

# Librerías y Modelos de Regresión



caracter-sticas-polin-micas-regularizaci-n-ibm-ml.ipynb  
File

Gracias por subir el notebook titulado "**características polinómicas y regularización**". Procederé a analizar el contenido celda por celda, explicando detalladamente cada bloque de código, su funcionamiento y fundamentos teóricos. Al final, haré una conclusión completa sobre el flujo del notebook, seguida de un desarrollo adicional con conceptos teóricos relacionados.

Comienzo con la primera celda de código. Un momento mientras la analizo...

## 📌 Celda 1: Importación de librerías

python

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
```

## 🔍 Explicación Detallada

### 1. numpy (np):

- Es la librería estándar de Python para operaciones numéricas con arrays.
- Se usa principalmente para manejar vectores, matrices y funciones matemáticas complejas (e.g., `np.linspace`, `np.array`, etc.).

### 2. matplotlib.pyplot (plt):

- Librería para visualización. Permite crear gráficos, scatter plots, líneas, histogramas, etc.
- Se usa aquí para visualizar tanto los datos como los modelos ajustados.

### 3. sklearn.linear\_model:

- **LinearRegression**: modelo de regresión lineal simple, sin regularización.
- **Ridge**: regresión lineal con regularización L2 (penaliza los coeficientes grandes).
- **Lasso**: regresión lineal con regularización L1 (fuerza algunos coeficientes a cero → selección de variables).

### 4. PolynomialFeatures:

- Se utiliza para transformar un conjunto de características en su versión polinómica, es decir, se generan nuevas características combinando las originales en potencias y productos cruzados.

- Ejemplo: si tienes una variable  $x$ , podrías generar  $x$ ,  $x^2$ ,  $x^3$ , etc.

#### 5. `make_pipeline`:

- Permite encadenar múltiples transformaciones y un modelo en una sola estructura.
- Por ejemplo, se puede crear un pipeline que primero transforme las variables con `PolynomialFeatures` y luego aplique `LinearRegression`.

## Fundamento Matemático

### • Regresión Lineal:

Modelo de la forma:

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

Se ajusta minimizando el error cuadrático medio (MSE).

### • Regresión Polinómica:

- Variante de la regresión lineal donde se agregan potencias de las variables independientes.
- Si  $x$  es la variable original, en grado 3 se tendría:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \varepsilon$$

### • Regularización (Ridge y Lasso):

- Se agregan términos de penalización al MSE:

#### • Ridge:

$$\text{MSE} + \alpha \sum_j \beta_j^2$$

#### • Lasso:

$$\text{MSE} + \alpha \sum_j |\beta_j|$$

- Donde  $\alpha$  controla la fuerza de la penalización.

¿Pasamos a la siguiente celda de código?



si

Perfecto, continuamos con la siguiente celda del notebook:

## Celda 2: Generación de los datos

python

```
np.random.seed(0)
n = 15
x = np.sort(np.random.rand(n))
y = np.sin(2 * np.pi * x) + np.random.normal(0, 0.1, n)
```

## Explicación Detallada

### 1. `np.random.seed(0)`:

- Fija la semilla del generador de números aleatorios de NumPy para asegurar que los resultados sean reproducibles.
- Esto significa que cada vez que ejecutes este código, obtendrás los mismos valores aleatorios.

### 2. `n = 15`:

- Se define que se generarán 15 puntos de datos.

### 3. `x = np.sort(np.random.rand(n))`:

- Se generan `n = 15` números aleatorios entre 0 y 1 con `np.random.rand(n)`.
- Luego se ordenan de menor a mayor con `np.sort(...)`.
- Esto facilita la visualización posterior en gráficas.

### 4. `y = np.sin(2 * np.pi * x) + np.random.normal(0, 0.1, n)`:

- Genera los valores de salida `y` a partir de una función seno:

$$y = \sin(2\pi x) + \text{ruido}$$

- El ruido se genera con `np.random.normal(0, 0.1, n)` → ruido gaussiano con media 0 y desviación estándar 0.1.

## Fundamentos Teóricos

### • Simulación de datos:

- Se está generando un conjunto de datos sintéticos basados en una función conocida ( $\sin(2\pi x)$ ), con la intención de ver cómo distintos modelos (lineal, polinómico, regularizado) pueden ajustarse a datos no lineales y ruidosos.

### • Modelo subyacente:

- La función seno es periódica y no lineal, por lo que una regresión lineal simple tendrá dificultades para capturar su comportamiento sin transformaciones.

### • Ruido Gaussiano:

- Se agrega para simular errores o incertidumbre en las mediciones reales. Este es un supuesto clásico en modelos estadísticos: que los errores son distribuidos normalmente.

¿Avanzamos a la tercera celda de código?



## CONTINUA

Perfecto. Vamos con la tercera celda del notebook:

### Celda 3: Visualización de los datos generados

```
python

plt.figure(figsize=(8, 4))
plt.plot(x, y, 'o', label='Datos')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Datos generados con ruido')
plt.legend()
plt.show()
```

### Explicación Detallada

#### 1. `plt.figure(figsize=(8, 4))`:

- Crea una nueva figura de 8 unidades de ancho por 4 de alto (en pulgadas, por defecto).
- Sirve para personalizar el tamaño del gráfico.

#### 2. `plt.plot(x, y, 'o', label='Datos')`:

- Dibuja los puntos (x, y) como círculos 'o'.
- No conecta los puntos con líneas, por lo que solo se visualiza la dispersión de los datos.
- Se le asigna una etiqueta "Datos" que aparecerá en la leyenda.

#### 3. `plt.xlabel('x'), plt.ylabel('y')`:

- Etiquetas para los ejes X e Y.

#### 4. `plt.title('Datos generados con ruido')`:

- Agrega un título al gráfico.

#### 5. `plt.legend()`:

- Muestra la leyenda con la etiqueta proporcionada anteriormente.

#### 6. `plt.show()`:

- Muestra el gráfico en pantalla.



### Resultado esperado

Se visualiza una nube de puntos distribuidos de forma ondulatoria, como una senoide perturbada por ruido, lo que representa los datos que el modelo intentará ajustar.



### Fundamento gráfico

Este paso es fundamental en cualquier análisis de datos o aprendizaje automático:

- Permite inspeccionar visualmente la relación entre las variables.
- Ayuda a detectar no linealidades (como en este caso).
- Nos sugiere la necesidad de usar modelos más complejos (como regresión polinómica o regularización) en lugar de una regresión lineal simple.

¿Paso a la siguiente celda donde se ajusta el primer modelo?



continua

Perfecto, ahora pasamos a la siguiente celda donde se comienza con el **ajuste de modelos polinómicos**. Aquí se entrena una serie de modelos con distintos grados polinómicos.

## Celda 4: Ajuste de modelos polinómicos y visualización

python

```
X = x[:, np.newaxis]
y = y[:, np.newaxis]

for degree in [1, 4, 15]:
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    model.fit(X, y)
    y_pred = model.predict(X)

    plt.plot(x, y_pred, label=f'Degree {degree}')

plt.plot(x, y, 'o', label='Datos')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Ajustes con diferentes grados polinómicos')
plt.legend()
plt.show()
```

## Explicación detallada por bloques

### ◆ 1. Preparación de datos

python

```
X = x[:, np.newaxis]
y = y[:, np.newaxis]
```

- Convierte los arrays unidimensionales **x** e **y** en matrices columna 2D (**shape (n,1)**), como espera **scikit-learn**.
- **np.newaxis** agrega una nueva dimensión para que se interprete como una matriz de entrenamiento en regresión supervisada.

## ♦ 2. Entrenamiento de modelos con distintos grados

```
python

for degree in [1, 4, 15]:
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    model.fit(X, y)
    y_pred = model.predict(X)
```

- Se recorre una lista de grados polinómicos: 1 (lineal), 4 (mediano), y 15 (muy complejo).
- `make_pipeline(...)`: crea una cadena donde:
  1. `PolynomialFeatures(degree)`: transforma las entradas `X` en sus potencias hasta el grado deseado.
  2. `LinearRegression()`: ajusta un modelo lineal sobre esas nuevas características.
- `model.fit(...)`: entrena el modelo con los datos.
- `model.predict(X)`: genera predicciones para graficarlas.

## ♦ 3. Visualización de los modelos ajustados

```
python

plt.plot(x, y_pred, label=f'Degree {degree}')
```

- Dibuja la curva ajustada por el modelo para cada grado de polinomio.

## ♦ 4. Visualización de los datos reales

```
python

plt.plot(x, y, 'o', label='Datos')
```

- Se superponen los datos reales como puntos para comparar visualmente qué tan bien se ajusta cada modelo.



## Resultado esperado

- Grado 1: línea recta → **subajuste (underfitting)**.
- Grado 4: curva suave que sigue el patrón → **buen ajuste**.
- Grado 15: curva muy oscilante que pasa por todos los puntos → **sobreajuste (overfitting)**.



## Teoría asociada

### Regresión polinómica:

- Aumentar el grado polinómico puede capturar relaciones no lineales, pero **exagera el ajuste a datos con ruido** si se abusa del grado → esto es **overfitting**.

## Underfitting vs Overfitting:

- **Underfitting:** modelo demasiado simple para los datos → alto sesgo.
- **Overfitting:** modelo demasiado complejo que se ajusta incluso al ruido → alta varianza.

¿Quieres que avancemos a la siguiente celda donde se aplica **regularización** para controlar el sobreajuste?



continúa

Perfecto, avanzamos a la siguiente celda del notebook, donde se introducen **técnicas de regularización** para mitigar el sobreajuste observado con polinomios de alto grado.

## Celda 5: Aplicación de Ridge y Lasso sobre modelo polinómico de grado 15

```
python

model_ridge = make_pipeline(PolynomialFeatures(15), Ridge(alpha=1e-3))
model_lasso = make_pipeline(PolynomialFeatures(15), Lasso(alpha=1e-3, max_iter=10000))

model_ridge.fit(X, y)
model_lasso.fit(X, y)

plt.plot(x, model_ridge.predict(X), label='Ridge')
plt.plot(x, model_lasso.predict(X), label='Lasso')
plt.plot(x, y, 'o', label='Datos')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Regularización: Ridge y Lasso (grado 15)')
plt.legend()
plt.show()
```

## Explicación detallada

### ♦ 1. Creación de modelos con regularización

```
python

model_ridge = make_pipeline(PolynomialFeatures(15), Ridge(alpha=1e-3))
model_lasso = make_pipeline(PolynomialFeatures(15), Lasso(alpha=1e-3, max_iter=10000))
```

- Se aplican tanto **Ridge** como **Lasso** sobre un modelo polinómico de grado 15.
- **alpha=1e-3:** es el parámetro de regularización; un valor pequeño que permite una penalización leve.
- **max\_iter=10000:** necesario en Lasso para garantizar convergencia (puede necesitar más iteraciones que Ridge).

## ♦ 2. Entrenamiento

```
python

model_ridge.fit(X, y)
model_lasso.fit(X, y)
```

- Se ajustan los modelos a los datos. Ambos intentan **limitar los coeficientes polinómicos extremos** inducidos por el grado 15.

## ♦ 3. Visualización

```
python

plt.plot(x, model_ridge.predict(X), label='Ridge')
plt.plot(x, model_lasso.predict(X), label='Lasso')
```

- Se dibujan las curvas predichas por ambos modelos.
- Se espera que ambas curvas sean más suaves y menos oscilantes que la del modelo polinómico sin regularización.

## Fundamento teórico

### Ridge (regularización L2):

- Penaliza la **suma de los cuadrados de los coeficientes**.
- Reduce la magnitud de todos los coeficientes pero no los lleva exactamente a cero.
- Tiende a **suavizar** la curva.

### Lasso (regularización L1):

- Penaliza la **suma de los valores absolutos de los coeficientes**.
- Tiene la propiedad de llevar algunos coeficientes a **cero exacto**, lo que también sirve como selección de variables.
- Tiende a generar modelos más **esparsos**.

## Resultado esperado

- Ambas curvas deben seguir la tendencia de los datos, pero sin las oscilaciones extremas del modelo polinómico de grado 15 sin regularización.
- La **curva Ridge** suele ser más suave.
- La **curva Lasso** puede parecer más segmentada si elimina varios términos.

¿Pasamos a la última celda del notebook o deseas detenerte a revisar algo de esta parte?





continúa

Perfecto. Ahora pasamos a la **última celda de código** del notebook, donde se visualiza el efecto de la regularización modificando el parámetro **alpha** en Ridge.

## 📌 Celda 6: Exploración del efecto de alpha en Ridge

```
python

alphas = [1e-5, 1e-3, 1e-1, 1, 10]
plt.figure(figsize=(10, 5))

for alpha in alphas:
    model = make_pipeline(PolynomialFeatures(15), Ridge(alpha=alpha))
    model.fit(X, y)
    y_pred = model.predict(X)
    plt.plot(x, y_pred, label=f'alpha={alpha}')

plt.plot(x, y, 'o', label='Datos')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Efecto del parámetro alpha en Ridge')
plt.legend()
plt.show()
```

## 🔍 Explicación Detallada

### ♦ 1. Lista de valores de alpha

```
python

alphas = [1e-5, 1e-3, 1e-1, 1, 10]
```

- **alpha** controla la **intensidad de la regularización**:
  - Valores **pequeños** → regularización débil → curva puede sobreajustarse.
  - Valores **grandes** → regularización fuerte → curva se suaviza demasiado.

### ♦ 2. Ciclo de entrenamiento y predicción

```
python

for alpha in alphas:
    model = make_pipeline(PolynomialFeatures(15), Ridge(alpha=alpha))
    model.fit(X, y)
    y_pred = model.predict(X)
    plt.plot(x, y_pred, label=f'alpha={alpha}')
```

- Se ajusta un modelo Ridge para cada valor de alpha.
- Se predicen valores de salida con el modelo ajustado.

- Se grafica cada curva resultante para comparar visualmente.

### ♦ 3. Visualización final

```
python

plt.plot(x, y, 'o', label='Datos')
```

- Se superponen los datos reales como referencia.
- Se observa cómo cambia el modelo Ridge en función de **alpha**.

## Teoría Complementaria

### Regularización y complejidad del modelo:

- A medida que **alpha** aumenta:
  - Los coeficientes de los términos polinómicos se acercan a cero.
  - La curva se aplanan → menor varianza, pero potencialmente más sesgo.
- Con **alpha** muy pequeño:
  - El modelo se comporta como una regresión polinómica estándar (sin regularización).
- Con **alpha** muy grande:
  - El modelo se aproxima a una regresión casi constante (subajuste).

## Resultado Esperado

- Para **alpha** = **1e-5**: curva muy ajustada (parecida a overfitting).
- Para **alpha** = **1e-3**: aún flexible pero más suave.
- Para **alpha** = **0.1** o **1**: curvas más regulares.
- Para **alpha** = **10**: curva casi recta o muy simplificada → posible underfitting.

## CONCLUSIÓN DEL NOTEBOOK

Este notebook demuestra paso a paso cómo:

1. Se **generan datos no lineales** con ruido.
2. Se prueba una **regresión polinómica** con distintos grados para mostrar:
  - Cómo el aumento de complejidad puede llevar al sobreajuste.
3. Se aplican **técnicas de regularización (Ridge y Lasso)** para suavizar los modelos.
4. Se estudia el **impacto del hiperparámetro alpha** sobre la forma del modelo Ridge.

💡 El mensaje clave: **modelos complejos requieren regularización para generalizar bien**. Hay que encontrar el equilibrio correcto entre sesgo y varianza.

## 📁 EXPLICACIÓN ADICIONAL Y TEORÍA RELACIONADA

### 📌 1. Regularización: El Dilema de Bias-Varianza

- **Bias** (sesgo): error debido a suposiciones demasiado rígidas del modelo.
- **Varianza**: sensibilidad del modelo a pequeñas variaciones en los datos.
- La regularización ayuda a encontrar el punto medio óptimo.

### 📌 2. Comparación entre Ridge y Lasso

Aspecto	Ridge	Lasso
Penalización	L2 (cuadrados)	L1 (valores absolutos)
Coefficientes 0	No (todos $\neq 0$ )	Algunos pueden ser 0
Interpretación	Todos los predictores	Selección de variables

### 📌 3. Pipeline en Scikit-Learn

- Es una herramienta clave para **automatizar flujos de trabajo** de Machine Learning.
- Permite aplicar transformaciones (como polinomios) seguidas de un modelo de forma organizada y reproducible.

### 📌 4. PolynomialFeatures

- Para  $x = [x_1]$  y grado 3, se obtienen:

$$[1, x_1, x_1^2, x_1^3]$$

- En múltiples dimensiones se incluyen también términos cruzados como  $x_1x_2, x_1^2x_2$ , etc.

✅ **AVISO: YA NO HAY MÁS CELDAS DE CÓDIGO PARA EXPLICAR.**

¿Te gustaría que convirtamos este notebook en un informe PDF con los gráficos y explicaciones? ¿O deseas practicar aplicando esto a tus propios datos?