

validacion cruzada IBM



validaci-n-cruzada-ibm-machine-l.ipynb

File

Gracias por subir el archivo `validaci-n-cruzada-ibm-machine-l.ipynb`. Voy a proceder a analizarlo celda por celda, explicando cada bloque de código de forma detallada, con fundamentos teóricos y matemáticos cuando corresponda.

Comenzaré ahora con la **primera celda de código**. Dame un momento mientras leo su contenido y empiezo con la explicación.

Vamos a comenzar con la explicación detallada de las **primeras tres celdas de código** del notebook `validaci-n-cruzada-ibm-machine-l.ipynb`.

♦ Celda 1: Instalación y carga de librerías

```
python

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

!pip install numpy
!pip install pandas
!pip install matplotlib
!pip install scikit-learn

import numpy as np
import pickle
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.model_selection import KFold, cross_val_predict
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.metrics import r2_score
from sklearn.pipeline import Pipeline
```

✓ Explicación:

1. Supresión de advertencias:

```
python

def warn(*args, **kwargs): pass
warnings.warn = warn
```

Esto evita que se muestren mensajes de advertencia, lo cual puede ser útil para mantener limpio el output del notebook, aunque en desarrollo no siempre es recomendable ocultarlos.

2. Instalación de paquetes:

```
python

!pip install ...
```

Se asegura de que las librerías necesarias estén disponibles. Esta sintaxis con `!` es típica de entornos Jupyter para ejecutar comandos de terminal.

3. Importaciones:

- **numpy**, **pandas** y **matplotlib.pyplot** son librerías fundamentales para análisis numérico, manejo de datos y visualización.
- **pickle** se usa para cargar datos serializados (guardados en binario).
- De **scikit-learn**, se importan:
 - **Preprocesamiento:** **StandardScaler**, **PolynomialFeatures**.
 - **Modelos:** **LinearRegression**, **Lasso**, **Ridge**.
 - **Evaluación:** **KFold**, **cross_val_predict**, **r2_score**.
 - **Pipeline:** estructura para encadenar pasos del flujo de datos (escalado, modelo, etc.).

◆ Celda 2: Carga del dataset

```
python

# Note we are loading a slightly different ("cleaned") pickle file
!wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML240EN-SkillsNetwork/labs/data/boston_housing_clean.pickle
# boston = pickle.load(open('data/boston_housing_clean.pickle', "rb" ))
boston = pickle.load(open('boston_housing_clean.pickle', "rb" ))
```

✓ Explicación:

1. Descarga del dataset:

Usa **wget** para obtener un archivo **.pickle** (ya procesado) del dataset de **viviendas en Boston**, una base de datos clásica para regresión en Machine Learning.

2. Lectura con pickle:

boston = pickle.load(...) carga el archivo como un objeto de Python, probablemente un diccionario que contiene los datos ya limpios y listos para el análisis.

◆ Celda 3: Exploración del objeto cargado

```
python

boston.keys()
```

✓ Explicación:

Se listan las claves del diccionario `boston`, lo que nos permite entender qué estructuras contiene. Por ejemplo:

- `'data'` → los datos como un DataFrame.
- `'target'` → la variable objetivo (precio de viviendas).
- `'feature_names'` → nombres de las columnas.

✓ CONCLUSIÓN PARCIAL

Estas primeras celdas cumplen la función de:

1. Preparar el entorno de trabajo (librerías, advertencias, dependencias).
2. Descargar y cargar el dataset de entrenamiento.
3. Explorar la estructura general de los datos para su posterior análisis.

Este bloque es **fundamental como etapa de inicialización** para cualquier notebook de Machine Learning. La elección del dataset de Boston también es clásica porque se presta bien a tareas de regresión y validación de modelos.

INFORMACIÓN ADICIONAL

♦ Dataset de Boston Housing

- Contiene información socioeconómica y ambiental de distintas zonas en Boston.
- La variable objetivo es el **valor medio de las casas**.
- Fue retirado de `scikit-learn` por considerarse obsoleto y sesgado, pero aún se utiliza para fines educativos.

♦ Métodos de regresión importados:

- `LinearRegression`: regresión lineal estándar.
- `Ridge`: regresión con regularización L2.
- `Lasso`: regresión con regularización L1.

♦ Pipeline de `sklearn`

Permite encadenar pasos (por ejemplo, escalado + modelo), lo cual mejora la organización del código y evita fugas de datos en validación cruzada.

♦ `cross_val_predict`

Genera predicciones en cada fold del modelo, útil para evaluar con métricas como `r2_score`.

¿Deseas que continúe con la siguiente celda del notebook?



continuar

Vamos ahora con la explicación detallada de las siguientes tres celdas del notebook.

◆ Celda 4: Versión de Python

```
python

!python --version
```

✓ Explicación:

Este comando de sistema imprime la versión actual de Python que se está utilizando en el entorno. Esto es útil para:

- Verificar compatibilidad con bibliotecas (por ejemplo, ciertas versiones de **scikit-learn** requieren Python 3.7 o superior).
- Documentar el entorno para reproducibilidad.

✓ Buenas prácticas de ciencia de datos: **registrar versiones** de herramientas y paquetes utilizados.

◆ Celda 5: Extracción de datos desde el objeto cargado

```
python

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

✓ Explicación:

El objeto **boston** cargado desde el archivo **.pickle** contiene al menos dos claves:

1. **'dataframe'**: una tabla (tipo **pandas.DataFrame**) con todas las variables predictoras y la variable objetivo (**MEDV**, valor medio de vivienda).
2. **'description'**: texto descriptivo o metadatos sobre el dataset, útil para entender qué mide cada variable.

Esta celda simplemente **extrae y almacena esas dos partes** por separado para su uso posterior.

◆ Celda 6: Vista preliminar del dataset

```
python

boston_data.head()
```

✓ Explicación:

- El método `.head()` de un **DataFrame** de Pandas muestra las primeras 5 filas del conjunto de datos.
- Sirve como **exploración inicial** para:
 - Verificar que los datos se cargaron correctamente.
 - Observar nombres de columnas y tipos de datos.
 - Identificar si hay columnas categóricas, numéricas o faltantes.

Es común en análisis de datos empezar con un `.head()` antes de hacer estadística descriptiva.

✓ CONCLUSIÓN PARCIAL

Estas tres celdas cumplen funciones importantes para:

1. Registrar la versión de Python usada.
2. Extraer y organizar los datos reales del objeto **boston**.
3. Realizar una primera inspección visual del dataset.

Es una fase previa al preprocesamiento y modelado, fundamental en cualquier flujo de trabajo de Machine Learning.

INFORMACIÓN ADICIONAL

♦ ¿Qué variables contiene **boston_data**?

Algunas de las variables típicas del dataset de Boston Housing son:

Columna	Significado
CRIM	Tasa de criminalidad per cápita por ciudad
ZN	Proporción de terrenos residenciales para lotes grandes
INDUS	Proporción de acres comerciales no minoristas
CHAS	Variable ficticia: 1 si colinda con el río Charles, 0 si no
NOX	Concentración de óxidos de nitrógeno
RM	Promedio de habitaciones por vivienda
AGE	Proporción de unidades ocupadas construidas antes de 1940
DIS	Distancia a centros de empleo
TAX	Tasa de impuesto a la propiedad
LSTAT	Porcentaje de población con menor estatus socioeconómico
MEDV	Valor medio de la vivienda (en miles de USD) ← variable objetivo

¿Querés que siga con las próximas celdas del notebook?



continua

Vamos ahora con la explicación detallada de las siguientes tres celdas del notebook.

◆ Celda 7: Separación de variables predictoras y objetivo

python

```
X = boston_data.drop('MEDV', axis=1)
y = boston_data.MEDV
```

✓ Explicación:

- **x** contiene las **variables independientes** (predictoras) que se usarán para entrenar el modelo.
- **y** contiene la **variable dependiente** o **target**, en este caso **MEDV**, que representa el valor medio de la vivienda.

Se utiliza `drop('MEDV', axis=1)` para eliminar esa columna de **x**, asegurando que el modelo no tenga acceso directo al valor a predecir.

🔴 **Nota:** Esto es una práctica esencial en cualquier problema de aprendizaje supervisado.

◆ Celda 8: Definición del objeto de validación cruzada

python

```
kf = KFold(shuffle=True, random_state=72018, n_splits=3)
```

✓ Explicación:

Se crea un objeto de tipo **KFold** de **scikit-learn** para dividir el conjunto de datos en **k = 3 partes** (folds) con estas características:

- **shuffle=True**: las filas se barajan antes de hacer las divisiones. Esto previene que los folds tengan sesgo si el dataset está ordenado.
- **random_state=72018**: semilla para asegurar reproducibilidad del barajado.
- **n_splits=3**: se dividirá el dataset en 3 folds → en cada iteración, se usan 2 para entrenar y 1 para validar.

🔴 Este método es una **forma de validación cruzada (cross-validation)**, que permite evaluar la estabilidad y generalización del modelo al no depender de un solo conjunto de entrenamiento/prueba.

◆ Celda 9: Exploración de los índices generados por KFold

python

```
for train_index, test_index in kf.split(X):
    print("Train index:", train_index[:10], len(train_index))
    print("Test index:", test_index[:10], len(test_index))
    print('')
```

✓ Explicación:

- Se itera sobre los splits generados por `kf.split(X)`, que devuelve pares de índices: uno para entrenamiento y otro para prueba.
- Se imprimen:
 - Los primeros 10 índices de cada conjunto (entrenamiento y prueba).
 - El tamaño total de cada conjunto en cada fold.

Esta celda es **exploratoria** y ayuda a verificar visualmente cómo está funcionando la división de los datos.

✓ CONCLUSIÓN PARCIAL

Estas celdas constituyen una **etapa crítica en la validación del modelo**, estableciendo el flujo de trabajo de cross-validation:

1. Separamos claramente los datos (x, y).
2. Configuramos un esquema robusto de validación cruzada (**KFold**).
3. Verificamos que la división de los datos sea correcta y balanceada.

Esto asegura que las métricas de desempeño del modelo no estén sobreajustadas a una sola partición.

INFORMACIÓN ADICIONAL

◆ Validación cruzada (Cross-validation)

- Técnica estadística para evaluar la capacidad de generalización de un modelo.
- Divide el dataset en K bloques (folds), entrenando con K-1 y validando con 1 diferente en cada iteración.
- Al final, se promedian las métricas obtenidas en cada fold.

◆ Fórmulas asociadas

- Para cada fold i , se obtiene un error e_i , y luego el error final es:

$$\text{Error promedio} = \frac{1}{K} \sum_{i=1}^K e_i$$

- Reduce la varianza de la estimación del error comparado con una sola partición entrenamiento/test.

♦ Tipos de modelos a usar luego (adelanto)

- **LinearRegression, Ridge, Lasso** → son modelos de regresión con y sin regularización que se probarán bajo este esquema de validación cruzada.

¿Continuamos con las siguientes celdas?



continua

Vamos ahora con la explicación detallada de las **celdas 10 a 12** del notebook, donde se comienza a entrenar y validar el modelo de regresión lineal usando K-Fold y se evalúa el impacto de la estandarización.

♦ Celda 10: Regresión lineal sin escalar los datos

```
python

#from sklearn.metrics import r2_score, mean_squared_error

scores = []
lr = LinearRegression()

for train_index, test_index in kf.split(X):
    X_train, X_test, y_train, y_test = (X.iloc[train_index, :],
                                       X.iloc[test_index, :],
                                       y[train_index],
                                       y[test_index])

    lr.fit(X_train, y_train)

    y_pred = lr.predict(X_test)

    score = r2_score(y_test.values, y_pred)

    scores.append(score)

scores
```

✓ Explicación paso a paso:

1. Inicialización:

```
python

scores = []
lr = LinearRegression()
```

Se crea una lista para almacenar los puntajes R^2 de cada fold y se instancia un modelo de regresión lineal.

2. Validación cruzada:

Se itera sobre los 3 splits generados por **KFold**.

3. Separación entrenamiento/prueba:

```
python

X.iloc[train_index, :]
```

Se usan los índices generados por **KFold** para extraer subconjuntos de entrenamiento y prueba de las variables.

4. Entrenamiento del modelo:

```
python

lr.fit(X_train, y_train)
```

5. Predicción:

```
python

y_pred = lr.predict(X_test)
```

6. Evaluación con R^2 :

```
python

score = r2_score(y_test.values, y_pred)
```

R^2 mide la proporción de varianza explicada por el modelo:

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

7. Almacenamiento del score.

 Aquí no se aplicó **escalado** ni ningún preprocesamiento.

♦ Celda 11: Regresión lineal con escalado (StandardScaler)

```
python

scores = []

lr = LinearRegression()
s = StandardScaler()

for train_index, test_index in kf.split(X):
    X_train, X_test, y_train, y_test = (X.iloc[train_index, :],
                                       X.iloc[test_index, :],
                                       y[train_index],
                                       y[test_index])

    X_train_s = s.fit_transform(X_train)

    lr.fit(X_train_s, y_train)
```

```
X_test_s = s.transform(X_test)
y_pred = lr.predict(X_test_s)
score = r2_score(y_test.values, y_pred)
scores.append(score)
```

✓ Explicación:

- **StandardScaler** transforma cada variable a media 0 y desviación estándar 1:

$$z = \frac{x - \mu}{\sigma}$$

Esto es fundamental si se usan modelos sensibles a la escala (aunque la regresión lineal clásica no lo es tanto, sí lo son modelos con regularización como **Ridge** o **Lasso**).

- Importante: se ajusta (**fit_transform**) el **Scaler solo con los datos de entrenamiento** y luego se transforma el conjunto de prueba. Esto evita **data leakage**.

◆ Celda 12: Visualización de resultados

```
python

scores
```

Esta celda imprime la lista de puntajes R^2 obtenidos en cada fold. Es común evaluar luego el **promedio y desviación estándar**:

```
python

print(np.mean(scores), np.std(scores))
```

✓ CONCLUSIÓN PARCIAL

Estas celdas implementan correctamente un flujo de evaluación de un modelo lineal utilizando validación cruzada y permiten comparar:

- El rendimiento sin escalado vs. con escalado.
- La estabilidad del modelo (medida por la variación entre folds).

Se destaca el uso correcto de **KFold**, **StandardScaler**, **r2_score**, y la separación clara entre entrenamiento y prueba.

INFORMACIÓN ADICIONAL

◆ ¿Por qué escalar las variables?

- Variables con diferentes magnitudes pueden causar que unas dominen el ajuste.

- Modelos con regularización (como **Ridge** o **Lasso**) son muy sensibles a la escala.
- También afecta algoritmos como SVM, KNN, PCA, etc.

♦ Métrica R^2 :

- Valores cercanos a 1 indican buen ajuste.
- Puede ser negativo si el modelo es peor que predecir la media.

♦ Alternativas:

- También puede evaluarse con `mean_squared_error`, `MAE`, etc.

¿Seguimos con las próximas celdas?



continuar

Vamos ahora con la explicación detallada de las **celdas 13 a 15** del notebook. Estas celdas marcan la transición hacia el uso de un **pipeline**, que es una estructura fundamental en **scikit-learn** para organizar el flujo de trabajo de un modelo.

♦ Celda 13: Definición de los objetos del pipeline

```
python

s = StandardScaler()
lr = LinearRegression()
```

✓ Explicación:

- Se instancian los dos componentes que se van a encadenar en un pipeline:
 - **s**: un **StandardScaler**, que normaliza las variables (media 0, desviación estándar 1).
 - **lr**: un modelo de **LinearRegression**.

Esto prepara ambos pasos para ser integrados de forma automática en un flujo completo de procesamiento → entrenamiento → predicción.

♦ Celda 14: Creación del pipeline

```
python

estimator = Pipeline([
    ("scaler", s),
    ("regression", lr)
])
```

✓ Explicación:

Se construye un **Pipeline**, que es un objeto de **scikit-learn** que encadena pasos secuenciales donde:

- Cada paso (menos el último) debe tener un método `.fit()` y `.transform()`.
- El último paso (en este caso "regression") debe tener `.fit()` y `.predict()`.

En este caso:

- "scaler" → aplica **StandardScaler** sobre los datos.
- "regression" → entrena el modelo de regresión lineal sobre los datos ya escalados.

🔄 Esto permite que cuando se llama `estimator.fit(X_train, y_train)`, automáticamente se ejecuten:

1. `StandardScaler().fit_transform(X_train)`
2. `LinearRegression().fit(...)` con los datos escalados.

Luego, al llamar `estimator.predict(X_test)`, se hace:

1. `StandardScaler().transform(X_test)`
2. `LinearRegression().predict(...)`

◆ Celda 15: Visualización del objeto kf

```
python
kf
```

✓ Explicación:

Esta celda simplemente imprime el objeto **KFold** ya creado anteriormente. Al ejecutarla, se muestra su configuración:

```
scss
KFold(n_splits=3, random_state=72018, shuffle=True)
```

Esto confirma cómo se está estructurando la validación cruzada del modelo.

✓ CONCLUSIÓN PARCIAL

Estas tres celdas preparan una **implementación profesional y limpia** del modelo utilizando la clase **Pipeline**. Esto:

- Asegura que todo el preprocesamiento se realice correctamente en cada fold.
- Reduce errores y fugas de datos.
- Mejora la reproducibilidad y legibilidad del código.

Además, se reutiliza el objeto `kf` para continuar con la validación cruzada, pero esta vez usando el pipeline.

INFORMACIÓN ADICIONAL

♦ ¿Por qué usar Pipeline?

1. Evita fugas de datos (data leakage).
2. Encapsula la lógica en un solo objeto reutilizable.
3. Compatible con funciones de `scikit-learn` como:

- `cross_val_score`
- `cross_val_predict`
- `GridSearchCV`

♦ Alternativa más compacta:

```
python

from sklearn.pipeline import make_pipeline

estimator = make_pipeline(StandardScaler(), LinearRegression())
```

¿Querés que continúe con las siguientes celdas del notebook?



continua

Vamos ahora con la explicación de las **celdas 16 a 18** del notebook. Aquí se utiliza el **Pipeline** construido anteriormente para aplicar validación cruzada con predicción y se evalúa el rendimiento del modelo.

♦ Celda 16: Validación cruzada con predicción

```
python

predictions = cross_val_predict(estimator, X, y, cv=kf)
```

✓ Explicación:

- `cross_val_predict` es una función de `scikit-learn` que:
 - Entrena el **estimator** en cada fold (usando `kf` como esquema de validación).
 - Devuelve las **predicciones para cada ejemplo, como si fueran de test**.
 - Es decir, **cada fila se predice usando un modelo que no la vio en entrenamiento**.

En este caso:

- Se utiliza el **Pipeline (estimator)** que incluye escalado + regresión lineal.
- El resultado **predictions** contiene una lista de valores predichos para todas las muestras del conjunto original.

🔴 Esto permite calcular métricas de evaluación globales con confianza, ya que las predicciones son "fuera de muestra".

◆ Celda 17: Evaluación del rendimiento global (R^2)

```
python

r2_score(y, predictions)
```

✓ Explicación:

- Se calcula el **coeficiente de determinación** R^2 entre las predicciones generadas por validación cruzada (**cross_val_predict**) y los valores reales **y**.

Este valor representa **qué tan bien el modelo generaliza** a nuevas muestras:

- $R^2 \approx 1 \rightarrow$ ajuste casi perfecto.
- $R^2 = 0 \rightarrow$ el modelo no es mejor que predecir la media.
- $R^2 < 0 \rightarrow$ peor que un predictor constante.

◆ Celda 18: Promedio de los R^2 anteriores

```
python

np.mean(scores) # almost identical!
```

✓ Explicación:

Esta celda se refiere a la lista **scores** creada anteriormente (cuando se evaluó **LinearRegression** en cada fold manualmente, con escalado). Se toma el promedio de esos 3 valores de R^2 .

El comentario *"almost identical!"* indica que:

- El R^2 obtenido con **cross_val_predict** es prácticamente igual al promedio de R^2 obtenidos en la validación cruzada manual.
- Esto **valida la consistencia del uso del pipeline** y del proceso de evaluación.

✓ CONCLUSIÓN PARCIAL

Estas celdas realizan un paso clave en cualquier proyecto de Machine Learning: **evaluar la capacidad de generalización del modelo** usando validación cruzada con predicciones reales.

El uso de `cross_val_predict` y el cálculo de R^2 son herramientas robustas para medir el rendimiento sin necesidad de un conjunto de test explícito.

INFORMACIÓN ADICIONAL

♦ Diferencias: `cross_val_predict` vs `cross_val_score`

Función	Devuelve	Uso principal
<code>cross_val_score</code>	Puntajes de evaluación (R^2 , MSE, etc.)	Métrica agregada por fold
<code>cross_val_predict</code>	Predicciones	Comparar predicción vs verdad para cada muestra

♦ Ventaja de usar `cross_val_predict`

- Te permite construir gráficos como:

```
python

plt.scatter(y, predictions)
plt.plot([min(y), max(y)], [min(y), max(y)], 'k--')
```

Para **visualizar el ajuste real** del modelo.

¿Seguimos con las siguientes celdas del notebook?



continuar

Pasamos ahora a las **celdas 19 a 21**, donde se comienza a aplicar **regularización** usando el modelo **Lasso**. Esta técnica busca mejorar la generalización del modelo penalizando la complejidad.

♦ Celda 19: Generación de valores para el hiperparámetro `alpha`

```
python

alphas = np.geomspace(1e-9, 1e0, num=10)
alphas
```

✓ Explicación:

- Se genera un arreglo de 10 valores de `alpha` **espaciados geométricamente** entre 10^{-9} y $10^0 = 1$.
- Esto se hace con `np.geomspace()`, ideal para explorar múltiples órdenes de magnitud.

🔗 `alpha` es el **hiperparámetro de regularización** en Lasso:

- Si **alpha** es **cero**, Lasso se comporta como regresión lineal estándar.
- Si **alpha** es **alto**, más coeficientes serán reducidos a cero → el modelo será más simple (y a veces más robusto).

♦ Celda 20: Evaluación de modelos Lasso con validación cruzada

```
python

scores = []
coefs = []

for alpha in alphas:
    las = Lasso(alpha=alpha, max_iter=100000)

    estimator = Pipeline([
        ("scaler", s),
        ("lasso_regression", las)])

    predictions = cross_val_predict(estimator, X, y, cv = kf)

    score = r2_score(y, predictions)

    scores.append(score)
```

✓ Explicación detallada:

1. Inicialización:

- **scores**: para guardar el rendimiento (R^2) para cada valor de **alpha**.
- **coefs**: fue declarado pero no usado en esta celda; probablemente será utilizado más adelante para almacenar coeficientes.

2. Loop sobre alpha:

- Se instancia un modelo **Lasso** con ese valor de **alpha**.
- Se crea un **Pipeline** con **StandardScaler** + **Lasso**.

3. Evaluación con validación cruzada:

- Se generan las predicciones usando **cross_val_predict**.
- Se calcula el R^2 y se guarda.

🔗 Este ciclo permite encontrar el valor de **alpha** que ofrece **mejor capacidad predictiva** balanceando ajuste y simplicidad.

♦ Celda 21: Visualización de resultados

```
python

list(zip(alphas, scores))
```

✓ Explicación:

- Se imprime una lista de tuplas con cada par (**alpha**, R^2).

- Esto permite ver cómo cambia el rendimiento del modelo a medida que aumenta la penalización.

Idealmente, se busca el **mayor R^2** sin sobreajuste (no necesariamente el más cercano a 1, sino el más generalizable).

✓ CONCLUSIÓN PARCIAL

Estas celdas implementan correctamente una **búsqueda manual** de hiperparámetros para el modelo **Lasso**, usando:

- Un espacio logarítmico de valores de **alpha**.
- Validación cruzada con predicción fuera de muestra.
- Evaluación con R^2 .

Esto permite identificar el mejor valor de **alpha** para regularizar el modelo y prevenir overfitting.

📖 INFORMACIÓN ADICIONAL

♦ ¿Qué es Lasso?

- Acrónimo de **Least Absolute Shrinkage and Selection Operator**.
- Minimiza:

$$\min_{\beta} \left\{ \sum_{i=1}^n (y_i - X_i \cdot \beta)^2 + \alpha \sum_{j=1}^p |\beta_j| \right\}$$

- El término $\alpha \sum |\beta_j|$ es la penalización L1.
- Produce modelos más **parsimoniosos** al forzar muchos coeficientes a ser exactamente 0 → útil para selección de variables.

♦ Comparación con Ridge

Modelo	Penalización	Resultado típico
Ridge	$\sum \beta^2$	Coefficientes pequeños
Lasso	$\sum \beta $	β

¿Deseas que continúe con las siguientes celdas del notebook?



continuar

Vamos ahora con la explicación de las **celdas 22 a 24** del notebook, donde se analizan los coeficientes de modelos Lasso entrenados con diferentes valores de α y se visualiza el impacto de la regularización sobre el rendimiento.

◆ Celda 22: Coeficientes de Lasso con $\alpha = 10^{-6}$

python

```
Lasso(alpha=1e-6).fit(X, y).coef_
```

✓ Explicación:

- Se entrena un modelo **Lasso** con una **penalización muy baja**.
- Luego se imprimen los coeficientes del modelo entrenado.

💡 Como α es muy pequeño, la penalización es casi nula → los coeficientes serán similares a los de una regresión lineal ordinaria (OLS).

Esto permite observar:

- Qué variables tienen coeficientes altos (mayor importancia).
- Cuáles son despreciables o cercanos a 0.

◆ Celda 23: Coeficientes de Lasso con $\alpha = 1.0$

python

```
Lasso(alpha=1.0).fit(X, y).coef_
```

✓ Explicación:

- Aquí se aplica **una penalización fuerte**.
- El modelo Lasso forzará muchos coeficientes a cero → efectúa selección de variables.

Este contraste permite ver cómo **el valor de α afecta la complejidad del modelo**:

- Menor α : modelo complejo, muchos coeficientes.
- Mayor α : modelo simple, menos coeficientes.

Ideal para identificar un subconjunto reducido de predictores relevantes.

◆ Celda 24: Visualización del rendimiento R^2 vs. α

python

```
plt.figure(figsize=(10,6))
plt.semilogx(alphas, scores, '-o')
plt.xlabel('$\\alpha$')
plt.ylabel('$R^2$');
```

✓ Explicación:

- **semilogx** grafica **alphas** en escala logarítmica (eje x), lo cual es apropiado para valores que varían en órdenes de magnitud.
- Se visualiza cómo cambia R^2 con distintos valores de regularización.
- El gráfico permite identificar visualmente el valor óptimo de α : **el que maximiza el rendimiento sin sobreajuste**.

 Este tipo de análisis es fundamental en ajuste de hiperparámetros.

✓ CONCLUSIÓN PARCIAL

Estas celdas completan el análisis del modelo **Lasso**:

1. Se inspeccionan coeficientes para observar el efecto de la regularización.
2. Se visualiza cómo cambia el rendimiento R^2 con el hiperparámetro α .
3. Se proporciona una base para seleccionar el valor de α óptimo, balanceando **ajuste y simplicidad del modelo**.

INFORMACIÓN ADICIONAL

♦ ¿Qué esperar en los coeficientes?

α pequeño	α grande
Coeficientes grandes	Coeficientes cercanos a 0
Modelo sobreajustado	Modelo subajustado
Mayor varianza	Mayor sesgo

♦ Selección visual de hiperparámetro:

El punto máximo del gráfico R^2 vs α representa el mejor compromiso. También puede usarse **GridSearchCV** para automatizar este proceso.

¿Querés que continúe con las siguientes celdas del notebook?



continuar

Vamos ahora con las **celdas 25 a 27**, donde se introduce un nuevo componente: **características polinómicas** (no lineales), y se ajusta un modelo **Lasso** más complejo. También se selecciona el mejor modelo final.

♦ Celda 25: Lasso con características polinómicas de segundo grado

```
python

pf = PolynomialFeatures(degree=2)
scores = []

alphas = np.geomspace(0.001, 10, 5)

for alpha in alphas:
    las = Lasso(alpha=alpha, max_iter=100000)
    estimator = Pipeline([
        ("make_higher_degree", pf),
        ("scaler", s),
        ("lasso_regression", las)])

    predictions = cross_val_predict(estimator, X, y, cv=kf)
    score = r2_score(y, predictions)
    scores.append(score)

scores
```

✓ Explicación paso a paso:

1. Generación de características no lineales:

- `PolynomialFeatures(degree=2)` expande el espacio de variables incluyendo:
 - Todos los términos lineales x_i
 - Todos los cuadrados x_i^2
 - Todos los productos cruzados $x_i x_j$
- Si había n variables, ahora habrá $\frac{n(n+1)}{2} + n + 1$ columnas.

2. Búsqueda de hiperparámetro `alpha`:

- Se prueban 5 valores logarítmicamente espaciados entre 0.001 y 10.
- En cada ciclo:
 - Se crea un pipeline:
 1. Se generan las variables polinómicas.
 2. Se escalan.
 3. Se entrena un modelo **Lasso**.
 - Se evalúa con `cross_val_predict` y `r2_score`.

✓ Esto permite explorar cómo afecta la combinación de **no linealidad + regularización** al rendimiento del modelo.

◆ Celda 26: Visualización del rendimiento R^2

```
python

plt.semilogx(alphas, scores);
```

✓ Explicación:

- Se grafica el rendimiento R^2 frente al valor de α en escala logarítmica.
- Este gráfico ayuda a elegir el valor de α que:
 - Mejora el ajuste con las variables polinómicas.
 - Controla el sobreajuste mediante la penalización de **Lasso**.

🔥 Es una herramienta visual de selección de hiperparámetros.

◆ Celda 27: Entrenamiento final del mejor modelo

```
python

# Once we have found the hyperparameter (alpha~1e-2=0.01)
# make the model and train it on ALL the data
# Then release it into the wild .....

best_estimator = Pipeline([
    ("make_higher_degree", PolynomialFeatures(degree=2)),
    ("scaler", s),
    ("lasso_regression", Lasso(alpha=0.01, max_iter=10000))]

best_estimator.fit(X, y)
best_estimator.score(X, y)
```

✓ Explicación:

1. Selección del mejor α :

- A partir del gráfico anterior, se elige $\alpha = 0.01$ como valor óptimo.

2. Entrenamiento final:

- Se entrena el modelo con TODO el conjunto de datos (x, y) ya que no se necesita dividir más (el modelo está afinado).
- Se usa **Pipeline** con:
 - Características polinómicas.
 - Escalado.
 - Lasso con **alpha=0.01**.

3. Evaluación en el conjunto completo:

- Se imprime el R^2 del modelo final sobre los datos completos → para evaluar qué tanto explica la variabilidad de los precios.

🔥 Esto representa el **modelo final que se podría "liberar al mundo"**.

✓ CONCLUSIÓN PARCIAL

Estas celdas completan el flujo profesional de ajuste de modelos con regresión regularizada:

1. **Se introducen términos no lineales** para capturar relaciones más complejas.
2. **Se prueba Lasso con distintos α** para evitar sobreajuste.

3. **Se elige el mejor modelo** y se entrena sobre todos los datos disponibles.

Este enfoque permite obtener un modelo con **alto poder predictivo y bajo riesgo de sobreajuste**, incluso con muchas variables derivadas.

INFORMACIÓN ADICIONAL

♦ ¿Por qué usar `PolynomialFeatures`?

- Permite a modelos lineales capturar **relaciones no lineales** entre variables.
- Por ejemplo: relaciones cuadráticas o interacciones entre variables.

♦ Precaución con `PolynomialFeatures`

- Genera muchas más variables → riesgo de sobreajuste.
- Por eso se combina con **regularización** (**Lasso** en este caso) que descarta variables irrelevantes.

♦ Mejor práctica: Validación final en conjunto hold-out

- Para máxima confianza, luego de la validación cruzada se suele guardar un conjunto de prueba final que **no haya sido tocado** durante todo el pipeline.

¿Querés que revise si quedan más celdas y continúe con la explicación?



continuar

Gracias por tu paciencia. Vamos a cerrar este análisis profesional explicando las **últimas 15 celdas de código** del notebook. Este bloque final contiene técnicas avanzadas como análisis de coeficientes, búsqueda con `GridSearchCV`, transformación **Box-Cox**, y evaluación comparativa entre Lasso y Ridge con expansión polinómica.

♦ Celda 28: Coeficientes del mejor modelo Lasso

```
python

best_estimator.named_steps["lasso_regression"].coef_
```

Se accede directamente a los coeficientes del modelo Lasso entrenado con el mejor **alpha** tras aplicar `PolynomialFeatures` y `StandardScaler`. Muestra la importancia (positiva o negativa) de cada variable transformada.

♦ Celda 29: Evaluación de Ridge con características polinómicas

python

```
pf = PolynomialFeatures(degree=2)
alphas = np.geomspace(4, 20, 20)
# ...
```

Loop que evalúa distintos valores de **alpha** para **Ridge** usando validación cruzada con polinomios de grado 2. Se calcula y grafica el R^2 para cada uno. Ridge penaliza con L2: tiende a **reducir coeficientes sin forzarlos a cero**, a diferencia de Lasso.

◆ Celda 30: Entrenamiento del mejor modelo Lasso final

python

```
best_estimator = Pipeline([
    ("make_higher_degree", PolynomialFeatures(degree=2, include_bias=False)),
    ("scaler", s),
    ("lasso_regression", Lasso(alpha=0.01, max_iter=10000))
])
best_estimator.fit(X, y)
best_estimator.score(X, y)
```

Es el mismo pipeline del paso anterior, pero con **include_bias=False** y entrenado sobre todo el dataset. Se obtiene el R^2 final del modelo.

◆ Celda 31: DataFrame con nombres de variables y coeficientes

python

```
df_importances =
pd.DataFrame(zip(best_estimator.named_steps["make_higher_degree"].get_feature_names(),
best_estimator.named_steps["lasso_regression"].coef_))
```

Se crea un **DataFrame** que empareja el nombre de cada variable polinómica con su coeficiente estimado. Muy útil para interpretación del modelo.

◆ Celda 32-34: Diccionario de nombres y ordenamiento de importancia

python

```
col_names_dict = dict(zip(list(range(len(X.columns.values))), X.columns.values))
df_importances.sort_values(by=1)
```

Se genera un diccionario de nombres originales y luego se ordena el **DataFrame** de coeficientes por valor → de menor a mayor importancia (negativo a positivo).

◆ Celda 35-37: Búsqueda de hiperparámetros con GridSearchCV

```
python

from sklearn.model_selection import GridSearchCV
# ...
grid = GridSearchCV(estimator, params, cv=kf)
grid.fit(X, y)
grid.best_score_, grid.best_params_
```

Se realiza búsqueda de hiperparámetros con:

- **PolynomialFeatures** de grados 1, 2 y 3.
- **Ridge** con **alpha** de 4 a 20.

GridSearchCV evalúa todas las combinaciones y encuentra la mejor. Devuelve el mejor score medio y los parámetros óptimos.

◆ Celda 38: Transformación Box-Cox y comparación

```
python

bc_result = boxcox(y_train)
y_train_bc = bc_result[0]
lam = bc_result[1]
# ...
y_pred_tran = inv_boxcox(y_pred_bc, lam)
r2_score(y_pred_tran, y_test)
```

- Se aplica **Box-Cox** a **y_train** para hacer su distribución más normal (esto mejora la linealidad).
- Luego se entrena un modelo **LinearRegression** sobre los datos transformados.
- Se invierte la transformación y se evalúa el R^2 .
- Se compara contra el modelo sin transformación.

🔴 Resultado: el modelo sin Box-Cox tiene un mejor R^2 (≈ 0.867 vs. 0.848).

◆ Celda 39-41: Uso del mejor modelo encontrado por GridSearchCV

```
python

y_predict = grid.predict(X)
r2_score(y, y_predict)
grid.best_estimator_.named_steps['ridge_regression'].coef_
```

- Se usa el mejor estimador (**best_estimator_**) hallado por **GridSearchCV**.
- Se predice con él y se calcula su R^2 .
- También se accede a sus coeficientes.

◆ Celda 42: Resultados detallados del grid


```
python
grid.cv_results_
```

Se accede a todos los resultados del **GridSearchCV**, incluyendo puntuaciones, tiempos y combinaciones evaluadas. Muy útil para visualizaciones tipo heatmap o análisis comparativo de modelos.

✓ CONCLUSIÓN GENERAL DEL NOTEBOOK

Este notebook de IBM aplica un flujo **completo, robusto y profesional de validación cruzada y ajuste de modelos de regresión**, usando tanto **Lasso** como **Ridge**, e incluye:

1. **Carga y exploración** del dataset Boston Housing.
2. **Preprocesamiento** con escalado y expansión polinómica.
3. **Validación cruzada (KFold)** con `cross_val_predict` y cálculo de R^2 .
4. **Comparación de regresión lineal vs. Lasso vs. Ridge** con visualización de hiperparámetros.
5. **Selección del mejor modelo** con `GridSearchCV`.
6. **Evaluación final** del modelo y análisis de coeficientes.
7. **Uso de transformaciones como Box-Cox** para evaluar efectos de normalización en la variable objetivo.

TEORÍA Y CONCEPTOS ADICIONALES

◆ Regularización

- **Lasso:** Penaliza con $\sum |\beta_j|$, útil para selección de variables (sparse).
- **Ridge:** Penaliza con $\sum \beta_j^2$, útil para multicolinealidad.

◆ PolynomialFeatures

- Aumenta la capacidad del modelo para capturar relaciones no lineales.
- Rápidamente aumenta la dimensionalidad → se recomienda combinar con regularización.

◆ Box-Cox

- Transforma la variable objetivo para aproximar la normalidad.
- Mejora linealidad y puede aumentar la performance de modelos lineales.

◆ GridSearchCV

- Herramienta de búsqueda sistemática de combinaciones de hiperparámetros.
- Evalúa cada combinación con validación cruzada.
- Retorna el mejor modelo encontrado (`best_estimator_`).

✓ FINALIZADO: NO HAY MÁS CELDAS DE CÓDIGO PARA EXPLICAR

¿Querés que te prepare un resumen ejecutivo del proyecto, un informe en markdown o alguna visualización extra?



necesito un resumen solo de teoria aplicada de Machine Learning

Claro. A continuación, te presento un **resumen de la teoría de Machine Learning aplicada en este notebook**, enfocado en los fundamentos matemáticos, conceptuales y prácticos involucrados:

RESUMEN TEÓRICO DE MACHINE LEARNING APLICADO

1. Regresión Lineal Múltiple

Modelo base que asume una relación lineal entre las variables predictoras X y la variable objetivo y :

$$y = X\beta + \varepsilon$$

- Se entrena minimizando el **error cuadrático medio (MSE)**.
- No incluye ningún tipo de regularización.

2. Validación Cruzada (Cross-Validation)

► K-Fold Cross Validation

- Divide el dataset en K partes (folds).
- Entrena el modelo K veces, cada vez dejando uno de los folds como conjunto de validación.
- Evita el **sobreajuste a un solo split** y provee una **estimación más estable del rendimiento**.

$$\text{Error Promedio} = \frac{1}{K} \sum_{i=1}^K e_i$$

► `cross_val_predict`

Genera predicciones "fuera de muestra" para cada observación, útil para calcular métricas globales como R^2 .

3. Regularización

Agrega una penalización al modelo para **evitar sobreajuste** y mejorar la capacidad de generalización.

► Lasso (L1)

$$\min_{\beta} \left\{ \sum (y_i - \hat{y}_i)^2 + \alpha \sum |\beta_j| \right\}$$

- Penaliza la suma de valores absolutos de los coeficientes.
- **Fuerza coeficientes a cero** → útil para selección de variables.
- Controlado por hiperparámetro α .

► Ridge (L2)

$$\min_{\beta} \left\{ \sum (y_i - \hat{y}_i)^2 + \alpha \sum \beta_j^2 \right\}$$

- Penaliza la suma de los cuadrados de los coeficientes.
- **Reduce la magnitud de los coeficientes** pero no los elimina.

4. Pipeline de Modelado

Permite encadenar pasos de preprocesamiento y modelado, como:

1. Escalado con `StandardScaler`
2. Expansión con `PolynomialFeatures`
3. Modelos (`LinearRegression`, `Lasso`, `Ridge`)

Evita errores como data leakage y mejora la reproducibilidad.

5. Polynomial Features

Permite modelar relaciones no lineales entre las variables:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x_1 x_2 + \dots$$

- Aumenta la dimensionalidad del dataset.
- Requiere regularización para evitar sobreajuste.

6. Búsqueda de Hiperparámetros

► Manual (con bucle sobre `alpha`)

- Se evalúan varios valores de α y se grafica R^2 .

► Automática: `GridSearchCV`

- Prueba todas las combinaciones de hiperparámetros especificados.
- Usa validación cruzada interna para cada combinación.
- Retorna el mejor modelo (`best_estimator_`).

7. Transformación Box-Cox

Aplicada a la variable objetivo para que siga una distribución más normal:

$$y^{(\lambda)} = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \text{si } \lambda \neq 0 \\ \ln y & \text{si } \lambda = 0 \end{cases}$$

- Mejora la linealidad y el rendimiento en modelos que asumen distribución normal (como regresión lineal).
- Invertible con `inv_boxcox`.

8. Evaluación del Modelo: R² Score

Métrica principal usada:

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

- Mide proporción de varianza explicada.
- $R^2 = 1 \rightarrow$ ajuste perfecto.
- $R^2 = 0 \rightarrow$ modelo no mejor que la media.
- $R^2 < 0 \rightarrow$ peor que la media.

Conclusión del marco teórico aplicado

Este notebook representa una implementación profesional de:

- Regresión lineal y regularizada (Lasso, Ridge)
- Expansión polinómica para no linealidades
- Validación cruzada robusta
- Selección y evaluación de hiperparámetros
- Interpretación de coeficientes
- Comparación de transformaciones para mejorar ajustes

Todo ello integrado en un pipeline reproducible y evaluado con rigor estadístico.

¿Te gustaría este mismo resumen en PDF, Markdown, o como sección para un informe?