

# detalles-de-la-regularizaci-n-ibm-machine-l



detalles-de-la-regularizaci-n-ibm-machine-l.ipynb

File

El motor de búsqueda no logró encontrar resultados usando la consulta general. Para poder ayudarte con la explicación detallada del notebook que subiste (**detalles-de-la-regularizaci-n-ibm-machine-l.ipynb**), voy a comenzar abriéndolo directamente y extrayendo todas las celdas de código para analizarlas una por una como me pediste.

Voy a hacerlo ahora. Un momento...

Voy a comenzar explicando las primeras **5 celdas de código** del notebook **detalles-de-la-regularizaci-n-ibm-machine-l.ipynb**. Como pediste, haré una **explicación detallada por bloque de código**, con fundamentos y teoría adicional cuando sea pertinente. Al finalizar estas 5 celdas, te haré un resumen y luego continuaré con las siguientes si así lo deseas.

## ◆ Celda 1: Instalación de librerías

```
python
```

```
import piplite
await piplite.install(['tqdm', 'seaborn', 'pandas', 'numpy'])
```

### Explicación:

- Esta celda está pensada para **entornos de ejecución en el navegador**, como **Pyodide** o **JupyterLite**, donde no se puede usar `pip install`.
- `piplite` permite instalar paquetes de Python en el entorno del navegador.
- Se instalan las siguientes librerías:
  - `tqdm`: para barras de progreso.
  - `seaborn`: para visualizaciones estadísticas.
  - `pandas`: manipulación de datos en estructuras tipo DataFrame.
  - `numpy`: para operaciones numéricas.

### Teoría adicional:

- Este entorno utiliza `await`, por lo tanto requiere una **función asíncrona**. En JupyterLite, cada celda puede tener `await` de forma implícita.
- En un entorno local de Jupyter, esta celda debería comentarse.

## ◆ Celda 2: Importaciones, configuración de warnings y display

```
python

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# Suprimir warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn

np.set_printoptions(precision=3, suppress=True)
```

### Explicación:

- Se importan las librerías clave:
  - **numpy** como **np**: para operaciones matemáticas.
  - **pandas** como **pd**: para trabajar con tablas de datos.
  - **matplotlib.pyplot**: para graficar.
- **%matplotlib inline** es un **magic command** para que los gráficos se muestren embebidos en el notebook.
- Se define una función vacía **warn** para anular los warnings del sistema.
- Se configura **numpy** para que:
  - Muestre los números flotantes con **3 decimales**.
  - **No use notación científica** (gracias a **suppress=True**).

### Teoría adicional:

- Esto es útil cuando se quiere mantener una salida limpia y legible, especialmente en notebooks educativos.

## ◆ Celda 3: Semilla aleatoria y funciones auxiliares

```
python

np.random.seed(72018)

def to_2d(array):
    return array.reshape(array.shape[0], -1)

def plot_exponential_data():
    data = np.exp(np.random.normal(size=1000))
    plt.hist(data)
    plt.show()
    return data

def plot_square_normal_data():
    data = np.square(np.random.normal(loc=5, size=1000))
    plt.hist(data)
```

```
plt.show()
return data
```

### Explicación:

- `np.random.seed(72018)`: fija la semilla para resultados reproducibles en generación aleatoria.
- `to_2d(array)`: convierte un array 1D o cualquier forma a una matriz 2D con `array.shape[0]` filas.
- `plot_exponential_data()`:
  - Genera 1000 números aleatorios con distribución normal ( $\sim N(0,1)$ ).
  - Aplica exponencial: los datos ahora siguen una **distribución log-normal**.
  - Se grafica el histograma.
- `plot_square_normal_data()`:
  - Genera 1000 números aleatorios con media 5.
  - Eleva al cuadrado → distribución sesgada hacia la derecha.
  - Se grafica el histograma.

### Teoría adicional:

- La distribución exponencial y la distribución cuadrada son ejemplos de **distribuciones no normales**.
- Este tipo de transformaciones se usa para ilustrar **escalamiento** o técnicas de **regularización** en aprendizaje automático.

## ◆ Celda 4: Descarga de datos vía pyodide

```
python

from pyodide.http import pyfetch

async def download(url, filename):
    response = await pyfetch(url)
    if response.status == 200:
        with open(filename, "wb") as f:
            f.write(await response.bytes())

path = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML240EN-SkillsNetwork/labs/data/boston_housing_clean.pickle"
await download(path, "boston_housing_clean.pickle")

# Import pandas library
import pandas as pd
```

### Explicación:

- `pyfetch` es una API de `pyodide` que simula `fetch` de JavaScript.
- La función `download` descarga un archivo remoto de datos y lo guarda localmente.
- Se baja el dataset limpio de Boston Housing, almacenado como un archivo `pickle` (formato binario de Python).

## Teoría adicional:

- `async / await` es el sistema de **programación asíncrona** de Python 3.7+.
- El dataset de **Boston Housing** es un clásico en ML supervisado. Cada fila representa un barrio de Boston, y el objetivo suele ser predecir el valor medio de las casas.

## ◆ Celda 5: Carga del dataset y vista preliminar

```
python

with open('boston_housing_clean.pickle', 'rb') as to_read:
    boston = pd.read_pickle(to_read)

boston_data = boston['dataframe']
boston_description = boston['description']

# show the first 5 rows using dataframe.head() method
print("The first 5 rows of the dataframe")
boston_data.head()
```

## Explicación:

- Se abre el archivo `.pickle` en modo lectura binaria.
- `pd.read_pickle` recupera el contenido (objeto serializado) en formato original.
- El archivo contiene dos objetos:
  - `dataframe`: los datos.
  - `description`: descripción textual o metadatos.
- `boston_data.head()` muestra las primeras 5 filas para entender la estructura de los datos.

## Teoría adicional:

- `pickle` es rápido y eficiente para guardar/leer estructuras de datos en Python.
- Este paso es importante para hacer el **análisis exploratorio de datos (EDA)**.

## ✅ Resumen de las primeras 5 celdas

1. **Instala las librerías necesarias** en un entorno WebAssembly como JupyterLite.
2. **Configura el entorno**: warnings, estilo de impresión, y gráficas embebidas.
3. **Define funciones auxiliares** para visualización y transformación de datos sintéticos.
4. **Descarga un dataset real** de regresión (Boston Housing) usando `pyfetch`.
5. **Carga el dataset** en un `DataFrame` de Pandas y muestra sus primeras filas.

¿Quieres que continúe con las siguientes celdas del notebook? AVISAME y sigo explicando.



continua

Vamos a continuar con la explicación detallada de las **celdas 6 a 10** del notebook.

## ◆ Celda 6: Separación de variables predictoras y objetivo

```
python

y_col = "MEDV"

X = boston_data.drop(y_col, axis=1)
y = boston_data[y_col]
```

### Explicación:

- Se define **MEDV** (Median value of owner-occupied homes in \$1000's) como la variable objetivo **y**.
- **x** contiene todas las columnas excepto **MEDV**.
- **axis=1** indica que se está eliminando una **columna** (no una fila).

### Teoría adicional:

- Este es un paso clave en cualquier problema de **aprendizaje supervisado**, donde:
  - **x** representa los **atributos o características (features)**.
  - **y** es el **target o etiqueta** que queremos predecir.
- En regresión lineal,  $y \approx X\beta + \epsilon$ .

## ◆ Celda 7: Estandarización de las variables predictoras

```
python

from sklearn.preprocessing import StandardScaler

s = StandardScaler()
X_ss = s.fit_transform(X)
```

### Explicación:

- Se importa **StandardScaler** del módulo **sklearn.preprocessing**.
- **fit\_transform(X)**:
  - Calcula la media y desviación estándar de cada variable.
  - Transforma los datos para que tengan media 0 y desviación estándar 1.

### Teoría adicional:

- Esta técnica de **normalización o estandarización Z-score** es fundamental en muchos algoritmos de Machine Learning, especialmente en:
  - Regresión con regularización (Ridge, Lasso).
  - KNN, SVM, PCA.
- Mejora la **estabilidad numérica** y evita que atributos con unidades diferentes dominen la solución.

### ◆ Celda 8: Ejemplo de arreglo NumPy

```
python

#Hint:

a = np.array([[1, 2, 3],
              [4, 5, 6]])
print(a) # 2 rows, 3 columns
```

#### Explicación:

- Se crea una matriz **a** de 2x3.
- Es un recordatorio (hint) sobre cómo se comportan los arrays de NumPy en operaciones de agregación.

### ◆ Celda 9: Promedio por columnas

```
python

a.mean(axis=0) # mean along the *columns*
```

#### Explicación:

- **axis=0** calcula la media **por columnas**.
- Resultado:  $[(1+4)/2, (2+5)/2, (3+6)/2] = [2.5, 3.5, 4.5]$ .

### ◆ Celda 10: Promedio por filas

```
python

a.mean(axis=1) # mean along the *rows*
```

#### Explicación:

- **axis=1** calcula la media **por filas**.
- Resultado:  $[(1+2+3)/3, (4+5+6)/3] = [2.0, 5.0]$ .

#### Teoría adicional:

- Estas celdas están pensadas para reforzar el uso de **funciones de agregación** en NumPy.
- En el contexto de Machine Learning, entender los ejes es crucial para operaciones como:
  - Normalización.
  - Cálculo de métricas.
  - Agrupamientos o transformaciones de vectores.

## ✓ Resumen de las celdas 6 a 10

1. Se define la variable objetivo **MEDV** y se separan las características (x) del target (y).
2. Se aplicó **escalado estándar (Z-score)** a los datos, esencial para regularización.
3. Se mostró un ejemplo pedagógico de cómo se calculan **promedios por filas y columnas** usando `np.mean`.

¿Deseas que siga con las siguientes celdas del notebook? Avísame y continúo.



continua

Vamos ahora con la explicación detallada de las **celdas 11 a 15** del notebook.

### ♦ Celda 11: Verificación manual de estandarización

```
python

### BEGIN SOLUTION
X2 = np.array(X)
man_transform = (X2 - X2.mean(axis=0)) / X2.std(axis=0)
np.allclose(man_transform, X_ss)
### END SOLUTION
```

#### Explicación:

- Se convierte el DataFrame **x** a un **ndarray** de NumPy con `np.array(X) → X2`.
- Se calcula **manualmente** la transformación Z-score:

$$Z = \frac{X - \mu}{\sigma}$$

- `X2.mean(axis=0)`: promedio por columna.
- `X2.std(axis=0)`: desviación estándar por columna.
- `np.allclose(...)` verifica si los valores de `man_transform` son numéricamente iguales (con tolerancia) a `X_ss`, que se obtuvo usando `StandardScaler`.

#### Teoría adicional:

- Esta celda sirve como **verificación matemática** de que `StandardScaler` realiza efectivamente una estandarización por columnas.
- Buena práctica para ilustrar el trasfondo matemático del preprocesamiento automático.

## ◆ Celda 12: Importación del modelo lineal

```
python

from sklearn.linear_model import LinearRegression
```

### Explicación:

- Se importa la clase `LinearRegression` desde el módulo `linear_model` de `scikit-learn`.
- Este modelo estima la relación lineal entre las variables independientes y la variable dependiente:

$$y = \beta_0 + \beta_1 x_1 + \dots + \beta_n x_n + \epsilon$$

## ◆ Celda 13: Preparación para entrenamiento del modelo

```
python

lr = LinearRegression()

y_col = "MEDV"

X = boston_data.drop(y_col, axis=1)
y = boston_data[y_col]
```

### Explicación:

- Se instancia el modelo lineal `lr`.
- Se vuelve a preparar `x` y `y` a partir de `boston_data`, como en la celda 6.

🔴 Nota: Esto repite código anterior, probablemente por segmentación en secciones del notebook.

## ◆ Celda 14: Entrenamiento del modelo lineal

```
python

lr.fit(X, y)
print(lr.coef_) # min = -18
```

### Explicación:

- `lr.fit(X, y)`: entrena el modelo de regresión lineal con los datos.
- `lr.coef_`: imprime los coeficientes (pesos) asociados a cada variable predictora.

### Teoría adicional:



- El modelo ajusta los coeficientes  $\beta$  que minimizan el **error cuadrático medio (MSE)**:

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

- El comentario indica que **al menos un coeficiente es negativo**, lo que es común: algunas variables pueden tener relación inversa con el precio.

## ♦ Celda 15: Importación redundante

```
python

from sklearn.preprocessing import StandardScaler
```

### Explicación:

- Esta importación ya se realizó en la **celda 7**, por lo que aquí es **redundante**.
- Posiblemente se repite por organización del notebook o por separar secciones conceptuales.

## ✓ Resumen de las celdas 11 a 15

1. Se verifica manualmente que la estandarización calculada con fórmula coincide con la de **StandardScaler**.
2. Se importa y configura un modelo de regresión lineal.
3. Se entrena el modelo con los datos originales (sin escalar).
4. Se imprimen los coeficientes del modelo.
5. Hay una importación duplicada de **StandardScaler**.

¿Quieres que siga con las próximas celdas del notebook? AVISAME y continúo.



continua

Continuamos con la explicación detallada de las **celdas 16 a 20** del notebook.

## ♦ Celda 16: Estandarización de x nuevamente

```
python

s = StandardScaler()
X_ss = s.fit_transform(X)
```

### Explicación:

- Se aplica nuevamente **StandardScaler** sobre los datos **X**.
- Esta vez será para usar los datos estandarizados con otro modelo lineal.

🔪 Esta celda repite lo hecho en la **celda 7**, lo cual indica que el flujo del notebook se reestablece para un análisis más profundo (ej. comparación de coeficientes con y sin escalado).

## ♦ Celda 17: Entrenamiento con datos estandarizados

```
python

lr2 = LinearRegression()
lr2.fit(X_ss, y)
print(lr2.coef_) # coefficients now "on the same scale"
```

### Explicación:

- Se instancia un nuevo modelo **lr2**.
- Se entrena sobre **X\_ss** (ya estandarizado).
- Se imprimen los coeficientes.

### Teoría adicional:

- Cuando se trabaja con **regresión lineal regularizada**, o cuando se quieren **comparar magnitudes de los coeficientes**, es obligatorio estandarizar los datos.
- Ahora los coeficientes están “en la misma escala” y permiten interpretar mejor la importancia relativa de cada variable.

## ♦ Celda 18: Visualización ordenada de los coeficientes

```
python

### BEGIN SOLUTION
pd.DataFrame(zip(X.columns, lr2.coef_)).sort_values(by=1)
### END SOLUTION
```

### Explicación:

- Se crea un **DataFrame** que relaciona cada variable con su coeficiente aprendido por **lr2**.
- Se usa **zip** para unir nombres de columnas con coeficientes.
- Se ordena por el valor del coeficiente (**by=1** → segunda columna del **DataFrame**).

### Interpretación:

- Esta celda busca responder:
  - ¿Qué variables tienen mayor impacto positivo?
  - ¿Qué variables influyen negativamente en el precio de la vivienda?

- ¿Cuál es la más relevante (en valor absoluto)?

## ◆ Celda 19: Importación de Lasso y PolynomialFeatures

```
python

from sklearn.linear_model import Lasso
from sklearn.preprocessing import PolynomialFeatures
```

### Explicación:

- Se importan dos herramientas importantes para regularización y complejidad del modelo:
  - **Lasso**: regresión lineal con regularización **L1**.
  - **PolynomialFeatures**: expande las variables lineales a polinomios de orden superior, aumentando la capacidad del modelo.

### Teoría adicional:

- **Lasso** impone una penalización basada en la **suma de los valores absolutos** de los coeficientes:

$$\text{Loss} = \sum (y - \hat{y})^2 + \alpha \sum |\beta_i|$$

- Esta penalización favorece coeficientes **exactamente cero**, haciendo que el modelo seleccione automáticamente variables → **regularización + selección de variables**.
- **PolynomialFeatures** transforma  $x$  en  $x, x^2, x_1x_2, \dots$ , aumentando la dimensionalidad y permitiendo capturar relaciones no lineales.

## ◆ Celda 20: Generación de características polinómicas

```
python

pf = PolynomialFeatures(degree=2, include_bias=False,)
X_pf = pf.fit_transform(X)
```

### Explicación:

- Se crea un generador de características polinómicas de **grado 2** sin incluir el término de sesgo (**include\_bias=False**).
- **fit\_transform(X)**:
  - Genera todas las combinaciones de grado 2:
    - cuadrados:  $x_i^2$
    - interacciones:  $x_i \cdot x_j$
- **X\_pf** ahora es una **matriz con muchas más columnas**, representando relaciones cuadráticas.

### Teoría adicional:

- Esta técnica se conoce como **expansión de características**.
- Permite aplicar modelos lineales sobre relaciones no lineales entre variables.
- La combinación con **regularización (ej. Lasso)** es útil para controlar el sobreajuste que podría surgir de esta expansión.

## ✓ Resumen de las celdas 16 a 20

1. Se estandarizan nuevamente los datos y se entrena un nuevo modelo lineal.
2. Se visualizan los coeficientes en orden para interpretar su impacto.
3. Se importan herramientas para **regularización L1 (Lasso)** y **expansión polinómica**.
4. Se generan características de **segundo grado** para capturar relaciones cuadráticas entre variables.

¿Quieres que continúe con las siguientes celdas? Puedo seguir hasta el final y avisarte cuando no haya más que explicar.



continua

Seguimos con la explicación detallada de las **celdas 21 a 25** del notebook.

### ♦ Celda 21: Estandarización de las características polinómicas

```
python

X_pf_ss = s.fit_transform(X_pf)
```

#### Explicación:

- Se reutiliza el objeto **StandardScaler (s)** para escalar las nuevas variables polinómicas (**X\_pf**).
- El resultado es **X\_pf\_ss**, que contiene **todas las combinaciones cuadráticas estandarizadas**.

#### Importancia:

- Esta estandarización es crítica porque:
  - Los términos cuadráticos e interacciones pueden tener magnitudes muy distintas.
  - La **regularización Lasso es muy sensible a la escala** de los datos.

### ♦ Celda 22: Entrenamiento con Lasso sin regularización explícita

```
python

las = Lasso()
las.fit(X_pf_ss, y)
las.coef_
```

### Explicación:

- Se instancia un modelo de regresión Lasso con el valor por defecto de regularización **alpha=1.0**.
- Se entrena el modelo sobre los datos polinómicos estandarizados.
- `las.coef_` devuelve los coeficientes del modelo entrenado.

### Teoría adicional:

- Lasso con características polinómicas busca:
  - Ajustar un modelo no lineal.
  - Eliminar automáticamente muchas variables irrelevantes al fijar sus coeficientes a **cero**.

## ◆ Celda 23: Lasso con alpha = 0.1

```
python

### BEGIN SOLUTION
las01 = Lasso(alpha = 0.1)
las01.fit(X_pf_ss, y)
print('sum of coefficients:', abs(las01.coef_).sum() )
print('number of coefficients not equal to 0:', (las01.coef_!=0).sum())
```

### Explicación:

- Se entrena un modelo Lasso con menor penalización **alpha=0.1**.
- Se imprime:
  - La **suma de los valores absolutos** de los coeficientes → indica el “tamaño total” del modelo.
  - El número de coeficientes **diferentes de cero** → indica cuántas variables fueron consideradas relevantes.

### Teoría adicional:

- Disminuir **alpha** reduce la fuerza de la regularización:
  - Menor sparsity (menos coeficientes en cero).
  - Mayor riesgo de sobreajuste.
- Este análisis permite **comparar distintos niveles de regularización**.

## ◆ Celda 24: Lasso con alpha = 1

python

```
las1 = Lasso(alpha = 1)
las1.fit(X_pf_ss, y)
print('sum of coefficients:',abs(las1.coef_).sum() )
print('number of coefficients not equal to 0:',(las1.coef_!=0).sum())
### END SOLUTION
```

### Explicación:

- Se entrena otro modelo Lasso, ahora con **alpha=1** (fuerte regularización).
- Se imprimen las mismas métricas:
  - Suma de coeficientes.
  - Número de variables activas.

🔗 Esto permite comparar con el modelo anterior (alpha = 0.1).

## ♦ Celda 25: Evaluación del modelo con $R^2$

python

```
### BEGIN SOLUTION
from sklearn.metrics import r2_score
r2_score(y, las.predict(X_pf_ss))
### END SOLUTION
```

### Explicación:

- Se importa **r2\_score**, una métrica que evalúa el **poder explicativo** del modelo:

$$R^2 = 1 - \frac{RSS}{TSS}$$

- RSS: suma de errores al cuadrado.
- TSS: variabilidad total en los datos.
- Se evalúa el modelo Lasso original (**las**, con **alpha=1.0**).
- El valor de  $R^2 \in (-\infty, 1]$ , y valores cercanos a 1 indican buen ajuste.

## ✓ Resumen de las celdas 21 a 25

1. Se estandarizan las variables polinómicas generadas previamente.
2. Se entrena un modelo Lasso sobre esas variables.
3. Se prueban distintos valores de **alpha** (0.1 y 1) y se observa:
  - Cuántos coeficientes son distintos de cero.
  - Qué tan complejo y regularizado es el modelo.
4. Se evalúa el modelo con la métrica  $R^2$  para medir su desempeño explicativo.

¿Quieres que siga con las siguientes celdas del notebook? Estoy cerca del final y te avisaré cuando no haya más.



continua

Vamos con la **última tanda de celdas (26 a 30)** del notebook. Aquí se aplican técnicas de **validación, entrenamiento y evaluación en datos separados**.

### ♦ Celda 26: Importación del método de partición

```
python

from sklearn.model_selection import train_test_split
```

#### Explicación:

- Se importa `train_test_split`, una función fundamental en validación de modelos.
- Sirve para dividir los datos en **conjunto de entrenamiento** y **conjunto de prueba**.

### ♦ Celda 27: División de los datos

```
python

X_train, X_test, y_train, y_test = train_test_split(X_pf, y, test_size=0.3,
                                                    random_state=72018)
```

#### Explicación:

- Se separan los datos expandidos polinómicamente (`X_pf`) y el objetivo `y` en:
  - 70% para entrenamiento.
  - 30% para prueba.
- `random_state=72018` garantiza que la división sea **reproducible**.

### ♦ Celda 28: Evaluación del modelo Lasso ( $\alpha=1$ )

```
python

X_train_s = s.fit_transform(X_train)
las.fit(X_train_s, y_train)
X_test_s = s.transform(X_test)
y_pred = las.predict(X_test_s)
r2_score(y_test, y_pred)
```

## Explicación:

- Se estandarizan los datos de entrenamiento (`fit_transform`) y de prueba (`transform`).
- Se entrena el modelo `las` (Lasso con `alpha=1`).
- Se predicen valores en `X_test_s` y se calcula el  $R^2$  contra `y_test`.

### ◆ Celda 29: Evaluación del modelo Lasso (`alpha=0.1`)

```
python

X_train_s = s.fit_transform(X_train)
las01.fit(X_train_s, y_train)
X_test_s = s.transform(X_test)
y_pred = las01.predict(X_test_s)
r2_score(y_test, y_pred)
```

## Explicación:

- Mismo procedimiento que la celda anterior, pero usando `las01` (Lasso con `alpha=0.1`).
- Esto permite **comparar el impacto de una regularización más débil**.

### ◆ Celda 30: Comparación completa entre Lasso y regresión lineal

```
python

### BEGIN SOLUTION

# Part 1

# Decreasing regularization and ensuring convergence
las001 = Lasso(alpha = 0.001, max_iter=100000)

# Transforming training set to get standardized units
X_train_s = s.fit_transform(X_train)

# Fitting model to training set
las001.fit(X_train_s, y_train)

# Transforming test set using the parameters defined from training set
X_test_s = s.transform(X_test)

# Finding prediction on test set
y_pred = las001.predict(X_test_s)

# Calculating r2 score
print("r2 score for alpha = 0.001:", r2_score(y_test, y_pred))

# Part 2

# Using vanilla Linear Regression
lr = LinearRegression()

# Fitting model to training set
lr.fit(X_train_s, y_train)

# predicting on test set
y_pred_lr = lr.predict(X_test_s)

# Calculating r2 score
```



```
print("r2 score for Linear Regression:", r2_score(y_test,y_pred_lr))

# Part 3
print('Magnitude of Lasso coefficients:', abs(las001.coef_).sum())
print('Number of coefficients not equal to 0 for Lasso:', (las001.coef_!=0).sum())

print('Magnitude of Linear Regression coefficients:', abs(lr.coef_).sum())
print('Number of coefficients not equal to 0 for Linear Regression:', (lr.coef_!=0).sum())
### END SOLUTION
```

## Explicación por partes:

### ● Parte 1: Lasso con $\alpha=0.001$

- Se entrena un modelo **Lasso con muy poca regularización**.
- `max_iter=100000` asegura convergencia porque hay muchas variables.
- Se calcula el  $R^2$  en test set.

### ● Parte 2: Regresión lineal clásica

- Se entrena un modelo de regresión sin regularización (OLS).
- Se calcula el  $R^2$  en el mismo test set.

### ● Parte 3: Comparación de coeficientes

- Se comparan magnitudes totales de los coeficientes (suma de valores absolutos).
- También se compara **cuántos coeficientes son distintos de cero**, es decir:
  - **Lasso** hace selección de variables (sparsity).
  - **OLS** no hace selección (todos  $\neq 0$ ).

## ✓ CONCLUSIÓN GENERAL DEL NOTEBOOK

Este notebook explora **regresión lineal regularizada**, especialmente **Lasso**, sobre el dataset de viviendas de Boston:

### 1. Carga y preprocesamiento:

- Se cargó el dataset serializado `.pickle`.
- Se separaron variables predictoras y objetivo.
- Se aplicó escalado estándar (Z-score).

### 2. Modelado lineal:

- Se entrenó un modelo de regresión lineal clásico.
- Se visualizaron e interpretaron coeficientes.

### 3. Regularización y expansión:

- Se aplicó expansión polinómica de grado 2.

- Se entrenaron modelos Lasso con distintos niveles de  **$\alpha$** .
- Se analizó el número de variables activas y magnitudes de los coeficientes.

#### 4. Evaluación en conjunto de prueba:

- Se compararon modelos con diferentes regularizaciones.
- Se midió desempeño usando  $R^2$ .
- Se compararon modelos en términos de **complejidad y poder predictivo**.

## TEORÍA ADICIONAL Y CONCEPTOS CLAVE

### Regresión Lineal

- Modelo que busca ajustar una combinación lineal de variables para predecir un valor continuo.
- Objetivo: minimizar el **Error Cuadrático Medio (MSE)**.

### Regularización Lasso (L1)

- Penaliza la **suma de los valores absolutos** de los coeficientes.
- Tiende a hacer **sparse models** → muchos coeficientes en cero.
- Útil para selección automática de variables.

### Estandarización (Z-score)

- Convierte los datos a una media 0 y desviación estándar 1.
- Necesaria para comparar coeficientes o usar regularización correctamente.

### Validación cruzada / Separación train-test

- Entrenar y evaluar en el mismo conjunto lleva a sobreajuste.
- Por eso se divide el dataset en train y test, o se hace K-Fold.

### Métrica $R^2$

- Proporción de varianza explicada por el modelo.
- Rango:  $(-\infty, 1]$ , donde 1 es ajuste perfecto.

### NO HAY MÁS CELDAS DE CÓDIGO PARA EXPLICAR.

¿Querés que te ayude a convertir esto en un informe profesional o agregar gráficos de interpretación de coeficientes y regularización?