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1049	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1090	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1179	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1216	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1218	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1232	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1321	NaN	NaN	NaN	NaN	0	NaN	0	0	0	
1412	NaN	NaN	NaN	NaN	0	NaN	0	0	0	

Modificamos el dataset de forma correcta directamente ya que son solo 2 variables para tratar directamente en conjunto de BsmtFinType2 y BsmtExposure donde se les asigna la categoría específica de 'No', ya que tienen sótano pero se considera que no disponen de esa característica y por lo tanto debe ser marcado como tal.

In [16]:
```
#Reemplazar variables concretas
df['BsmtFinType2'][332]='No'
df['BsmtExposure'][948]='No'
```

En el resto de casos, donde el sótano no está entrado se asume que no tienen y por lo tanto las variables relacionadas con el sótano se van a tratar como las de garaje, reemplazando los NaN por un nuevo tipo llamado 'None' como en el caso de los

garajes.

In [17]:
```
# Los valores faltantes en las variables Bsmnt implican que no hay ningun sotano.
df["BsmtFinType2"].fillna("None",inplace=True)
df["BsmtExposure"].fillna("None",inplace=True)
df["BsmtQual"].fillna("None",inplace=True)
df["BsmtFinType1"].fillna("None",inplace=True)
df["BsmtCond"].fillna("None",inplace=True)
```

En este punto ya se han corregido la mayoría de valores missing, pero aún faltan unos pocos. Los próximos que se van a tratar son los relacionados con la mampostería.

En este caso no hay ninguna relación que pueda indicar el tipo de material ni el área relacionadas con la mampostería, por lo tanto, se decide asumir que si los valores faltan no hay mampostería, y por lo tanto se reemplazan los missing por el tipo llamado "None" y el área se considera que es 0, ya que al no existir mampostería tampoco habría ninguna área a tener en cuenta para ello.

In [18]:
```
# Las variables con valores nulo en la maposteria impilcan que probablemente no hay maposteria, por lo
df["MasVnrType"].fillna("None",inplace=True)
df["MasVnrArea"].fillna(0.0,inplace=True)
```

En este punto solo queda el atributo llamado "Electrical", en este caso solo es una única variable, y observando como se reparten los distintos tipos se puede ver que la gran mayoría es del tipo SBrkr y por lo tanto asignamos este tipo a la variable faltante.

In [19]:
```
# El valor restante en electricidad, hemos puesto el sistema estandard pues es el mas comun
df["Electrical"].fillna("SBrkr", inplace=True)
```

Por último se revisa que no exista ningún valor del tipo NaN en ninguna de las variables.

In [20]:
```
(((df.isnull().sum()/ len(df)).sort_values(ascending=False)[:10])*100)
```

Out[20]:
```
SalePrice        0.0
SaleCondition    0.0
Exterior1st      0.0
Exterior2nd      0.0
MasVnrType       0.0
MasVnrArea       0.0
ExterQual        0.0
ExterCond        0.0
Foundation       0.0
BsmtQual         0.0
dtype: float64
```

4. Problema de regresión

El problema de regresión es el primero que se va a afrontar. En este caso se realizará la predicción de la variable de 'SalePrice' con árbol de decisión y con random forest para poder comparar su desempeño.

En este punto, sería interesante realizar una normalización de los datos para mejorar los resultados, ya que algunas de las variables y sobretodo la variable de interés tienen valores altos. Un punto importante es que esta normalización requiere de que se pueda deshacer para comprobar los resultados de las predicciones. En este caso se ha decidido no realizarla, ya que el punto de la actividad es comprar los árboles de decisión frente a los random forest, y sin hacer la normalización ya se puede realizar.

In [21]:
```
df.iloc[:,:-1]
```

Out[21]:

	MSSubClass	MSZoning	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	...	OpenP
0	60	RL	8450	Pave	Reg	Lvl	AllPub	Inside	Gtl	CollgCr	...	
1	20	RL	9600	Pave	Reg	Lvl	AllPub	FR2	Gtl	Veenker	...	
2	60	RL	11250	Pave	IR1	Lvl	AllPub	Inside	Gtl	CollgCr	...	
3	70	RL	9550	Pave	IR1	Lvl	AllPub	Corner	Gtl	Crawfor	...	
4	60	RL	14260	Pave	IR1	Lvl	AllPub	FR2	Gtl	NoRidge	...	
...	...	...	...	...	...	...	...	...	...	...	...	
1455	60	RL	7917	Pave	Reg	Lvl	AllPub	Inside	Gtl	Gilbert	...	
1456	20	RL	13175	Pave	Reg	Lvl	AllPub	Inside	Gtl	NWAmes	...	
1457	70	RL	9042	Pave	Reg	Lvl	AllPub	Inside	Gtl	Crawfor	...	

	MSSubClass	MSZoning	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	...	OpenP
1458	20	RL	9717	Pave	Reg	Lvl	AllPub	Inside	Gtl	NAmes	...	
1459	20	RL	9937	Pave	Reg	Lvl	AllPub	Inside	Gtl	Edwards	...	

1460 rows × 71 columns

In [22]:
```
X = df.iloc[:,:-1]
y = df["SalePrice"]
```

Se crea una funcion para mostrar los valores de los distintos errores que tendra la predicción realizada en el problema de regresión.

In [23]:
```
def regresion(X_train, X_test, y_train, y_test, regresor, metricas_error_off=False):
    regresor.fit(X_train, y_train)

    predicciones = regresor.predict(X_test)

    if not(metricas_error_off):
        print(f"MSE: {mean_squared_error(y_test, predicciones)}")
        print(f"MAE: {mean_absolute_error(y_test, predicciones)}")
        print(f"RMSE: {mean_squared_error(y_test, predicciones, squared=False)}")
        print(f"RMSLE: {np.sqrt(mean_squared_log_error(y_test, predicciones))}")

    return predicciones
```

En python, para entrenar un modelo para un problema de regresión no se pueden utilizar variables cualitativas, como estas son de interés, se realiza una función con la que transformar estas variables cualitativas en variables numéricas. Para ello se crean nuevas columnas con cada posible valor de las variables cualitativas y en cada fila se indica de forma booleana si la clase esa corresponde o no a esa fila. Una cosa a remarcar es que en la conversión se elimina una columna de cada una de las cualitativas, ya que el no hacerlo podría dar lugar a problemas.

In [24]:
```
def cual2num(df_,feature):
    df = df_.copy(deep=True)
    dummy = pd.get_dummies(df[feature])
    dummy = dummy.drop(dummy.columns[-1],axis=1)
    df = pd.concat([df,dummy],axis=1)
    return df.drop(feature, axis=1)

#Se obtienen los nuevos datos catagoricos despues de eliminar las distintas columnas
new_categoricos = [column for column in df.columns if df.dtypes[column] == 'object']
```

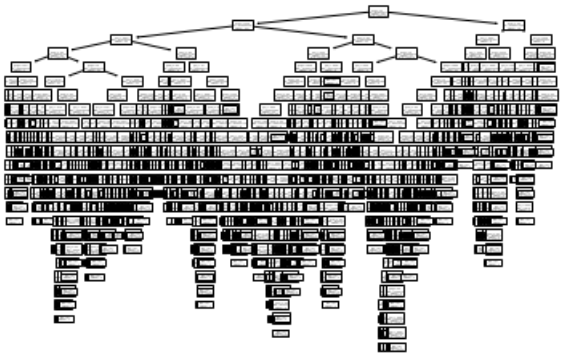
En este punto se separan los conjuntos de datos de entrenamiento de los de test, para ello se utiliza un 80% de los datos para el entrenamiento y el 20% restante corresponden a los de test. Estos datos se van a usar tento para el árbol de decisión como para el random forest, de esta forma se evitará cualquier possible cambio en el error de predicción debido a la separación de los datos.

In [25]:
```
X_train, X_test, y_train, y_test = train_test_split(cual2num(X,new_categoricos), y, test_size = 0.2 ,r
```

Árbol de decisión

Primero de todo, se crea el árbol de decisión completo y se muestra su representación gráfica para comprovar si es necesario hacer la poda.

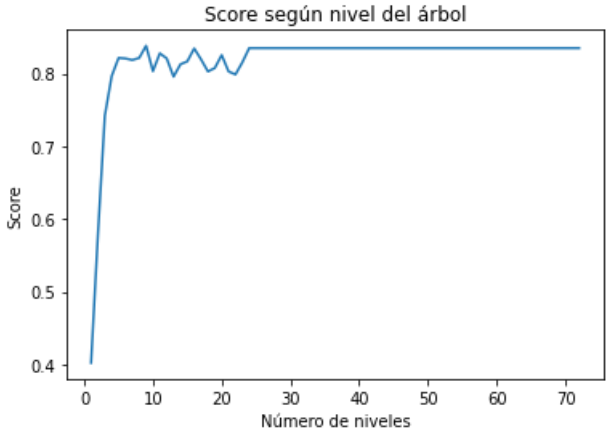
In [26]:
```
dt = tree.DecisionTreeRegressor(random_state=42)

fig = plt.figure()
dt.fit(X_train, y_train)
tree.plot_tree(dt);
```



Solo con observar la imagen del árbol ya se puede ver que es necesario podarlo porque es demasiado grande y esto provocaría overfitting.

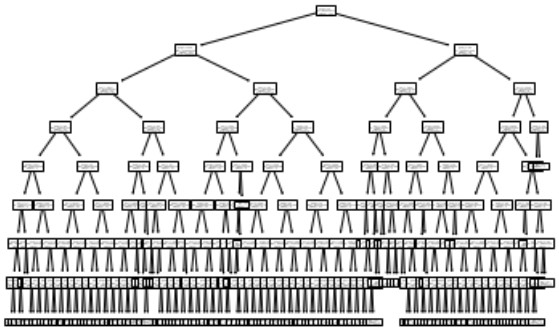
Para realizar la poda del árbol, primero se tiene que obtener la profundidad más óptima para el árbol, para ello se representa en una gráfica el score del árbol obtenido según los niveles usados para entrenarlo.

In [27]:
```
scores = []
num_variables = len(list(df))
n_niveles = range(1, num_variables + 1)
for n in n_niveles:
    dt.set_params(max_depth = n)
    dt.fit(X_train, y_train)
    scores.append(dt.score(X_test, y_test))

plt.title("Score según nivel del árbol")
plt.xlabel("Número de niveles")
plt.ylabel("Score")
plt.plot(n_niveles, scores)
plt.show()
```



Como indica la comprobación, el valor de profundidad igual a 8 es donde se obtiene el valor máximo de score y se usa este valor para crear de nuevo el árbol podado con profundidad de valor 8 niveles. Aunque no es estable hasta la profundidad cercana a 28, el nivel 8 tiene el mismo puntuaje, por lo tanto el árbol tendrá prácticamente la misma efectividad aunque el resultado pueda ser un poco peor con según que datos de test, pero dejándolo con 28 niveles el árbol es difícilmente interpretable.

In [28]:
```
dt = tree.DecisionTreeRegressor(random_state=42, max_depth = 8)

fig = plt.figure()
dt.fit(X_train, y_train)
tree.plot_tree(dt);
```



Se puede observar como el árbol ha mejorado bastante y es mucho más interpretable, este árbol ya sirve para realizar predicciones genéricas.

En este caso se muestran los distintos errores de predicción para poder evaluar lo bueno que es el modelo entrenado.

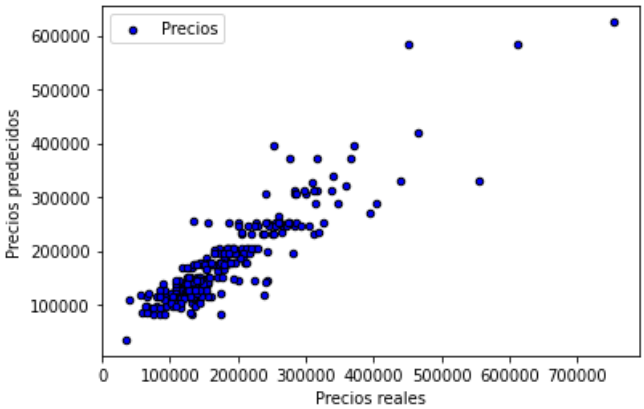
In [29]:
```
predicciones=regresion(X_train, X_test, y_train, y_test, dt);
```

```
MSE: 1369737813.635676
MAE: 23660.747644963725
RMSE: 37009.96911152016
RMSLE: 0.20069722958436367
```

Como se puede apreciar, los errores tienen valores bastante grandes aparte del RMSLE, esto es debido a que los precios de las casas son valores muy altos, y un error que en estos rangos puede ser proporcionalmente pequeño, hace que sean valores muy altos.

Para comparar se utilizará la métrica del error medio cuadrático (MSE). En este caso tiene un valor relativamente grande, ya que tratando con precios de las viviendas, este error aumenta en gran cantidad. Para comprobar lo correcto que es el árbol, se representa el gráfico de dispersión de los valores reales contra las predicciones.

In [30]:
```
# Generar plot basado en la distribución de datos de prueba y datos de predicción
def dibujar_dispersion(real, predicted):
    plt.scatter(real, predicted, s=20, edgecolor="black", c="blue", label="Precios")
    plt.xlabel("Precios reales")
    plt.ylabel("Precios predecidos")
    plt.legend()
    plt.show()

dibujar_dispersion(y_test, predicciones)
```



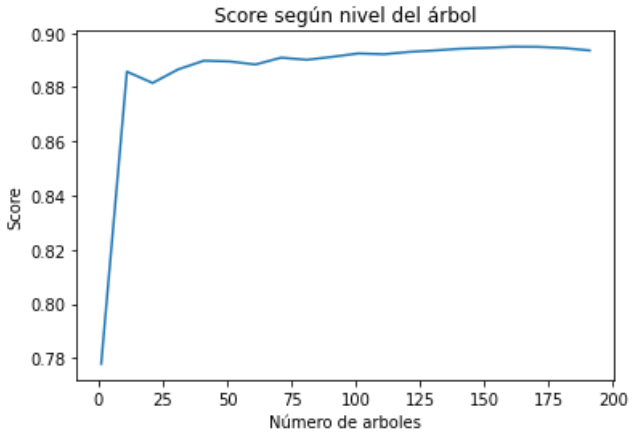
Como se puede ver, los valores se encuentran bastante centrados en la diagonal, por lo tanto se confirma que el modelo de arbol es correcto.

Random forest

Para realizar la comparación entre el modelo de árbol de decisión y el de random forest, se crean distintos bosques utilizando los mismos conjuntos de entrenamiento y test que para el árbol de decisión y cambiando el número de árboles entre ellos para encontrar el valor más óptimo.

In [31]:
```
rf = RandomForestRegressor(random_state=42, max_features='auto')

scores = []
for n in np.arange(1,200,10):
    rf.set_params(n_estimators=n)
```

```
    rf.fit(X_train, y_train)
    scores.append(rf.score(X_test, y_test))

plt.title("Score según nivel del árbol")
plt.xlabel("Número de arboles")
plt.ylabel("Score")
plt.plot(np.arange(1,200,10), scores)
plt.show()
```



Como se puede observar, el valor de 50 árboles es el que tiene valor más optimo para utilizar ya que se acerca al máximo de score y tiene un número reducido de arboles, por lo tanto se crea el bosque con este número para ser comparado con los resultados del árbol de decisión.

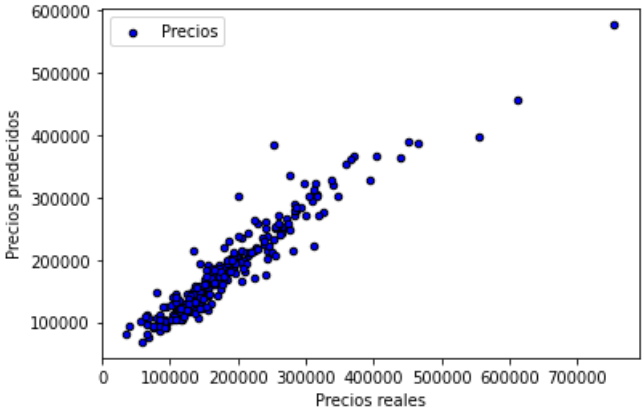
In [32]:
```
rf = RandomForestRegressor(n_estimators = 50, random_state=42, max_features='auto')
predicciones_rf = regresion(X_train, X_test, y_train, y_test, rf);
```

```
MSE: 851929445.8549521
MAE: 17241.877602739725
RMSE: 29187.83044104087
RMSLE: 0.15706842421917117
```

Como se ha comentado para el arbol, se usará la metrica de MSE para analizar el error del bosque y comparar este con el arbol de decisión.

Por un lado, observando el valor de la metrica de error MSE se puede ver que tiene un valor alto de unos 852 millones, pero en este rango de valores es normal tener estos números como se ha comentado anteriormente. Para ello se repersenta la gráfica de comparación como se ha realizado para el árbol.

In [33]:
```
dibujar_dispersion(y_test, predicciones_rf)
```



Por otro lado, haciendo la comparación entre el error del árbol de decisión y el del random forest, el valor de MSE es bastante menor en el bosque siendo este de unos 500 millones menos y por lo tanto, en un caso real se utilizaría un random forest, ya que predice con más precisión. Además se puede ver como en el bosque los datos están mucho más centrados en la diagonal, por lo tanto a simple vista ya indica que es mejor. Como añadido a la comparación con la métrica MSE, el resto de variables de error también salen menores con los random forest.

Aunque los dos modelos son válidos, y un punto a favor que tiene el árbol es que se puede interpretar viendo la imagen, no como el bosque que es un modelo de caja negra. Por lo tanto según el caso, se podría llegar a seleccionar el árbol por delante del bosque.

5. Problema de clasificación

Este es el segundo problema que se va a afrontar. Para este caso, se hará una clasificación de los precios en tres clases distintas:

- Grupo 1: SalePrice menor o igual a 100000.
- Grupo 2: SalePrice entre 100001 y 500000.
- Grupo 3: SalePrice mayor o igual a 500000.

Primero de todo, se crea una función parecida a la utilizada en la regresión, la cual a partir de los datos de resultado, se obtienen las métricas de error correspondientes a la clasificación.

In [34]:
```
def clasificacion(X_train, X_test, y_train, y_test, clasificador, prints_off=False):
    clasificador.fit(X_train, y_train)

    predicciones = clasificador.predict(X_test)

    if not(prints_off):
        print(f"Exactitud: {accuracy_score(y_test, predicciones)}")
        #print(f"Recall: {recall_score(y_test, predicciones, average='weighted')}")
        print(f"F1_Score: {f1_score(y_test, predicciones, average='weighted')}")
        print(f"Matriz de confusion:\n {confusion_matrix(y_test, predicciones)}")

    return predicciones
```

Un paso totalmente indispensable es preparar las categorias mencionadas para la variable. Para ello creamos una nueva columna la cual asignamos la clase en la que se encuentra cada edificio en venta.

In [35]:
```
df_clf = df.copy(deep=True)

df_clf["Categoria"] = 1
df_clf.loc[(df_clf["SalePrice"] > 100000) & (df_clf["SalePrice"] < 500000), "Categoria"] = 2
df_clf.loc[df_clf["SalePrice"] >= 500000, "Categoria"] = 3
```

Se muestra la distribución de las clases.

In [36]:
```
df_clf.Categoria.value_counts()
```

Out[36]:
```
2    1328
1     123
3       9
Name: Categoria, dtype: int64
```

Como se puede observar, la clase 3, tiene un número muy reducido de variables, y por lo tanto se tendrá que trabajar con cuidado en la división del conjunto de entrenamiento y test, ya que si en el conjunto de entrenamiento no existe la clase 3, el algoritmo no podrá predecir la clase 3 en ningún caso y se obtendrán errores.

Se elimina la columna de 'SalePrice' ya que si se dejara esta daria sobreajuste en el modelo.

In [37]:
```
df_clf.drop("SalePrice", axis=1, inplace=True)
```

Se dividen los conjuntos de entrenamiento y test, utilizando un 80% de los datos para entrenamiento y el 20% restante para el test. En este caso como el tercer grupo es muy reducido dividimos los datos en 5 folds y así se asegura de que para el conjunto de entrenamiento exista la clase 3.

In [38]:
```
df_clf.sample(frac=1).reset_index(drop=True)
df_clf["kfold"] = -1
y = df_clf.Categoria.values
kf = StratifiedKFold(5, shuffle=True)
for fold, (x, y) in enumerate(kf.split(X=df_clf, y=y)):
    df_clf.loc[y, "kfold"] = fold
```

Se comprueva que en los distintos folds hay la misma proporción de cada clase.

In [39]:
```
for fold in range(0,5):
    print(f"Fold: {fold}")
    print(df_clf[df_clf["kfold"] == fold].Categoria.value_counts())
    print()
```

```
Fold: 0
2    266
1     25
```

```
3      1
Name: Categoria, dtype: int64

Fold: 1
2    266
1     24
3      2
Name: Categoria, dtype: int64

Fold: 2
2    266
1     24
3      2
Name: Categoria, dtype: int64

Fold: 3
2    265
1     25
3      2
Name: Categoria, dtype: int64

Fold: 4
2    265
1     25
3      2
Name: Categoria, dtype: int64
```

A continuación se realiza la separación de los datos, en la que se convierten los datos categóricos a numéricos como se ha realizado para la regresión.

In [40]:
```
new_numericos = [column for column in df_clf.drop(["Categoria","kfold"],axis=1).columns if df_clf.dtyp
new_categoricos = [column for column in df_clf.drop(["Categoria","kfold"],axis=1).columns if df_clf.dt
df_ = cual2num(df_clf,new_categoricos)

X_train = df_[df_.kfold != 1].drop(["Categoria","kfold"],axis=1)
y_train = df_[df_.kfold != 1].Categoria
X_test = df_[df_.kfold == 1].drop(["Categoria","kfold"],axis=1)
y_test = df_[df_.kfold == 1].Categoria
```

Se asegura de que en los datos de entrenamiento existen elementos de la tercera clase.

In [41]:
```
y_train.value_counts()
```

Out[41]:
```
2    1062
1      99
3       7
Name: Categoria, dtype: int64
```

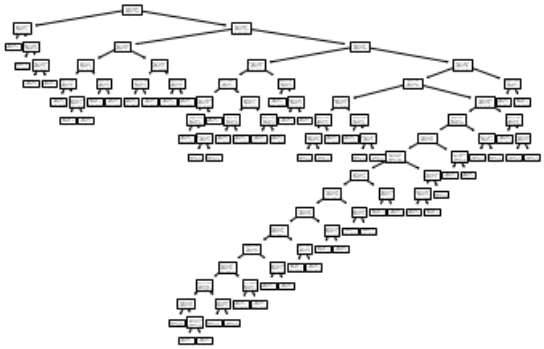
Como se puede ver, existen valores de las 3 clases en el conjunto de entrenamiento, por lo tanto, se continua con esta división de los datos.

Cabe mencionar, que al realizar la separación de los datos de entranamiento y de test con este metodo y sin realizar un metodo de validación k-fold completo, cada vez que se repita la comprovación, puede haber cierta varianza en los resultados del arbol de decisión y del random forest, ya que no simpre se va a entrenar con los mismos datos, pero aún así, se pueden extraer las mismas conclusiones.

Árbol de decisión

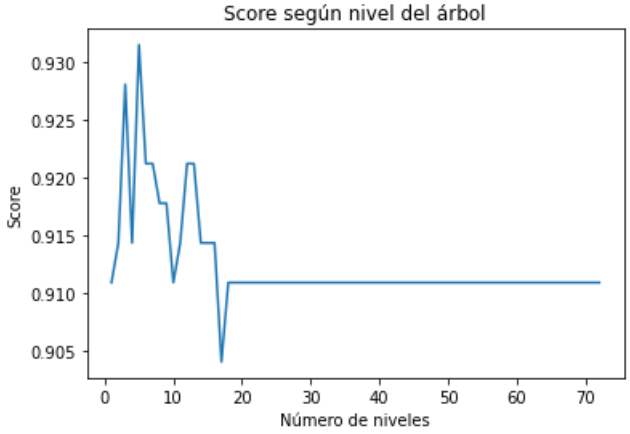
Primero de todo creamos y mostramos el árbol de decisión para evaluar si podarlo o no.

In [42]:
```
dt_clf = DecisionTreeClassifier(random_state=42)

fig = plt.figure()
dt_clf.fit(X_train, y_train)
tree.plot_tree(dt_clf);
```



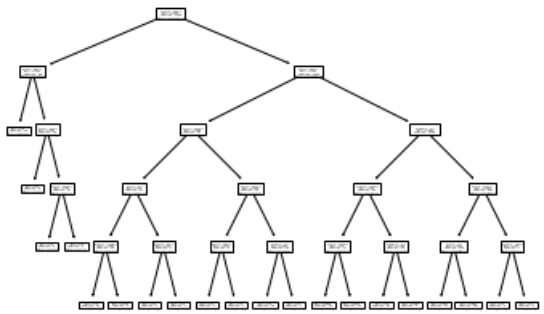
Como se puede ver, el árbol queda ya muy reducido, se podria seguir con el arbol sin podar, pero se decide buscar con que profundidad da mejores resultados para comparar su mejor versión con los bosques.

In [43]:
```
scores = []
num_variables = len(list(df))
n_niveles = range(1, num_variables + 1)


for n in n_niveles:
    dt_clf.set_params(max_depth = n)
    dt_clf.fit(X_train, y_train)
    scores.append(dt_clf.score(X_test, y_test))

plt.title("Score según nivel del árbol")
plt.xlabel("Número de niveles")
plt.ylabel("Score")
plt.plot(n_niveles, scores)
plt.show()
```



Con una profundidad de 5 se obtiene el mejor valor para el arbol, por lo tanto en este caso se va a usar este valor de profundidad, aunque no se estabilize hasta una profundidad cercana a los 20 niveles.

In [51]:
```
dt_clf = DecisionTreeClassifier(max_depth = 5, random_state=42)

fig = plt.figure()
dt_clf.fit(X_train, y_train)
tree.plot_tree(dt_clf);
```



In [52]:
```
predicciones = clasificacion(X_train, X_test, y_train, y_test, dt_clf);
```

```
Exactitud: 0.9315068493150684
F1_Score: 0.9241864173371022
Matriz de confusion:
 [[ 12  12   0]
 [  6 260   0]
 [  0   2   0]]
```

Las métricas de error salen muy buenas, con una exactitud del 0.932%, la matriz de confusión marca que los datos para la clase 2 son buenos, para la clase 1 acierta bastante, pero para la tercera clase no tiene buena clasificación debido a que esa tercera clase contiene muy pocas muestras y no marca ningún dato como cierto, si se hubiese usado un árbol de profundidad mayor, es posible haber podido predecir algún valor de la clase 3 gracias a tener más profundidad con una clase tan reducida, pero por culpa del overfitting se hubiera perdido efectividad en las otras dos clases.

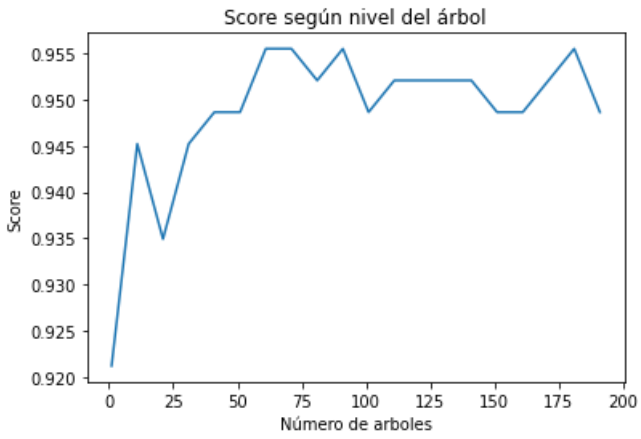
Random forests

Para los bosques se va a seguir el mismo proceso que para los árboles. Para evitar añadir error por la separación de los datos, se va a usar exactamente la misma que para el arbol de decisión.

In [46]:
```
rf_clf = RandomForestClassifier(random_state=42)
```

Se grafican los resultados de exactitud con diferentes números de árboles en el random forest.

In [47]:
```
values = []
for n in np.arange(1,200,10):
    rf_clf.set_params(n_estimators=n)
    rf_clf.fit(X_train, y_train)
    values.append(rf_clf.score(X_test, y_test))

plt.title("Score según nivel del árbol")
plt.xlabel("Número de arboles")
plt.ylabel("Score")
plt.plot(np.arange(1,200,10), values)
plt.show()
```



Los mejores valores se obtienen en el valor de 60 árboles, por lo tanto se usa este número para el random forest.

In [53]:
```
rf_clf = RandomForestClassifier(n_estimators = 60, random_state=42)
predicciones = clasificacion(X_train, X_test, y_train, y_test, rf_clf);
```

```
Exactitud: 0.952054794520548
F1_Score: 0.9437515903982356
Matriz de confusion:
 [[ 13  11   0]
 [  1 265   0]
 [  0   2   0]]
```

En este caso se obtiene un valor de exactitud de 0.952, que es un valor muy bueno y esto indica que el bosque predecirá con gran acierto. Observando la matriz de confusión, se puede ver que para la clase 3 tampoco hay verdaderos positivos, y que predice bien la clase 2, pero la clase 1 la predice con cierto error.

En este caso, el bosque supera al árbol por un margen bajo, por lo tanto se considera que el modelo de random forest es mejor en cuanto a métricas. En ambas hay mala clasificación para la tercera variable, por lo que se podría valorar el juntar la clase 2 con la clase número 3, pero como lo que se ganaría no aportaría ninguna diferencia de como se encuentra actualmente, se ha considerado que mantener las tres clases es la mejor opción para poder ver exactamente los fallos de los modelos.

Comparando los 2 modelos se puede encontrar la curiosidad de que los dos predicen prácticamente igual de bien la clase 1, pero los bosques predicen mejor la clase 2 y por ello su exactitud es mejor, ya que se aciertan un total de 5 predicciones más para esta clase. Para la clase 3 ambos clasificadores realizan exactamente la misma clasificación, designan esos edificios

como pertenecientes a la clase 2, seguramente con un dataset con mayor cantidad de elementos de la clase 3 y menor número de la clase 2, se encontrarían modelos mejores.

Observando las métricas de F1 de ambos modelos, se vuelve a confirmar que el bosque tiene mejor resultado por un margen mayor que con la exactitud, aunque sigue siendo menor.

Pruebas extra

Se realizan pruebas extra con random forest utilizando el GridSearchCV y el cross validation, para comprovar si estas funcionalidades aportan un cambio significativo en los resultados que se han obtenido anteriormente.

Grid Search

Se obtienen los parametros más ótpimos para crear el random forest.

In [54]:
```
params = {"n_estimators":(20,50,100,250,500),
          "criterion":("gini","entropy"),
          "max_depth":(10,30,50,None),
          "max_leaf_nodes":(None,4,10,20,40)}
rf_grid = RandomForestClassifier(random_state = 42)
grid_search = GridSearchCV(rf_grid, params,cv=5)
```

In [55]:
```
grid_search.fit(X_train,y_train)
```

Out[55]:
```
GridSearchCV(cv=5, estimator=RandomForestClassifier(),
             param_grid={'criterion': ('gini', 'entropy'),
                         'max_depth': (10, 30, 50, None),
                         'max_leaf_nodes': (None, 4, 10, 20, 40),
                         'n_estimators': (20, 50, 100, 250, 500)})
```

In [56]:
```
grid_search.best_params_
```

Out[56]:
```
{'criterion': 'entropy',
 'max_depth': 50,
 'max_leaf_nodes': 40,
 'n_estimators': 20}
```

Cross Validation

In [61]:
```
clasificador = RandomForestClassifier(random_state = 42, n_estimators = 20, max_depth = 50, max_leaf_n

prediction_list = []
accuracy_list = []
f1_score_list = []

for i in df_.kfold.unique():
    X_train = df_[df_.kfold != i].drop(["Categoria","kfold"],axis=1)
    y_train = df_[df_.kfold != i].Categoria
    X_test = df_[df_.kfold == i].drop(["Categoria","kfold"],axis=1)
    y_test = df_[df_.kfold == i].Categoria

    clasificador.fit(X_train, y_train)

    predicciones = clasificador.predict(X_test)

    prediction_list.append(predicciones)
    accuracy_list.append(accuracy_score(y_test, predicciones))
    f1_score_list.append(f1_score(y_test, predicciones, average='weighted'))
```

In [62]:
```
accuracy_list
```

Out[62]:
```
[0.9452054794520548,
 0.9383561643835616,
 0.952054794520548,
 0.934931506849315,
 0.9486301369863014]
```

In [63]:
```
f1_score_list
```

Out[63]:
```
[0.9344924481910782,
 0.930283757338552,
 0.9445927343540156,
 0.9200424303351853,
 0.9376417176772791]
```

Al probar la validación cruzada, usando 5 capas y el metodo *Stratified*, el cual permite mantener el mismo balance entre clases que en el conjunto de datos original, podemos ver como en las distintas capas la precisión cambia, siendo superior en las dos

últimas. Por ende, a la hora de escoger nuestro modelo, se podría probar a unir los distintos modelos obtenidos para cada capa con un metodo de ensamble o limitarnos a coger el modelo con más precisión de los 5 en la evaluación.

Comparando estos resultados otra vez con el arbol de decisión, el peor modelo obtenido con el cross-validation, sigue siendo mejor que los resultados del arbol, por lo tanto, los bosques aleatórios se vuelven a imponer a los árboles.