gomez_del_hierro_laboratorio

May 18, 2024

Asignatura

```
        </td>
        <td style="border: 1px #0098cd solid; background-color:#E6F4F9; color:#0098CD"><center>
        </td>
        <td style="border: 1px #0098cd solid; background-color:#E6F4F9; color:#0098CD; width:10
        </td>
    </tr>
    <tr style="width:100%;font-size:11pt">
        <td rowspan="2" style="border: 1px #0098cd solid"><center><b>Técnicas Multivaraintes y
        </td>
        <td style="border: 1px #0098cd solid; text-align:left">Apellidos: Gómez del Hierro</td>
        <td rowspan="2" style="border: 1px #0098cd solid"><center>20/05/2024</center>
        </td>
    </tr>
    <tr>
        <td style="border: 1px #0098cd solid; text-align:left">Nombres: Gonzalo Miguel
        </td>
    </tr>
        <tr style="width:100%;font-size:11pt">
        <td style="border: 1px #0098cd solid; background-color:#E6F4F9; color:#0098CD"><center>
        </td>
        <td colspan="2" style="border: 1px #0098cd solid; text-align:left">Laboratorio: Resolve
        </td>
    </tr>
```

[56]:
```
# importamos algunas librerías y funciones para usar durante el desarrollo de␣
 ↪la práctica
import sklearn
from sklearn.datasets import make_regression
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import itertools
import copy as cp
```

A partir de mi NI (71204274) y según las reglas recogidas en el enunciado de la práctica, definimos la semilla siguiente. La definimos como un string para poder acceder a las posiciones de manera más

cómoda. También, definimos los parámetros de entrada que necesita la función make_regression de sklearn tal y como se describe en el enunciado de la práctica.

[57]:
```
semilla = '72224274'
```

[58]:
```
# definimos las variables de entrada para crear la colección de datos según se␣
 ↪indica
n_samples = 200 + 10 * int(semilla[0])
n_features = 10 +  int(semilla[1]) + int(semilla[2])
n_informative = 10 + int(semilla[1])
noise = 10 * int(semilla[3])
```

[59]:
```
# generamos la colección de datos
X, y = make_regression(n_samples=n_samples, n_features=n_features,␣
 ↪n_informative=n_informative,
                       bias=2, noise=noise, random_state=int(semilla),␣
 ↪shuffle=False)
```

De esta manera, hemos generado una muestra con 270 observaciones (n_samples) compuestas por 14 variables independientes (n_features) y una variable objetivo. De estas 14 variables independientes, 2 son redundantes (n_features - n_informative), como también veremos más adelante con un método paso a paso guiado por el p-valor. Como se puede ver en la documentación de la librería, el parámetro noise se corresponde con la desviación estándar del ruido que se aplica sobre la muestra perfectamente lineal, para romper dicha linealidad. Como también se puede ver en la documentación, el parámetro bias es la ordenada en el origen del modelo lineal del que se parte. Es decir, si el parámetro noise fuera 0, este bias sería la constante (β_0) en el modelo lineal $y = \beta_0 + \beta_1 \cdot x_1 \cdot ... \cdot \beta_{14} \cdot x_{14}$. Se define el parámetro shuffle como False para que no se "barajen" observaciones y características.

A continuación, vamos a almacenar la información que sale de la función make_regression y vamos a recabar sus características haciendo uso de métodos propios de los DataFrames de Pandas.

[60]:
```
# Generamos un dataset apropiado con la info que tenemos
# etiquetas para las n_features variables independientes + la variable␣
 ↪independiente
columns = ['x{}'.format(i) for i in range(X.shape[1])]
columns.append('y')
```

[61]:
```
# generamos un dataframe apropiado
df = pd.DataFrame(np.hstack((X,y[:, None])), columns=columns)
```

[62]:
```
# describimos nuestro conjunto de datos con algunas funciones útiles
df
```

[62]:

	x0	x1	x2	x3	x4	x5	x6	\
0	0.315281	-1.284600	0.564583	-0.727306	0.263381	0.017403	-2.332820	
1	-0.602440	-0.831936	0.693117	-0.427931	1.100620	0.245356	1.559427	
2	0.794278	1.778578	0.552433	1.011247	0.259250	0.009512	-0.757721	

```
3   -0.996813 -0.415302 -0.370038  1.650984 -0.188385 -0.176537  0.026197
4   -0.679804  1.217816 -0.259442 -1.609947  0.669880  2.281330 -0.939036
..        …         …         …         …         …         …
265  0.444094 -1.122727 -0.883724  1.785362 -1.368745 -0.270355  0.766365
266 -0.238318  0.771682  1.776297 -0.638129 -0.565051 -0.185949  1.595666
267 -0.832529 -0.114558  1.164128 -0.743954  0.353762 -0.518621  1.125528
268  0.084676 -0.584892  0.267026 -2.096889  0.239488 -0.864785  1.345757
269 -0.620912  0.756763 -0.884207  0.233496 -0.589103 -1.274796 -0.403543

           x7        x8        x9       x10       x11       x12       x13  \
0    1.738110  0.016395  0.733640 -2.004806 -0.969114 -0.623471  1.322573
1   -1.025479  1.018934  0.008634 -0.363674  0.331092  0.422991 -2.527868
2    0.972232  0.112370 -0.318427 -0.524120 -0.484009 -0.957438  0.724373
3    0.150881 -0.290190  1.181442 -0.100682 -0.043348  1.309407 -0.927046
4    1.601044 -1.189196  0.018630 -0.759211 -1.379096  0.925069 -0.198745
..        …         …         …         …         …         …
265  2.478700  1.958399 -1.320300  1.714167  0.075454 -0.062392  1.504630
266 -0.039494 -0.783031 -0.774059 -0.673031  0.162668 -0.928633  0.129669
267  0.561812  0.398975 -0.317631  0.865496 -0.510306 -0.363431  0.972170
268  0.985853 -0.451914  0.114965 -0.776839  0.428191 -0.757286 -0.789057
269  0.568941  1.386052  0.006668 -0.091288  1.582465 -0.253343  0.041913

              y
0   -451.745047
1    108.002722
2     29.343925
3     64.951889
4   -192.166408
..          …
265  296.593400
266   41.641172
267   32.869988
268 -170.486885
269  -50.133525

[270 rows x 15 columns]
```

[63]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 270 entries, 0 to 269
Data columns (total 15 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   x0      270 non-null    float64
 1   x1      270 non-null    float64
 2   x2      270 non-null    float64
 3   x3      270 non-null    float64
```

```
 4   x4      270 non-null    float64
 5   x5      270 non-null    float64
 6   x6      270 non-null    float64
 7   x7      270 non-null    float64
 8   x8      270 non-null    float64
 9   x9      270 non-null    float64
 10  x10     270 non-null    float64
 11  x11     270 non-null    float64
 12  x12     270 non-null    float64
 13  x13     270 non-null    float64
 14  y       270 non-null    float64
dtypes: float64(15)
memory usage: 31.8 KB
```

[64]:
```
df.describe()
```

[64]:

	x0	x1	x2	x3	x4	x5
count	270.000000	270.000000	270.000000	270.000000	270.000000	270.000000
mean	-0.081585	-0.097058	-0.039994	-0.027497	-0.104199	0.050485
std	1.035451	0.961844	1.011112	0.991279	0.974281	0.941727
min	-2.754765	-2.642654	-2.682709	-2.641308	-2.299527	-2.441561
25%	-0.768866	-0.814567	-0.768364	-0.729374	-0.786953	-0.559967
50%	-0.119491	-0.106750	-0.048220	-0.051611	-0.139248	0.034343
75%	0.614231	0.602314	0.633426	0.637455	0.529370	0.583760
max	3.398219	2.014683	2.840378	2.590261	2.802959	2.419353

	x6	x7	x8	x9	x10	x11
count	270.000000	270.000000	270.000000	270.000000	270.000000	270.000000
mean	-0.020608	0.156459	-0.026023	-0.057981	0.104659	0.054468
std	0.998258	1.027737	1.051168	0.983672	0.953814	1.000935
min	-2.878151	-2.964314	-2.884744	-3.208742	-3.095005	-2.304660
25%	-0.658967	-0.505317	-0.722046	-0.705362	-0.554559	-0.711380
50%	-0.044463	0.194328	-0.094226	-0.083917	0.089433	0.062546
75%	0.687367	0.851522	0.724171	0.560182	0.663197	0.730935
max	2.503547	2.910772	3.446197	2.778748	2.889454	3.034686

	x12	x13	y
count	270.000000	270.000000	270.000000
mean	-0.053897	-0.053776	3.890033
std	0.974444	0.994570	186.003767
min	-2.698280	-2.637598	-652.962806
25%	-0.676127	-0.702190	-114.400174
50%	-0.094599	-0.085843	2.808690
75%	0.576332	0.614925	132.729315
max	2.750551	3.182717	555.141179

Del describe anterior podemos observar que todas las features (variables independientes) $x_0, ...x_{13}$ se distribuyen de maneras muy similar.

[65]:
```
df.head()
```

[65]:
```
          x0        x1        x2        x3        x4        x5        x6  \
0  0.315281 -1.284600  0.564583 -0.727306  0.263381  0.017403 -2.332820
1 -0.602440 -0.831936  0.693117 -0.427931  1.100620  0.245356  1.559427
2  0.794278  1.778578  0.552433  1.011247  0.259250  0.009512 -0.757721
3 -0.996813 -0.415302 -0.370038  1.650984 -0.188385 -0.176537  0.026197
4 -0.679804  1.217816 -0.259442 -1.609947  0.669880  2.281330 -0.939036

          x7        x8        x9       x10       x11       x12       x13  \
0  1.738110  0.016395  0.733640 -2.004806 -0.969114 -0.623471  1.322573
1 -1.025479  1.018934  0.008634 -0.363674  0.331092  0.422991 -2.527868
2  0.972232  0.112370 -0.318427 -0.524120 -0.484009 -0.957438  0.724373
3  0.150881 -0.290190  1.181442 -0.100682 -0.043348  1.309407 -0.927046
4  1.601044 -1.189196  0.018630 -0.759211 -1.379096  0.925069 -0.198745

             y
0 -451.745047
1  108.002722
2   29.343925
3   64.951889
4 -192.166408
```

[66]:
```
df.tail()
```

[66]:
```
           x0        x1        x2        x3        x4        x5        x6  \
265  0.444094 -1.122727 -0.883724  1.785362 -1.368745 -0.270355  0.766365
266 -0.238318  0.771682  1.776297 -0.638129 -0.565051 -0.185949  1.595666
267 -0.832529 -0.114558  1.164128 -0.743954  0.353762 -0.518621  1.125528
268  0.084676 -0.584892  0.267026 -2.096889  0.239488 -0.864785  1.345757
269 -0.620912  0.756763 -0.884207  0.233496 -0.589103 -1.274796 -0.403543

           x7        x8        x9       x10       x11       x12       x13  \
265  2.478700  1.958399 -1.320300  1.714167  0.075454 -0.062392  1.504630
266 -0.039494 -0.783031 -0.774059 -0.673031  0.162668 -0.928633  0.129669
267  0.561812  0.398975 -0.317631  0.865496 -0.510306 -0.363431  0.972170
268  0.985853 -0.451914  0.114965 -0.776839  0.428191 -0.757286 -0.789057
269  0.568941  1.386052  0.006668 -0.091288  1.582465 -0.253343  0.041913

              y
265  296.593400
266   41.641172
267   32.869988
268 -170.486885
269  -50.133525
```

Veamos la matriz de correlación de las variables independientes y la variable dependendiente, con el objetivo de tener una primera sensación de qué variable son más explicativas y también de si

podemos tener variables redundantes.

[67]:
```
corr = df.corr(method='pearson')
print(corr)
# Mapa de calor sobre las ocrrelaciones entre variables
sns.heatmap(corr, cmap='coolwarm')
```

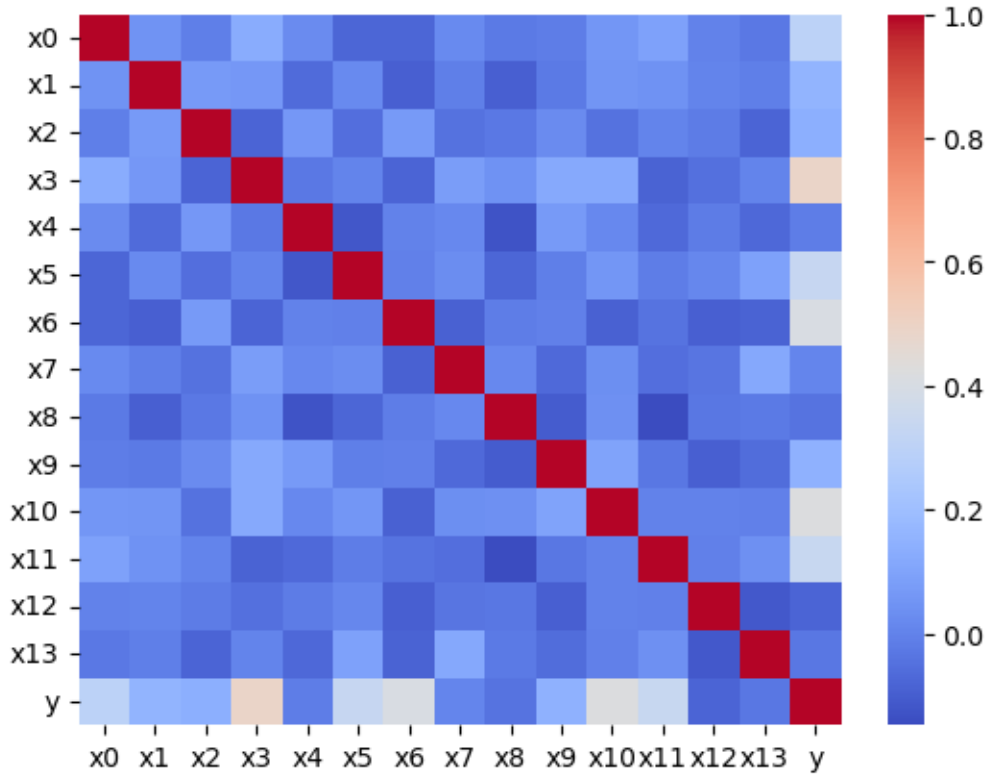
```
          x0        x1        x2        x3        x4        x5        x6  \
x0   1.000000  0.047332 -0.011633  0.129664  0.025830 -0.078759 -0.077773
x1   0.047332  1.000000  0.068519  0.058937 -0.067256  0.020036 -0.094876
x2  -0.011633  0.068519  1.000000 -0.084839  0.060474 -0.054768  0.068116
x3   0.129664  0.058937 -0.084839  1.000000 -0.030884  0.003324 -0.082605
x4   0.025830 -0.067256  0.060474 -0.030884  1.000000 -0.117472 -0.001086
x5  -0.078759  0.020036 -0.054768  0.003324 -0.117472  1.000000 -0.007852
x6  -0.077773 -0.094876  0.068116 -0.082605 -0.001086 -0.007852  1.000000
x7   0.021354 -0.009929 -0.045517  0.077318  0.013932  0.028494 -0.090591
x8  -0.023806 -0.096948 -0.030257  0.042612 -0.128249 -0.079345 -0.015678
x9  -0.017417 -0.024117  0.025911  0.117759  0.069818 -0.009953 -0.006790
x10  0.054988  0.049806 -0.046960  0.119235  0.013778  0.054078 -0.091679
x11  0.093502  0.043629  0.000696 -0.085921 -0.068100 -0.016295 -0.042769
x12 -0.000599  0.000527 -0.020067 -0.049748 -0.021411  0.009633 -0.095556
x13 -0.031297 -0.012387 -0.083664 -0.000254 -0.076547  0.089692 -0.089214
y    0.299256  0.156535  0.137085  0.492711 -0.017189  0.335290  0.406003

          x7        x8        x9       x10       x11       x12       x13  \
x0   0.021354 -0.023806 -0.017417  0.054988  0.093502 -0.000599 -0.031297
x1  -0.009929 -0.096948 -0.024117  0.049806  0.043629  0.000527 -0.012387
x2  -0.045517 -0.030257  0.025911 -0.046960  0.000696 -0.020067 -0.083664
x3   0.077318  0.042612  0.117759  0.119235 -0.085921 -0.049748 -0.000254
x4   0.013932 -0.128249  0.069818  0.013778 -0.068100 -0.021411 -0.076547
x5   0.028494 -0.079345 -0.009953  0.054078 -0.016295  0.009633  0.089692
x6  -0.090591 -0.015678 -0.006790 -0.091679 -0.042769 -0.095556 -0.089214
x7   1.000000  0.015899 -0.068974  0.033080 -0.056933 -0.037642  0.112974
x8   0.015899  1.000000 -0.103883  0.038049 -0.148341 -0.033422 -0.026280
x9  -0.068974 -0.103883  1.000000  0.096678 -0.032208 -0.095900 -0.060502
x10  0.033080  0.038049  0.096678  1.000000 -0.002815 -0.000532 -0.009077
x11 -0.056933 -0.148341 -0.032208 -0.002815  1.000000 -0.006429  0.037216
x12 -0.037642 -0.033422 -0.095900 -0.000532 -0.006429  1.000000 -0.112982
x13  0.112974 -0.026280 -0.060502 -0.009077  0.037216 -0.112982  1.000000
y    0.004628 -0.044128  0.144972  0.419797  0.340270 -0.084490 -0.035540

           y
x0   0.299256
x1   0.156535
x2   0.137085
x3   0.492711
x4  -0.017189
x5   0.335290
```

```
x6    0.406003
x7    0.004628
x8   -0.044128
x9    0.144972
x10   0.419797
x11   0.340270
x12  -0.084490
x13  -0.035540
y     1.000000
```

[67]: `<Axes: >`

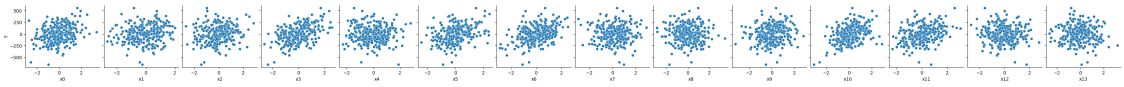


Como se ve en la figura, las variables independiente no están apenas correladas entre sí, y sí lo están con la variable independiente, aunque uno ya puede ver que las variables 4, 7, 8, 12 y 13 tienen un índice de correlación más bajo con la variable objetivo.

Vamos a representar la variable objetivo en función de cada una de las variables independientes, como se pide en el enunciado, pero no resulta muy informativo.

[68]:
```
sns.pairplot(df, x_vars=columns[0:-1], y_vars=columns[-1])
```

[68]: `<seaborn.axisgrid.PairGrid at 0x1c679cb7450>`



Podríamos dividir nuestro conjunto en simplemente las primeras 200 observaciones para el conjunto de entrenamiento y las restantes 70 para el conjunto de validación, pero por introducir herramientas establecidas en la industria, generemos nuestros conjuntos de manera aleatoria sobre la muestra.

[69]:
```
# Dividimos nuestro conjunto en validación y test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=70/270,␣
 ↪random_state=40)
df_train = pd.DataFrame(np.hstack((X_train,y_train[:, None])), columns=columns)
df_test = pd.DataFrame(np.hstack((X_test,y_test[:, None])), columns=columns)
```

Construimos el modelo de regresión lineal múltiple usando la muestra de entrenamiento.

[70]:
```
from sklearn.linear_model import LinearRegression
# Regresión lineal habitual por mínimos cuadrados
reg_ls = LinearRegression().fit(X_train, y_train)
```

[71]:
```
# evalúamos la función obtenida en la muestra de variables independientes␣
 ↪reservada para test
y_test_pred = reg_ls.predict(X_test)
```

[91]:
```
from sklearn.metrics import mean_squared_error, r2_score
# Coeficientes del modelo
print("Coeficientes: \n", reg_ls.coef_, "\n", "Intercepto: \n", reg_ls.
 ↪intercept_)
# Error cuadrático medio sobre la muestra de test
mean_error_lsm = cp.copy(mean_squared_error(y_test, y_test_pred))
print("Error cuadrático medio: %.2f" % r2_lsm)
# Coeficiente de determinación 1 significa predicción perfecta
r2_lsm = cp.copy(r2_score(y_test, y_test_pred))
print("Coeficiente de determinación R2: %.2f" % r2_lsm)
```

```
Coeficientes:
 [43.94602382 22.46183943 29.83640725 92.26844468 10.59352384 71.49254563
 97.85547196  3.45748611  9.52834508 15.67491137 72.3429387  73.63585407
  0.68818835 -0.58378327]
 Intercepto:
 2.6948483674190102
Error cuadrático medio: 0.99
Coeficiente de determinación R2: 0.99
```

La predicción es muy buena. Si atendemos a los coeficientes obtenidos para la regresión es claro que las variables 12 y 13 tienen un peso despreciable en el cáclulo de y, y por tanto podremos eliminarlas sin perder capacidad explicativa de la variable objetivo.

Podemos repetir la operación anterior excluyendo estas variables de una en una siguiendo algún criterio. También podríamos llevar a cabo, por ejemplo, un análisis de componentes principales y quedarnos con las componentes que necesitásemos para explicar la mayor parte de la varianza de la variable objetivo.

Solo en esta ocasión, sin falta de generalidad para las posteriores regresiones, probamos la normalidad de los residuos sobre la muestra de test.

[73]:
```
# obtener residuos sobre la muestra de entrenamiento
y_pred = reg_ls.predict(X_train)
residuos = y_train - y_pred
y_resta = residuos ** 2
import scipy.stats as stats
sh_result = stats.shapiro(residuos)

# dar formato a la salida
print("Test Shapiro-Wilk, p.valor: %5.5f" %(sh_result.pvalue))
print("Como p.valor > 0.05, no se rechaza la hipótesis nula y se da normalidad␣
 ↪en los residuos.")
```

```
Test Shapiro-Wilk, p.valor: 0.84126
Como p.valor > 0.05, no se rechaza la hipótesis nula y se da normalidad en los
residuos.
```

[74]:
```
import statsmodels.api as sm
import statsmodels.stats.api as sms
X2 = sm.add_constant(df_train[columns[0:-1]])
est = sm.OLS(y_train,X2) # ordinary least squares method for regression
est2 = est.fit()
bp1 = sms.het_breuschpagan(resid = est2.resid, exog_het = est2.model.exog)[1]
print("El resultado del test Breusch-Pagan es: p.valor = %5.3f"%(bp1))
print("Como p.valor > 0.05, no se rechaza la hipótesis nula y se da␣
 ↪homocedasticidad.")
```

```
El resultado del test Breusch-Pagan es: p.valor = 0.396
Como p.valor > 0.05, no se rechaza la hipótesis nula y se da homocedasticidad.
```

[75]:
```
# calcular ICbeta1

# calcular numerador sb1^2
s2 = sum(y_resta)/(len(y_train)-2)

# calcular denominador sb1^2
den = np.var(X) * len(X)
# calcular sb1
sb1 = (s2/den) ** 0.5
amplitud = 1.96 * sb1
print("El IC al 0.95 de b1 es:", reg_ls.coef_, "+/-", amplitud)
```

```
print("El intervalo de confianza contiene al 0 para los dos últimos␣
 ↪coeficientes, luego, las dos últimas variables (x12 y x13) no son␣
 ↪significativas.")
```

```
El IC al 0.95 de b1 es: [43.94602382 22.46183943 29.83640725 92.26844468
10.59352384 71.49254563
 97.85547196  3.45748611  9.52834508 15.67491137 72.3429387  73.63585407
  0.68818835 -0.58378327] +/- 2.349341987407838
El intervalo de confianza contiene al 0 para los dos últimos coeficientes,
luego, las dos últimas variables (x12 y x13) no son significativas.
```

[76]:
```
# Hagamos algo de análisis estadístico extra sobre la regresión lineal que se␣
 ↪obtiene meidante la librería statsmodel
print(est2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.989
Model:                            OLS   Adj. R-squared:                  0.988
Method:                 Least Squares   F-statistic:                     1216.
Date:                Sat, 18 May 2024   Prob (F-statistic):          8.28e-174
Time:                        09:57:54   Log-Likelihood:                -877.72
No. Observations:                 200   AIC:                             1785.
Df Residuals:                     185   BIC:                             1835.
Df Model:                          14
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.6948      1.506      1.789      0.075      -0.276       5.666
x0            43.9460      1.451     30.290      0.000      41.084      46.808
x1            22.4618      1.647     13.638      0.000      19.213      25.711
x2            29.8364      1.513     19.716      0.000      26.851      32.822
x3            92.2684      1.548     59.618      0.000      89.215      95.322
x4            10.5935      1.550      6.835      0.000       7.536      13.651
x5            71.4925      1.562     45.784      0.000      68.412      74.573
x6            97.8555      1.483     65.994      0.000      94.930     100.781
x7             3.4575      1.548      2.234      0.027       0.404       6.511
x8             9.5283      1.407      6.773      0.000       6.753      12.304
x9            15.6749      1.518     10.324      0.000      12.679      18.670
x10           72.3429      1.504     48.091      0.000      69.375      75.311
x11           73.6359      1.438     51.206      0.000      70.799      76.473
x12            0.6882      1.432      0.481      0.631      -2.136       3.513
x13           -0.5838      1.493     -0.391      0.696      -3.529       2.361
==============================================================================
Omnibus:                        0.457   Durbin-Watson:                   1.890
Prob(Omnibus):                  0.796   Jarque-Bera (JB):                0.600
Skew:                           0.068   Prob(JB):                        0.741
```

```
Kurtosis:                       2.769   Cond. No.                         1.61
==============================================================================
```

```
Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Nótese que para este análisis más exhaustivo hemos tenido que volver a calcular los parámetros de la regresión para aprovechar la infraestructura que ofrece statsmodels (cuando ya habíamos empleado la clase LinearRegression de sklearn). Además este análisis es como "matar moscas a cañonazos", puesto que por el momento no nos vamos a fijar mucho más que en los R^2, los coeficientes y los p-valores.

Atendiendo a la columna P>|t| (mayor valor implica menor significatividad estadística) de nuevo podemos observarque las variables x_{13} (x_{12} en el df) y x_{14} (x_{13} en el df) son las candidatas de no ser significativas.

Con un algoritmo paso a paso, guiados por el p-valor y con una condición de parada basada en el rango en que se encuentren todos los p-valores, vamos a eliminar las variable con mayor p-valor, rehacer la regresión, calcular los p-valores y rehacer la operación hasta que se cumpla la condición de salida.

[77]:
```
tol = 1
# comenzamos considerando todas las variables
x_columns = columns[0:-1]
idx = np.argwhere(est2.pvalues == max(est2.pvalues))[0][0]
iter = 0
removed_var = []
while tol > 1e-2 and idx != 0:
    iter += 1
    removed_var.append(x_columns[idx-1])
    x_columns.remove(x_columns[idx-1])
    X2 = sm.add_constant(df_train[x_columns])
    est_aux = sm.OLS(y_train,X2).fit()
    idx = np.argwhere(est_aux.pvalues == max(est_aux.pvalues))[0][0]
    tol = abs(max(est_aux.pvalues) - min(est_aux.pvalues))
print("Variables eliminadas:", removed_var)
print("Variables significativas:", x_columns)
print(est_aux.summary())
```

```
Variables eliminadas: ['x13', 'x12']
Variables significativas: ['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8',
'x9', 'x10', 'x11']
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.989
Model:                            OLS   Adj. R-squared:                  0.989
Method:                 Least Squares   F-statistic:                     1430.
Date:                Sat, 18 May 2024   Prob (F-statistic):          7.19e-177
Time:                        09:57:54   Log-Likelihood:                -877.95
```

```
No. Observations:                 200   AIC:                             1782.
Df Residuals:                     187   BIC:                             1825.
Df Model:                          12
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.7123      1.488      1.823      0.070      -0.223       5.647
x0            44.0283      1.435     30.676      0.000      41.197      46.860
x1            22.4708      1.640     13.702      0.000      19.236      25.706
x2            29.8990      1.503     19.888      0.000      26.933      32.865
x3            92.2451      1.541     59.870      0.000      89.206      95.285
x4            10.6756      1.529      6.983      0.000       7.660      13.692
x5            71.3962      1.542     46.314      0.000      68.355      74.437
x6            97.8362      1.465     66.782      0.000      94.946     100.726
x7             3.4146      1.540      2.218      0.028       0.377       6.452
x8             9.4998      1.397      6.798      0.000       6.743      12.256
x9            15.6263      1.501     10.413      0.000      12.666      18.587
x10           72.3530      1.498     48.314      0.000      69.399      75.307
x11           73.5675      1.428     51.510      0.000      70.750      76.385
==============================================================================
Omnibus:                        0.311   Durbin-Watson:                   1.888
Prob(Omnibus):                  0.856   Jarque-Bera (JB):                0.461
Skew:                           0.051   Prob(JB):                        0.794
Kurtosis:                       2.788   Cond. No.                         1.60
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

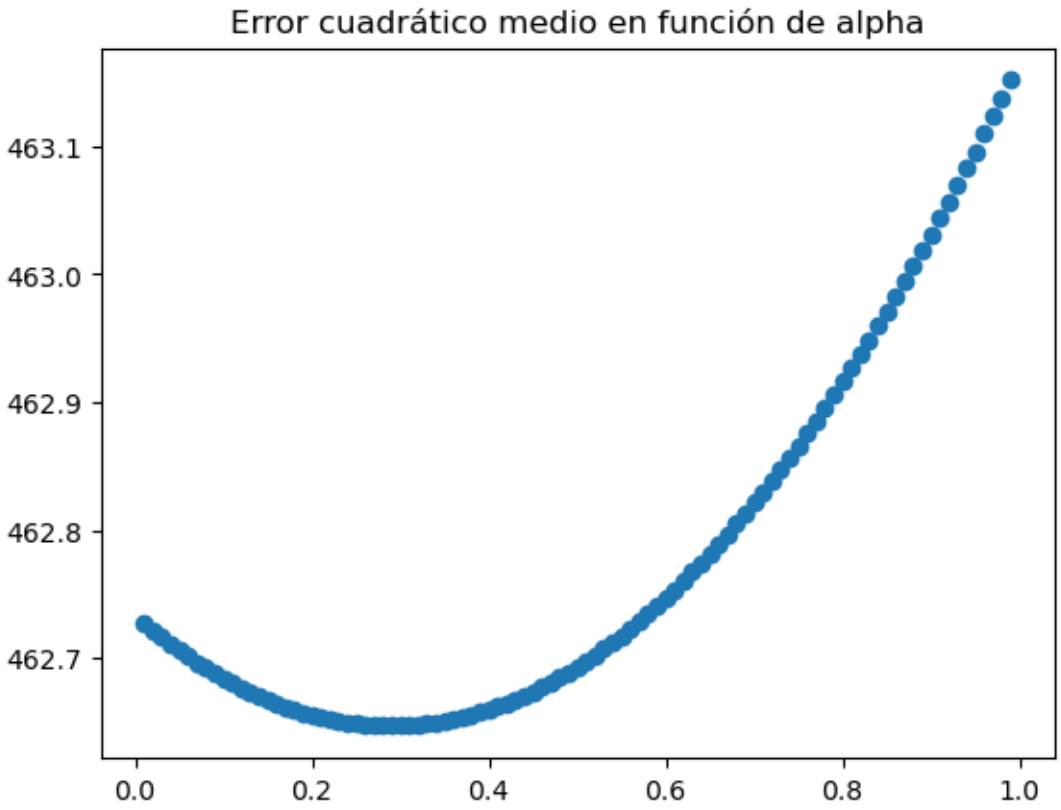
Como ya habíamos anticipado a través de varios indicadores, las variables no significativas eran las dos últimas, y al eliminarlas no hemos perdido bondad en el ajuste.

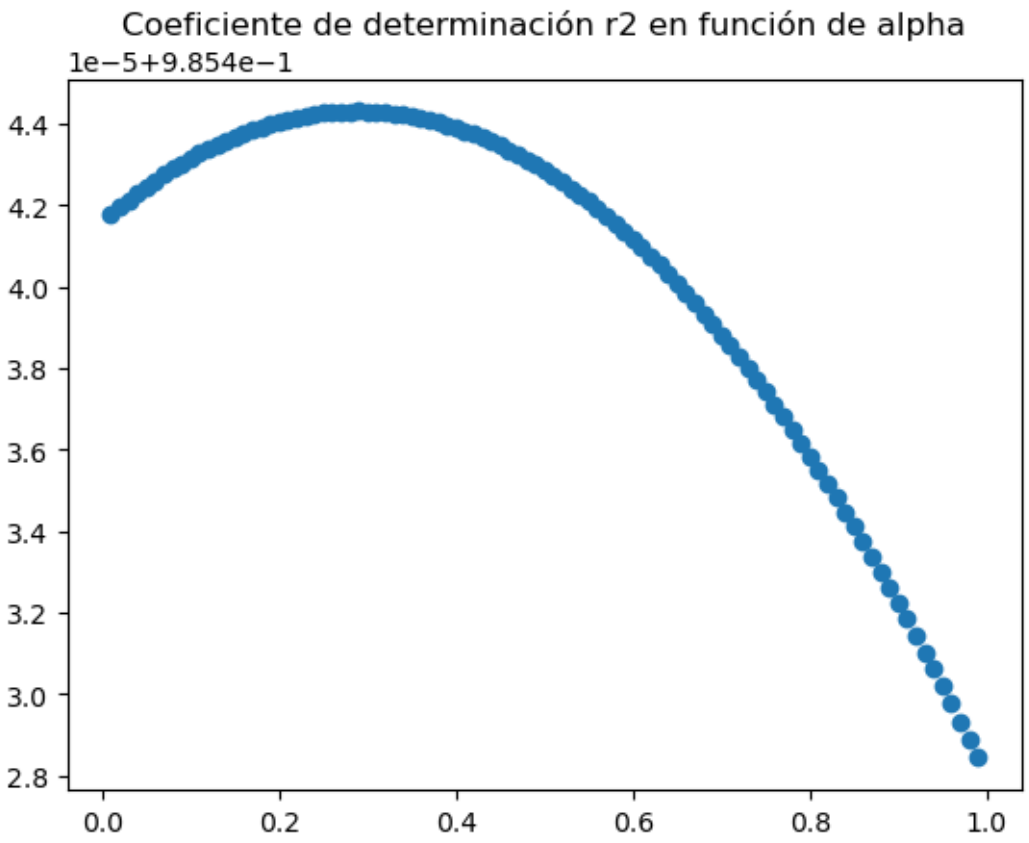
A continuación, sobre las variables ya determinadas como significativas: ridge, lasso y red elástica, a través de sklearn.

La regresión de Ridge es una regresión por mínimos cuadrados con una penalización $\mathcal{L}^2$, es decir, una penalización proporcional a las normas 2, pesada por el parámetro α.

[78]:
```python
"""Ridge linear regression depending on alpha"""
from sklearn.linear_model import Ridge
def ridge_test(alpha_list):
    ridge_mean_error = []
    ridge_r2 = []
    for a in alpha_list:
        Ridge_aux = Ridge(alpha=a).fit(df_train[x_columns], y_train)
        y_pred = Ridge_aux.predict(df_test[x_columns])
        ridge_mean_error.append(mean_squared_error(y_test,y_pred))
```

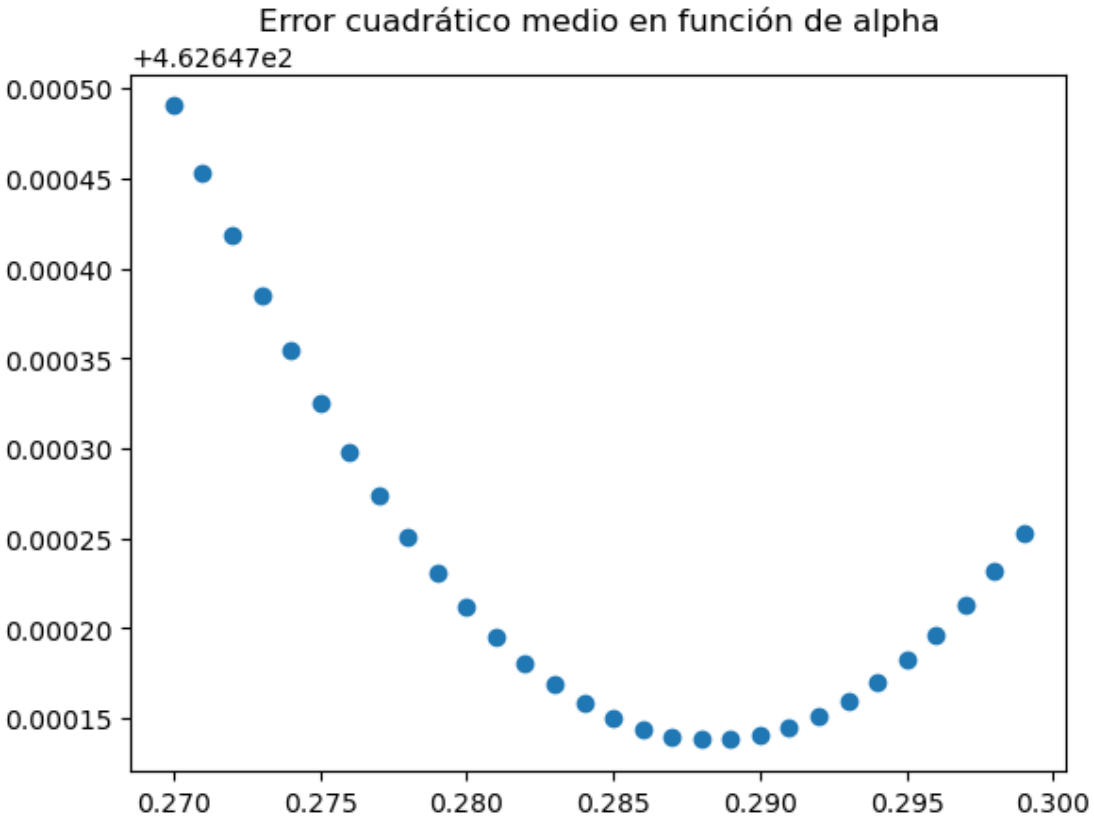
```
        ridge_r2.append(r2_score(y_test,y_pred))
    f, ax = plt.subplots()
    ax.scatter(alpha_list, ridge_mean_error)
    ax.set_title('Error cuadrático medio en función de alpha')
    plt.show()
    f, ax = plt.subplots()
    ax.scatter(alpha_list, ridge_r2)
    ax.set_title('Coeficiente de determinación r2 en función de alpha')
    plt.show()
    return ridge_mean_error, ridge_r2
mean_error, r2_values = ridge_test(np.arange(0.01,1,0.01))
```

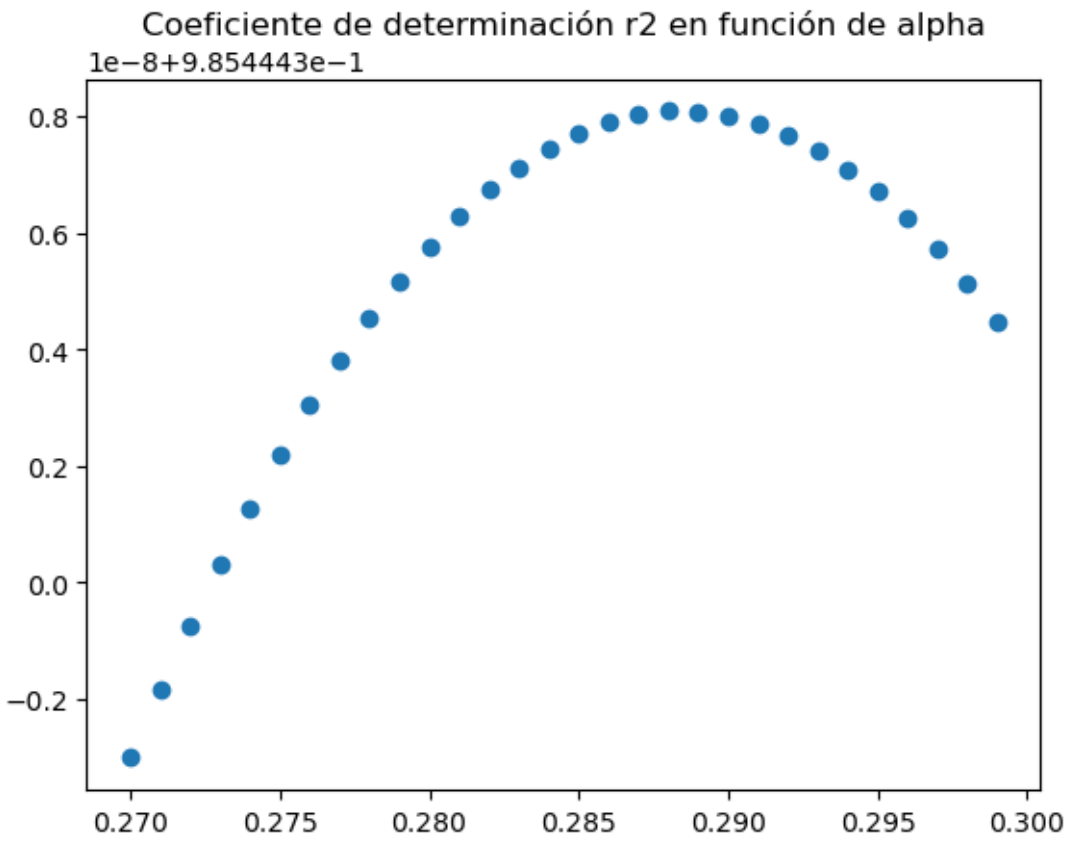




De las gráficas anteriores se observa un mínimo en cuanto a error y máximo en cuanto a R^2 en torno a *alpha* $= 0.3$, pero podemos determinarlo de manera numérica (lógicamente, el valor de alpha que minimiza el error cuadrático medio también maximiza el coeficiente de determinación, así que sería suficiente con hallar el extremo de la curva $R^2(\alpha)$ para obtener un valor óptimo, al menos localmente).

[79]:
```
# hagamos zoom en torno a la zona óptima
mean_error_ridge, r2_values_ridge = ridge_test(np.arange(0.27,0.30,0.001))
```


Error cuadrático medio en función de alpha
+4.62647e2
0.00050
0.00045
0.00040
0.00035
0.00030
0.00025
0.00020
0.00015
0.270
0.275
0.280
0.285
0.290
0.295
0.300

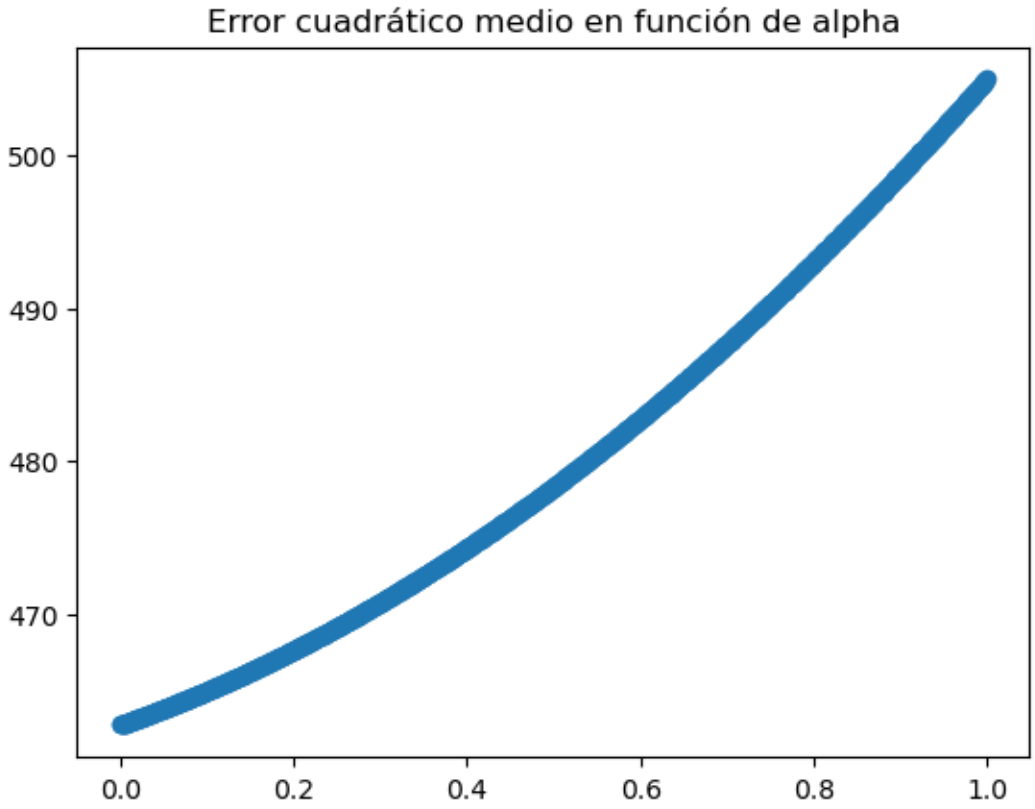


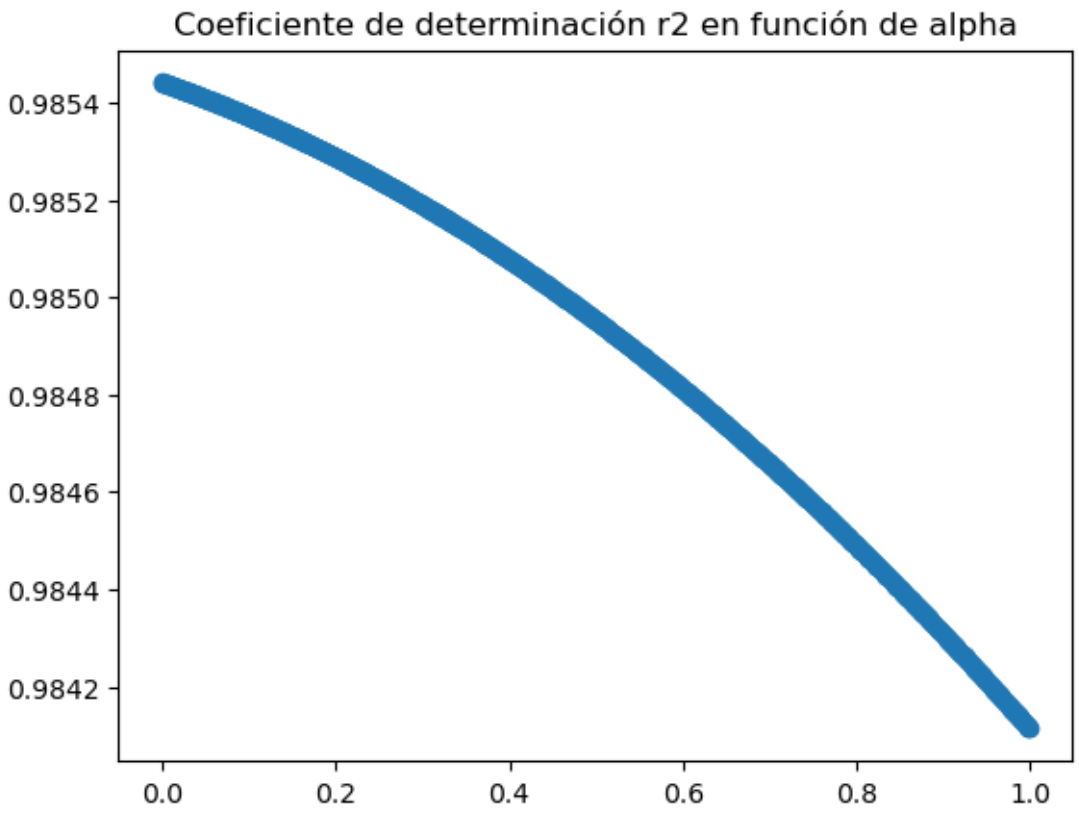
[80]:
```
idx_optim = np.argwhere(r2_values_ridge == max(r2_values_ridge))[0][0]
ridge_r2_optim = cp.copy(r2_values[idx_optim])
ridge_mean_error_optim = cp.copy(mean_error[idx_optim])
print("El valor alpha óptimo es: {:.4f}".format((np.arange(0.27,0.30,0.
  ↪001))[idx_optim]))
print("El R2 para la regresión de Ridge y alpha óptimo es: {:.7f}".
  ↪format(r2_values[idx_optim]))
print("El error cuadrático medio para la regresión de Ridge y alpha óptimo es:␣
  ↪{:.2f}".format(mean_error[idx_optim]))
```

```
El valor alpha óptimo es: 0.2880
El R2 para la regresión de Ridge y alpha óptimo es: 0.9854440
El error cuadrático medio para la regresión de Ridge y alpha óptimo es: 462.66
```

Pasemos a hacer un tratamiento análogo pero con la regresión de Lasso. La regresión de Lasso es una regresión por mínimos cuadrados con una penalización $\mathcal{L}^1$, es decir, una penalización proporcional a las normas 1, pesada por el parámetro α. Como vemos a continuación, esta penalización no parece ser una buena opción para nuestra muestra, pues nuestro óptimo en cuanto a bondad del ajuste lo encontramos para el menor valor de α considerado.

[81]:
```
"""Lasso linear regression depending on alpha"""
from sklearn.linear_model import Lasso
def lasso_test(alpha_list):
    lasso_mean_error = []
    lasso_r2 = []
    for a in alpha_list:
        lasso_aux = Lasso(alpha=a).fit(df_train[x_columns], y_train)
        y_pred = lasso_aux.predict(df_test[x_columns])
        lasso_mean_error.append(mean_squared_error(y_test,y_pred))
        lasso_r2.append(r2_score(y_test,y_pred))
    f, ax = plt.subplots()
    ax.scatter(alpha_list, lasso_mean_error)
    ax.set_title('Error cuadrático medio en función de alpha')
    plt.show()
    f, ax = plt.subplots()
    ax.scatter(alpha_list, lasso_r2)
    ax.set_title('Coeficiente de determinación r2 en función de alpha')
    plt.show()
    return lasso_mean_error, lasso_r2
mean_error_lasso, r2_values_lasso = lasso_test(np.arange(0.00001,1,0.001))
```





[82]:
```
idx_optim = np.argwhere(r2_values_lasso == max(r2_values_lasso))[0][0]
lasso_r2_optim = cp.copy(r2_values_lasso[idx_optim])
lassos_mean_error_optim = cp.copy(mean_error_lasso[idx_optim])
print("El valor alpha óptimo es: {:.5f}".format((np.arange(0.00001,1,0.
  ↪001))[idx_optim]))
print("El R2 para la regresión de Lasso y alpha óptimo es: {:.7f}".
  ↪format(r2_values_lasso[idx_optim]))
print("El error cuadrático medio para la regresión de Lasso y alpha óptimo es:␣
  ↪{:.2f}".format(mean_error_lasso[idx_optim]))
```

```
El valor alpha óptimo es: 0.00001
El R2 para la regresión de Lasso y alpha óptimo es: 0.9854416
El error cuadrático medio para la regresión de Lasso y alpha óptimo es: 462.73
```

Ahora hagamos lo mismo pero con el método de la red elástica, donde en vez de un parámetro manejamos dos (r y α). Vamos a hacer una función similar a las anteriores, pero que tomará como entrada dos listas en vez de una. En la documentación referida a esta clase en sklearn, se específica que el parámetro r debe estar entre 0 y 1, y que para $r = 1$ recuperaríamos el modelo de Lasso.

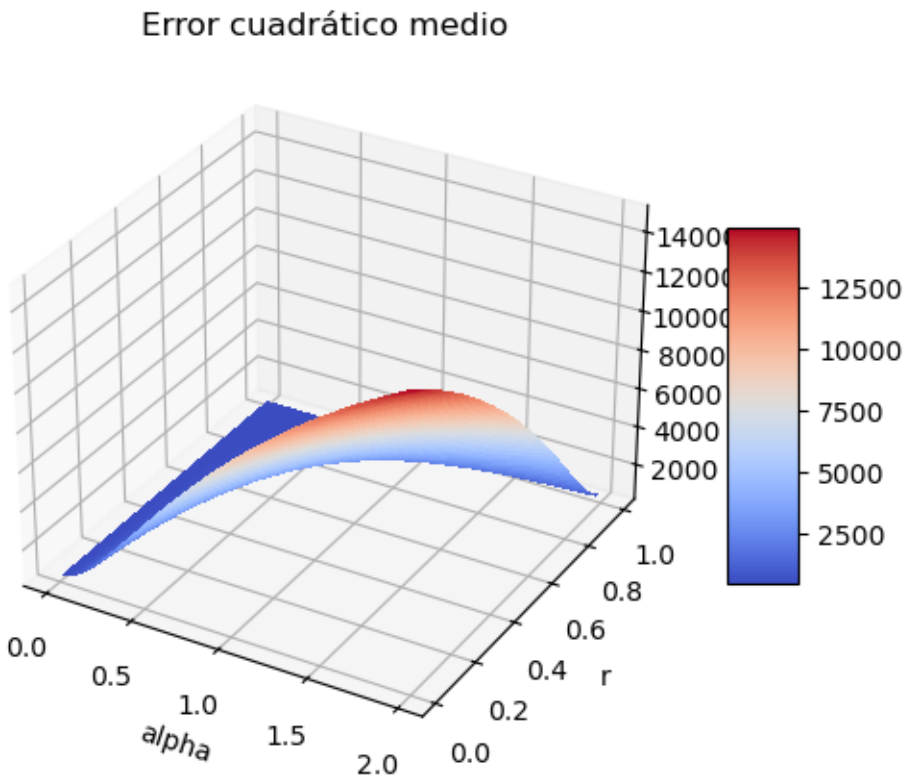
En este caso, atendiendo a la documentación de sklearn, donde podemos encontrar lo siguiente:

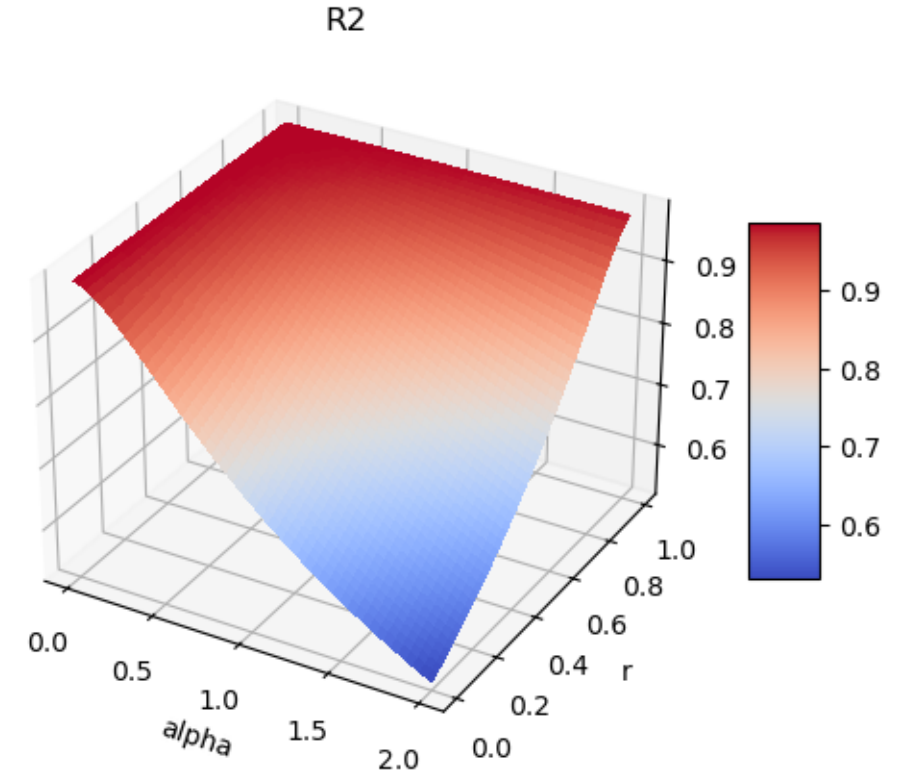
Minimizes the objective function:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

Atendiendo a los valores óptimos que encontramos, la penalización en norma 2 será menos significativa que la penalización en norma 1, lo que concuerda con que en la regresión de Lasso obtuviéramos un α tan pequeño, y es que parece que no es el mejor camino para ajustar nuestra muestra.

[85]:
```
"""Elastic Net linear regression depending on alpha and r"""
from sklearn.linear_model import ElasticNet
from matplotlib import cm
def elastic_net_test(alpha_list, r_list):
    a, r = np.meshgrid(alpha_list,r_list)
    en_mean_error = np.ones_like(a)
    en_r2 = np.ones_like(en_mean_error)
    for j in range(len(r_list)):
        for i in range(len(alpha_list)):
            en_aux = ElasticNet(alpha=alpha_list[i], l1_ratio=r_list[j]).
 ↪fit(df_train[x_columns], y_train)
            y_pred = en_aux.predict(df_test[x_columns])
            en_mean_error[j,i] = mean_squared_error(y_test,y_pred)
            en_r2[j][i] = r2_score(y_test,y_pred)
    f1, ax1 = plt.subplots(subplot_kw={"projection": "3d"})
    surf1 = ax1.plot_surface(a, r, en_mean_error, linewidth=0,cmap=cm.coolwarm,␣
 ↪antialiased=False)
    ax1.set_xlabel('alpha')
    ax1.set_ylabel('r')
    ax1.set_title('Error cuadrático medio')
    f1.colorbar(surf1, shrink=0.5, aspect=5)
    plt.show()
    f2, ax2 = plt.subplots(subplot_kw={"projection": "3d"})
    surf2 = ax2.plot_surface(a, r, en_r2, linewidth=0,cmap=cm.coolwarm,␣
 ↪antialiased=False)
    ax2.set_xlabel('alpha')
    ax2.set_ylabel('r')
    ax2.set_title('R2')
    f2.colorbar(surf2, shrink=0.5, aspect=5)
    plt.show()
    return en_mean_error, en_r2
mean_error_en, r2_values_en = elastic_net_test(np.arange(0.01,2,0.01), np.
 ↪arange(0.01,1,0.01))
```


Error cuadrático medio
14000
12000
10000
8000
6000
4000
2000
12500
10000
7500
5000
2500
0.0
0.5
1.0
1.5
2.0
alpha
0.0
0.2
0.4
0.6
0.8
1.0
r


R2
0.9
0.8
0.7
0.6
0.9
0.8
0.7
0.6
0.0
0.5
1.0
1.5
2.0
alpha
0.0
0.2
0.4
0.6
0.8
1.0
r

[89]:
```
idx_optim = np.argwhere(r2_values_en == np.max(r2_values_en))
idx_optim = idx_optim[0]
en_r2_optim = cp.copy(r2_values_en[idx_optim[0]][idx_optim[1]])
en_mean_error_optim = cp.copy(mean_error_en[idx_optim[0]][idx_optim[1]])
print("El valor alpha óptimo es: {:.5f}".format((np.arange(0.1,10,0.
 ↪1))[idx_optim[1]]))
print("El valor r óptimo es: {:.5f}".format((np.arange(0.01,1,0.
 ↪01))[idx_optim[0]]))
print("El R2 para alpha y r óptimos: {:.7f}".
 ↪format(r2_values_en[idx_optim[0]][idx_optim[1]]))
print("El error cuadrático medio para alpha y r óptimos: {:.2f}".
 ↪format(mean_error_en[idx_optim[0]][idx_optim[1]]))
```

```
El valor alpha óptimo es: 0.10000
El valor r óptimo es: 0.84000
El R2 para alpha y r óptimos: 0.9854382
El error cuadrático medio para alpha y r óptimos: 462.84
```

Por último, comparemos los R^2 y los errores cuadráticos medios obtenidos en el mejor ajuste por cada uno de los métodos. Nótese que estos indicadores se han obtenido a partir de la comparación de la predicción y los valores reales en la muestra de test.

[93]:
```
list_reg_method = ['LS Basic Method', 'Ridge', 'Lasso', 'Elastic Net']
r2_list = [r2_lsm, ridge_r2_optim, lasso_r2_optim, en_r2_optim]
mean_error_list = [mean_error_lsm, ridge_mean_error_optim,␣
 ↪lassos_mean_error_optim, en_mean_error_optim]
data_reg = {'Method' : list_reg_method, 'R2': r2_list, 'MSE': mean_error_list}
df_reg = pd.DataFrame(data=data_reg)
print(df_reg)
```

```
            Method        R2         MSE
0  LS Basic Method  0.985388  464.446828
1            Ridge  0.985444  462.657176
2            Lasso  0.985442  462.733718
3      Elastic Net  0.985438  462.840047
```

Con el nivel de precisión con el que se ha llevado a cabo el desarrollo, podemos afirmar que ciertamente los métodos de Ridge, Lasso y red elástica mejoran a la regresión por mínimos cuadrados básica, pero no exhibiría conclusiones muy fuertes sobre la posterior comparación entre ellos. Sí que, guiados por las resultados obtenidos, podemos afirmar que el camino de Ridge (penalización en norma 1) es algo más apropiada que la penalización en norma 2 que se emplea en Lasso y también en la cuerda elástica, pero teniendo en cuenta la similitud entre los resultados y los grids de parámetros sobre los que hemos evaluado las regresiones, no afirmaría que red elástica sea peor que Lasso (aunque extrictamente el resultado obtenido sí lo sea).

Bibliografía: - Trevor, et al. The elements of statistical learning: data mining, inference, and prediction. New York: springer, 2009. - Docu-

mentación sobre las funciones principales utilizadas de sklearn: - https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html - https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html - https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html#sklearn.linear_model.Lasso - https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html#sklearn.linear_model. - https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html#sklearn.linear_model.Ridg - Documentación sobre ajuste lineal de statsmodel (OLS) : https://www.statsmodels.org/dev/generated/statsmodels.formula.api.ols.html#statsmodels.formula.api.ols