Tarea 3 v2

June 26, 2025

Universidad Autónoma Metropolitana - Unidad Iztapalapa (UAM-I)

Maestría en Matemáticas Aplicadas e Industriales (MCMAI)

Taller de Modelado Matemático II - Parte I

Trimestre 25-P

Profesor:

Dr. Alejandro Román Vásquez

Alumnos:

Alan Badillo Salas

Brandon Eduardo Antonio Gómez

Diego Armando Arce Montes de Oca

1 Fase 1 - Adquisición de los datos

En esta fase reconstruiremos de forma directa las transformaciones aplicadas a las `Casas.csv` y guardaremos el imputador de los datos para poder aplicar las mismas transformaciones a los demás datos de `Casa_Kaggle.csv`.

1.1 Cargamos las librerías necesarias

Usaremos las librerías tradicionales.

[2]:
```
import numpy
import pandas

import matplotlib.pyplot as pyplot
import seaborn
```

1.2 Se carga el conjunto de datos de entrenamiento

Obtenemos el conjunto de datos y aplicamos las tranformaciones de las 22 variables construidas en la tarea anterior.

Guardamos la matriz X_2 que contiene las 22 variables procesadas y la matriz X_3 que contiene las mismas variables normalizadas. Además de la variable de respuesta y correspondiente a *SalePrice*.

[3]:

```
# Carga de datos

casas = pandas.read_csv("Casas.csv")

# Selección de Columnas de Análisis

columnas_analisis = [
    "MSZoning",
    "LotArea",
    "Street",
    "Neighborhood",
    "YearBuilt",
    "OverallCond",
    "ExterQual",
    "GrLivArea",
    "FullBath",
    "GarageArea",
    "BsmtCond",
    "FireplaceQu",
    "Electrical",
    "LotFrontage",
    "KitchenQual",
    "PavedDrive",
    "SalePrice",
]

casas_analisis = casas[columnas_analisis]

# Selección de Ejes de Datos

MSZoning = casas_analisis["MSZoning"]
LotArea = casas_analisis["LotArea"]
Street = casas_analisis["Street"]
Neighborhood = casas_analisis["Neighborhood"]
YearBuilt = casas_analisis["YearBuilt"]
OverallCond = casas_analisis["OverallCond"]
ExterQual = casas_analisis["ExterQual"]
GrLivArea = casas_analisis["GrLivArea"]
FullBath = casas_analisis["FullBath"]
GarageArea = casas_analisis["GarageArea"]
BsmtCond = casas_analisis["BsmtCond"]
FireplaceQu = casas_analisis["FireplaceQu"]
Electrical = casas_analisis["Electrical"]
LotFrontage = casas_analisis["LotFrontage"]
KitchenQual = casas_analisis["KitchenQual"]
PavedDrive = casas_analisis["PavedDrive"]
SalePrice = casas_analisis["SalePrice"]
```

```python
# Mean Encoder

Ejes_Cats = [
    ("MSZoning", MSZoning),
    ("Neighborhood", Neighborhood),
    ("OverallCond", OverallCond),
    ("BsmtCond", BsmtCond),
    ("FireplaceQu", FireplaceQu),
    ("Electrical", Electrical),
]

for nombre, eje in Ejes_Cats:
    eje = eje.fillna("NA")
    casas_analisis.loc[:, [nombre]] = eje
    eje_mean = pandas.merge(left=eje, right=pandas.DataFrame([eje, SalePrice]).
 ↪T.groupby(nombre).mean(), on=nombre)["SalePrice"]
    casas_analisis.loc[:, [f"{nombre}_mean"]] = eje_mean

# One-Hot Encoder (Dummies)

Ejes_Dums = [
    ("ExterQual", ExterQual),
    ("FullBath", FullBath),
    ("KitchenQual", KitchenQual),
    ("PavedDrive", PavedDrive),
]

for nombre, eje in Ejes_Dums:
    columnas = []
    eje = eje.fillna("NA")
    casas_analisis.loc[:, [nombre]] = eje
    for i, cat in enumerate(eje.unique()):
        eje_dummy = (eje == cat).astype(int)
        casas_analisis.loc[:, [f"{nombre}_{cat}_dummy{i}"]] = eje_dummy
        columnas.append(f"{nombre}_{cat}_dummy{i}")

# Selección de variables

x1 = casas_analisis["MSZoning_mean"]            # mean encoder
x2 = casas_analisis["LotArea"]                  # continua
x3 = casas_analisis["Neighborhood_mean"]        # mean encoder
x4 = casas_analisis["YearBuilt"]                # continua
x5 = casas_analisis["OverallCond_mean"]         # mean encoder
x6 = casas_analisis["ExterQual_Gd_dummy0"]      # dummy
x7 = casas_analisis["ExterQual_Ex_dummy2"]      # dummy
x8 = casas_analisis["ExterQual_Fa_dummy3"]      # dummy
```

```python
x9 = casas_analisis["GrLivArea"]                # continua
x10 = casas_analisis["FullBath_1_dummy1"]       # dummy
x11 = casas_analisis["FullBath_3_dummy2"]       # dummy
x12 = casas_analisis["FullBath_0_dummy3"]       # dummy
x13 = casas_analisis["GarageArea"]              # continua
x14 = casas_analisis["BsmtCond_mean"]           # mean encoder
x15 = casas_analisis["FireplaceQu_mean"]        # mean encoder
x16 = casas_analisis["Electrical_mean"]         # mean encoder
x17 = casas_analisis["LotFrontage"]             # continua*
x18 = casas_analisis["KitchenQual_Gd_dummy0"]   # dummy
x19 = casas_analisis["KitchenQual_Ex_dummy2"]   # dummy
x20 = casas_analisis["KitchenQual_Fa_dummy3"]   # dummy
x21 = casas_analisis["PavedDrive_N_dummy1"]     # dummy
x22 = casas_analisis["PavedDrive_P_dummy2"]     # dummy

X = pandas.DataFrame([
    x1, x2, x3, x4, x5, x6, x7, x8, x9, x10,
    x11, x12, x13, x14, x15, x16, x17, x18, x19, x20,
    x21, x22
], index=[
    "x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8", "x9", "x10",
    "x11", "x12", "x13", "x14", "x15", "x16", "x17", "x18", "x19", "x20",
    "x21", "x22"
]).T

# Imputación de Datos

X1 = X.copy().dropna(subset=["x17"])

y17 = X1["x17"]

from sklearn.ensemble import RandomForestRegressor

reg = RandomForestRegressor(random_state=123)

reg.fit(X1, y17)

print(reg.score(X1, y17))

# Obtenemos los registros con datos faltantes
Xmiss = X[X1.columns][X["x17"].isna()]

# Predecimos los datos faltantes
yp = reg.predict(Xmiss)

# Reintegramos los datos faltantes (imputados) a la matriz original de variables
X.loc[:, ["x17_imp"]] = X["x17"]
```

```python
X.loc[X["x17"].isna(), ["x17_imp"]] = yp

x17 = X["x17_imp"]

casas_analisis.loc[:, ["LotFrontage_imp"]] = X["x17_imp"]

# Eliminación de Puntos Atípicos

xs = [
    ("x2", x2),
    ("x4", x4),
    ("x9", x9),
    ("x13", x13),
    ("x17_imp", x17)
]

for i, (nombre, x) in enumerate(xs):
    Q1 = x.quantile(0.25)
    Q3 = x.quantile(0.75)
    IQR = Q3 - Q1
    xmin = Q1 - 1.5 * IQR
    xmax = Q3 + 1.5 * IQR
    xp = x.copy().astype(float)
    xp[xp >= xmax] = xmax
    xp[xp <= xmin] = xmin

    X.loc[:, [f"{nombre}_in"]] = xp

X2 = X.copy()

del X2["x2"]
del X2["x4"]
del X2["x9"]
del X2["x13"]
del X2["x17"]
del X2["x17_imp"]

# print(X2.columns)

columns = [
    "x1", "x2_in", "x3", "x4_in", "x5", "x6", "x7", "x8", "x9_in", "x10",
    "x11", "x12", "x13_in", "x14", "x15", "x16", "x17_imp_in", "x18", "x19",␣
 ↪"x20",
    "x21", "x22"
]

X2 = X2[columns]
```

```python
X2.columns = [f"x{j + 1}" for j in range(len(columns))]

# print(X2.head())

# Normalización

n, m = X2.shape

X3 = numpy.zeros((n, m))

for j, column in enumerate(X2.columns):
    xj = X2[column]
    X3[:, j] = (xj - xj.mean()) / xj.std()

X3 = pandas.DataFrame(X3, columns=X2.columns)
y = SalePrice

# Guardamos los datos transformados y el regresor de la imputación de x17

casas_analisis.to_csv("Casas_analisis.csv", index=False)
X2.to_csv("Casas_X2.csv", index=False)
X3.to_csv("Casas_X3.csv", index=False)
y.to_csv("Casas_y.csv", index=False)

# Guardamos el regresor de la imputación para x17

import pickle

pickle.dump(reg, open("reg_imp_x17.pickle", "wb"))
```

```
0.9963294003653655
```

2 Fase 2 - Transformación de los datos de Kaggle

Aplicamos las mismas trasformaciones al conjunto de datos de Kaggle, usando los resultados de las casas originales en la codificación.

[4]:
```python
casas_analisis_original = pandas.read_csv("Casas_analisis.csv")

casas_analisis_original.head()
```

[4]:
```
  MSZoning  LotArea Street Neighborhood  YearBuilt  OverallCond ExterQual  \
0       RL     8450   Pave      CollgCr       2003            5        Gd
1       RL     9600   Pave      Veenker       1976            8        TA
2       RL    11250   Pave      CollgCr       2001            5        Gd
3       RL     9550   Pave      Crawfor       1915            5        TA
```

```
4       RL    14260   Pave      NoRidge       2000            5        Gd

   GrLivArea  FullBath  GarageArea  … FullBath_3_dummy2 FullBath_0_dummy3  \
0       1710         2         548  …                 0                 0
1       1262         2         460  …                 0                 0
2       1786         2         608  …                 0                 0
3       1717         1         642  …                 0                 0
4       2198         2         836  …                 0                 0

  KitchenQual_Gd_dummy0  KitchenQual_TA_dummy1 KitchenQual_Ex_dummy2  \
0                     1                      0                     0
1                     0                      1                     0
2                     1                      0                     0
3                     1                      0                     0
4                     1                      0                     0

  KitchenQual_Fa_dummy3  PavedDrive_Y_dummy0  PavedDrive_N_dummy1  \
0                     0                    1                    0
1                     0                    1                    0
2                     0                    1                    0
3                     0                    1                    0
4                     0                    1                    0

   PavedDrive_P_dummy2  LotFrontage_imp
0                    0             65.0
1                    0             80.0
2                    0             68.0
3                    0             60.0
4                    0             84.0

[5 rows x 39 columns]
```

Realizamos codificación por la media (mean encoding) para datos de altas categóricas, mientras que hicimos codificación one-hot para 4 o menos categorías, también la imputación y eliminación de datos atípicos

Definimos una función de transformación para las `Casa_Kaggle.csv` basado en las transformaciones de `Casas.csv` (casas originales).

Aquí en el *Mean-Encoder* obtenemos el precio medio de cada categoría resultante de la anterior (casas originales) y en las *Dummies* de la codificación *One-Hot-Encoder* usamos el mismo orden para las categorías de las casas originales.

Además al aplicar la imputación cargamos el mismo imputador para hacer las predicciones sobre x_{17} en casas kaggle entrenado con las casas originales.

[5]:
```python
def transformaciones(casas, casas_analisis_original):
    # Selección de Columnas de Análisis
```

```python
columnas_analisis = [
    "MSZoning",
    "LotArea",
    "Street",
    "Neighborhood",
    "YearBuilt",
    "OverallCond",
    "ExterQual",
    "GrLivArea",
    "FullBath",
    "GarageArea",
    "BsmtCond",
    "FireplaceQu",
    "Electrical",
    "LotFrontage",
    "KitchenQual",
    "PavedDrive",
]

casas_analisis = casas[columnas_analisis]

# Mean Encoding

Ejes_Cats = [
    "MSZoning",
    "Neighborhood",
    "OverallCond",
    "BsmtCond",
    "FireplaceQu",
    "Electrical",
]

for nombre in Ejes_Cats:
    eje = casas_analisis[nombre]
    casas_analisis.loc[:, [nombre]] = eje
    # NOTA: Recuperamos la media de la categoría del conjunto original
    eje_mean = pandas.merge(left=eje,␣
↪right=casas_analisis_original[[nombre, f"{nombre}_mean"]],␣
↪on=nombre)[f"{nombre}_mean"]
    casas_analisis.loc[:, [f"{nombre}_mean"]] = eje_mean

# One-Hot Encoder (Dummies)

Ejes_Dums = [
    "ExterQual",
    "FullBath",
    "KitchenQual",
```

```python
        "PavedDrive",
    ]

    for nombre in Ejes_Dums:
        eje = casas_analisis[nombre]
        casas_analisis.loc[:, [nombre]] = eje
        # NOTA: Recuperamos el orden de las categorías de la original
        for i, cat in enumerate(casas_analisis_original[nombre].unique()):
            eje_dummy = (eje == cat).astype(int)
            casas_analisis.loc[:, [f"{nombre}_{cat}_dummy{i}"]] = eje_dummy

    # print(casas_analisis.head())

    # Selección de variables

    x1 = casas_analisis["MSZoning_mean"]             # mean encoder
    x2 = casas_analisis["LotArea"]                   # continua
    x3 = casas_analisis["Neighborhood_mean"]         # mean encoder
    x4 = casas_analisis["YearBuilt"]                 # continua
    x5 = casas_analisis["OverallCond_mean"]          # mean encoder
    x6 = casas_analisis["ExterQual_Gd_dummy0"]       # dummy
    x7 = casas_analisis["ExterQual_Ex_dummy2"]       # dummy
    x8 = casas_analisis["ExterQual_Fa_dummy3"]       # dummy
    x9 = casas_analisis["GrLivArea"]                 # continua
    x10 = casas_analisis["FullBath_1_dummy1"]        # dummy
    x11 = casas_analisis["FullBath_3_dummy2"]        # dummy
    x12 = casas_analisis["FullBath_0_dummy3"]        # dummy
    x13 = casas_analisis["GarageArea"]               # continua
    x14 = casas_analisis["BsmtCond_mean"]            # mean encoder
    x15 = casas_analisis["FireplaceQu_mean"]         # mean encoder
    x16 = casas_analisis["Electrical_mean"]          # mean encoder
    x17 = casas_analisis["LotFrontage"]              # continua*
    x18 = casas_analisis["KitchenQual_Gd_dummy0"]    # dummy
    x19 = casas_analisis["KitchenQual_Ex_dummy2"]    # dummy
    x20 = casas_analisis["KitchenQual_Fa_dummy3"]    # dummy
    x21 = casas_analisis["PavedDrive_N_dummy1"]      # dummy
    x22 = casas_analisis["PavedDrive_P_dummy2"]      # dummy

    X = pandas.DataFrame([
        x1, x2, x3, x4, x5, x6, x7, x8, x9, x10,
        x11, x12, x13, x14, x15, x16, x17, x18, x19, x20,
        x21, x22
    ], index=[
        "x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8", "x9", "x10",
        "x11", "x12", "x13", "x14", "x15", "x16", "x17", "x18", "x19", "x20",
        "x21", "x22"
    ]).T
```

```
# print(X.head())

# Imputación de Datos

X1 = X.copy().dropna(subset=["x17"])

# Obtenemos los registros con datos faltantes
Xmiss = X[X1.columns][X["x17"].isna()]

# NOTA: Usamos el regresor de imputación del conjunto original
import pickle

reg = pickle.load(open("reg_imp_x17.pickle", "rb"))
# reg = pickle.load(open("reg_imp_best.pickle", "rb"))

# Predecimos los datos faltantes
yp = reg.predict(Xmiss)

# Reintegramos los datos faltantes (imputados) a la matriz original de␣
↪variables
X.loc[:, ["x17_imp"]] = X["x17"]
X.loc[X["x17"].isna(), ["x17_imp"]] = yp

x17 = X["x17"]

# Eliminación de Puntos Atípicos

xs = [
    ("x2", "LotArea"),
    ("x4", "YearBuilt"),
    ("x9", "GrLivArea"),
    ("x13", "GarageArea"),
    ("x17_imp", "LotFrontage_imp"),
]

for i, (nombre, column) in enumerate(xs):
    # print(nombre, column, casas_analisis_original.columns[-1])
    # NOTA: Recuperamos los rangos del conjunto original
    x = casas_analisis_original[column]
    Q1 = x.quantile(0.25)
    Q3 = x.quantile(0.75)
    IQR = Q3 - Q1
    xmin = Q1 - 1.5 * IQR
    xmax = Q3 + 1.5 * IQR
    # xp = x.copy().astype(float)
    xp = X[nombre].copy().astype(float)
```

```
        xp[xp >= xmax] = xmax
        xp[xp <= xmin] = xmin

        X.loc[:, [f"{nombre}_in"]] = xp

    # print(X.head())

    X2 = X.copy()

    del X2["x2"]
    del X2["x4"]
    del X2["x9"]
    del X2["x13"]
    del X2["x17"]
    del X2["x17_imp"]

    # print(X2.columns)

    columns = [
        "x1", "x2_in", "x3", "x4_in", "x5", "x6", "x7", "x8", "x9_in", "x10",
        "x11", "x12", "x13_in", "x14", "x15", "x16", "x17_imp_in", "x18", "x19", "x20",
        "x21", "x22"
    ]

    X2 = X2[columns]

    X2.columns = [f"x{j + 1}" for j in range(len(columns))]

    # print(X2.head())

    # Normalización

    n, m = X2.shape

    X3 = numpy.zeros((n, m))

    for j, column in enumerate(X2.columns):
        xj = X2[column]
        X3[:, j] = (xj - xj.mean()) / xj.std()

    X3 = pandas.DataFrame(X3, columns=X2.columns)

    return casas_analisis, X2, X3
```

2.1 Aplicación de las transformaciones

Ahora aplicamos las transformaciones para los datos de las Casas Kaggle usando los datos de las Casas Originales.

A partir de los datos de prueba generamos los archivos csv para los datos con la matriz X_2 que selecciona las variables finales de la matriz X tomando en cuenta las variables que fueron transformadas e imputadas, y para X_3 que contiene a los datos estandarizados

[6]:
```
casas_kaggle = pandas.read_csv("Casas_Kaggle.csv")

casas_analisis_kaggle, X2, X3 = transformaciones(casas_kaggle,␣
↪casas_analisis_original)

casas_analisis_kaggle.to_csv("Casas_Kaggle_analisis.csv", index=False)
X2.to_csv("Casas_Kaggle_X2.csv", index=False)
X3.to_csv("Casas_Kaggle_X3.csv", index=False)

X2.head()
```

[6]:
```
           x1       x2         x3      x4           x5   x6   x7   x8  \
0  131558.375  11622.0  145847.08  1961.0  153961.59127  0.0  0.0  0.0
1  131558.375  14267.0  145847.08  1958.0  153961.59127  0.0  0.0  0.0
2  131558.375  13830.0  145847.08  1997.0  153961.59127  0.0  0.0  0.0
3  131558.375   9978.0  145847.08  1998.0  153961.59127  0.0  0.0  0.0
4  131558.375   5005.0  145847.08  1992.0  153961.59127  1.0  0.0  0.0

       x9  x10  …    x13          x14            x15            x16   x17  \
0   896.0  1.0  …  730.0  183632.6209  141331.482609  186825.113193  80.0
1  1329.0  1.0  …  312.0  183632.6209  141331.482609  186825.113193  81.0
2  1629.0  0.0  …  482.0  183632.6209  141331.482609  186825.113193  74.0
3  1604.0  0.0  …  470.0  183632.6209  141331.482609  186825.113193  78.0
4  1280.0  0.0  …  506.0  183632.6209  141331.482609  186825.113193  43.0

   x18  x19  x20  x21  x22
0  0.0  0.0  0.0  0.0  0.0
1  1.0  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0  0.0
3  1.0  0.0  0.0  0.0  0.0
4  1.0  0.0  0.0  0.0  0.0

[5 rows x 22 columns]
```

3 Fase 3 - Ajuste de los modelos

Una vez construidas las matriz X_2 (para las Casas Originales) y $X_2^{(k)}$ (para las Casas Kaggle). Realizamos el ajuste mediante los modelos de regulrairzación Ridge, Lasso, árboles de decisión y bosques aleatorios, para asegurar reproducibilidad utilizamos la *semilla* 123.

Cargamos los datos de X_2 de las casas originales.

[7]:
```
X2 = pandas.read_csv("Casas_X2.csv")

X2.head()
```

[7]:
```
              x1       x2             x3      x4             x5   x6   x7  \
0  191004.994787   8450.0  197965.773333  2003.0  203146.914738  1.0  0.0
1  191004.994787   9600.0  238772.727273  1976.0  155651.736111  0.0  0.0
2  191004.994787  11250.0  197965.773333  2001.0  203146.914738  1.0  0.0
3  191004.994787   9550.0  210624.725490  1915.0  203146.914738  0.0  0.0
4  191004.994787  14260.0  335295.317073  2000.0  203146.914738  1.0  0.0

    x8      x9  x10  …    x13            x14            x15            x16  \
0  0.0  1710.0  0.0  …  548.0  183632.620900  141331.482609  186825.113193
1  0.0  1262.0  0.0  …  460.0  183632.620900  205723.488818  186825.113193
2  0.0  1786.0  0.0  …  608.0  183632.620900  205723.488818  186825.113193
3  0.0  1717.0  1.0  …  642.0  213599.907692  226351.415789  186825.113193
4  0.0  2198.0  0.0  …  836.0  183632.620900  205723.488818  186825.113193

    x17  x18  x19  x20  x21  x22
0  65.0  1.0  0.0  0.0  0.0  0.0
1  80.0  0.0  0.0  0.0  0.0  0.0
2  68.0  1.0  0.0  0.0  0.0  0.0
3  60.0  1.0  0.0  0.0  0.0  0.0
4  84.0  1.0  0.0  0.0  0.0  0.0

[5 rows x 22 columns]
```

Visualizamos la información de las 22 variables predictivas.

[8]:
```
X2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 22 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   x1      1460 non-null   float64
 1   x2      1460 non-null   float64
 2   x3      1460 non-null   float64
 3   x4      1460 non-null   float64
 4   x5      1460 non-null   float64
 5   x6      1460 non-null   float64
 6   x7      1460 non-null   float64
 7   x8      1460 non-null   float64
 8   x9      1460 non-null   float64
 9   x10     1460 non-null   float64
 10  x11     1460 non-null   float64
```

```
 11  x12     1460 non-null   float64
 12  x13     1460 non-null   float64
 13  x14     1460 non-null   float64
 14  x15     1460 non-null   float64
 15  x16     1460 non-null   float64
 16  x17     1460 non-null   float64
 17  x18     1460 non-null   float64
 18  x19     1460 non-null   float64
 19  x20     1460 non-null   float64
 20  x21     1460 non-null   float64
 21  x22     1460 non-null   float64
dtypes: float64(22)
memory usage: 251.1 KB
```

Cargamos los datos de X_3 que son las mismas 22 variables, pero normalizadas.

[9]:
```
X3 = pandas.read_csv("Casas_X3.csv")

X3.head()
```

[9]:
```
         x1        x2        x3        x4        x5        x6        x7  \
0  0.387032 -0.333130  0.290473  1.052885  0.790080  1.410829 -0.192111
1  0.387032 -0.013184  0.985904  0.156125 -0.898279 -0.708318 -0.192111
2  0.387032  0.445869  0.290473  0.986459  0.790080  1.410829 -0.192111
3  0.387032 -0.027095  0.506207 -1.869888  0.790080 -0.708318 -0.192111
4  0.387032  1.283293  2.630839  0.953245  0.790080  1.410829 -0.192111

         x8        x9       x10  …       x13       x14       x15      x16  \
0 -0.098363  0.428489 -0.89550  …  0.373381  0.150550 -0.919147  0.30418
1 -0.098363 -0.502177 -0.89550  … -0.051523  0.150550  0.575830  0.30418
2 -0.098363  0.586370 -0.89550  …  0.663088  0.150550  0.575830  0.30418
3 -0.098363  0.443031  1.11593  …  0.827255  1.814463  1.054744  0.30418
4 -0.098363  1.442250 -0.89550  …  1.763975  0.150550  0.575830  0.30418

        x17       x18      x19      x20      x21       x22
0 -0.147210  1.220838 -0.27107 -0.16561 -0.25622 -0.144792
1  0.707824 -0.818548 -0.27107 -0.16561 -0.25622 -0.144792
2  0.023797  1.220838 -0.27107 -0.16561 -0.25622 -0.144792
3 -0.432222  1.220838 -0.27107 -0.16561 -0.25622 -0.144792
4  0.935834  1.220838 -0.27107 -0.16561 -0.25622 -0.144792

[5 rows x 22 columns]
```

Cargamos la variable respuesta y (SalePrice).

[10]:
```
y = pandas.read_csv("Casas_y.csv")

y.head()
```

```
[10]:    SalePrice
      0     208500
      1     181500
      2     223500
      3     140000
      4     250000
```

3.1 Partición de los datos

Ahora generamos los conjuntos de entrenamiento y pruebas con la semilla aleatoria fijada.

```
[11]: from sklearn.model_selection import train_test_split

X2_train, X2_test, y_train, y_test = train_test_split(X2, y["SalePrice"],␣
 ↪random_state=123)

X2_train.shape, X2_test.shape, y_train.shape, y_test.shape
```

```
[11]: ((1095, 22), (365, 22), (1095,), (365,))
```

Podemos hacer lo mismo para los datos normalizados.

```
[12]: from sklearn.model_selection import train_test_split

X3_train, X3_test, y_train, y_test = train_test_split(X3, y["SalePrice"],␣
 ↪random_state=123)

X3_train.shape, X3_test.shape, y_train.shape, y_test.shape
```

```
[12]: ((1095, 22), (365, 22), (1095,), (365,))
```

3.2 Ajuste por Ridge y Lasso

Como la respuesta y o *SalePrice* es continua nos enfrentamos a un problema de **regresión**, por lo que usaremos la regresión lineal, pero con regularizaciones L_2 (Ridge) y L_1 (Lasso).

La diferencia entre Ridge (regularización suave o cuadrática) y Lasso (regularización rígida o absoluta) es que los coeficientes de regresión encontrados serán más cercanos a cero en Lasso para las variables con poca influencia lineal en la respuesta.

Los modelos son adaptaciones al modelo de regresión lineal usando el factor de regularización λ.

Midelo lineal

El modelo lineal base se expresa como:

$$y = X\beta + \varepsilon$$

donde $y \in \mathbb{R}^n$ es el vector de respuesta, $X \in \mathbb{R}^{n\times(k+1)}$ es la matriz de diseño (incluyendo la columna de unos para el intercepto), $\beta \in \mathbb{R}^{k+1}$ es el vector de coeficientes, y ε es el vector de errores aleatorios.

Modelo Ridge

La regresión Ridge busca estimar los coeficientes β minimizando la siguiente función de pérdida:

$$\mathcal{L}_{\text{Ridge}}(\beta) = (y - X\beta)^\top (y - X\beta) + \lambda \|\beta\|_2^2,$$

donde $\|\beta|_2^2 = \sum_{j=1}^{k} \beta_j^2$ es la norma cuadrada del vector de coeficientes (a menudo se excluye $ _0 $ del término de penalización), y $ 0$ es un hiperparámetro que controla la fuerza de la regularización.

O visto en el problema de minimización la función de pérdida $\mathcal{L}_{\text{pérdida}}(\beta) = SS_E$ (en el caso de regresión) es la suma de los errores cuadráticos que se obtiene como:

$$SS_E = (y - \hat{y})^\top (y - \hat{y}) + \lambda \|\beta\|_2^2$$

donde $\hat{y} = X\beta$

Tiene una solución análitica derivada de las ecuaciones normales:

$$\hat{\beta}_{Ridge} = (X^\top X + \lambda I)^{-1} X^\top y$$

donde I es la matriz identidad del tamaño adecuado (ajustada si no se penaliza β_0).

Modelo Lasso

La regresión Lasso también es una técnica de regularización, pero a diferencia de Ridge, puede llevar algunos coeficientes a ser exactamente cero, actuando así como una forma de selección automática de variables.

El modelo base también es:

$$y = X\beta + \varepsilon,$$

y la función de pérdida que Lasso minimiza es:

$$\mathcal{L}_{\text{Lasso}}(\beta) = (y - X\beta)^\top (y - X\beta) + \lambda \|\beta\|_1,$$

donde $\|\beta\|_1 = \sum_{j=1}^{k} |\beta_j|$ es la norma ℓ_1 del vector de coeficientes (usualmente también sin incluir β_0.

Este modelo no tiene solución analítica, por lo que requiere de un optimizador que busque los mejores coefientes β.

Para que el ajuste sea apropiado las covariables o predictores deben de tener la misma escala por lo que se utiliza la matriz X_3 que contiene los datos estandarizados

3.2.1 Ajuste por Ridge

Importamos la clase Ridge desde el módulo *linear model* de la biblioteca SciKit Learn.

[ ]:
```python
from sklearn.linear_model import Ridge

reg = Ridge(alpha=1, random_state=123)

reg.fit(X3_train, y_train)

pandas.DataFrame(reg.coef_, columns=["Coeficiente"])
```

[ ]:

	0
0	-846.486566
1	6886.279794
2	22863.255175
3	9665.150058
4	-1012.246991
5	4953.332558
6	9580.695817
7	-996.344053
8	25151.043691
9	4945.508586
10	6422.075682
11	1759.469582
12	8252.970411
13	4421.501127
14	6553.096982
15	361.364159
16	979.902765
17	4844.301303
18	11659.826201
19	194.178162
20	936.995237
21	-1241.354897

Cáculo del RMS_E La raíz del error cuadrático medio es una medida que indica la pérdida positiva acumulada entre los valores de respuesta conocidos y la predicción obtenida en el ajuste. Este sirve para determinar qué tan alejada está la predicción de los datos reales y es útil para que el optimizador minimice esta medida, ya que si es cero la predicción será exacta.

Primero calculamos el error cuadrático medio MS_E

El error cuadrático medio MS_E se obtiene mediante:

$$MS_E = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

donde

- y_i : es el valor observado (real)
- $\hat{y}_i$: es el valor que predice el modelo

- n : número de observaciones

Como el MS_E eleva los residuos al cuadrado, para poder interpretar mejor los resultados se le extrae la raíz cuadrada,

Así la raíz del error cuadrático medio RMS_E es:

$$RMS_E = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} = \sqrt{MS_E}$$

[37]:
```
y_pred = reg.predict(X3_test)

e = (y_test.values - y_pred) ** 2

rmse = e.mean() ** 0.5

rmse
```

[37]:
```
np.float64(32564.9352529349)
```

Observamos que una predicción promedio se aleja aproximadamente $32,564$ unidades promedio de dato real, cómo la respuesta está en dólares, la predicción falla cerca de 30 mil dólares en la estimación promedio del precio de la casa.

Búsqueda del mejor parámetro de Ridge Podemos buscar el mejor hiperparámetro λ posible para Ridge, buscando entre un conjunto de λ's.

Podemos proponer diferentes λ en un espacio logarítmico entre 10^{-3} y 10^{3}.

[38]:
```
numpy.logspace(-3, 3)
```

[38]:
```
array([1.00000000e-03, 1.32571137e-03, 1.75751062e-03, 2.32995181e-03,
       3.08884360e-03, 4.09491506e-03, 5.42867544e-03, 7.19685673e-03,
       9.54095476e-03, 1.26485522e-02, 1.67683294e-02, 2.22299648e-02,
       2.94705170e-02, 3.90693994e-02, 5.17947468e-02, 6.86648845e-02,
       9.10298178e-02, 1.20679264e-01, 1.59985872e-01, 2.12095089e-01,
       2.81176870e-01, 3.72759372e-01, 4.94171336e-01, 6.55128557e-01,
       8.68511374e-01, 1.15139540e+00, 1.52641797e+00, 2.02358965e+00,
       2.68269580e+00, 3.55648031e+00, 4.71486636e+00, 6.25055193e+00,
       8.28642773e+00, 1.09854114e+01, 1.45634848e+01, 1.93069773e+01,
       2.55954792e+01, 3.39322177e+01, 4.49843267e+01, 5.96362332e+01,
       7.90604321e+01, 1.04811313e+02, 1.38949549e+02, 1.84206997e+02,
       2.44205309e+02, 3.23745754e+02, 4.29193426e+02, 5.68986603e+02,
       7.54312006e+02, 1.00000000e+03])
```

Realizamos búsqueda por cuadrícula para encontrar el hiperparámetro de regularización óptimo.

[16]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge
```

```python
reg = Ridge(random_state=123)

cv = GridSearchCV(reg, {
    "alpha": numpy.logspace(-3, 3, 100)
})
cv.fit(X3, y)

pandas.DataFrame(cv.cv_results_).sort_values(by = "rank_test_score")
```

[16]:

```
    mean_fit_time  std_fit_time  mean_score_time  std_score_time  param_alpha  \
76       0.001699      0.000489         0.001061        0.000150    40.370173
75       0.001925      0.000565         0.001365        0.000558    35.111917
74       0.001499      0.000211         0.001395        0.000992    30.538555
77       0.002053      0.001178         0.001086        0.000234    46.415888
73       0.001810      0.000626         0.001052        0.000101    26.560878
..            …             …              …               …           …
95       0.001784      0.001034         0.000931        0.000203   572.236766
96       0.001273      0.000108         0.001473        0.001228   657.933225
97       0.001207      0.000062         0.000869        0.000084   756.463328
98       0.001645      0.000871         0.000857        0.000050   869.749003
99       0.001581      0.000526         0.000882        0.000053  1000.000000

                         params  split0_test_score  split1_test_score  \
76   {'alpha': 40.37017258596558}           0.840911           0.811198
75  {'alpha': 35.111917342151344}           0.840653           0.811008
74  {'alpha': 30.538555088334185}           0.840416           0.810837
77   {'alpha': 46.41588833612782}           0.841190           0.811407
73  {'alpha': 26.560877829466893}           0.840199           0.810683
..                             …                …                  …
95    {'alpha': 572.236765935022}           0.840187           0.814426
96   {'alpha': 657.9332246575682}           0.838208           0.813322
97    {'alpha': 756.463327554629}           0.835639           0.811689
98   {'alpha': 869.7490026177834}           0.832360           0.809388
99             {'alpha': 1000.0}           0.828237           0.806264

    split2_test_score  split3_test_score  split4_test_score  mean_test_score  \
76           0.834541           0.828027           0.770295         0.816994
75           0.834772           0.828133           0.770405         0.816994
74           0.834966           0.828219           0.770493         0.816986
77           0.834266           0.827898           0.770157         0.816984
73           0.835129           0.828290           0.770563         0.816973
..                …                  …                  …                …
95           0.800625           0.808445           0.748574         0.802451
96           0.794893           0.804864           0.744761         0.799210
97           0.788348           0.800685           0.740358         0.795344
98           0.780897           0.795810           0.735276         0.790746
```

```
99          0.772438          0.790126          0.729411          0.785295

    std_test_score  rank_test_score
76        0.025362                1
75        0.025323                2
74        0.025290                3
77        0.025407                4
73        0.025261                5
..             …              …
95        0.030031               96
96        0.030774               97
97        0.031597               98
98        0.032493               99
99        0.033453              100

[100 rows x 14 columns]
```

Observamos que el mejor $\lambda = 40.37$ alcanza una puntuación media en las pruebas de 0.816994.

[17]:
```python
pandas.DataFrame(cv.cv_results_).sort_values(by =␣
↪"rank_test_score")[["param_alpha", "mean_test_score", "rank_test_score"]]
```

[17]:
```
    param_alpha  mean_test_score  rank_test_score
76    40.370173         0.816994                1
75    35.111917         0.816994                2
74    30.538555         0.816986                3
77    46.415888         0.816984                4
73    26.560878         0.816973                5
..          …              …              …
95   572.236766         0.802451               96
96   657.933225         0.799210               97
97   756.463328         0.795344               98
98   869.749003         0.790746               99
99  1000.000000         0.785295              100

[100 rows x 3 columns]
```

El mejor hiperparámetro encontrado por la técnica búsqueda por cuadríctula es:

[18]:
```python
cv.best_estimator_.alpha
```

[18]:
```
np.float64(40.37017258596558)
```

El mejor valor del hiperparámetro λ (o alpha en python) es: $\lambda_{Ridge} = 40.37017258596558$

Ajustamos el modelo con la λ_{Ridge} óptima

[19]:
```python
from sklearn.linear_model import Ridge
```

```
reg = Ridge(alpha=cv.best_estimator_.alpha, random_state=123)

reg.fit(X3_train, y_train)

pandas.DataFrame(reg.coef_)
```

[19]:

	0
0	-511.988093
1	6885.581340
2	21826.578380
3	8760.614163
4	-669.380337
5	5285.643348
6	9757.630797
7	-929.957095
8	23380.459212
9	3416.581682
10	6481.123022
11	1530.592225
12	8500.088683
13	4393.200747
14	7007.701415
15	468.466873
16	1299.474203
17	4693.430915
18	11494.595402
19	89.820228
20	767.208373
21	-1249.787441

Calculamos el RMS_E

[39]:
```
y_pred = reg.predict(X3_test)

e = (y_test.values - y_pred) ** 2

rmse = e.mean() ** 0.5

rmse
```

[39]: `np.float64(32564.9352529349)`

Realizamos la validación cruzada

[21]:
```
from sklearn.linear_model import RidgeCV

cv = RidgeCV(
```

```
    alphas= numpy.logspace(-3,3, 100), cv =5
)

cv.fit(X3, y)

cv.best_score_
```

[21]:
```
np.float64(0.8169944466785761)
```

Ajustamos el modelo y obtenemos los coeficientes β

[22]:
```
from sklearn.linear_model import Ridge

reg = Ridge(alpha=cv.best_score_, random_state=123)

reg.fit(X3_train, y_train)

pandas.DataFrame(reg.coef_)
```

[22]:

	0
0	-848.213467
1	6886.213668
2	22868.415460
3	9670.284272
4	-1014.000770
5	4951.360904
6	9579.615545
7	-996.711660
8	25160.518561
9	4953.692238
10	6421.673106
11	1760.697906
12	8251.550503
13	4421.572732
14	6550.636346
15	360.814680
16	978.298392
17	4845.134136
18	11660.625522
19	194.722438
20	937.902467
21	-1241.279649

El RMS_E fue de:

[ ]:
```
y_pred = reg.predict(X3_test)

e = (y_test.values - y_pred) ** 2
```

```
rmse = e.mean() ** 0.5

rmse
```

[ ]: np.float64(32565.86434937793)

[24]:
```
from sklearn.metrics import root_mean_squared_error

root_mean_squared_error(y_test, y_pred)
```

[24]: 32565.86434937793

3.2.2 Ajuste por Lasso

Ajustamos el modelo Lasso de forma similar que con el modelo Ridge

[25]:
```
from sklearn.linear_model import Lasso

reg = Lasso(alpha=1, random_state=123)

reg.fit(X3_train, y_train)

y_pred =reg.predict(X3_test)

root_mean_squared_error(y_test, y_pred)
```

[25]: 32569.8635208258

[26]:
```
from sklearn.linear_model import LassoCV

cv = LassoCV(

    alphas= numpy.logspace(-3,3, 100), cv =5
)

cv.fit(X3_train, y_train)

cv.alpha_
```

[26]: np.float64(657.9332246575682)

[27]:
```
reg = Lasso(alpha=cv.alpha_, random_state=123)

reg.fit(X3_train, y_train)

y_pred =reg.predict(X3_test)
```

```
root_mean_squared_error(y_test, y_pred)
```

[27]: 32592.332401382308

3.2.3 Ajuste por árboles de decisión

Podemos experimentar con un regresor por árboles de decisión para compararlo a los modelos de Ridge y Lasso.

[28]:
```
from sklearn.tree import DecisionTreeRegressor

reg =DecisionTreeRegressor(

    criterion="squared_error",
    #criterion = "absolute_error",
    #criterion = "poisson",

    max_depth=20,
    min_samples_leaf=5,
    min_samples_split=5,
    random_state=123
)


reg.fit(X2_train, y_train)

y_pred =reg.predict(X2_test)

root_mean_squared_error(y_test, y_pred)
```

[28]: 41395.30782181962

Observamos que este genera más error cuadrático medio (RMS_E).

Búsqueda por cuadrícula Buscamos la mejor combinación de algunos hiperparámetros propuestos.

[29]:
```
from sklearn.model_selection import GridSearchCV

reg = DecisionTreeRegressor(random_state= 123)

cv = GridSearchCV(reg,
    param_grid={
        "criterion": ["squared_error", "absolute_error", "poisson"],
        "max_depth":[ 6, 8, 10, 12, 14],
        "min_samples_leaf": [ 6, 8, 10, 12, 14],
        "min_samples_split":[8, 10],
    },
```

```
    cv=5
)

cv.fit(X2_train, y_train)
cv.best_estimator_
```

[29]:
```
DecisionTreeRegressor(criterion='absolute_error', max_depth=12,
                      min_samples_leaf=8, min_samples_split=8,
                      random_state=123)
```

Observamos que el mejor comportamiento se da para el error absoluto con una profundidad máxima del árbol de 12 niveles y 8 muestras mínimas por hoja y corte.

[40]:
```python
from sklearn.tree import DecisionTreeRegressor

reg =DecisionTreeRegressor(

    criterion = "absolute_error",
    max_depth=12,
    min_samples_leaf=8,
    min_samples_split=8,
    random_state=123
)


reg.fit(X2_train, y_train)

y_pred =reg.predict(X2_test)

root_mean_squared_error(y_test, y_pred)
```

[40]:
```
40402.463693595826
```

El error cuadrático medio (la raíz) logra descender un poco, pero sigue siendo mayor que en Ridge y Lasso.

El RMS_E es más alto que con Ridge y Lasso, como vimos en clase la desventaja es que no tiene un poder de predicción muy grande

Ajuste por Bosques Aleatorios Los bósques aleatorios tienen mayor poder de predicción que los árboles de decisión ya que construyen un bósque aleatorio de árboles de decisión, combinando sus resultados.

[31]:
```python
# entrenamiento
from sklearn.ensemble import RandomForestRegressor

reg =RandomForestRegressor(n_estimators=200,

    #criterion="squared_error",
```

```
    #criterion = "absolute_error",
    #criterion = "poisson",
    bootstrap=True,

    #max_depth=20,
    #min_samples_leaf=5,
    #min_samples_split=5,
    random_state=123
)


reg.fit(X2_train, y_train)

y_pred =reg.predict(X2_test)

root_mean_squared_error(y_test, y_pred)
```

[31]: 29561.827275505428

Observamos que el error cuadrático medio (la raíz) es mucho menor que en Ridge y Lasso.

Búsqueda por cuadrícula Buscamos minimizar aún más el error buscando entre diferentes combinaciones de parámetros para ver si mejora.

[32]:
```
# Ajuste de hiperparámetros (optimización de hiperparámetros)
from sklearn.model_selection import GridSearchCV

reg = RandomForestRegressor(random_state= 123)

cv = GridSearchCV(reg,
    param_grid={
        "n_estimators": [100, 200, 300],
        "criterion": ["squared_error", "absolute_error", "poisson"],
        "max_depth":[None,10, 20],
        "min_samples_leaf": [2, 6, 8],
        "min_samples_split":[2, 8, 10],
    },
    cv=5
)

cv.fit(X2_train, y_train)

cv.best_estimator_
```

[32]: RandomForestRegressor(min_samples_leaf=2, n_estimators=300, random_state=123)

Observamos que la mejor combinación se da con el error cuadrático (a diferencia del absoluto en el árbol de decisión) y tamaños de 2 en lugar de 8 en las mínimas muestras por hoja y corte.

[43]:
```python
from sklearn.ensemble import RandomForestRegressor
# por búsqueda por cuadrícula
reg =RandomForestRegressor(

    criterion="squared_error",
    #criterion = "absolute_error",
    #criterion = "poisson",
    n_estimators=300,
    bootstrap=True,
    max_depth=None,
    min_samples_leaf=2,
    min_samples_split=2,
    random_state=123
)


reg.fit(X2_train, y_train)

y_pred =reg.predict(X2_test)

root_mean_squared_error(y_test, y_pred)
```

[43]: 29703.352903651416

Sin embargo, el error no logra descender más.

Optimización Bayesiana Además de la búsqueda por rejilla o cuadrícula, podemos intentar encontrar los hiperparámetros mediante una optimización Bayesiana.

Usando la librería de SciKit Learn Optimize.

[ ]:
```python
! pip install scikit-optimize
```

Construimos el buscador Bayesiano con validación cruzada.

[52]:
```python
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer

# Definimos el modelo base
reg = RandomForestRegressor(random_state=123)

# Definimos el espacio de búsqueda
search_space = {
    "n_estimators": Integer(100, 300),
    "criterion": Categorical(["squared_error", "absolute_error", "poisson"]),
    "max_depth": Categorical([None, 10, 12, 20]),
    "min_samples_leaf": Integer(2, 8),
    "min_samples_split": Integer(2, 10),
}
```

```python
# Definir el optimizador bayesiano
cv = BayesSearchCV(
    estimator=reg,
    search_spaces=search_space,
    cv=5,
    n_iter=32,  # número de combinaciones a explorar
    random_state=123,
    n_jobs=-1,  # usa todos los núcleos disponibles
    verbose=0
)

# Ajustar el modelo
cv.fit(X2_train, y_train)

# Mostrar el mejor modelo encontrado
cv.best_estimator_
```

[52]:
```
RandomForestRegressor(min_samples_leaf=2, n_estimators=300, random_state=123)
```

Y vemos que los resultados son similares a los de la búsqueda por cudrícula.

[50]:
```python
from sklearn.ensemble import RandomForestRegressor
# por búsqueda por cuadrícula
reg =RandomForestRegressor(

    criterion="squared_error",
    n_estimators=300,
    bootstrap=True,
    max_depth=None,
    min_samples_leaf=2,
    min_samples_split=3,
    random_state=123
)



reg.fit(X2_train, y_train)

y_pred =reg.predict(X2_test)

root_mean_squared_error(y_test, y_pred)
```

[50]:
```
29703.352903651416
```

Obtuvimos los mismos resultados tanto por búsqueda por cuadrícula como por optimización Bayesiana, $RMS_E = 29703.352903651416$

4 Fase 4 Prueba en Kaggle

Finalmente optamos por el modelo de Bosques aleatorios con los hiperparámetros obtenidos por búsqueda por cuadrícula

[51]:
```
from sklearn.ensemble import RandomForestRegressor

reg =RandomForestRegressor(

    criterion="squared_error",
    n_estimators=300,
    max_depth=12,
    min_samples_leaf=2,
    min_samples_split=2,
    random_state=123
)



reg.fit(X2_train, y_train)

y_pred =reg.predict(X2_test)

root_mean_squared_error(y_test, y_pred)
```

[51]:
```
29693.494426570975
```

Finalmente el $RMS_E = 29693.494426570975$ es el más bajo

Generamos el archivo Casas_Kaggle_X2.csv

[ ]:
```
X2k = pandas.read_csv("Casas_Kaggle_X2.csv")
```

Obtenemos las respuesta predichas por el mejor modelo

[ ]:
```
y_pred =reg.predict(X2k)
y_pred
```

[ ]:
```
array([126585.88356128, 147210.17306478, 168707.79345317, …,
       241707.81839441, 194481.27527108, 286983.64981548])
```

Generamos un archivo con las columnas Id y la predicción (creada por el mejor modelo).

[ ]:
```
casas_analisis_kaggle =pandas.read_csv("Casas_Kaggle.csv")

data = pandas.DataFrame(y_pred, columns=["SalePrice"])
data["Id"]= casas_analisis_kaggle["Id"]

data[["Id", "SalePrice"]].to_csv("Kaggle_submit1.csv", index=False)
```

5 Conclusiones

En esta tarea hemos replicado las transformaciones de las Casas Originales (`Casas.csv`) a las Casas Kaggle (`Casas_Kaggle.csv`), identificando que las transformaciones en Kaggle debían replicar los mismos comportamientos que en las casas originales, por ejemplo, para las codificaciones por la media (*Mean Encoder*) usadas en ejes de 5 o más categorías, estos debían usar el mismo promedio que en las casas originales. También preservar el mismo orden en las *Dummies* (con la codificación *One-Hot Encoder*) y sobre todo aplicar el mismo imputador (el predictor construido para rellenar los datos faltantes de x_{17} o *LotFrontage*).

Al lograr transformar los datos y extraer las mismas 22 variables de predicción en la matriz X_2, aplicamos los modelos Ridge, Lasso, Árbol de Decisión y Bósque Aleatorio para construir un regresor que lograra predecir la variable de respuesta (*SalePrice*) y tener una estimación del valor de la Casa mediante las 22 variables predictivas.

Además, como los modelos son dependientes de hiperparámetros desconocidos (ajustados a $\lambda = 1$ para Ridge y Lasso), se tuvieron que aplicar técnicas de Validación Cruzada para encontrar los mejores hiperparámetros y tratar de reducir aún más la medida de error.

En este caso usamos la medida de error RMS_E (Raíz del Error Cuadrático Medio / Root Mean Squared Error), para medir la diferencia entre la predicción $\hat{y}$ y la variable de respuesta conocida y (y_{test} para ser precisos).

Al realizar las pruebas sobre las Casas Kaggle, solo logramos reducir el error a cerca de $29,500$ unidades, lo que es un error aún bastante grande obteniendo una puntuación de 0.26 en Kaggle, muy superior al 0.014 de los primeros usuarios, quedando así en el Rank $4k$ ($4,215$ para ser exactos) en el tablero de mejores usuarios.

Aunque no logramos reducir más la puntuación, debemos recordar que solo se utilizaron las 16 variables propuestas para las tareas, qué quizás con más variables (usando las 80 columnas completas), se pudiera mejorar la puntuación.