Regularización

La regularización es el proceso de "penalizar" un modelo o utilizar un modelo más sencillo al que inicialmente habíamos apuntado con la intención de conseguir que generalice mejor (es decir que la diferencia entre error de train y test sea reduzca aunque ambos errores sean grandes). La regularización es un proceso que se aplica a cualquier modelo, pero en esta sesión vamos a verlo en modelos de regresión lineal.

En concreto para ver el impacto de la regularización en modelos de regresión lineal vamos a crear un modelo de regresión lineal sin regularización y luego varios modelos en los que aplicaremos penalizaciones L2 y L1 (o regularizaciones L2 y L1) también basados en la regresión lineal (los que hemos llamado Ridge y Lasso)

Problema y Datos

Consideramos un estudio médico realizado en 97 hombres con cáncer de próstata. El enfoque se centra en la relación entre el antígeno prostático específico (psa), que se eleva en hombres con cáncer de próstata, y otras medidas clínicas. Las otras medidas clínicas son las variables predictoras, recogidas en un examen médico, y la cantidad de expresión del antígeno asociado con la detección del cáncer es la variable de respuesta (lpsa).

Así, el marco de datos se compone de 97 observaciones sobre 9 variables:

- lcavol: logaritmo del volumen del cáncer
- lweight: logaritmo del peso de la próstata
- edad: edad del paciente en años
- lbph: logaritmo de la cantidad de hiperplasia prostática benigna
- svi: invasión de vesícula seminal
- lcp: logaritmo de la penetración capsular
- gleason: puntuación de Gleason
- pgg45: porcentaje de la puntuación de Gleason 4 o 5
- lpsa: logaritmo del antígeno prostático específico

El objetivo es encontrar modelos que predigan la respuesta lpsa.

Imports

In [13]:
```
import numpy as np
import pandas as pd
from sklearn import linear_model, metrics, model_selection
from sklearn.linear_model import ElasticNet
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge
import seaborn as sns
```

Carga de datos y primera visulización

In [14]:
```
lpsa_data = pd.read_csv('data/prostate_dataset.txt', delimiter='\t')
lpsa_data = lpsa_data.loc[:, 'lcavol':]
lpsa_data.head()
```

Out[14]:

	lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45	lpsa
0	-0.579818	2.769459	50	-1.386294	0	-1.386294	6	0	-0.430783
1	-0.994252	3.319626	58	-1.386294	0	-1.386294	6	0	-0.162519
2	-0.510826	2.691243	74	-1.386294	0	-1.386294	7	20	-0.162519
3	-1.203973	3.282789	58	-1.386294	0	-1.386294	6	0	-0.162519
4	0.751416	3.432373	62	-1.386294	0	-1.386294	6	0	0.371564

In [15]:
```
lpsa_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 97 entries, 0 to 96
Data columns (total 9 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   lcavol   97 non-null     float64
 1   lweight  97 non-null     float64
 2   age      97 non-null     int64  
 3   lbph     97 non-null     float64
 4   svi      97 non-null     int64  
 5   lcp      97 non-null     float64
 6   gleason  97 non-null     int64  
 7   pgg45    97 non-null     int64  
 8   lpsa     97 non-null     float64
dtypes: float64(5), int64(4)
memory usage: 6.9 KB
```

Data train/test splitting

Podemos hacer un split como queramos, si suponemos que ya hay un orden aleatorio en los datos, podemos hacer una selección de las n primeras filas para train y las siguientes para test. Aunque se aconseja usar sklearn en general:

In [16]:
```
# train : the first rows
# test : the last rows
n_split = 60
```

```
train_set = lpsa_data.iloc[:n_split] # without pgg45
test_set = lpsa_data.iloc[n_split:]
target = "lpsa"
X_train = train_set.drop(target, axis = 1)
y_train = train_set[target]
X_test = test_set.drop(target, axis = 1)
y_test = test_set[target]
```

In [17]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(60, 8)
(37, 8)
(60,)
(37,)
```

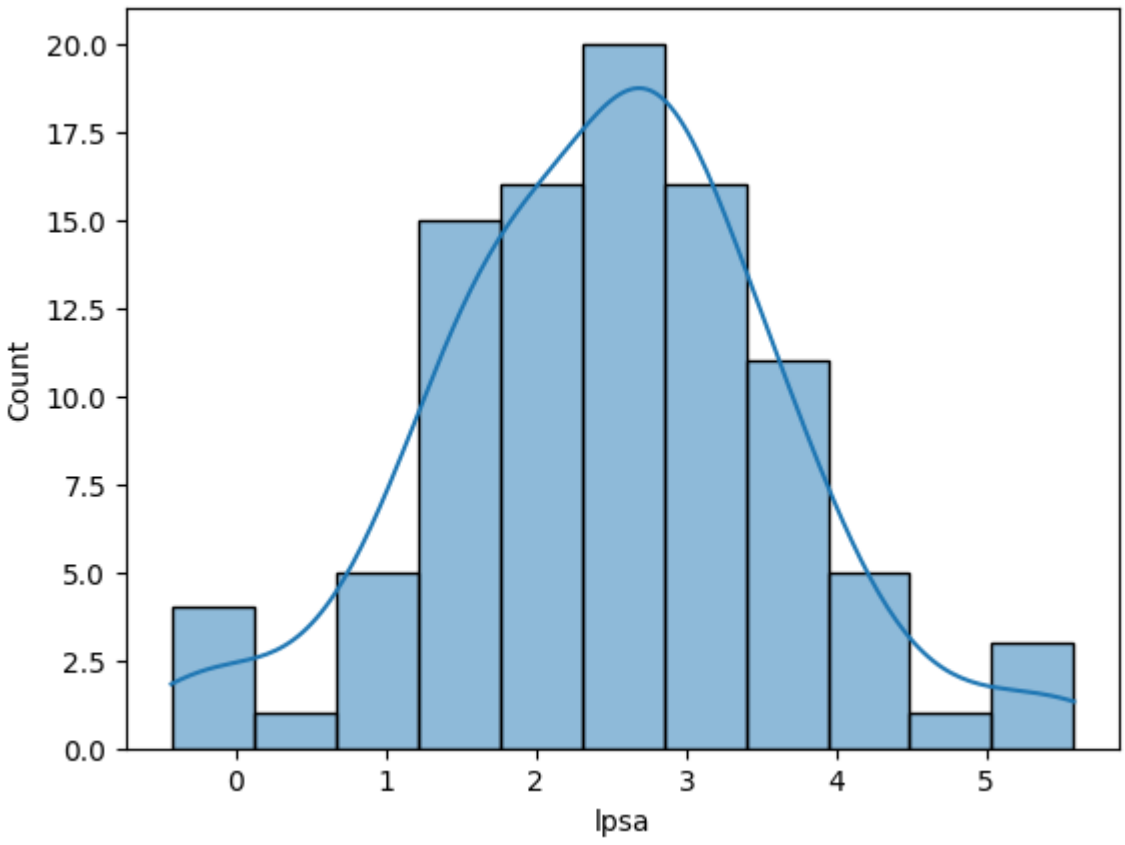
EDA y seleccion de features

Tenemos un dataset numérico continuo (de pontenciales features numéricas continuas) y aunque deberíamos hacer un EDA un poco más exhaustivo, nos vamos a centrar en el análisis de correlaciones para la selección de features. Pero antes analicemos el target univariantemente.

Target

In [18]:
```
sns.histplot(lpsa_data["lpsa"], kde= True)
```

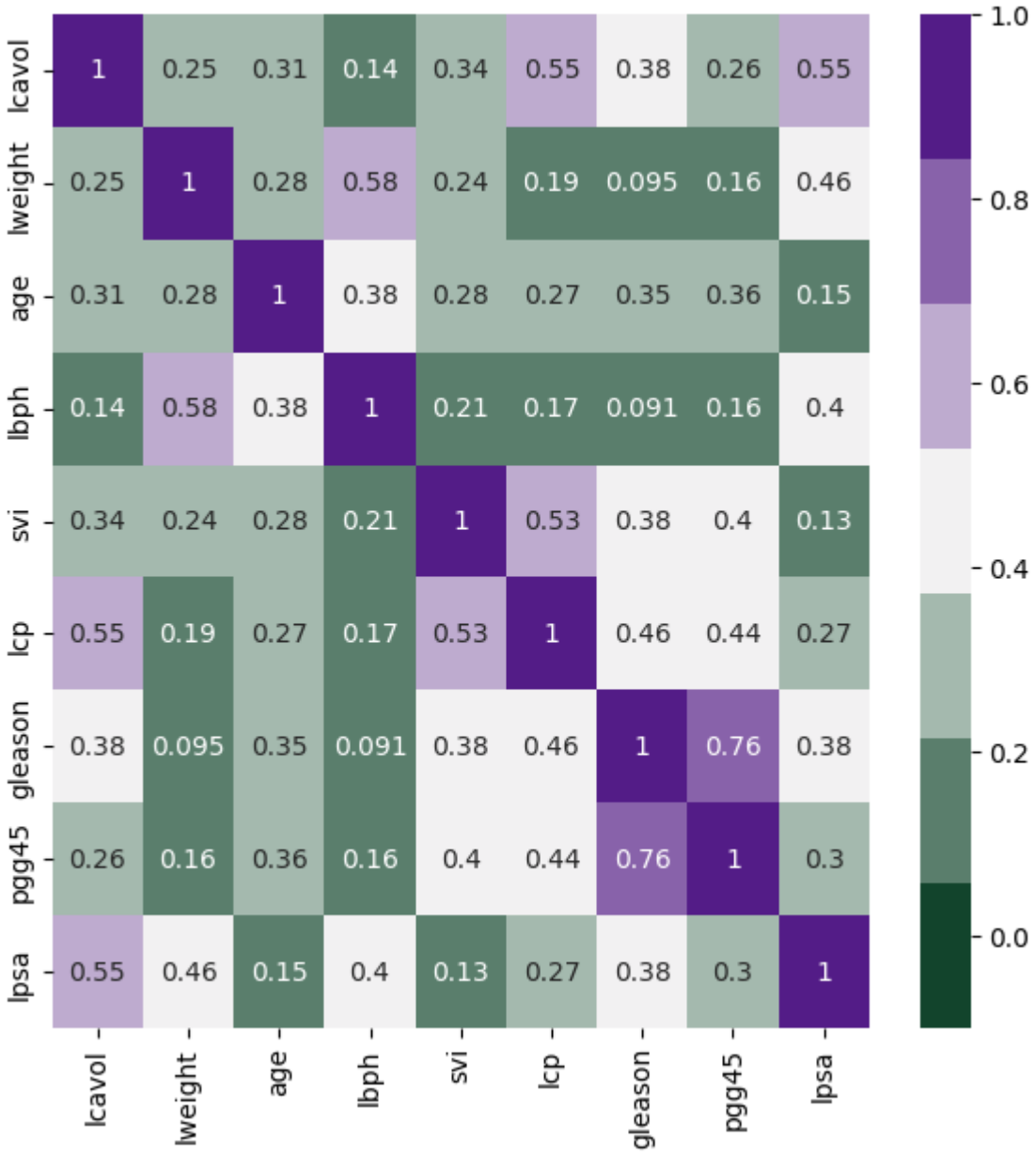
Out[18]:
```
<Axes: xlabel='lpsa', ylabel='Count'>
```



La variable target tiene una distribución casi normal, lo que es bueno para aplicar regresiones lineales.

Veamos las correlaciones:

In [19]:
```
plt.figure(figsize=(7,7))
sns.heatmap(train_set.corr(),
            vmin=-0.1,
            vmax=1,
            cmap=sns.diverging_palette(145, 280, s=85, l=25, n=7),
            annot=True);
```



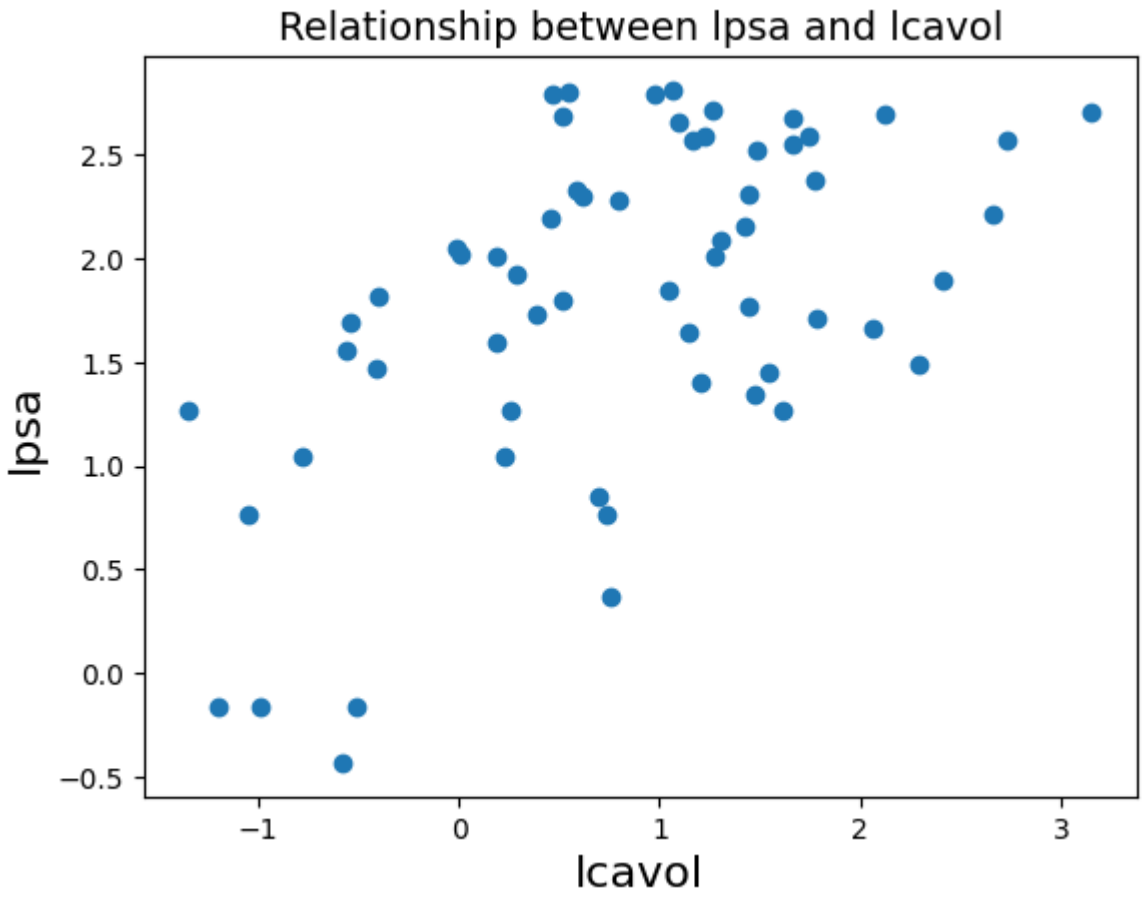
- La variable más correlacionada con la respuesta lpsa es lcavol. Por lo tanto, en un análisis de datos, la variable lcavol debe incluirse como predictor.
- La matriz de correlación muestra que gleason y pgg45 están realmente correlacionados. De hecho, la variable pgg45 mide el porcentaje de puntuaciones de

Gleason 4 o 5 que se registraron antes de la puntuación final actual de Gleason. Descartaremos pgg45 por esta relación.

- Observa que hay otras correlaciones entre las posibles features que nos invitarían a hacer un pequeño análisis de colinearidad (svi-lcp, 0.68 de correlación, pej.), pero no tenemos hueco, aunque probablemente sería una mejora del modelo.

Vamos a mostrar gráficamente la relación entre la respuesta lpsa y la característica lcavol.

In [20]:
```
plt.scatter(train_set['lcavol'], train_set['lpsa'])
plt.xlabel('lcavol', fontsize=16)
plt.ylabel('lpsa', fontsize=16)
plt.title("Relationship between lpsa and lcavol", fontsize=14)
plt.show()
```



In [12]:
```
exclude = ["pgg45"]
X_train = X_train.drop(columns= exclude)
X_test = X_test.drop(columns = exclude)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[12], line 2
      1 exclude = ["pgg45"]
----> 2 X_train = X_train.drop(columns= exclude)
      3 X_test = X_test.drop(columns = exclude)

File /usr/local/lib/python3.8/dist-packages/pandas/core/frame.py:5258, in DataFra
me.drop(self, labels, axis, index, columns, level, inplace, errors)
   5110 def drop(
   5111     self,
   5112     labels: IndexLabel = None,
   (...)
   5119     errors: IgnoreRaise = "raise",
   5120 ) -> DataFrame | None:
   5121     """
   5122     Drop specified labels from rows or columns.
   5123
   (...)
   5256             weight  1.0     0.8
   5257     """
-> 5258     return super().drop(
   5259         labels=labels,
   5260         axis=axis,
   5261         index=index,
   5262         columns=columns,
   5263         level=level,
   5264         inplace=inplace,
   5265         errors=errors,
   5266     )

File /usr/local/lib/python3.8/dist-packages/pandas/core/generic.py:4549, in NDFra
me.drop(self, labels, axis, index, columns, level, inplace, errors)
   4547 for axis, labels in axes.items():
   4548     if labels is not None:
-> 4549         obj = obj._drop_axis(labels, axis, level=level, errors=errors)
   4551 if inplace:
   4552     self._update_inplace(obj)

File /usr/local/lib/python3.8/dist-packages/pandas/core/generic.py:4591, in NDFra
me._drop_axis(self, labels, axis, level, errors, only_slice)
   4589         new_axis = axis.drop(labels, level=level, errors=errors)
   4590     else:
-> 4591         new_axis = axis.drop(labels, errors=errors)
   4592     indexer = axis.get_indexer(new_axis)
   4594 # Case for non-unique axis
   4595 else:

File /usr/local/lib/python3.8/dist-packages/pandas/core/indexes/base.py:6699, in
Index.drop(self, labels, errors)
   6697 if mask.any():
   6698     if errors != "ignore":
-> 6699         raise KeyError(f"{list(labels[mask])} not found in axis")
   6700     indexer = indexer[~mask]
   6701 return self.delete(indexer)

KeyError: "['pgg45'] not found in axis"
```

Modelados

Baseline a partir de un regresor lineal

In [21]:
```
lr = linear_model.LinearRegression()

lr.fit(X_train, y_train)

y_pred = lr.predict(X_test)
#baseline_error = metrics.mean_squared_error(y_test, y_pred)

print("Train MSE:", metrics.mean_squared_error(y_train, lr.predict(X_train)))
print("Test MSE:", metrics.mean_squared_error(y_test, lr.predict(X_test)))
# print("Test RMSE:", np.sqrt(metrics.mean_squared_error(y_test, lr.predict(X_te
```

```
Train MSE: 0.28652244970584706
Test MSE: 2.8208388737948624
```

Fíjate que existe una gran diferencia entre el error de Test y el error de Train, eso nos dice que el modelo generaliza mal y que fallará bastante con datos nuevos. Es hora de ganar bias (tener un error mayor en general) al precio de reducir varianza (ganar generalización) y para ello aplicaremos la regularización.

Aplicamos Regularización (particularizado para regresión lineal)

Para evitar el sobreajuste, empleamos la regularización, que básicamente es controlar la complejidad del modelo (o ponerle "trabas"). En las sesioines teóricas hemos visto mecanismos de regularización basados en incluir una penalización adicional a la función de error. Esta "penalización" se denomina término de regularización ($\lambda Reg(\beta)$). Y en general esta compuesta por dos partes:

- $Reg(\beta)$ que es una forma de penalizar (depende del método, y se usará si quiero limitar los coeficientes menos útiles, el rango de los mismos, etc) -> Depende del método de regularización.
- Y lambda que es el peso que le damos a la penalización (a mayor lambda más estamos "fastidiando" al modelo y más lo estamos regularizando)

Lambda (o alpha en algunas versiones) es nuestro primer "hiperparámetros", recuerda esos valores que sí controlamos desde fuera del modelo (y que además en este caso vamos a poder aplicar a casi todos los modelos) y que nos ayudan a gestionar el entrenamiento. Los mejores valores de los hiperparámetros y no solo los de regularización, se buscan con diferentes técnicas que veremos cuando tratemos específicamente el "ajuste" o *tuning* de los modelos. Volviendo a la regularización que hemos visto en teoría:

Regresión Ridge

En la regresión Ridge, el término de regularización es $Reg(\beta) = \sum_{j=0}^{j=p}\beta_j^2$. (recuerda que β son los coeficientes o pesos de la

regresión y p el número de coeficientes que hayamos incluido, tantos como numero de features más 1)

La regresión Ridge permite reducir la magnitud de los pesos β_i de la regresión lineal, y así evitar el sobreaprendizaje.

La regresión Ridge tiene un efecto de selección agrupada: las variables correlacionadas tienen los mismos pesos.

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

ridgeR = Ridge(alpha = 10) # alpha es lambda
ridgeR.fit(X_train, y_train)

print("Train MSE sin regularización:", round(metrics.mean_squared_error(y_train,
print("Test MSE sin regularización:", round(metrics.mean_squared_error(y_test, l

print("Train MSE:", round(metrics.mean_squared_error(y_train, ridgeR.predict(X_t
print("Test MSE:", round(metrics.mean_squared_error(y_test, ridgeR.predict(X_tes
# print("Test RMSE:", np.sqrt(metrics.mean_squared_error(y_test, ridgeR.predict(
```

```
Train MSE sin regularización: 0.29
Test MSE sin regularización: 2.82
Train MSE: 0.31
Test MSE: 1.97
```

Regresión LASSO (Least Absolute Shrinkage and Selection Operator)

El Lasso realiza una selección de características del modelo: para variables correlacionadas, retiene solo una variable y establece otras variables correlacionadas en cero. La contraparte es que obviamente induce una pérdida de información que resulta en menor precisión.

En Lasso, el término de regularización se define por $Reg(\beta) = \sum_{j=0}^{j=p}|\beta_j|$. Al igual que Ridge, tiene un peso para ese término (alpha o lambda).

In [23]:
```
from sklearn.linear_model import Lasso

lassoR = Lasso(alpha=0.1)
lassoR.fit(X_train, y_train)

print("Train MSE sin regularización:", round(metrics.mean_squared_error(y_train,
print("Test MSE sin regularización:", round(metrics.mean_squared_error(y_test, l

print("Train MSE: %0.4f" % metrics.mean_squared_error(y_train, lassoR.predict(X_
print("Test MSE: %0.4f" % metrics.mean_squared_error(y_test, lassoR.predict(X_te
```

```
Train MSE sin regularización: 0.29
Test MSE sin regularización: 2.82
Train MSE: 0.3677
Test MSE: 1.8816
```

In [24]:
```
lassoR = Lasso(alpha = 0.02)
lassoR.fit(X_train, y_train)
```

```
print("Train MSE: %0.4f" % metrics.mean_squared_error(y_train, lassoR.predict(X_
print("Test MSE: %0.4f" % metrics.mean_squared_error(y_test, lassoR.predict(X_te
```

```
Train MSE: 0.3062
Test MSE: 1.7655
```

Regresion Elastic net

El método Elastic Net es un híbrido de la regresión Ridge y el Lasso. El término de regularización combina tanto las regularizaciones L_1 (Lasso) como L_2 (Ridge).

Más precisamente, el término de regularización se establece en $Reg(\beta) = \lambda((l1_{ratio})||\beta||_1+(1-l1_{ratio})||\beta||_2^2)$ donde $l1_{ratio}$ es un hiperparámetro adicional a ajustar.

El Elastic Net tiene un efecto de selección sobre las variables como el Lasso, pero mantiene variables correlacionadas como la regresión Ridge. Por lo tanto, el modelo Elastic Net es menos disperso que el Lasso, conservando más información. Sin embargo, el modelo demanda más recursos computacionales.

In [25]:
```
from sklearn.linear_model import ElasticNet

elastic_net = ElasticNet(alpha = 0.5, l1_ratio = 0.5)
elastic_net.fit(X_train, y_train)

print("Train MSE: %0.4f" % metrics.mean_squared_error(y_train, elastic_net.predi
print("Test MSE: %0.4f" % metrics.mean_squared_error(y_test, elastic_net.predict
```

```
Train MSE: 0.4765
Test MSE: 2.4897
```

In [26]:
```
elastic_net = ElasticNet(alpha = 0.02, l1_ratio=1)
elastic_net.fit(X_train, y_train)

print("Train MSE: %0.4f" % metrics.mean_squared_error(y_train, elastic_net.predi
print("Test MSE: %0.4f" % metrics.mean_squared_error(y_test, elastic_net.predict
```

```
Train MSE: 0.3062
Test MSE: 1.7655
```

En este caso, sin tunear exahustivamente los hiperparámetros, parece interesante aplicar una Ridge Regression, o una Regularización Ridge a la regresión lineal.

RESUMEN IMPORTANTE:

¿Y sólo hay estos métodos para la regresión lineal?

No, se pueden aplicar penalizaciones L2 y L1 a todas las funciones de pérdidas y veremos que los modelos tienen sus hiperparámetros relacionados para poder "castigar"/"regularizar" con las mismas ideas que Ridge y Lasso (limitar el valor de los coeficientes o parámetros, quitar coeficientes correlados). Eso sí, como no todos los modelos tienen "pesos", no siempre será igual...

Entonces, ¿qué lío?¿Con qué me quedo? Con lo siguiente:

- Todos los modelos tienen hiperparametros (argumentos que nosotros podemos controlar, aka dar el valor que queramos) que sirven para regularizarlos más o menos y controlar el overfitting. Aprenderemos de cada modelo cuáles son.
- El mejor valor de estos hiperparámetros dependerá del nivel de generalización que queramos frente al error (bias) particular en cada predicción. Y aprenderemos a ajustarlos una vez tengamos nuestro modelo ganador.
- Tres tipos de regularización son la Ridge, la Lasso y la combinación (Elastic Net)
- La regularización no sólo es "fastidiar" la función de pérdida, es simplificar el modelo (quitar features, por ejemplo, limitar los valores, usar modelos que internamente son más sencillos, etc)