P1_Casas_Boston

October 29, 2020

1 Práctica 1: Casas en Boston

Esta es la primera práctica de la asignatura de **Análisis de Datos** del grado de *Ingeniería Informática* de la *UC3M* durante el curso 2020/21.

Autores:
Alba Reinders Sánchez, 100383444, gr.83
Alejandro Valverde Mahou, 100383383, gr.83

El objetivo de esta práctica es predecir mediante un *modelo de regresión* una estimación del precio medio de la vivienda en distintas zonas de Boston.

El conjunto de datos está compuesto por **506** ejemplos, cada uno con **13** atributos. Más la clase a predecir.

Para ver qué representa cada uno de estos atributos, visitar el artículo original: Harrison Jr, David, and Daniel L. Rubinfeld. "Hedonichousingpricesand thedemandforcleanair."Journalof environmentaleconomicsand management5, no. 1 (1978): pp. 81–102

[1]:
```
# Cabecera de los datos (atributos)
header = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad',␣
 ↪'tax', 'ptratio',
```

```
          'black', 'lstat', 'medv']
```

1.1 Tarea 1: Carga de datos

Se va a utilizar la librería *Pandas* para realizar esta tarea.

[2]:
```
import pandas as pd
```

Debido a que las columnas del *.csv* están separadas por espacios irregulares, se ha realizado la siguiente conversión para que el separador sea la ',' y así poder cargar mejor los datos.

[3]:
```
with open("housing.csv") as f:
    lines = f.readlines()

with open("housing_comas.csv", "w") as f2:
    for line in lines:
        line = line.split(' ')
        while '' in line:
            line.remove('')
        f2.write(','.join(map(str, line)))
```

[4]:
```
data = pd.read_csv("./housing_comas.csv", header=None, names=header)

assert data.shape == (506, 14), 'Los datos deben ser de la forma: 506 filas y 14␣
 ↪columnas'

display(data)
```

```
        crim    zn  indus  chas    nox     rm   age     dis  rad    tax  \
0    0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296.0
1    0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242.0
2    0.02729   0.0   7.07     0  0.469  7.185  61.1  4.9671    2  242.0
3    0.03237   0.0   2.18     0  0.458  6.998  45.8  6.0622    3  222.0
4    0.06905   0.0   2.18     0  0.458  7.147  54.2  6.0622    3  222.0
..       ...   ...    ...   ...    ...    ...   ...     ...  ...    ...
501  0.06263   0.0  11.93     0  0.573  6.593  69.1  2.4786    1  273.0
502  0.04527   0.0  11.93     0  0.573  6.120  76.7  2.2875    1  273.0
503  0.06076   0.0  11.93     0  0.573  6.976  91.0  2.1675    1  273.0
504  0.10959   0.0  11.93     0  0.573  6.794  89.3  2.3889    1  273.0
505  0.04741   0.0  11.93     0  0.573  6.030  80.8  2.5050    1  273.0

     ptratio   black  lstat  medv
0       15.3  396.90   4.98  24.0
1       17.8  396.90   9.14  21.6
2       17.8  392.83   4.03  34.7
3       18.7  394.63   2.94  33.4
4       18.7  396.90   5.33  36.2
```

```
..       ...     ...    ...   ...
501     21.0  391.99   9.67  22.4
502     21.0  396.90   9.08  20.6
503     21.0  396.90   5.64  23.9
504     21.0  393.45   6.48  22.0
505     21.0  396.90   7.88  11.9

[506 rows x 14 columns]
```

El número de filas y columnas es el esperado: 506 fila y 14 columnas.

1.2 Tarea 2: Análisis exploratorio

En primer lugar, se lleva a cabo un *resumen estadístico* para poder visualizar, de forma general, el aspecto de los datos.

[5]:
```
display(data.describe())
```

```
              crim          zn       indus        chas         nox          rm  \
count   506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean      3.613524   11.363636   11.136779    0.069170    0.554695    6.284634
std       8.601545   23.322453    6.860353    0.253994    0.115878    0.702617
min       0.006320    0.000000    0.460000    0.000000    0.385000    3.561000
25%       0.082045    0.000000    5.190000    0.000000    0.449000    5.885500
50%       0.256510    0.000000    9.690000    0.000000    0.538000    6.208500
75%       3.677082   12.500000   18.100000    0.000000    0.624000    6.623500
max      88.976200  100.000000   27.740000    1.000000    0.871000    8.780000

              age         dis         rad         tax     ptratio       black  \
count   506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean     68.574901    3.795043    9.549407  408.237154   18.455534  356.674032
std      28.148861    2.105710    8.707259  168.537116    2.164946   91.294864
min       2.900000    1.129600    1.000000  187.000000   12.600000    0.320000
25%      45.025000    2.100175    4.000000  279.000000   17.400000  375.377500
50%      77.500000    3.207450    5.000000  330.000000   19.050000  391.440000
75%      94.075000    5.188425   24.000000  666.000000   20.200000  396.225000
max     100.000000   12.126500   24.000000  711.000000   22.000000  396.900000

            lstat        medv
count  506.000000  506.000000
mean    12.653063   22.532806
std      7.141062    9.197104
min      1.730000    5.000000
25%      6.950000   17.025000
50%     11.360000   21.200000
75%     16.955000   25.000000
max     37.970000   50.000000
```

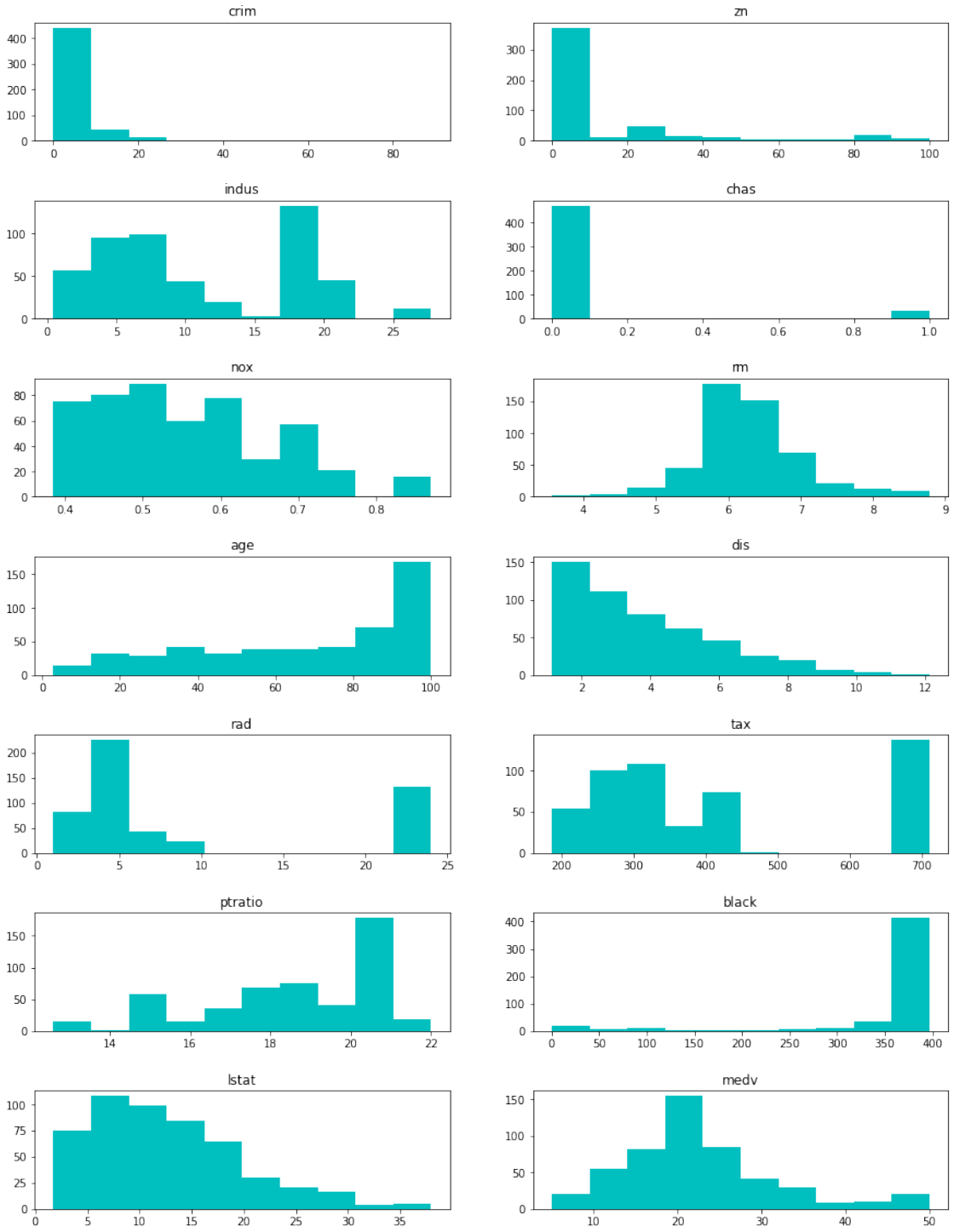
A pesar de que este resumen aporta información útil referente a los datos, no es suficiente para realizar una visualización real de los mismos, pero permite intuir su estructura y es una buena forma de empezar a estudiar los datos.

En este resumen no se ha encontrado ningún valor dispar que haga sospechar de posibles errores en los datos.

Una vez realizado este estudio previo, se realiza un *histograma* de cada uno de los atributos.

[6]:
```
import matplotlib.pyplot as plt
```

[7]:
```
plt.figure(figsize=(15,20))
plt.subplots_adjust(hspace=0.5)
for i in range(14):
    plt.subplot(7,2,i+1)
    plt.title(header[i])
    plt.hist(data[header[i]], color='c')
plt.show()
```



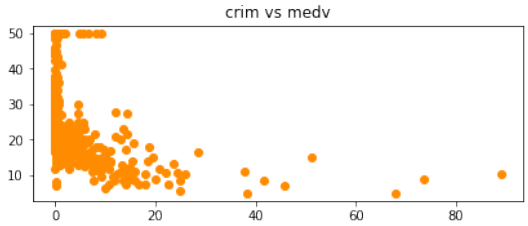
Esto permite ver la distribución de forma general de cada atributo por separado. Esta información sirve para conocer la estructura en la que se organizan los datos sin tener en cuenta su relación con el resto de atributos.

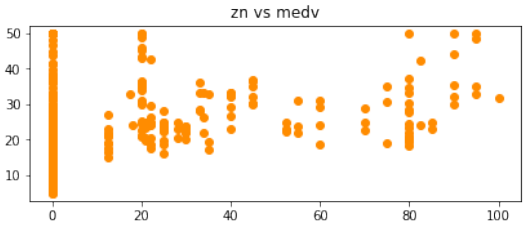
Por ejemplo, se puede ver que la variable *tax* se divide en dos grupos bien diferenciados y la

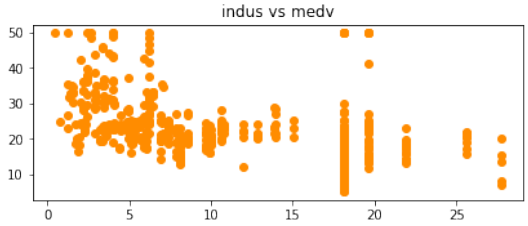
variable *age* tiene una cola hacia la derecha muy diferenciada.

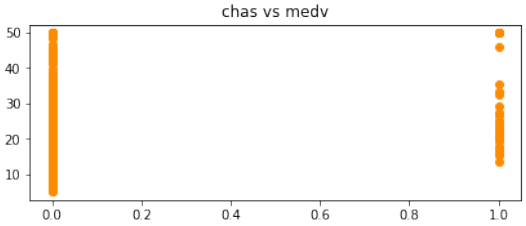
Pero estudiar estos atributos por separado no aporta demasiado conocimiento, por lo que es necesario analizar su **correlación** entre ellos. Dado que el atributo que se está intentando predecir es *medv*, se va a realizar este estudio de la **correlación** entre el resto de atributos y *medv*.

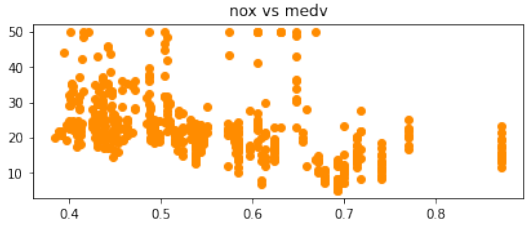
[8]:
```
plt.figure(figsize=(15,25))
plt.subplots_adjust(hspace=0.5)
for i in range(13):
    plt.subplot(7,2,i+1)
    plt.title(header[i] + " vs " + header[13])
    plt.scatter(data[header[i]], data[header[13]], color='darkorange', vmin=0)
plt.show()
```

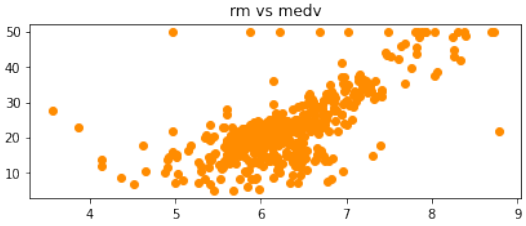

crim vs medv

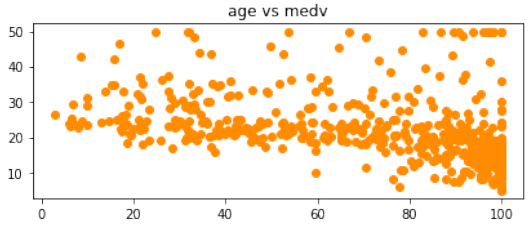

zn vs medv

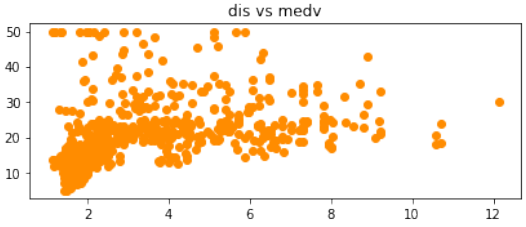

indus vs medv

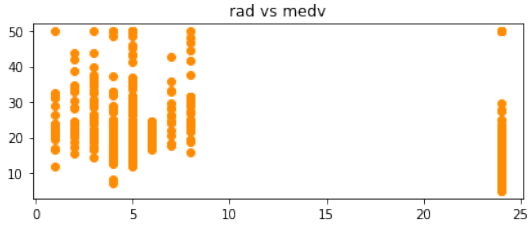

chas vs medv

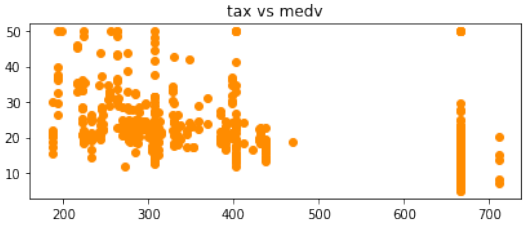

nox vs medv

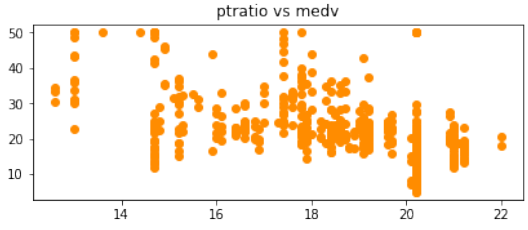

rm vs medv

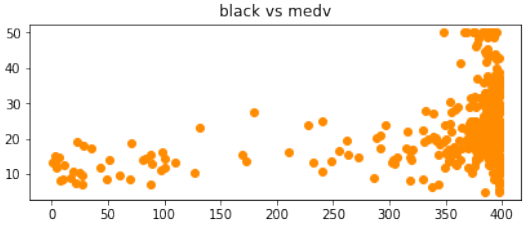

age vs medv

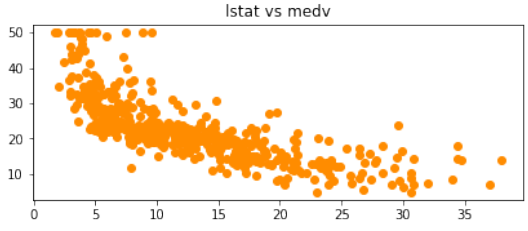

dis vs medv


rad vs medv


tax vs medv


ptratio vs medv

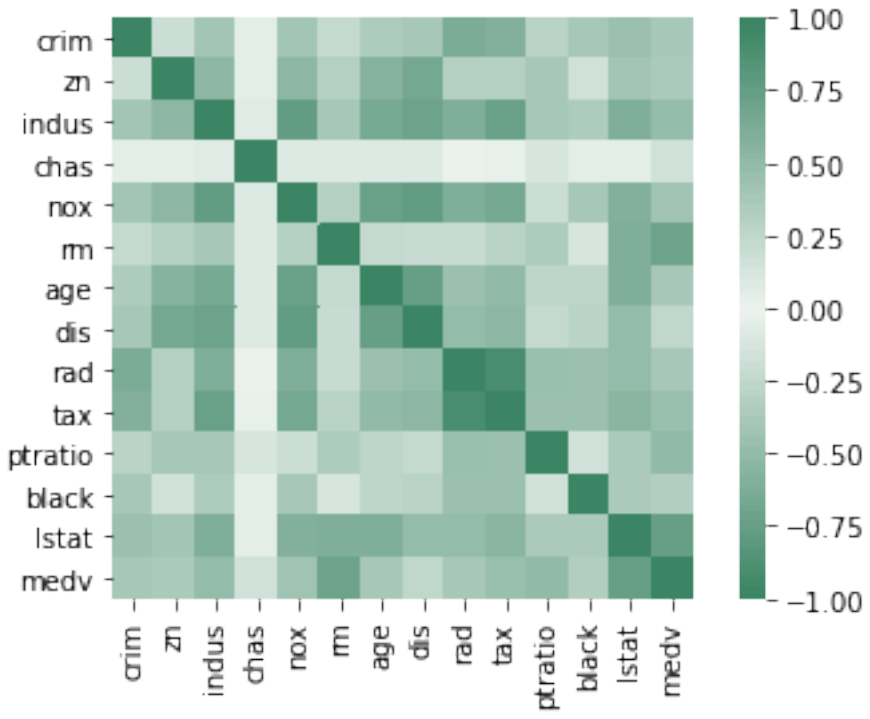

black vs medv


lstat vs medv

Las gráficas anteriores muestran una posible correlación sobre los datos. Para ver en mayor profundidad la relación entre las variables, se va a calcular además su **coeficiente de correlación de Pearson**, donde los valores que más se acerquen al *1* o al *-1* serán los mejores.

[9]:
```
from scipy import stats
import numpy as np
import seaborn as sns
```

[10]:
```
corr = data.corr()
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1, center=0,
    cmap=sns.diverging_palette(150, 150, n=100),
    square=True
)
```



De la gráfica de calor anterior, tan solo interesa la última fila o última columna, ya que es la que compara la variable a estudiar con en resto de variables.

[11]:
```
print("Coeficiente de correlación de Pearson de la variable a estudiar con el␣
 ↪resto:")
```

```
for name in header[:-1]:
    coef = data[name].corr(data[header[13]])
    print("\t- " +name + ": " + str(coef))
```

```
Coeficiente de correlación de Pearson de la variable a estudiar con el resto:
        - crim: -0.3883046085868114
        - zn: 0.3604453424505444
        - indus: -0.48372516002837346
        - chas: 0.17526017719029738
        - nox: -0.42732077237328137
        - rm: 0.6953599470715387
        - age: -0.3769545650045959
        - dis: 0.24992873408590385
        - rad: -0.38162623063977735
        - tax: -0.4685359335677663
        - ptratio: -0.507786685537561
        - black: 0.33346081965706603
        - lstat: -0.7376627261740145
```

Gracias a esta información se puede ver que las variables que tienen mayor relación con *medv* son, de mayor a menor correlación:

- ***lstat***
- *rm*
- *ptratio*

El resto tienen un valor por debajo de **0.5**, por lo que se ha considerado que no están fuertemente relacionadas.

1.3 Tarea 3: Construcción de un modelo de regresión lineal

Una vez se han analizado los datos, dado que la salida esperada es un número, es necesario construir un modelo de **regresión lineal**. Para ver su precisión, se evaluará con las siguientes técnicas:

- **Coeficiente** $R^2 \rightarrow \mathrm{R}^2 = \dfrac{\sigma_{XY}^2}{\sigma_X^2 \sigma_Y^2}$
- **RMSE** $\rightarrow \mathrm{RMSE} = \sqrt{\dfrac{\sum_{i=1}^{n}(y_i - x_i)^2}{n}}$
- **MAE** $\rightarrow \mathrm{MAE} = \dfrac{\sum_{i=1}^{n}|y_i - x_i|}{n}$

1.3.1 Creación del modelo de regresión

[12]:
```
from sklearn.linear_model import LinearRegression
```

[13]:
```
lr = LinearRegression()
```

1.3.2 Entrenamiento del modelo de regresión

Primero es necesario dividir los datos para separar las etiquetas de los datos a usar.

[14]:
```
X = np.array(data[header[:-1]])
y = np.array(data[header[-1]])
lr.fit(X, y)
```

[14]:
```
LinearRegression()
```

La formula de **regresión lineal** de este problema tiene la siguiente forma:

$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_{13} x_{13} + \varepsilon$

donde β_i toma los valores que se muestran en la siguiente celda:

[15]:
```
beta = np.array([lr.intercept_] + list(lr.coef_))
beta = pd.DataFrame(np.reshape(beta, (1, 14)))
beta.columns = ['b0'] + header[:-1]
display(beta)
```

```
          b0      crim       zn     indus      chas        nox        rm  \
0  36.459488 -0.108011  0.04642  0.020559  2.686734 -17.766611  3.809865

        age       dis       rad       tax   ptratio     black     lstat
0  0.000692 -1.475567  0.306049 -0.012335 -0.952747  0.009312 -0.524758
```

1.3.3 Evaluación del modelo de regresión

[16]:
```
y_pred = lr.predict(X)
```

[17]:
```
r2 = lr.score(X, y)
rmse = np.sqrt(np.mean((y - y_pred)**2))
mae = np.mean(np.abs(y - y_pred))


print("Coeficiente R² =", r2)
print("RMSE =", rmse)
print("MAE =", mae)
```

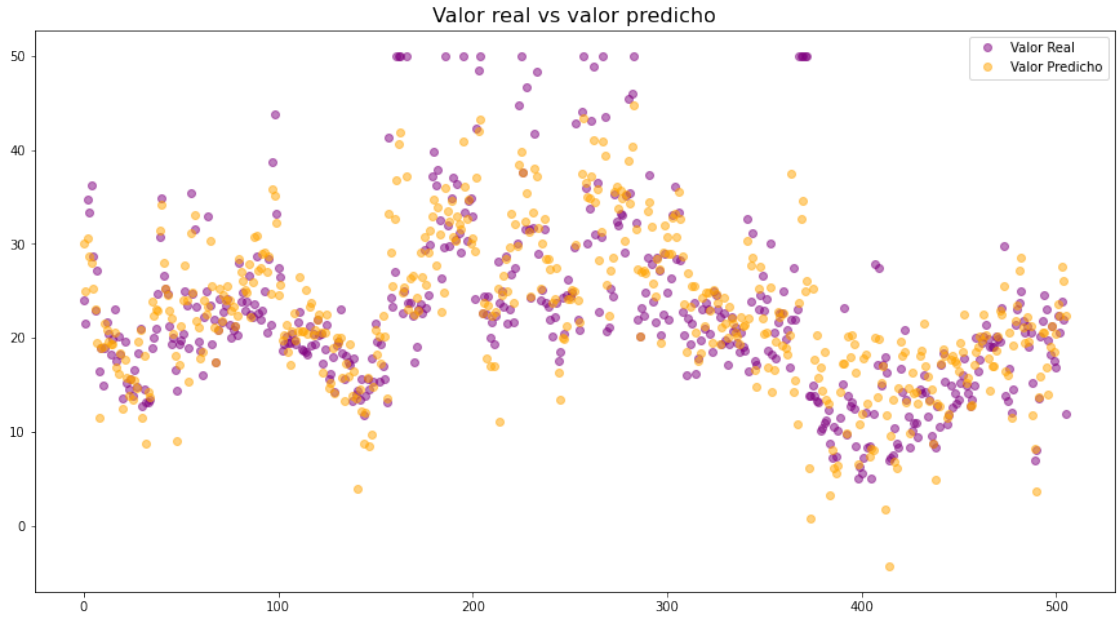
```
Coeficiente R² = 0.7406426641094095
RMSE = 4.679191295697281
MAE = 3.2708628109003177
```

La evaluación generada por el *MAE*(*Mean Absolute Error*) sirve para determinar que, de media, se equivoca en 3.27 unidades, o lo que es lo mismo, se equivoca de media por **$3270.86**, un error bastante grande para los valores que se están tratando.

Por otro lado, el coeficiente R^2 determina la calidad del modelo para replicar los resultados, y la

proporción de variación de los resultados que se puede explicar por el modelo. Cuanto más se ajusta a '1', mayor será el ajuste del modelo a la variable que se intenta explicar. En este caso toma un valor de **0.7406**, lo cual es un valor relativamente aceptable.

[18]:
```
plt.figure(figsize=(15,8))

plt.plot(y, 'o', color='purple', alpha=0.5, label='Valor Real')
plt.plot(y_pred, 'o', color='orange', alpha=0.5, label='Valor Predicho')

plt.legend()
plt.title("Valor real vs valor predicho", fontsize=16)

plt.show()
```



Los valores predichos se ajustan bastante bien a los datos reales, a excepción de los datos atípicos, que no es capaz de imitar.

1.4 Tarea 4: Mejora del modelo de regresión lineal

A continuación se intenta mejorar los resultados obtenidos con el modelo anterior. Para ello se van a usar dos técnicas de *regularización*.

1.4.1 Regularización

Las transformaciones de regularización tienen como finalidad reducir el nivel de complejidad de los modelos lineales. Para ello se modifican las funciones de coste del modelo básico.

La función de coste del modelo lineal es *Residual Sum of Squares*:

$$RSS = \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j xij)^2$$

Se llevan a cabo dos modificaciones distintas: **Ridge** y **Lasso**.

Ridge (L2) Usa una regresión muy similar a *RSS*, con el añadido de un parámetro de contracción (α) que minimiza el valor de los coeficientes reduciendo así su aportación al modelo en función de su importancia. Su función de coste es:

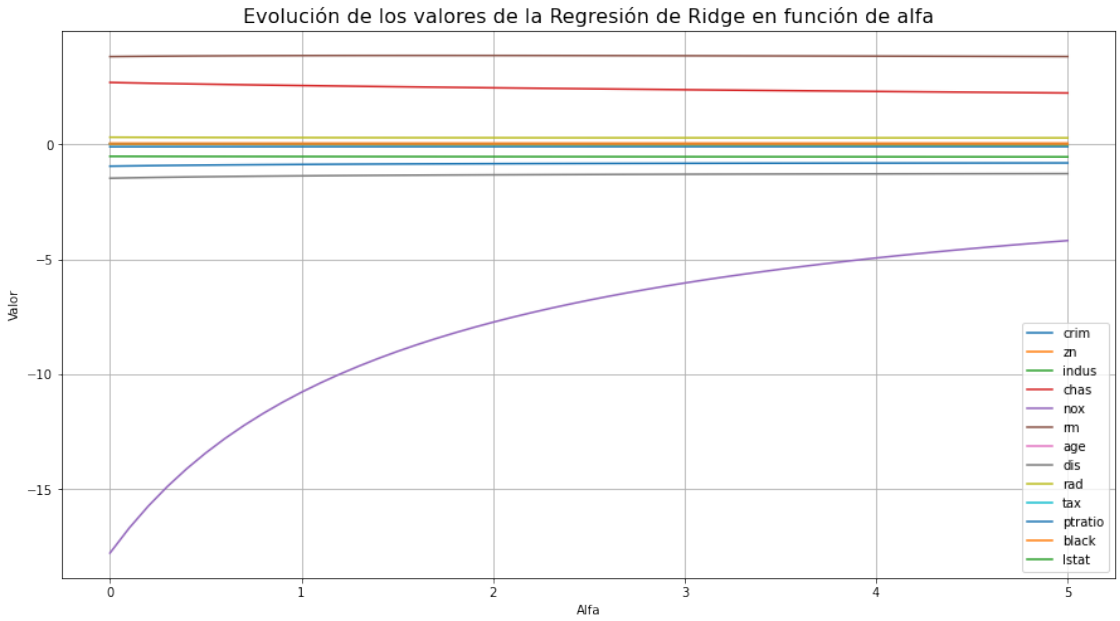
$$Ridge = RSS + \alpha \sum_{j=1}^{p}\beta_j^2$$

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

[20]:
```
ridge_scores = []
ridge_coefs = []
alpha_values = []

for i in range(51):
    rr = Ridge(alpha=i*0.1)
    rr.fit(X, y)
    y_pred = rr.predict(X)
    mae = np.mean(np.abs(y - y_pred))
    ridge_scores.append(mae)
    ridge_coefs.append(rr.coef_)
    alpha_values.append(i*0.1)
```

[21]:
```
ridge_coefs = np.array(ridge_coefs)
```

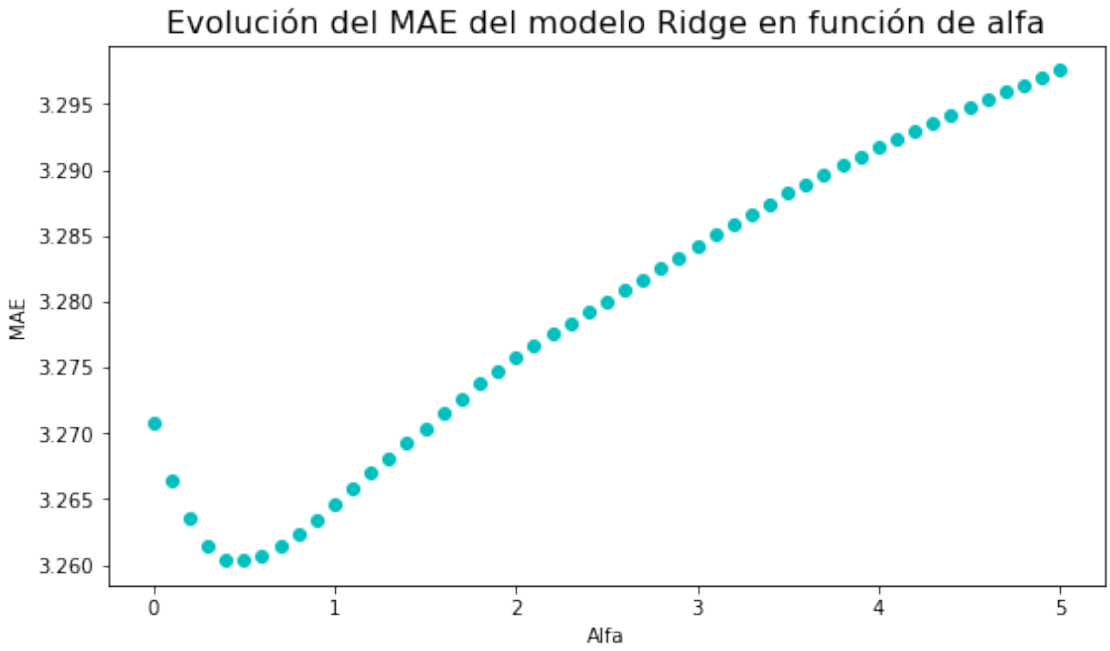
[22]:
```
plt.figure(figsize=(15,8))

for i, name in enumerate(header[:-1]):
    plt.plot(alpha_values, ridge_coefs[:,i], label=name)

plt.legend()
plt.title("Evolución de los valores de la Regresión de Ridge en función de␣
 ↪alfa", fontsize=16)
plt.xlabel("Alfa")
plt.ylabel("Valor")
plt.grid(True)
plt.show()
```



La gráfica anterior muestra la importancia de cada uno de los atributos. Por ejemplo, se puede ver que el atributo *nox* tenía mucha importancia en el modelo original pero la ha ido perdiendo según aumentaba alfa, lo que indica que no es un atributo relevante. Por el contrario, el atributo *rm* se ha mantenido cercano a su valor original durante la evolución de alfa, demostrando su importancia en el modelo.

Una vez visto esto, es necesario determinar el valor óptimo de alfa para crear el modelo que genera mejores resultados.

[23]:
```
plt.figure(figsize=(9,5))

plt.plot(alpha_values, ridge_scores, 'co')

plt.title("Evolución del MAE del modelo Ridge en función de alfa", fontsize=16)
plt.xlabel("Alfa")
plt.ylabel("MAE")

plt.show()
```



[24]:
```
best_alpha = ridge_scores.index(min(ridge_scores))*0.1

print("El mejor modelo usa un alfa de", best_alpha, "y tiene un MAE de",␣
 ↪min(ridge_scores))
```

```
El mejor modelo usa un alfa de 0.4 y tiene un MAE de 3.260328834895458
```

[25]:
```
# Modelo con el mejor alfa
rr = Ridge(alpha=best_alpha)
rr.fit(X, y)
y_pred = rr.predict(X)
```

[26]:
```
r2 = rr.score(X, y)
rmse = np.sqrt(np.mean((y - y_pred)**2))
mae = np.mean(np.abs(y - y_pred))


print("Coeficiente R² =", r2)
print("RMSE =", rmse)
print("MAE =", mae)
```

```
Coeficiente R² = 0.7401576284016107
RMSE = 4.6835646340049015
MAE = 3.260328834895458
```

A pesar de que el *MAE* sea algo mejor que el del modelo original, el resto de medidas muestran que empeora. Por lo tanto se puede asumir que no se ha generado un mejor modelo.

Esto puede deberse a que la regularización *Ridge* es capaz de quitarle importancia a los atributos menos útiles pero no es capaz de eliminarlos (nunca llegan a valer 0). Otro posible motivo es que se está evaluando con los mismos datos con los que se está entrenando, por lo que no se puede ver si estos peores resultados se deben a un aumento de la capacidad de generalización del modelo.

Lasso (L1) Es un tipo de regularización que intenta mejorar la capacidad de *Ridge*, gracias a que es capaz de eliminar por completo aquellos atributos que no son relevantes. Su función de coste es:

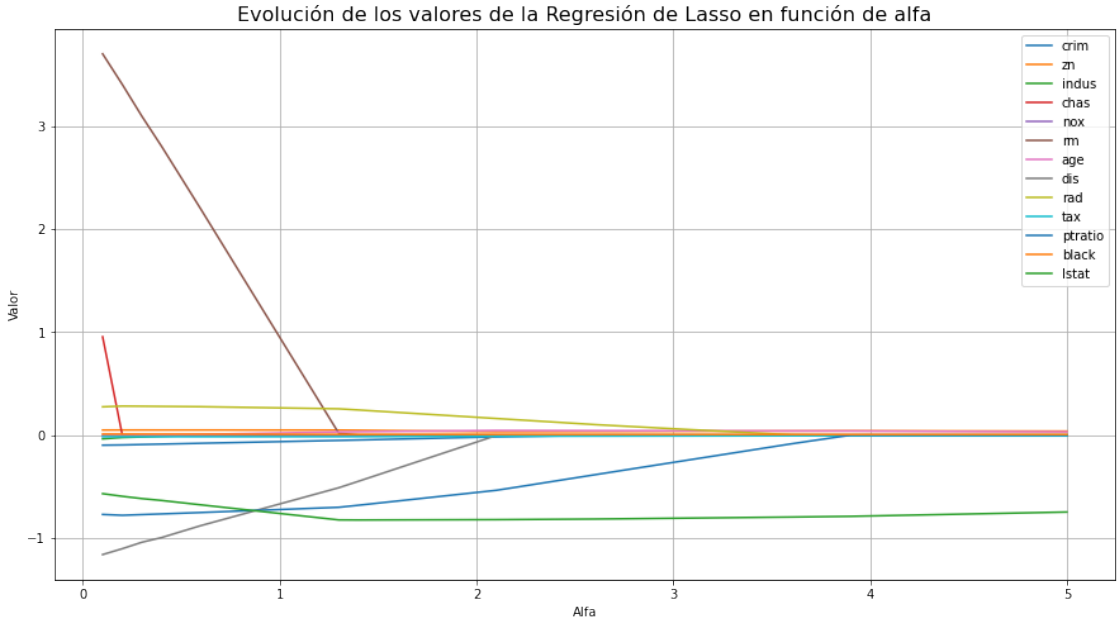
$$Lasso = RSS + \alpha \sum_{j=1}^{p} |\beta_j|$$

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

[28]:
```
lasso_scores = []
lasso_coefs = []
alpha_values = []

# Lasso con el valor de alfa igual a 0 no converge bien y saca un warning
for i in range(1, 51):
    ls = Lasso(alpha=i*0.1)
    ls.fit(X, y)
    y_pred = ls.predict(X)
    mae = np.mean(np.abs(y - y_pred))
    lasso_scores.append(mae)
    lasso_coefs.append(ls.coef_)
    alpha_values.append(i*0.1)
```

[29]:
```
lasso_coefs = np.array(lasso_coefs)
```

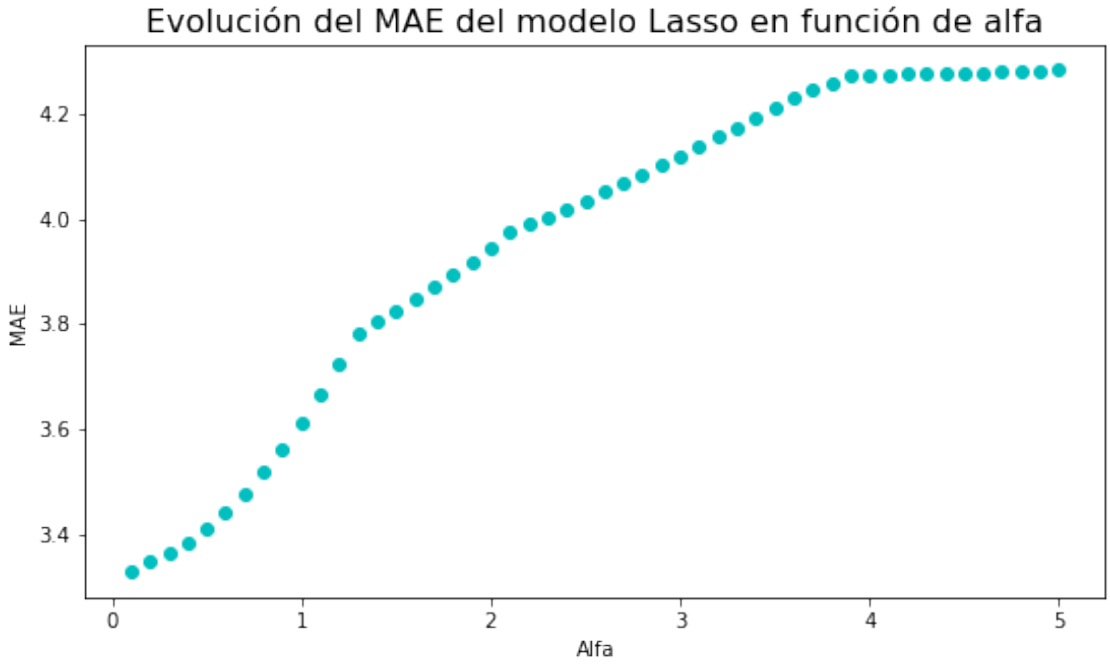
[30]:
```
plt.figure(figsize=(15,8))

for i, name in enumerate(header[:-1]):
    plt.plot(alpha_values, lasso_coefs[:,i], label=name)

plt.legend()
plt.title("Evolución de los valores de la Regresión de Lasso en función de␣
 ↪alfa", fontsize=16)
plt.xlabel("Alfa")
plt.ylabel("Valor")
plt.grid(True)
plt.show()
```



En esta gráfica se observa que el atributo *lstat* es el más relevante para el modelo ya que no solo se mantiene su valor sino que aumenta ligeramente. Sin embargo, por ejemplo el atributo *chas* no parece tener mucha relevancia.

Una vez visto esto, es necesario determinar el valor óptimo de alfa para crear el modelo que genera mejores resultados.

[31]:
```
plt.figure(figsize=(9,5))

plt.plot(alpha_values, lasso_scores, 'co')

plt.title("Evolución del MAE del modelo Lasso en función de alfa", fontsize=16)
plt.xlabel("Alfa")
plt.ylabel("MAE")

plt.show()
```



[32]:
```
best_alpha = (lasso_scores.index(min(lasso_scores))+1)*0.1

print("El mejor modelo usa un alfa de", best_alpha, "y tiene un MAE de",␣
 ↪min(lasso_scores))
```

```
El mejor modelo usa un alfa de 0.1 y tiene un MAE de 3.3282231192242113
```

[33]:
```
# Modelo con el mejor alfa
ls = Lasso(alpha=best_alpha)
ls.fit(X, y)
y_pred = ls.predict(X)
```

[34]:
```
r2 = ls.score(X, y)
rmse = np.sqrt(np.mean((y - y_pred)**2))
mae = np.mean(np.abs(y - y_pred))


print("Coeficiente R² =", r2)
print("RMSE =", rmse)
print("MAE =", mae)
```

```
Coeficiente R² = 0.7269834862602695
RMSE = 4.800826274008644
MAE = 3.3282231192242113
```

A pesar de que este modelo funciona peor dados sus errores, puede deberse a que es capaz de

generalizar a mayor nivel, pero todavía no puede verse hasta que no se realice la división de los datos en entrenamiento y test.

1.5 Tarea 5: Generalización del modelo de regresión lineal

Como ya se ha mencionado en el apartado anterior, la capacidad de generalización de un modelo es un aspecto muy relevante a tener en cuenta a la hora de crear un modelo. Para ello es necesario dividir el conjunto de datos en **entrenamiento** y **test**. La proporción propuesta es un **70%** datos de entrenamiento y **30%** para test.

1.5.1 División de los datos

Los dos conjuntos de datos, para realizar una evaluación efectiva, deben contener una muestra lo suficientemente representativa. La mejor forma de conseguir esto, en este caso concreto, es mezclar todos los datos de forma aleatoria, y luego separarlos en las proporciones deseadas.

[35]:
```
data = data.sample(frac=1).reset_index(drop=True)
```

[36]:
```
allData = np.array(data)
```

Hay que definir los tamaños de cada conjunto.

[37]:
```
train_size = int(len(allData) * 0.7)
test_size = len(allData) - train_size
print("El tamaño del conjunto de entrenamiento es", train_size, "y el del␣
 ↪conjunto de test es", test_size)
```

```
El tamaño del conjunto de entrenamiento es 354 y el del conjunto de test es 152
```

[38]:
```
X_train = allData[:train_size, :-1]
y_train = allData[:train_size, -1]

X_test = allData[train_size:, :-1]
y_test = allData[train_size:, -1]
```

1.5.2 Volver a crear el modelo de regresión lineal básico

[39]:
```
lr = LinearRegression()
```

[40]:
```
lr.fit(X_train, y_train)
y_pred_train = lr.predict(X_train)
y_pred_test = lr.predict(X_test)
```

[41]:
```
r2_train = lr.score(X_train, y_train)
rmse_train = np.sqrt(np.mean((y_train - y_pred_train)**2))
mae_train = np.mean(np.abs(y_train - y_pred_train))

r2_test = lr.score(X_test, y_test)
```

```
rmse_test = np.sqrt(np.mean((y_test - y_pred_test)**2))
mae_test = np.mean(np.abs(y_test - y_pred_test))


print("Error de entrenamiento")
print("Coeficiente R² =", r2_train)
print("RMSE =", rmse_train)
print("MAE =", mae_train)

print()

print("Error de test")
print("Coeficiente R² =", r2_test)
print("RMSE =", rmse_test)
print("MAE =", mae_test)
```

```
Error de entrenamiento
Coeficiente R² = 0.71366652310908
RMSE = 4.848161954755153
MAE = 3.3762016141322015

Error de test
Coeficiente R² = 0.7898022447470276
RMSE = 4.345131474920369
MAE = 3.179106202713651
```

Tal y como se ve, la evaluación del conjunto de entrenamiento y el conjunto de test parece bastante similar, y no parece haber indicio de sobre-entrenamiento, lo cual es una buena señal.

Este nuevo modelo muestra resultados ligeramente peores respecto a los obtenidos en el modelo original, pero no se podría considerar una gran diferencia. En cambio, es capaz de predecir correctamente con datos no vistos por el modelo previamente, demostrando que es capaz de generalizar.

A continuación, se volverá a repetir la regularización del modelo (Tarea 4), pero en este caso haciendo uso del conjunto de test para evaluar el valor de alfa.

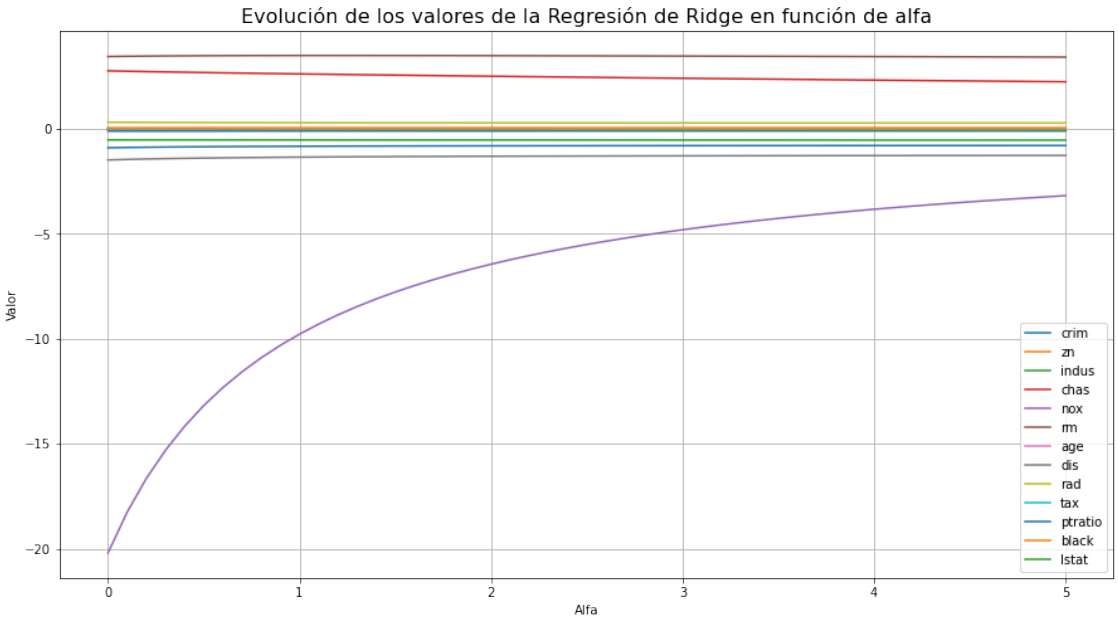
1.5.3 Volver a crear el modelo de regresión lineal Ridge

[42]:
```
ridge_scores = []
ridge_coefs = []
alpha_values = []

for i in range(51):
    rr = Ridge(alpha=i*0.1)
    rr.fit(X_train, y_train)
    y_pred = rr.predict(X_test)
    mae = np.mean(np.abs(y_test - y_pred))
    ridge_scores.append(mae)
```

```
    ridge_coefs.append(rr.coef_)
    alpha_values.append(i*0.1)
```

[43]:
```
ridge_coefs = np.array(ridge_coefs)
```

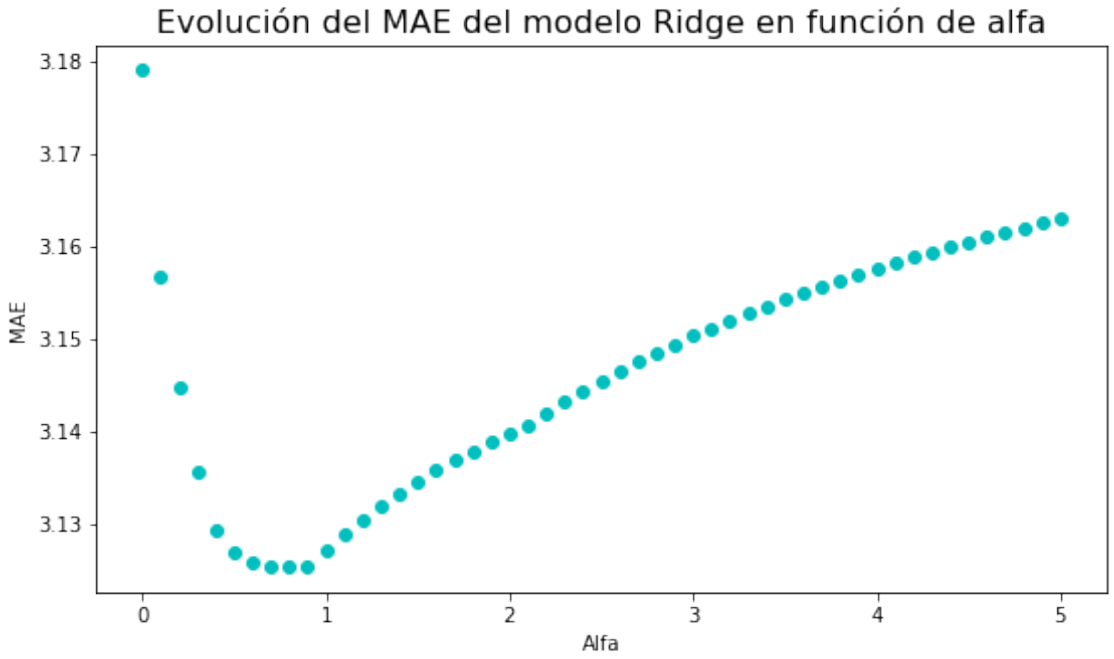
[44]:
```
plt.figure(figsize=(15,8))

for i, name in enumerate(header[:-1]):
    plt.plot(alpha_values, ridge_coefs[:,i], label=name)

plt.legend()
plt.title("Evolución de los valores de la Regresión de Ridge en función de␣
 ↪alfa", fontsize=16)
plt.xlabel("Alfa")
plt.ylabel("Valor")
plt.grid(True)
plt.show()
```



Los resultados que se muestran por la gráfica parecen ser bastante similares a los resultados originales.

[45]:
```
plt.figure(figsize=(9,5))

plt.plot(alpha_values, ridge_scores, 'co')

plt.title("Evolución del MAE del modelo Ridge en función de alfa", fontsize=16)
plt.xlabel("Alfa")
```

```
plt.ylabel("MAE")

plt.show()
```



[46]:
```
best_alpha = ridge_scores.index(min(ridge_scores))*0.1

print("El mejor modelo usa un alfa de", best_alpha, "y tiene un MAE de",␣
 ↪min(ridge_scores))
```

```
El mejor modelo usa un alfa de 0.7000000000000001 y tiene un MAE de
3.125285870464135
```

[47]:
```
# Modelo con el mejor alfa
rr = Ridge(alpha=best_alpha)
rr.fit(X_train, y_train)
y_pred_train = rr.predict(X_train)
y_pred_test = rr.predict(X_test)
```

[48]:
```
r2_train = rr.score(X_train, y_train)
rmse_train = np.sqrt(np.mean((y_train - y_pred_train)**2))
mae_train = np.mean(np.abs(y_train - y_pred_train))

r2_test = rr.score(X_test, y_test)
rmse_test = np.sqrt(np.mean((y_test - y_pred_test)**2))
mae_test = np.mean(np.abs(y_test - y_pred_test))
```

```
print("Error de entrenamiento")
print("Coeficiente R² =", r2_train)
print("RMSE =", rmse_train)
print("MAE =", mae_train)

print()

print("Error de test")
print("Coeficiente R² =", r2_test)
print("RMSE =", rmse_test)
print("MAE =", mae_test)
```

```
Error de entrenamiento
Coeficiente R² = 0.7112526220844647
RMSE = 4.868554996989833
MAE = 3.3594011314943044

Error de test
Coeficiente R² = 0.7923991786728523
RMSE = 4.318206617852234
MAE = 3.125285870464135
```

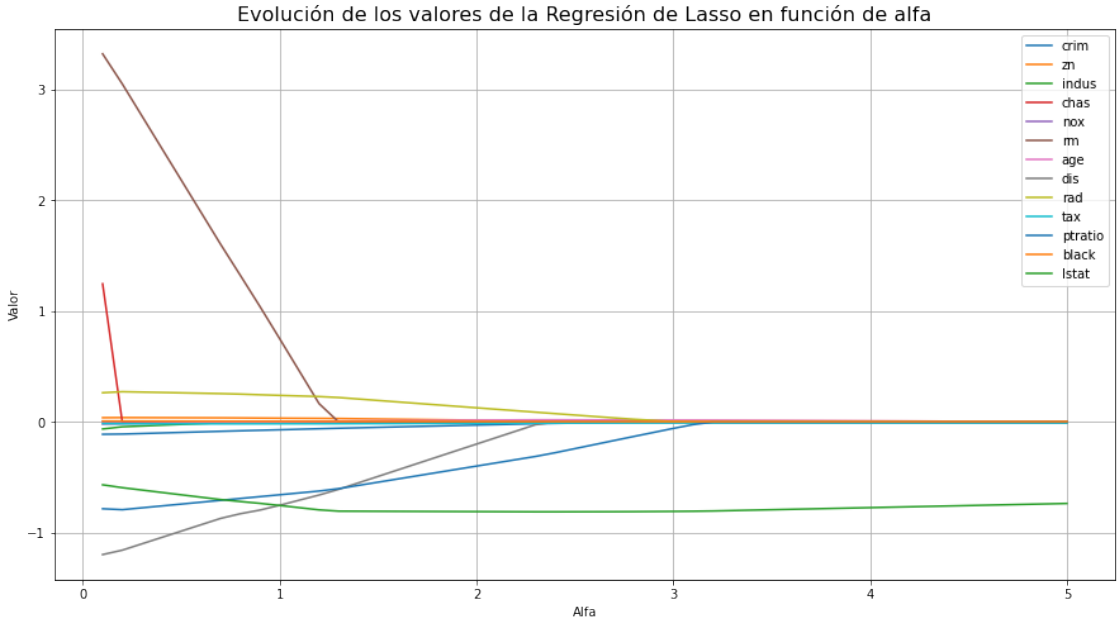
Los resultados son muy semejantes a los anteriores. No da la impresión de que este modelo tenga mayor capacidad de generalización, a pesar de que se pensaba lo contrario. Esto puede deberse a que el modelo ya sea suficientemente sencillo, y cualquier intento de simplificación disminuye su capacidad de predicción.

1.5.4 Volver a crear el modelo de regresión lineal Lasso

[49]:
```
lasso_scores = []
lasso_coefs = []
alpha_values = []

for i in range(1, 51):
    ls = Lasso(alpha=i*0.1)
    ls.fit(X_train, y_train)
    y_pred = ls.predict(X_test)
    mae = np.mean(np.abs(y_test - y_pred))
    lasso_scores.append(mae)
    lasso_coefs.append(ls.coef_)
    alpha_values.append(i*0.1)
```

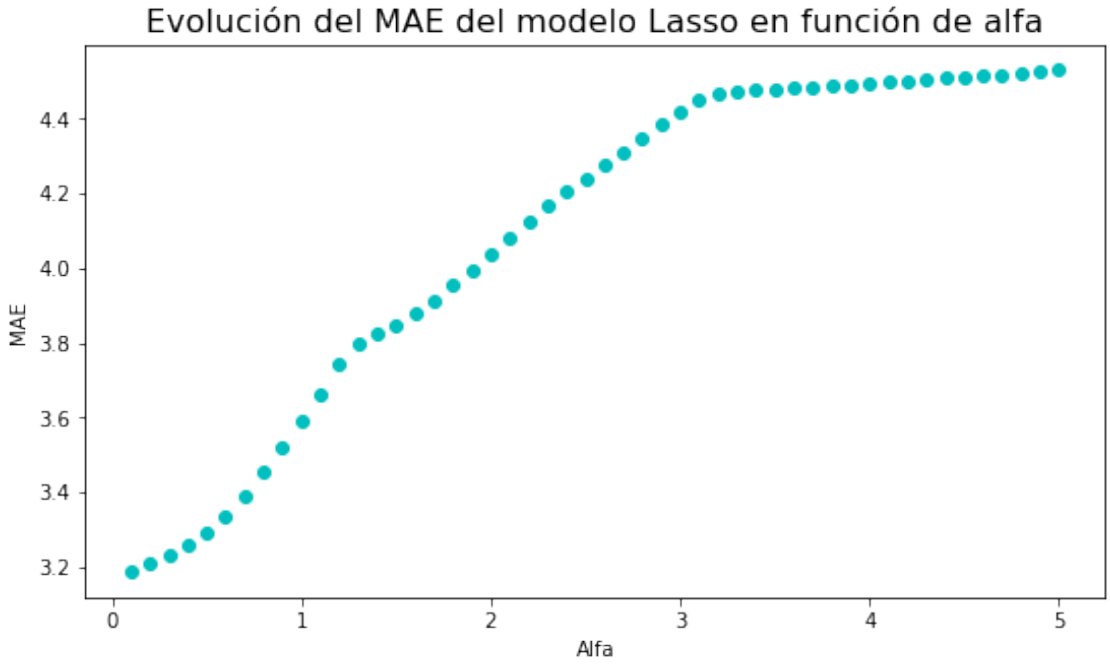
[50]:
```
lasso_coefs = np.array(lasso_coefs)
```

[51]:
```
plt.figure(figsize=(15,8))

for i, name in enumerate(header[:-1]):
    plt.plot(alpha_values, lasso_coefs[:,i], label=name)

plt.legend()
plt.title("Evolución de los valores de la Regresión de Lasso en función de␣
 ↪alfa", fontsize=16)
plt.xlabel("Alfa")
plt.ylabel("Valor")
plt.grid(True)
plt.show()
```



Al igual que ha sucedido con *Ridge,* parece que la gráfica original y esta son semejantes.

[52]:
```
plt.figure(figsize=(9,5))

plt.plot(alpha_values, lasso_scores, 'co')

plt.title("Evolución del MAE del modelo Lasso en función de alfa", fontsize=16)
plt.xlabel("Alfa")
plt.ylabel("MAE")

plt.show()
```



[53]:
```
best_alpha = (lasso_scores.index(min(lasso_scores))+1)*0.1

print("El mejor modelo usa un alfa de", best_alpha, "y tiene un MAE de",␣
 ↪min(lasso_scores))
```

```
El mejor modelo usa un alfa de 0.1 y tiene un MAE de 3.185733193017345
```

[54]:
```
# Modelo con el mejor alfa
ls = Lasso(alpha=best_alpha)
ls.fit(X_train, y_train)
y_pred_train = ls.predict(X_train)
y_pred_test = ls.predict(X_test)
```

[55]:
```
r2_train = ls.score(X_train, y_train)
rmse_train = np.sqrt(np.mean((y_train - y_pred_train)**2))
mae_train = np.mean(np.abs(y_train - y_pred_train))

r2_test = ls.score(X_test, y_test)
rmse_test = np.sqrt(np.mean((y_test - y_pred_test)**2))
mae_test = np.mean(np.abs(y_test - y_pred_test))


print("Error de entrenamiento")
print("Coeficiente R² =", r2_train)
print("RMSE =", rmse_train)
```

```
print("MAE =", mae_train)

print()

print("Error de test")
print("Coeficiente R² =", r2_test)
print("RMSE =", rmse_test)
print("MAE =", mae_test)
```

```
Error de entrenamiento
Coeficiente R² = 0.6983868443724749
RMSE = 4.975837582323464
MAE = 3.423050173589846

Error de test
Coeficiente R² = 0.7815665483693118
RMSE = 4.429436287069334
MAE = 3.185733193017345
```

Con este nuevo modelo se puede confirmar que la regularización en este problema no solo no proporciona mayor capacidad de generalización, sino que además resulta en peores resultados, aunque la diferencia no sea mucha.

Este tipo de técnicas son especialmente útiles cuando es necesario reducir la complejidad de los atributos, por ejemplo, cuando hay gran cantidad de ellos.

1.6 Conclusión

El modelo de regresión lineal que mejor se adecua a los datos en este caso es el modelo que no usa ningún tipo de regularización. A pesar de ello, ya se ha visto que los resultados todavía estan lejos de ser ideales, y tienen mucho error.

Otro posible acercamiento a este problema podría ser a través de algoritmos de regresión no lineales.

El modelo elegido es el que ha sido entrenado con el conjunto de entrenamiento, y evaluado con el de test, ya que es capaz de generalizar.

Los pesos obtenidos por este modelo son:

[56]:
```
beta = np.array([lr.intercept_] + list(lr.coef_))
beta = pd.DataFrame(np.reshape(beta, (1, 14)))
beta.columns = ['b0'] + header[:-1]
display(beta)
```

```
         b0      crim        zn     indus      chas       nox       rm  \
0  41.060804 -0.120454  0.035261  0.004045  2.748725 -20.20319  3.43025

        age      dis       rad       tax   ptratio     black     lstat
0 -0.006915 -1.49219  0.303696 -0.011438 -0.911382  0.006933 -0.539501
```

Gracias al estudio previo, se ha podido determinar cuales eran las variables con mayor correlación, y luego, mediante la regularización, se ha podido comprobar si estas variables que tenían más correlación eran realmente útiles en el problema o no.

A pesar de que estas técnicas no han supuesto una mejora en el modelo, es un estudio interesante que aporta información, y, en otros problemas, si puede llegar a suponer una mejora en el modelo.

1.7 Tarea Extra: Investigación de otros modelos

Dado que para este problema todavía no se ha encontrado un modelo capaz de sacar resultados con una precisión aceptable, se va a intentar realizar un acercamiento a través de un algoritmo de regresión no lineal. Concretamente, se va a usar el algoritmo de los **K-vecinos más cercanos**.

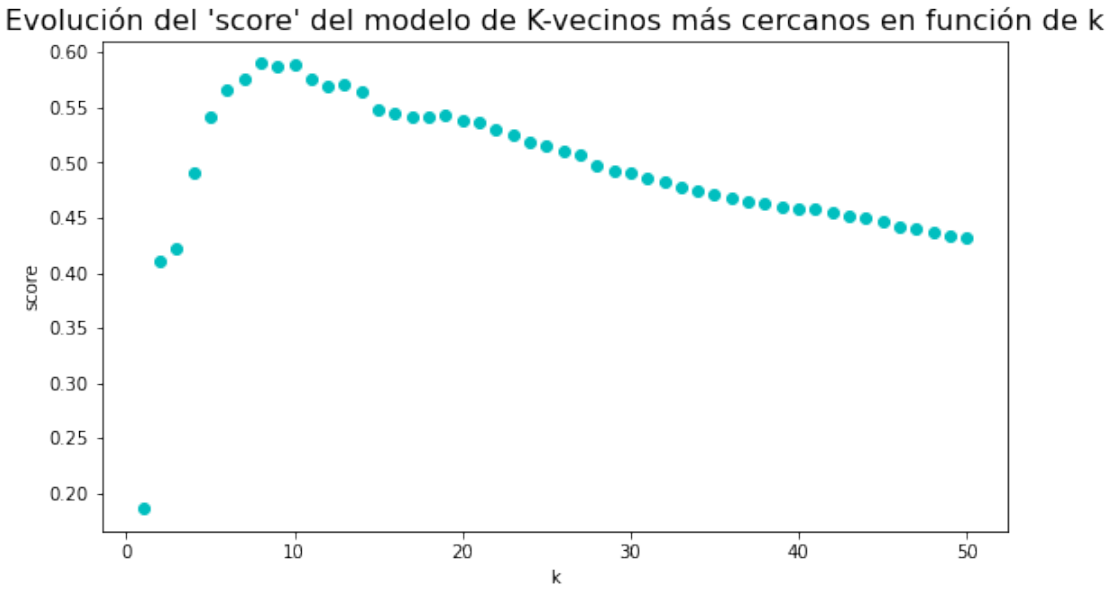
[57]:
```
from sklearn.neighbors import KNeighborsRegressor
```

Este algoritmo busca en las observaciones más cercanas a las que se está tratando de predecir y predice el punto de interés basado en la media de los datos que lo rodean.

El algoritmo básico hace la media de todos los datos que lo rodean, pero, en este caso, se va a usar la modificación del algoritmo que permite usar medias ponderadas usando las distancias a cada uno de ellos, de forma que los valores atípicos no generen demasiado error en la predicción.

[58]:
```
knn_scores = []
k_values = []

for i in range(1, 51):
    knn = KNeighborsRegressor(n_neighbors=i, weights='distance')
    knn.fit(X_train, y_train)
    knn_scores.append(knn.score(X_test, y_test))
    k_values.append(i)
```

[59]:
```
plt.figure(figsize=(9,5))

plt.plot(k_values, knn_scores, 'co')

plt.title("Evolución del 'score' del modelo de K-vecinos más cercanos en función␣
 ↪de k", fontsize=16)
plt.xlabel("k")
plt.ylabel("score")

plt.show()
```



[60]:
```
best_k = (knn_scores.index(max(knn_scores))+1)

print("El mejor modelo usa una k de", best_k, "y tiene un 'score' de",␣
 ↪max(knn_scores))
```

```
El mejor modelo usa una k de 8 y tiene un 'score' de 0.5899383026462905
```

[61]:
```
knn = KNeighborsRegressor(n_neighbors=best_k, weights='distance')
```

[62]:
```
knn.fit(X_train, y_train)
y_pred_train = knn.predict(X_train)
y_pred_test = knn.predict(X_test)
```

[63]:
```
r2_train = knn.score(X_train, y_train)
rmse_train = np.sqrt(np.mean((y_train - y_pred_train)**2))
mae_train = np.mean(np.abs(y_train - y_pred_train))

r2_test = knn.score(X_test, y_test)
rmse_test = np.sqrt(np.mean((y_test - y_pred_test)**2))
mae_test = np.mean(np.abs(y_test - y_pred_test))


print("Error de entrenamiento")
print("Coeficiente R² =", r2_train)
print("RMSE =", rmse_train)
print("MAE =", mae_train)
```

```
print()

print("Error de test")
print("Coeficiente R² =", r2_test)
print("RMSE =", rmse_test)
print("MAE =", mae_test)
```

```
Error de entrenamiento
Coeficiente R² = 1.0
RMSE = 0.0
MAE = 0.0

Error de test
Coeficiente R² = 0.5899383026462905
RMSE = 6.068948993469138
MAE = 4.410039021829905
```

Este algoritmo no supone una mejora respecto al modelo de regresión lineal, a pesar de usar técnicas no lineales. Esto se puede deber a que los valores estén, o muy separados, o muy juntos en el espacio de búsqueda, de forma que las medias supongan un error muy grande.

Se puede ver que este algoritmo no es adecuado para resolver este problema con este conjunto de datos.