Trabajo Final Machine Learning

Jaime García Lozano

20 de febrero de 2022

Índice

1. Introducción

En este trabajo se busca poner en práctica todos los modelos de *Machine Learning* vistos a lo largo del curso. Partiendo del *dataset diamonds*, la práctica constará de tres partes:

1. Predicción del precio del diamante como una variable continua. En este apartado utilizaremos las siguientes métricas para comparar los resultados de los modelos:

$$RMSE = \sqrt{\frac{1}{m}\sum_{j}^{m}\left[\hat{y}_j - y_j\right]^2} \tag{1}$$

$$MAE = \frac{1}{m}\sum_{j}^{m}\left|\hat{y}_j - y_j\right| \tag{2}$$

   Las ventajas de estas dos métricas es que están en las mismas unidades que la variable a predecir. La segunda, además, es más robusta a valores extremos.

2. Predicción del precio como variable categórica. Transformaremos el precio en una variable categórica y utilizaremos la métrica *Accuracy* (proporción de aciertos) para la comparación entre modelos.

3. Por último, se repetirá el apartado anterior, esta vez prediciendo el corte de los diamantes en función del resto de variables.

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

from matplotlib import style

plt.rcParams['image.cmap'] = "bwr"
#plt.rcParams['figure.dpi'] = "100"
plt.rcParams['savefig.bbox'] = "tight"
style.use('ggplot') or plt.style.use('ggplot')

import warnings
warnings.filterwarnings('ignore')
```

2. Descripción de los datos

Disponemos un *dataset* que contiene diferentes características de los diamantes junto al precio de cada uno de ellos. Esta última variable será la que buscaremos predecir.

[2]:
```
df = pd.read_csv('/Users/chewa/Documents/GitHub/MADM2021/Practica/diamonds.csv',␣
↪index_col=0)
df.head()
```

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Tabla 1: Estructura del *dataset*.

- **price**: precio en dolares.
- **carat**: peso.
- **cut**: calidad del corte.
- **color**: color del diamante.
- **clarity**: medida que establece cómo de claro el diamante es.
- **x,y,z**: longitud, anchura y profundidad en mm.
- **depth**:porcentaje de profundidad total= z/media(x,y).
- **table**: ancho de la parte superior del diamante en relación con el punto más ancho.

2.1. Tratamiento de los valores atípicos

[3]:
```
df.hist(column="price");
```

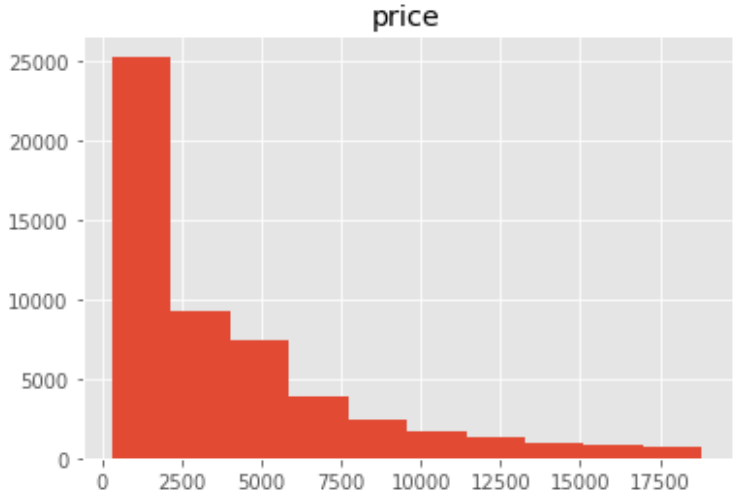

Figura 1: Distribución del precio.

[4]:
```
df2= df[["price","carat","table", "depth"]]
df2.plot(kind="box", subplots=True, layout=(2,2), figsize=(15,10));
```

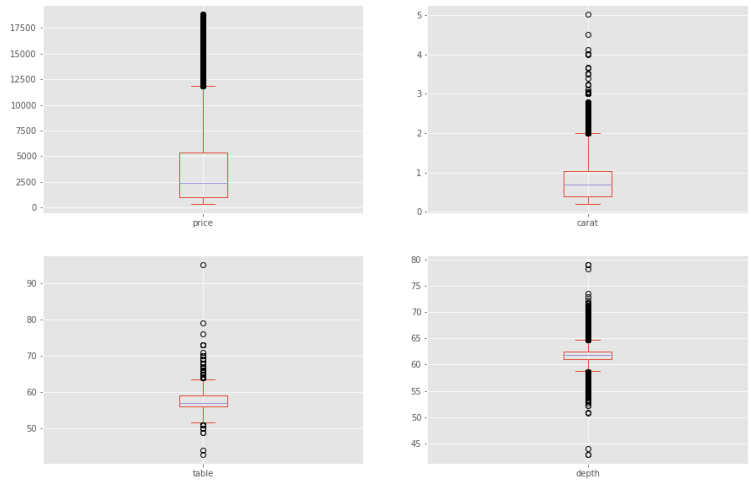

Figura 2: Boxplot de las cuatro variables cuantitativas más relevantes. Los puntos representan los valores atípicos.

Eliminamos todos los valores que estén fuera del rango intercuartílico. Fuente: [1].

[5]:
```
def outliers(var, data):
    a = []
    q1 = data[var].quantile(.25)
    q2 = data[var].quantile(.5)
    q3 = data[var].quantile(.75)
    iqr = q3-q1
    ulim = float(q3+(1.5*iqr))
    llim = float(q1-(1.5*iqr))

    for i in data[var]:
        if i > ulim:
            i = np.NaN
        elif i < llim:
            i = np.NaN
        else:
            i=i
        a.append(i)
    return a

for col in df.select_dtypes(exclude='object').columns:
    df[col] = outliers(col, df)
```

[6]:
```
df_new=df.dropna()
```

Porcentaje de datos que nos quedan tras quitar los outliers:

[7]:
```
np.round(df_new.shape[0]/df.shape[0],3)
```

[7]:
```
0.881
```

Por lo que seguimos disponiendo de la inmensa mayoría de los datos.

[8]:
```
df=df_new
```

2.2. Arreglo de las variables

Arreglo de las variables categóricas:

[9]:
```
categorias = df["cut"].astype('category')
color = df["color"].astype('category')
claridad  = df["clarity"].astype('category')
```

[10]:
```
df["cut_cat"]=categorias.cat.codes
df["color_cat"]= color.cat.codes
df["clarity_cat"]= claridad.cat.codes
df["price"]= df["price"].astype("float64")
```

[11]:
```
df.head()
```

	carat	cut	color	clarity	depth	table	price	x	y	z	cut_cat	color_cat	clarity_cat
1	0.23	Ideal	E	SI2	61.5	55.0	326.0	3.95	3.98	2.43	2	1	3
2	0.21	Premium	E	SI1	59.8	61.0	326.0	3.89	3.84	2.31	3	1	2
4	0.29	Premium	I	VS2	62.4	58.0	334.0	4.20	4.23	2.63	3	5	5
5	0.31	Good	J	SI2	63.3	58.0	335.0	4.34	4.35	2.75	1	6	3
6	0.24	Very Good	J	VVS2	62.8	57.0	336.0	3.94	3.96	2.48	4	6	7

Tabla 2: *Dataset* después de añadir las columnas con las variables categóricas transformadas a formato numérico

Para la segunda parte del trabajo añadimos una columna con la versión categórica del precio. A partir de la función *cut* definimos cuatro clases: barato, normal, caro y muy caro.

[12]:
```
df["price_cat"]=pd.cut(df["price"],4, labels=["barato","normal", "caro", "muy␣
↪caro"])
```

2.2.1. Análisis de las correlaciones

Veamos la correlación entre las variables a partir de un mapa de calor:

[14]:
```
df2=␣
↪df[["carat","depth","table","price","x","y","z","cut_cat","color_cat","clarity_cat"]]
```

```
plt.figure(figsize=(10,10))
cm = np.corrcoef(df2[df2.columns].values.T)
sns.set(font_scale=1.5)
hm = sns.heatmap(cm, cmap="vlag" ,cbar=True,annot=True, square=True, fmt='.2f',
                 annot_kws={'size': 15}, yticklabels=df2.columns,␣
 ↪xticklabels=df2.columns)
plt.show();
```

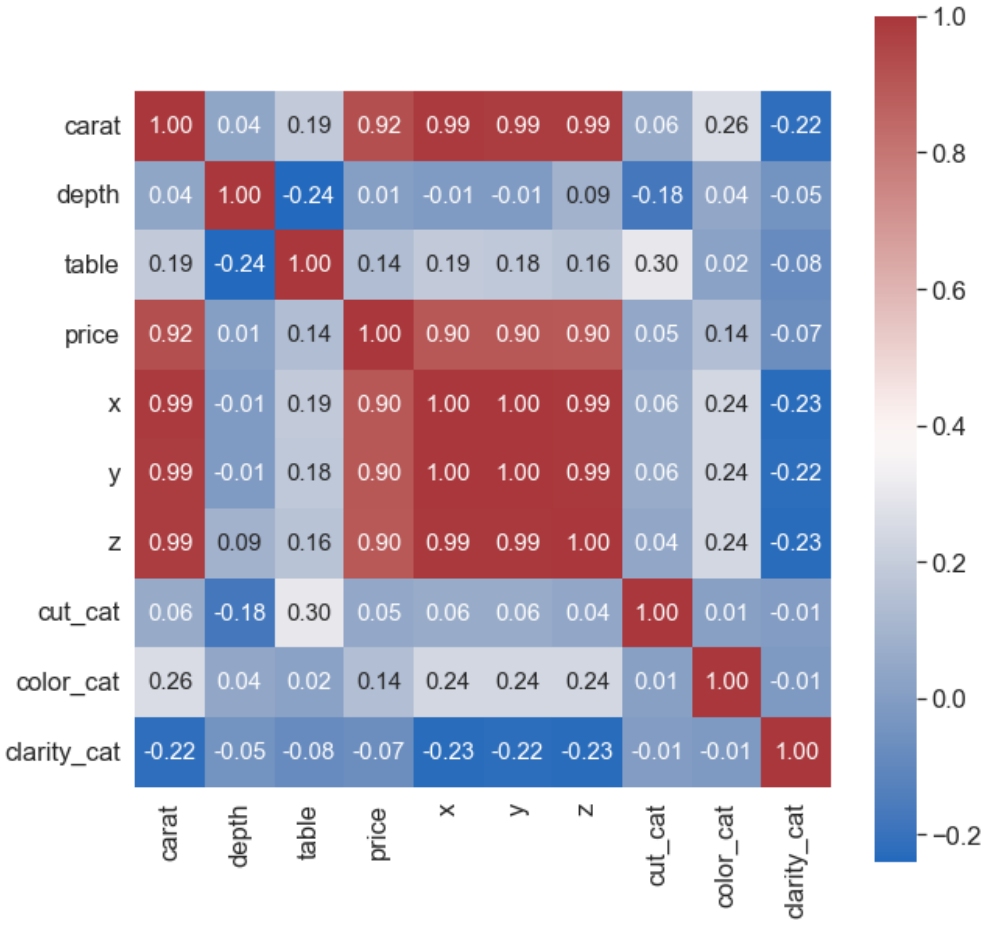

Figura 3: Matriz de correlaciones.

Observamos que *x,y* y *z* están muy correlacionadas tanto con el peso del diamante *carat* como entre ellas. Debido a que esta correlación es mayor que 0.7, descartaremos estas varables para que la multicolinealidad perfecta no sesgue los resultados.

3. Predicción del precio como variable continua

Definimos la variable dependiente y las independientes:

[15]:
```
X=df[["carat","depth","table","cut_cat","color_cat","clarity_cat"]]
y=df["price"]
```

Dividimos entre train (70 %) y test:

[16]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=3)
```

Creamos una tabla donde iremos guardando la raíz del error cuadrático medio (RMSE) y el error medio absoluto (MAE) de cada modelo:

[17]:
```
recopilatorio= pd.DataFrame({'RMSE':range(6),'MAE':range(6)},␣
 ↪index=['OLS','Ridge','Lasso','Polinómica','Red Elástica','Árbol'])
```

3.1. Mínimos Cuadrados Ordinarios

[18]:
```
from sklearn.linear_model import LinearRegression
```

[19]:
```
regr = LinearRegression(fit_intercept=False)
```

Entrenamos el modelo:

[20]:
```
regr.fit(X_train, y_train);
```

Predecimos:

[21]:
```
y_pred = regr.predict(X_test)
```

Creamos una función que nos devuelva las métricas de interés del modelo:

[22]:
```
import sklearn.metrics as metrics


def regression_results(y_true, y_pred, model):

    # Regression metrics

    mae=metrics.mean_absolute_error(y_true, y_pred)
    rmse=np.sqrt(metrics.mean_squared_error(y_true, y_pred) )
    r2=metrics.r2_score(y_true, y_pred)
    r2_adj= 1 - float(len(y_true)-1)/(len(y_true)-len(model.coef_)-1)*(1 -␣
 ↪metrics.r2_score(y_true,y_pred))

    return(pd.DataFrame({'R2':np.round(r2,3),'R2 ajustado':np.
 ↪round(r2_adj,3),'RMSE':np.round(rmse,3),'MAE':np.round(mae,3)}, index=[' ']))
```

[23]:
```
regression_results(y_test, y_pred,regr)
```

R^2	R^2 ajustado	RMSE	MAE
0.882	0.882	947.223	657.335

Tabla 3: Métricas MCO.

Observamos que la regresión lineal hace una buena predicción. El valor de R^2 es muy elevado y la raíz del error cuadrático medio (RMSE) es bastante bajo. Además, es notable el efecto de valores extremos sobre le RMSE al ser el MAE mucho menor.

[24]:
```
recopilatorio.loc['OLS']=[np.round(np.sqrt(metrics.mean_squared_error(y_test,␣
↪y_pred) ),2),np.round(metrics.mean_absolute_error(y_test, y_pred),2)]
```

[25]:
```
import statsmodels.api as sm


X2 = sm.add_constant(X_train)
est = sm.OLS(y_train, X_train)
est2 = est.fit()
print(est2.summary())
```

```
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                   price   R-squared (uncentered):
0.949
Model:                             OLS   Adj. R-squared (uncentered):
0.949
Method:                  Least Squares   F-statistic:
1.023e+05
Date:                 Sat, 19 Feb 2022   Prob (F-statistic):
0.00
Time:                         21:26:11   Log-Likelihood:
-2.7487e+05
No. Observations:               33266   AIC:
5.498e+05
Df Residuals:                   33260   BIC:
5.498e+05
Df Model:                           6
Covariance Type:            nonrobust
================================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
carat        7310.5220     14.910    490.303      0.000    7281.297    7339.747
depth          -9.4754      2.009     -4.716      0.000     -13.413      -5.537
table         -33.5561      2.246    -14.940      0.000     -37.959     -29.154
cut_cat         3.3916      5.736      0.591      0.554      -7.850      14.634
color_cat    -177.8908      3.167    -56.162      0.000    -184.099    -171.683
```

```
clarity_cat   226.1438      3.031     74.604      0.000     220.202     232.085
==============================================================================
Omnibus:                     9050.800   Durbin-Watson:                   2.002
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            41088.531
Skew:                           1.262   Prob(JB):                         0.00
Kurtosis:                       7.824   Cond. No.                         245.
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Vemos que todas las variables excepto *cut* son relevantes al 5 % de significación.

[26]:
```
plt.scatter(y_pred, y_test-y_pred);
plt.ylabel("$y_{test}-y_{pred}$");
plt.xlabel("$y_{pred}$");
```

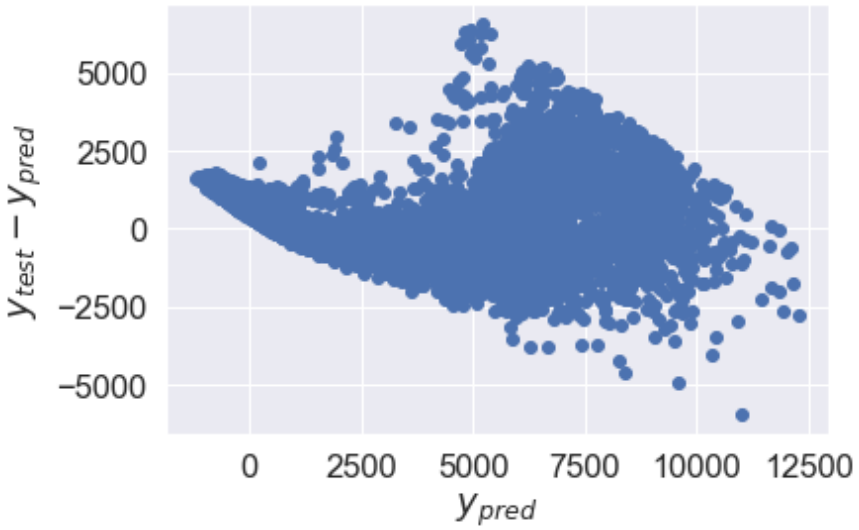

Figura 4: Dispersión del error de predicción de la Regresión Lineal.

En la Fig. 4 se aprecia un considerable sesgo, pues el error de predicción no tiene una distribución aleatoria.

3.2. Regresión Ridge

[27]:
```
from sklearn.linear_model import Ridge
```

Primeramente, encontraremos qué valor de α da mejores resultados:

[28]:
```
scores = []
best = (0,0)
alphas = np.logspace(-6, 6, 100)

for alpha in alphas:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train, y_train);

    sc = ridge.score(X_test,y_test)
    scores.append(sc)
    if sc > best[0]:
        best = sc, alpha

plt.plot(alphas, scores)
plt.xlabel("alpha")
plt.ylabel("scores")
plt.xscale("log")
plt.show()
```

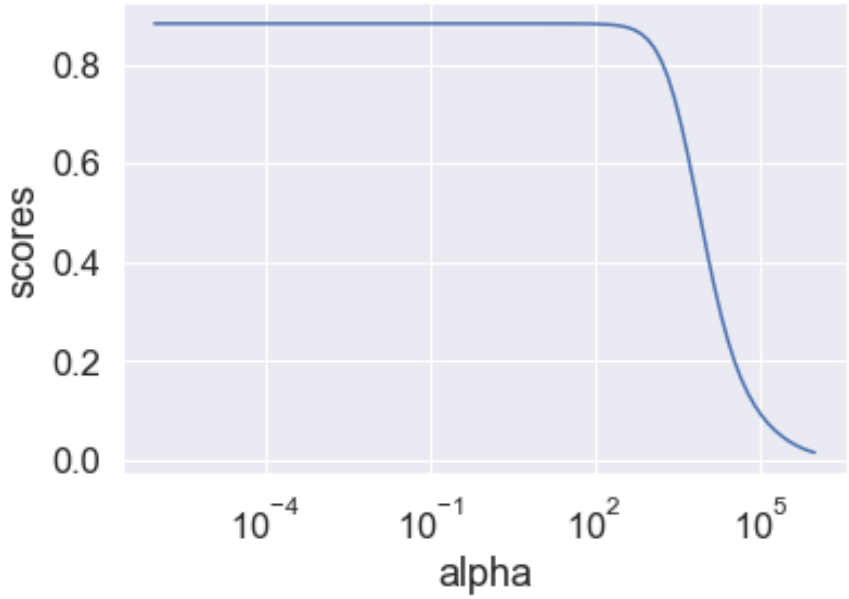

Figura 5: *Score* obtenido para distintos valores de α.

Hacemos un ajuste con el α con mayor *score*:

[29]:
```
ridge = Ridge(alpha=best[1])
```

[30]:
```
ridge.fit(X_train,y_train);
```

[31]:
```
y_pred=ridge.predict(X_test)
```

[32]:
```
regression_results(y_test, y_pred, ridge)
```

R^2	R^2 ajustado	RMSE	MAE
0.883	0.883	944.922	658.493

Tabla 4: Métricas Regresión *Ridge*.

[33]:
```
recopilatorio.loc['Ridge']=[np.round(np.sqrt(metrics.mean_squared_error(y_test,␣
↪y_pred) ),2),np.round(metrics.mean_absolute_error(y_test, y_pred),2)]
```

[34]:
```
plt.scatter(y_pred, y_test-y_pred);
plt.ylabel("$y_{test}-y_{pred}$");
plt.xlabel("$y_{pred}$");
```

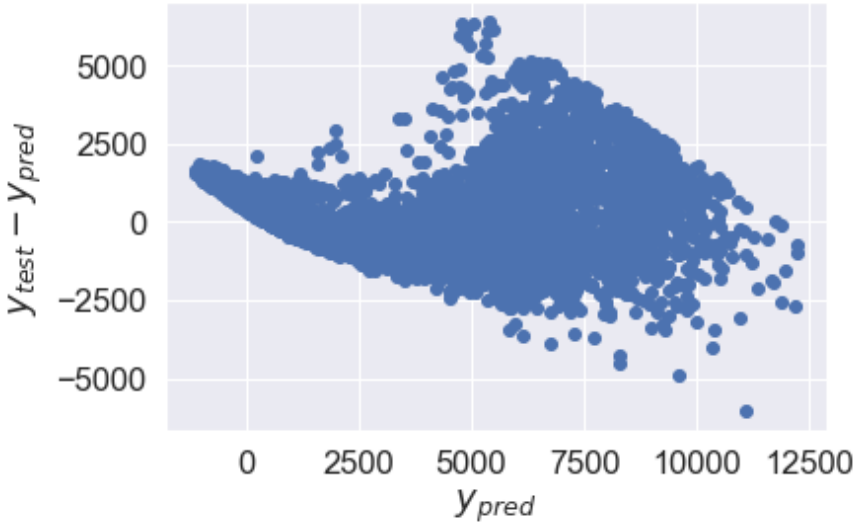

Figura 6: Dispersión del error de predicción de la regresión *Ridge*.

3.3. Regresión Lasso

[35]:
```
from sklearn.linear_model import Lasso
```

Encontraremos qué valor de α da mejores resultados:

[36]:
```
scores = []
best = (0,0)
alphas = np.logspace(-6, 6, 100)

for alpha in alphas:
    lasso = Lasso(alpha=alpha)
    lasso.fit(X_train, y_train);
```

```
    sc = lasso.score(X_test,y_test)
    scores.append(sc)
    if sc > best[0]:
        best = sc, alpha

plt.plot(alphas, scores)
plt.xlabel("alpha")
plt.ylabel("scores")
plt.xscale("log")
plt.show()
```

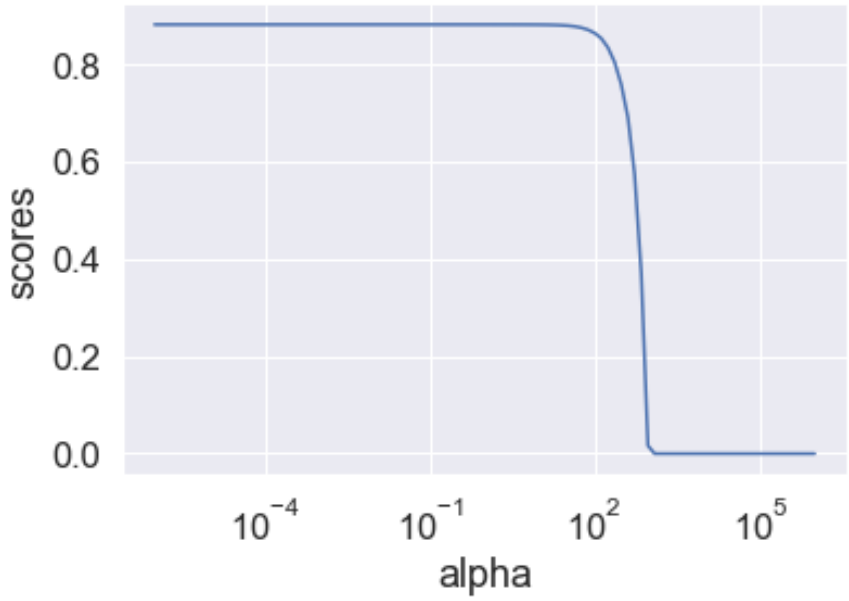

Figura 7: *Score* obtenido para distintos valores de α.

Con el valor de α encontrado ajustamos una nueva regresión: *Lasso*.

[37]:
```
lasso = Lasso(alpha=best[1])
lasso.fit(X_train, y_train);
```

[38]:
```
y_pred = lasso.predict(X_test)
```

[39]:
```
regression_results(y_test, y_pred, lasso)
```

R^2	R^2 ajustado	RMSE	MAE
0.883	0.883	944.922	658.493

Tabla 5: Métricas Regresión *Lasso*.

[40]:
```
recopilatorio.loc['Lasso']=[np.round(np.sqrt(metrics.mean_squared_error(y_test,␣
↪y_pred) ),2),np.round(metrics.mean_absolute_error(y_test, y_pred),2)]
```

[41]:
```
plt.scatter(y_pred, y_test-y_pred);
plt.ylabel("$y_{test}-y_{pred}$");
plt.xlabel("$y_{pred}$");
```

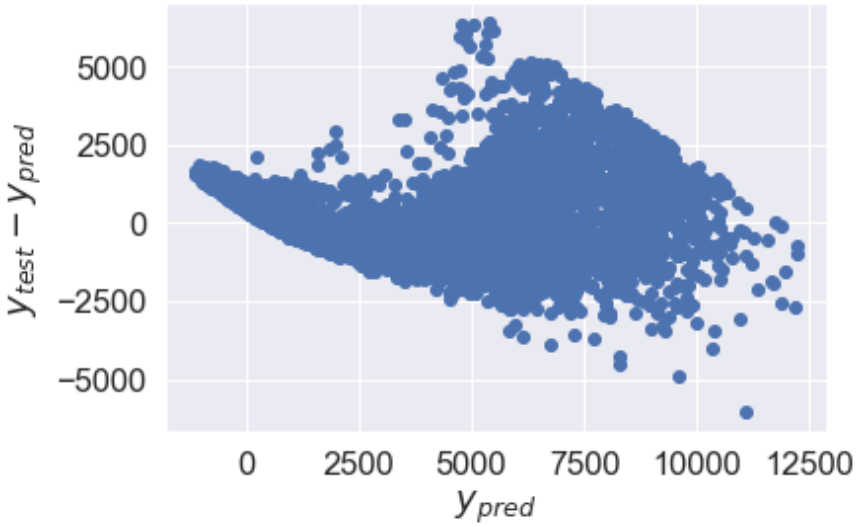

Figura 8: Dispersión del error de predicción de la regresión *Lasso*.

3.4. OLS vs Ridge vs Lassso

[42]:
```
df_coeficientes = pd.DataFrame(
                        {'predictor': X_train.columns,
                         'OLS': np.round(regr.coef_,4),
                         'Ridge': np.round(ridge.coef_,4),
                          'Lasso':np.round(lasso.coef_,4)}
                    )

df_coeficientes
```

	predictor	OLS	Ridge	Lasso
0	carat	7310.5220	7331.8830	7331.8830
1	depth	-9.4754	-72.0565	-72.0565
2	table	-33.5561	-56.9649	-56.9649
3	cut_cat	3.3916	2.4138	2.4138
4	color_cat	-177.8908	-177.4983	-177.4983
5	clarity_cat	226.1438	221.9005	221.9005

Tabla 6: Comparación de los coeficientes obtenidos.

Para el valor de α utilizado no hay ninguna diferencia entre *Lasso* y *Ridge*. Este primero no descarta ninguno de los predictores y ambos incrementan (en valor absoluto) los coeficientes de *carat*, *depth* y *table*, y disminuyen el resto. Además, si nos fijamos las Fig's 6 y 8, no solucionan el sesgo del MCO.

3.5. Regresión polinómica

[43]:
```
from sklearn.preprocessing import PolynomialFeatures
```

Calculamos el MSE para regresiones polinómicas de diferentes grados:

[44]:
```
mse_test=[]
mse_train=[]
grados= range(2,10)

for i in grados:
    polynomial_features = PolynomialFeatures(i)

    linear_regression = LinearRegression()
    polynomial_train = polynomial_features.fit_transform(X_train)
    polynomial_test = polynomial_features.transform(X_test)

    linear_regression.fit(polynomial_train,y_train)
    y_pred= linear_regression.predict(polynomial_test)
    y_pred_train=linear_regression.predict(polynomial_train)

    mse_test.append(metrics.mean_squared_error(y_test, y_pred))
    mse_train.append(metrics.mean_squared_error(y_train, y_pred_train))
```

[45]:
```
fig=plt.figure();
ax=fig.add_subplot(1,1,1)
ax.plot(grados,mse_test, label="test");
ax.plot(grados,mse_train, label="train");
ax.set_xlabel("Grados");
ax.set_ylabel("MSE");
ax.set_yscale("log");
ax.legend();
```

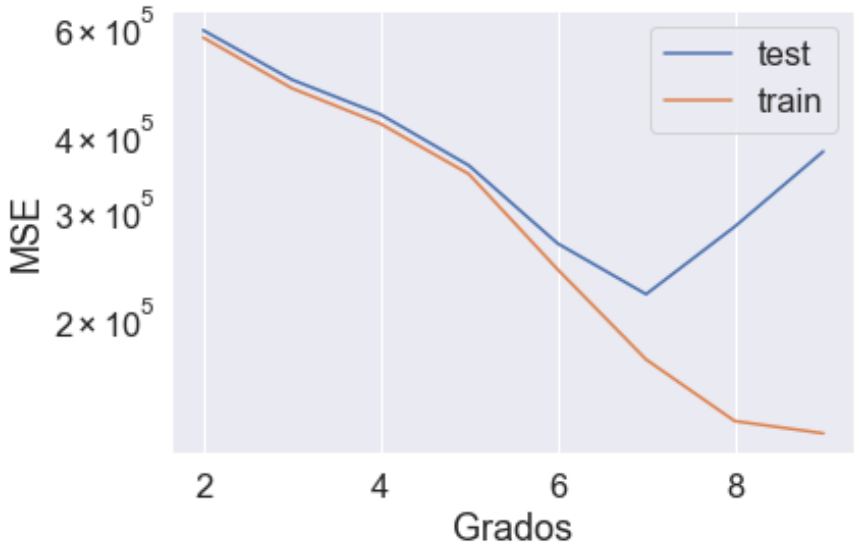

Figura 9: MSE en función del grado del ajute polinómico.

Hacemos un ajuste con el grado que da menor MSE:

[46]:
```
grados[int(np.where(mse_test==min(mse_test))[0])]
```

[46]:
```
7
```

[47]:
```
polynomial_features = PolynomialFeatures(grados[int(np.
 ↪where(mse_test==min(mse_test))[0])])

linear_regression = LinearRegression()
polynomial_train = polynomial_features.fit_transform(X_train)
polynomial_test = polynomial_features.transform(X_test)

linear_regression.fit(polynomial_train,y_train)

y_pred= linear_regression.predict(polynomial_test)
```

[48]:
```
regression_results(y_test, y_pred, linear_regression)
```

R^2	R^2 ajustado	RMSE	MAE
0.971	0.967	467.028	293.887

Tabla 7: Métricas de la Regresión Polinómica.

[49]:
```
recopilatorio.loc['Polinómica']=[np.round(np.sqrt(metrics.
 ↪mean_squared_error(y_test, y_pred) ),2),np.round(metrics.
 ↪mean_absolute_error(y_test, y_pred),2)]
```

[50]:
```
plt.scatter(y_pred, y_test-y_pred);
plt.ylabel("$y_{test}-y_{pred}$");
plt.xlabel("$y_{pred}$");
```

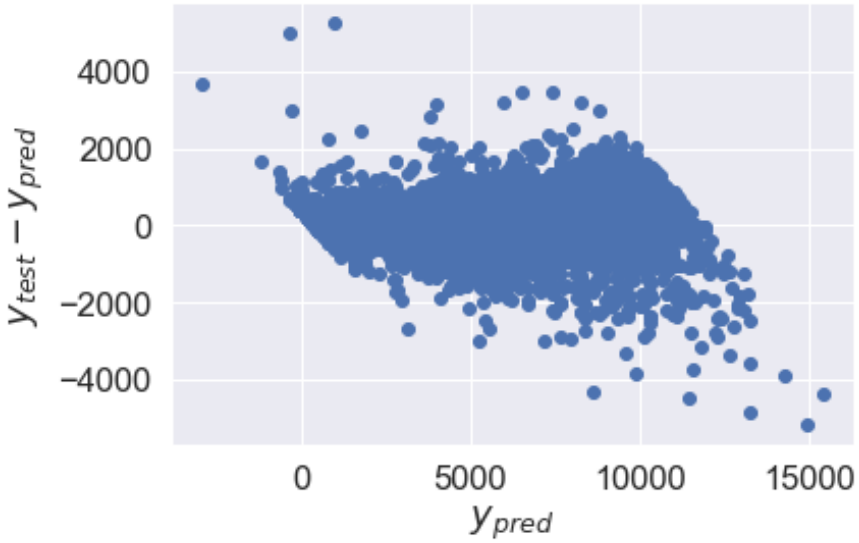

Figura 10: Disperción del error de predicción de la Regresión Polinómica.

3.6. Red Elástica

Aplicamos el método *Grid Search* para una búsqueda exhasutiva de los hiperparámetros de la red elástica α y *l1_ratio* óptimos.

[51]:
```
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import GridSearchCV
```

[52]:
```
param_grid ={'alpha':np.logspace(-10, 10, 10), 'l1_ratio':[0, 0.1, 0.5, 0.7, 0.
 ↪9, 0.95, 0.99]}

elast = ElasticNet()

best_elast = GridSearchCV(elast, param_grid)
best_elast.fit(X_train, y_train);
```

[53]:
```
pd.DataFrame(best_elast.best_params_, index=[' '])
```

	α	l1_ratio
	0.000464	0.99

Tabla 8: Valores óptimos de los hiperparámetros.

RMSE	MAE
944.92	658.48

Tabla 9: Métricas Red Elástica.

[54]:
```
y_pred=best_elast.predict(X_test)
```

[55]:
```
pd.DataFrame({'RMSE':np.round(np.sqrt(metrics.mean_squared_error(y_test,␣
 ↪y_pred)),2),
'MAE': np.round(metrics.mean_absolute_error(y_test, y_pred),2) }, index=[" "])
```

[56]:
```
recopilatorio.loc['Red Elástica']=[np.round(np.sqrt(metrics.
 ↪mean_squared_error(y_test, y_pred) ),2),np.round(metrics.
 ↪mean_absolute_error(y_test, y_pred),2)]
```

[57]:
```
plt.scatter(y_pred, y_test-y_pred);
plt.ylabel("$y_{test}-y_{pred}$");
plt.xlabel("$y_{pred}$");
```

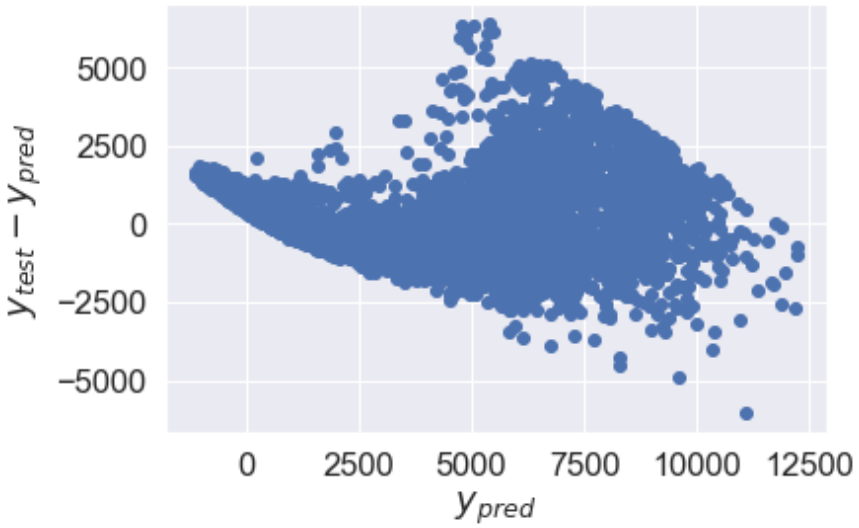

Figura 11: Dispersión del error de predicción de la Red Elástica.

3.7. Árbol de Regresión

[58]:
```
from sklearn.tree import DecisionTreeRegressor
```

En primer lugar, vamos a encontrar la profundidad del árbol que nos de mejores resultados.

[59]:
```
depths= range(1,30)

mse_test=[]
```

```
mse_train=[]

for i in depths:
    regr_tree = DecisionTreeRegressor(max_depth=i)
    regr_tree.fit(X_train, y_train);

    y_pred=regr_tree.predict(X_test)
    mse_test.append(metrics.mean_squared_error(y_test, y_pred))
    mse_train.append(metrics.mean_squared_error(y_train, y_pred_train))
```

[60]:
```
fig=plt.figure();
ax=fig.add_subplot(1,1,1)
ax.plot(depths,mse_test, label="test");
ax.plot(depths,mse_train, label="train");
ax.set_xlabel("Máxima profundidad");
ax.set_ylabel("MSE");
ax.set_yscale("log");
ax.legend();
```

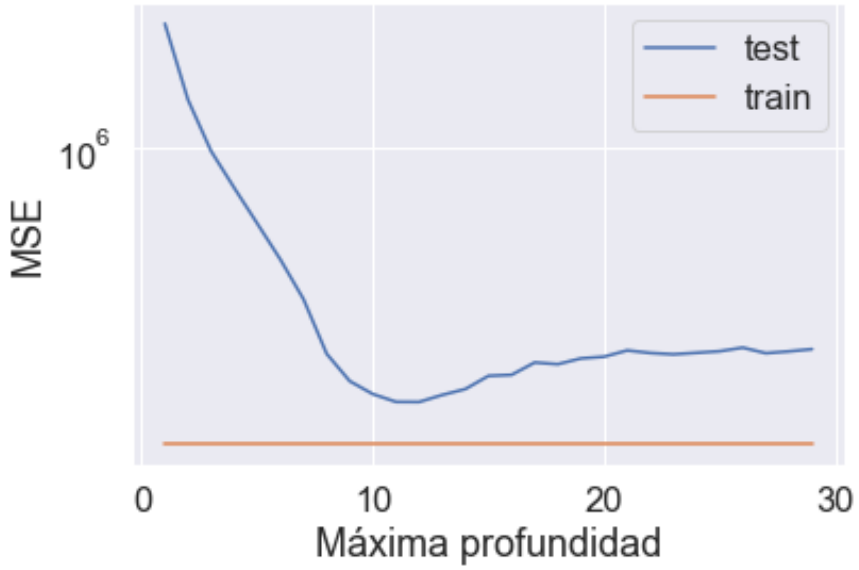

Figura 12: MSE en función de la profundidad máxima del árbol.

[61]:
```
depths[np.argmin(mse_test)]
```

[61]: 12

Ajustamos un árbol con esta profundidad:

[62]:
```
regr_tree = DecisionTreeRegressor(max_depth=depths[np.argmin(mse_test)])
regr_tree.fit(X_train, y_train);
```

```
y_pred=regr_tree.predict(X_test)


pd.DataFrame({'RMSE':np.round(np.sqrt(metrics.mean_squared_error(y_test,␣
 ↪y_pred)),2),
'MAE': np.round(metrics.mean_absolute_error(y_test, y_pred),2) }, index=[" "])
```

RMSE	MAE
413.66	239.32

Tabla 10: Métricas Árbol de Regresión.

[63]:
```
recopilatorio.loc['Árbol']=[np.round(np.sqrt(metrics.mean_squared_error(y_test,␣
 ↪y_pred) ),2),np.round(metrics.mean_absolute_error(y_test, y_pred),2)]
```

[64]:
```
plt.scatter(y_pred, y_test-y_pred);
plt.ylabel("$y_{test}-y_{pred}$");
plt.xlabel("$y_{pred}$");
```

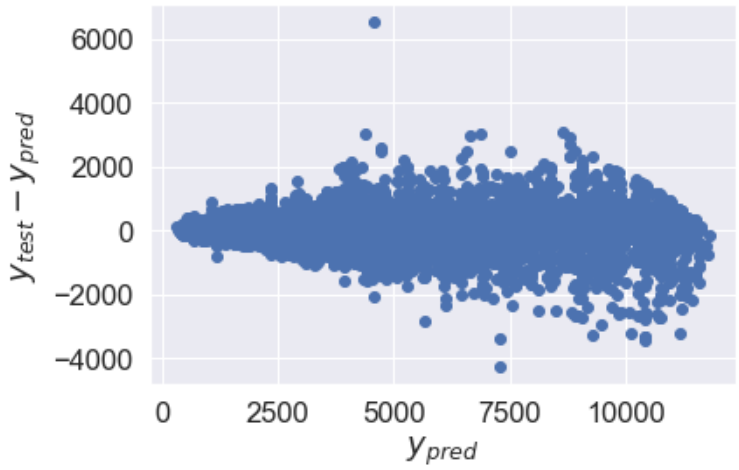

Figura 13: Dispersión del error de predicción del Árbol de regresión.

3.8. Resumen

[65]:
```
recopilatorio.transpose()
```

	OLS	Ridge	Lasso	Polinómica	Red Elástica	Árbol
RMSE	947.22	944.92	944.92	467.03	944.92	413.66
MAE	657.33	658.49	658.49	293.89	658.48	239.32

Tabla 11: Comparación de las métricas de los modelos utilizados.

El Árbol de Regresión es el que tiene menor error, seguido de la Regresión Polinómica. Ambos modelos muy susceptibles de hacer un sobre-ajuste y poco flexibles a la hora de adaptarse a nuevos datos. Por tanto, sería necesario comprobar mediante Validación Cruzada si realmente esos errores son representativos de la precisión del modelo.

Por otro lado, *Ridge, Lasso* y Red Elástica dan resultados prácticamente idénticos, ligeramente mejores que los de la Regresión Lineal.

4. Predicción del precio como variable categórica

[66]:
```
X=df[["carat","depth","table","cut_cat","color_cat","clarity_cat"]]
y=df["price_cat"]
```

[67]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=3)
```

[68]:
```
recopilatorio=pd.DataFrame({'Accuracy':range(7)}, index=['Perceptrón',␣
↪'Regresión Logística','SVM', 'Árbol','Bosque aleatorio','GTB','KNN'])
```

[69]:
```
from sklearn.multiclass import OneVsRestClassifier
```

En los clasificadores que no estén preparados para problemas con multiclase utilizaremos la técnica de *One vs the Rest*.

4.1. Perceptrón

[70]:
```
from sklearn.linear_model import Perceptron

perc = OneVsRestClassifier(Perceptron(tol=1e-3, random_state=3))
perc.fit(X_train, y_train);
```

[71]:
```
y_pred=perc.predict(X_test)
```

[72]:
```
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

      barato       0.99      0.89      0.94      8891
        caro       0.36      0.65      0.46      1425
    muy caro       0.00      0.00      0.00       757
      normal       0.62      0.71      0.66      3185

    accuracy                           0.78     14258
   macro avg       0.49      0.56      0.52     14258
weighted avg       0.79      0.78      0.78     14258
```

[73]:
```
recopilatorio.loc['Perceptrón']=np.round(metrics.accuracy_score(y_test,y_pred),3)
```

4.2. Regresión Logística

[74]:
```
from sklearn.linear_model import LogisticRegression

regr_log = LogisticRegression(random_state=3,multi_class='ovr')
regr_log.fit(X_train, y_train);
```

[75]:
```
y_pred=regr_log.predict(X_test)
```

[76]:
```
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

      barato       0.95      0.99      0.97      8891
        caro       0.60      0.20      0.31      1425
    muy caro       0.82      0.46      0.59       757
      normal       0.65      0.83      0.73      3185

    accuracy                           0.85     14258
   macro avg       0.75      0.62      0.65     14258
weighted avg       0.84      0.85      0.83     14258
```

[77]:
```
recopilatorio.loc['Regresión Logística']=np.round(metrics.
 ↪accuracy_score(y_test,y_pred),3)
```

4.3. Máquinas de Soporte de Vectores (SVM)

[78]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC , LinearSVC
```

[79]:
```
param_grid =  {'kernel':['linear', 'rbf'], 'C':[0.01,0.1]}
```

[80]:
```
svc = SVC(random_state=3)
clf = GridSearchCV(svc, param_grid)
clf.fit(X_train, y_train);
```

[81]:
```
pd.DataFrame(clf.best_params_, index=[' '])
```

	C	kernel
	0.1	linear

Tabla 12: Valores óptimos de los hiperparámetros.

[82]:
```
y_pred= clf.predict(X_test)
```

[83]:
```
print(metrics.classification_report(y_test, y_pred))
```

criterion	max_depth
gini	10

Tabla 13: Valores óptimos de los hiperparámetros.

```
              precision    recall  f1-score   support

      barato       0.96      0.98      0.97      8891
        caro       0.64      0.58      0.61      1425
    muy caro       0.76      0.63      0.69       757
      normal       0.79      0.83      0.81      3185

    accuracy                           0.89     14258
   macro avg       0.79      0.75      0.77     14258
weighted avg       0.88      0.89      0.88     14258
```

[84]:
```
recopilatorio.loc['SVM']=np.round(metrics.accuracy_score(y_test,y_pred),3)
```

4.4. Árbol de Decisión

[85]:
```
from sklearn.tree import DecisionTreeClassifier, plot_tree
```

[86]:
```
param_grid =  {'criterion':('entropy', 'gini'),"max_depth":[1,10,30]}
```

[87]:
```
tree = DecisionTreeClassifier( random_state=3)
clf=GridSearchCV(tree, param_grid)
clf.fit(X_train, y_train);
```

[88]:
```
pd.DataFrame(clf.best_params_, index=[' '])
```

[89]:
```
y_pred= clf.predict(X_test)
```

[90]:
```
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

      barato       0.98      0.98      0.98      8891
        caro       0.80      0.80      0.80      1425
    muy caro       0.83      0.84      0.83       757
      normal       0.89      0.90      0.89      3185

    accuracy                           0.93     14258
   macro avg       0.87      0.88      0.88     14258
weighted avg       0.93      0.93      0.93     14258
```

[91]:
```
recopilatorio.loc['Árbol']=np.round(metrics.accuracy_score(y_test,y_pred),3)
```

4.5. Bosques aleatorios de decisión

[92]:
```
from sklearn.ensemble import RandomForestClassifier
```

[93]:
```
param_grid =  {'criterion':('entropy', 'gini'), "n_estimators":
 ↪[1,20,50,100],"max_depth":[1,10,30]}
```

[94]:
```
rd_for = RandomForestClassifier(random_state=3)
clf = GridSearchCV(rd_for, param_grid)
clf.fit(X_train, y_train);
```

[95]:
```
pd.DataFrame(clf.best_params_, index=[' '])
```

	criterion	max_depth	n_estimators
	gini	30	100

Tabla 14: Valores óptimos de los hiperparámetros.

[96]:
```
y_pred=clf.predict(X_test)
```

[97]:
```
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

      barato       0.98      0.98      0.98      8891
        caro       0.82      0.83      0.83      1425
    muy caro       0.87      0.86      0.86       757
      normal       0.90      0.90      0.90      3185

    accuracy                           0.94     14258
   macro avg       0.89      0.89      0.89     14258
weighted avg       0.94      0.94      0.94     14258
```

[98]:
```
recopilatorio.loc['Bosque aleatorio']=np.round(metrics.
 ↪accuracy_score(y_test,y_pred),3)
```

4.6. Gradient Tree Boosting

[99]:
```
from sklearn.ensemble import GradientBoostingClassifier
```

[100]:
```
param_grid =  {"n_estimators":[1,20,50,100],"max_depth":[1,10,30],␣
 ↪'learning_rate':[0.01,0.1,1]}
```

[101]:
```
GBC = GradientBoostingClassifier(random_state=3)
clf = GridSearchCV(GBC, param_grid)
clf.fit(X_train, y_train);
```

[102]:
```
pd.DataFrame(clf.best_params_, index=[' '])
```

	learning_rate	max_depth	n_estimators
	0.1	10	20

Tabla 15: Valores óptimos de los hiperparámetros.

[103]:
```
y_pred=clf.predict(X_test)
```

[104]:
```
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

      barato       0.98      0.98      0.98      8891
        caro       0.83      0.83      0.83      1425
    muy caro       0.86      0.88      0.87       757
      normal       0.91      0.91      0.91      3185

    accuracy                           0.94     14258
   macro avg       0.89      0.90      0.90     14258
weighted avg       0.94      0.94      0.94     14258
```

[105]:
```
recopilatorio.loc['GTB']=np.round(metrics.accuracy_score(y_test,y_pred),3)
```

4.7. K-nearest neighbors

[106]:
```
from sklearn.neighbors import KNeighborsClassifier
```

[107]:
```
knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')
knn.fit(X_train, y_train)
y_pred=knn.predict(X_test)
```

[108]:
```
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

      barato       0.85      0.94      0.89      8891
        caro       0.57      0.45      0.51      1425
    muy caro       0.69      0.30      0.42       757
      normal       0.65      0.61      0.63      3185

    accuracy                           0.78     14258
```

```
   macro avg       0.69      0.58      0.61     14258
weighted avg       0.77      0.78      0.77     14258
```

[109]:
```
recopilatorio.loc['KNN']=np.round(metrics.accuracy_score(y_test,y_pred),3)
```

4.8. Resumen

[110]:
```
recopilatorio.transpose()
```

	Perceptrón	Regresión Logística	SVM	Árbol	Bosque aleatorio	GTB	KNN
Accuracy	0.781	0.845	0.886	0.934	0.942	0.944	0.783

Tabla 16: Comparación del *accuracy* de los distintos modelos utilizados.

Gradient Tree Boosting es el que da mejor *accuracy*, seguido de Bosque Aleatorio y Árbol de Decisión, con una diferencia muy pequeña entre ellos.

5. Predicción del corte

En esta sección nos limitaremos a mostrar los resultados obtenidos, ya que el proceso es exactamente igual al del apartado anterior.

Destacar que se ha descartado la variable *carat* al tener una muy alta correlación con el precio.

Además, para ahorrar carga computacional no se ha hecho ningún *Grid Search*.

[111]:
```
X=df[["depth","table","price","color_cat","clarity_cat"]]
y=df["cut_cat"]
```

[112]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=3)
```

[113]:
```
def resumen(X_train, y_train,X_test,y_test):

    modelos=[OneVsRestClassifier(Perceptron(tol=1e-3, random_state=3)),
             LogisticRegression(random_state=3,multi_class='ovr'),
             LinearSVC(random_state=3),
             DecisionTreeClassifier( random_state=3, criterion='gini',␣
 ↪max_depth=10),
             RandomForestClassifier(random_state=3,criterion='gini',␣
 ↪max_depth=10,n_estimators=100),
             GradientBoostingClassifier(random_state=3, n_estimators=100,␣
 ↪max_depth=10, learning_rate=1),
             KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')]
```

```
    results=pd.DataFrame({'Perceptrón':0,'Regresión Logística':0, 'SVM lineal':
↪0, 'Árbol de decisión':0, 'Bosques aleatorios':0, 'GTB':0, 'KNN':0},␣
↪index=['Accuracy'])
    kk=results.keys()

    for i in range(len(modelos)):
        clf=modelos[i]
        clf.fit(X_train,y_train)
        y_pred=clf.predict(X_test)
        results[kk[i]]= np.round(metrics.accuracy_score(y_test, y_pred),3)
    return(results)
```

[114]:
```
resumen(X_train, y_train,X_test, y_test)
```

	Perceptrón	Regresión Logística	SVM	Árbol	Bosque Aleatorio	GTB	KNN
Accuracy	0.093	0.571	0.446	0.733	0.74	0.561	0.556

Tabla 17: Comparación del *accuracy* de los distintos modelos utilizados.

En este caso los resultados no son tan buenos como en el anterior apartado. Era esperable si tenemos en cuenta la Fig. 3 , donde podemos ver que *cut* no tiene correlaciones tan importantes con el resto de variables como las que tiene *price*.

Bosque Aleatorio y Árbol de Decisión destacan por encima del resto, aunque el análisis que se ha hecho ha sido muy superficial.

Referencias

[1] *Diamonds analysis*, https://www.kaggle.com/heeraldedhia/regression-on-diamonds-dataset-95-score.