

# Informe:

## Práctica 6: Implementación de una canalización de aprendizaje automático con Amazon SageMaker

Jordi Blasco Lozano

Infraestructuras y Servicios Cloud

Universidad de Alicante

11 de diciembre de 2025

### Resumen

En esta práctica hemos implementado un pipeline completo de Machine Learning con Amazon SageMaker para detectar anomalías en la columna vertebral de pacientes. Comenzamos explorando el dataset, luego entrenamos un modelo XGBoost, lo desplegamos y evaluamos con diferentes umbrales de clasificación. Finalmente, experimentamos con ajuste automático de hiperparámetros para intentar mejorar el rendimiento. A través de todos estos pasos vimos cómo funcionan los servicios de AWS para ML desde cero hasta tener un modelo en producción.

## Índice

<b>1. Exploración de Datos (Lab 3.2)</b>	<b>3</b>
1.1. El Dataset . . . . .	3
1.2. Análisis Exploratorio (EDA) . . . . .	3
1.3. Tareas de desafíos . . . . .	3
<b>2. Codificación de Datos Categóricos (Lab 3.3)</b>	<b>3</b>
2.1. Qué Hicimos . . . . .	3
2.2. Dos Técnicas de Codificación . . . . .	3
2.3. Desafío: Añadir Más Variables . . . . .	4
<b>3. División de Datos y Entrenamiento (Lab 3.4)</b>	<b>5</b>
3.1. Preparación del Dataset . . . . .	5
3.2. División Estratificada . . . . .	5
3.3. Entrenamiento con SageMaker . . . . .	5
<b>4. Implementación y Predicciones (Lab 3.5)</b>	<b>6</b>
4.1. Deploy del Modelo . . . . .	6
4.2. Haciendo Predicciones . . . . .	6
4.3. Batch Transform . . . . .	6
4.4. Desafío: Experimentar con Umbrales . . . . .	6
<b>5. Evaluación del Modelo (Lab 3.6)</b>	<b>7</b>
5.1. La Matriz de Confusión . . . . .	7
5.2. Métricas Principales . . . . .	7
5.3. Desafío: Comparar Umbrales . . . . .	7
5.4. Curva ROC y AUC . . . . .	7

<b>6. Ajuste de Hiperparámetros (Lab 3.7)</b>	<b>8</b>
6.1. Qué Hicimos . . . . .	8
6.2. Configuración del Tuner . . . . .	8
6.3. Ejecución . . . . .	8
6.4. Desafío: Análisis de Resultados . . . . .	8
<b>7. Conclusiones</b>	<b>9</b>

# 1 Exploración de Datos (Lab 3.2)

## 1.1 El Dataset

Para esta práctica utilizamos el dataset `column_2C_weka.arff` del repositorio UCI, que contiene información sobre pacientes con columna vertebral. Tiene 310 instancias (100 normales y 210 con anomalías) y 6 atributos biomecánicos:

- Incidencia pélvica
- Inclinación pélvica
- Ángulo de lordosis lumbar
- Inclinación del sacro
- Radio pélvico
- Grado de espondilolistesis

El objetivo era clasificar a los pacientes en dos categorías: **Normal** o **Abnormal**.

## 1.2 Análisis Exploratorio (EDA)

Lo primero que hicimos fue cargar el dataset y explorar su estructura usando `shape`, `describe()` y `info()`. Luego generamos gráficos de densidad (KDE), histogramas y box plots para entender cómo se distribuían los datos.

La característica `degree_spondylolisthesis` tenía valores atípicos bastante extremos (alrededor de 400), mientras que el resto de variables estaban más centradas.

También vimos que había un desbalanceo moderado: 2/3 Abnormal y 1/3 Normal. Esto es importante tenerlo en cuenta más adelante con las métricas.

## 1.3 Tareas de desafios

Las tareas desafío de este lab tenían como objetivo entender como explorar manualmente todas las características e identificar sus valores atípicos mediante pandas. Para hacer la ultima tarea de desafío, para explorar sobre otros datos de UCI,simplemente me descargue el DataSet que estamos usando en nuestra asignatura de Aprendizaje Automático y le pase las mismas celdas de código. Mi data set era del zoo y tube estos resultados para el mapa de calor.

# 2 Codificación de Datos Categóricos (Lab 3.3)

## 2.1 Qué Hicimos

En este lab trabajamos con el dataset `imports-85.csv` que tiene información sobre automóviles. El archivo tiene 205 instancias, 25 atributos y no tenía encabezados, así que tuvimos que definirlos manualmente. El objetivo era aprender a codificar variables categóricas de dos formas diferentes.

## 2.2 Dos Técnicas de Codificación

Para variables **ordinales** (que tienen un orden), usamos un diccionario que mapeaba valores a números:

Bloque 1: Codificación ordinal

```
1 # Número de puertas: two=2, four=4
2 df['num_doors'] = df['num_doors'].map({"two": 2, "four": 4})
3
4 # Número de cilindros: tiene orden (two, three, four...)
```

```

5 df['num_cylinders'] = df['num_cylinders'].map({
6     "two": 2, "three": 3, "four": 4, "five": 5,
7     "six": 6, "eight": 8, "twelve": 12
8 })

```

Para variables **nominales** (sin orden natural), usamos `get_dummies()` que crea columnas binarias:

#### Bloque 2: Codificación one-hot

```

1 # drive-wheels: 4wd, fwd, rwd (sin orden)
2 df = pd.concat([df, pd.get_dummies(df['drive-wheels'],
3     prefix='drive')], axis=1)
4
5 # aspiration con drop_first=True evita colinealidad
6 df = pd.concat([df, pd.get_dummies(df['aspiration'],
7     drop_first=True)], axis=1)

```

### 2.3 Desafío: Añadir Más Variables

El desafío consistía en codificar variables adicionales: `fuel-type`, `body-style` y `engine-location`.

Lo que hicimos fue aplicar la misma lógica:

- **fuel-type**: One-hot (diesel, gas - sin orden)
- **body-style**: One-hot (convertible, hardtop, hatchback, sedan, wagon)
- **engine-location**: Ordinal o one-hot con `drop_first=True`

Después de codificar todo, nos quedó un dataset con muchas más columnas que las que teníamos al inicio..

**Lección aprendida:** La codificación ordinal es más sencilla (menos columnas) pero solo funciona si la variable tiene un orden. Con nominales, one-hot es la opción más segura para evitar que el modelo interprete un número como una magnitud cuando no la hay.

### 3 División de Datos y Entrenamiento (Lab 3.4)

#### 3.1 Preparación del Dataset

Lo primero fue convertir la clase a números: Abnormal = 1, Normal = 0. XGBoost en SageMaker tiene un requisito importante: **la columna objetivo debe estar en la primera posición**, así que reordenamos todas las columnas.

#### 3.2 División Estratificada

Dividimos el dataset en tres partes usando `train_test_split` con `stratify` para mantener la proporción de clases:

Bloque 3: División de datos

```
1 # 80% train, 20% rest
2 train, rest = train_test_split(
3     df, test_size=0.2, random_state=42,
4     stratify=df['class']
5 )
6
7 # 50/50 del resto
8 test, validate = train_test_split(
9     rest, test_size=0.5, random_state=42
10)
11
```

#### Distribución final:

- Train: 248 (80 %)
- Validation: 31 (10 %)
- Test: 31 (10 %)

Usar `stratify` asegura que cada conjunto tenga la misma proporción de Normal/Abnormal.

#### 3.3 Entrenamiento con SageMaker

Exportamos los CSV sin encabezados ni índices y los cargamos a S3. Luego configuramos XGBoost:

Bloque 4: Configuración XGBoost

```
1 hyperparams = {
2     "num_round": "42",      # 42 iteraciones de boosting
3     "eval_metric": "auc",   # Métrica: Area Under Curve
4     "objective": "binary:logistic" # Clasificación binaria
5 }
6
7 instance_type = 'ml.m4.xlarge' # Tipo de instancia EC2
8
```

SageMaker se encargó de entrenar el modelo en la nube. Con 248 instancias de entrenamiento y 6 características, el modelo aprendió a distinguir entre columnas normales y anormales.

## 4 Implementación y Predicciones (Lab 3.5)

### 4.1 Deploy del Modelo

Una vez entrenado, desplegamos el modelo como un endpoint en tiempo real:

Bloque 5: Despliegue

```
1 xgb_predictor = xgb_model.deploy(
2     initial_instance_count=1,
3     instance_type='ml.m4.xlarge',
4     serializer=CSVSerializer()
5 )
6
```

Esto pone el modelo en un servidor que puede recibir peticiones y devolver predicciones. Lo importante aquí es que sigue costando dinero mientras esté activo, así que hay que borrarlo cuando no se use.

### 4.2 Haciendo Predicciones

Enviamos datos al endpoint y nos devuelve una probabilidad:

Bloque 6: Predicción

```
1 result = xgb_predictor.predict(test_data_row)
2 # Devuelve algo como 0.87 (probabilidad de ser Abnormal)
3
```

Bloque 7: Conversión binaria

El modelo devuelve **probabilidades (0-1)**, no clases. Un valor de 0.87 significa que el modelo cree que hay 87% de probabilidad de que sea Abnormal. Necesitamos decidir un **umbral** para convertir eso a un 1 o 0 final.

```
1 def binary_convert(prob):
2     if prob > 0.45:
3         return 1 # Abnormal
4     else:
5         return 0 # Normal
6
```

### 4.3 Batch Transform

Para procesar muchos datos de una sola vez sin mantener un endpoint abierto, usamos Batch Transform, que es más barato. Al final borramos el endpoint con `delete_endpoint()` para no pagar por nada que no usemos.

### 4.4 Desafío: Experimentar con Umbral

El gran desafío de este lab fue entender que cambiar el umbral (0.25, 0.30, 0.45, 0.75) **cambia completamente las métricas**. Con umbral bajo captamos más casos positivos (bueno para detectar anomalías, malo para falsos positivos). Con umbral alto al revés. Los resultados de cada combinación se muestran en el lab siguiente.

## 5 Evaluación del Modelo (Lab 3.6)

### 5.1 La Matriz de Confusión

Para evaluar un modelo de clasificación, usamos la matriz de confusión que cruza lo que el modelo predijo vs lo que realmente pasó:

		Pred: Normal	Pred: Abnormal
Real: Normal	TN	FP	
Real: Abnormal	FN	TP	

Cuadro 1: Matriz de Confusión

En contexto médico, los **Falsos Negativos (FN)** son lo peor: son pacientes enfermos que dijimos que estaban sanos.

### 5.2 Métricas Principales

- **Sensibilidad:**  $\frac{TP}{TP+FN}$  = Detectar positivos reales
- **Especificidad:**  $\frac{TN}{TN+FP}$  = Detectar negativos reales
- **Precisión:**  $\frac{TP}{TP+FP}$  = Qué % de nuestros positivos acertamos
- **Exactitud:**  $\frac{TP+TN}{Total}$  = Aciertos generales
- **FPR:**  $\frac{FP}{FP+TN}$  = Falsas alarmas
- **FNR:**  $\frac{FN}{TP+FN}$  = Omisiones (casos perdidos)
- **VPN:**  $\frac{TN}{TN+FN}$  = Fiabilidad de negativos

### 5.3 Desafío: Comparar Umbrales

Probamos con umbrales 0.25, 0.30 y 0.75 para ver cómo afectaban a las métricas:

#### Umbrales 0.25 (Bajo)

- ↑ Sensibilidad
- ↓ Especificidad
- Más FP
- Detectamos más casos pero con más falsas alarmas

#### Umbrales 0.45 (Medio)

- Balance razonable
- Menos extremo en ambas direcciones
- Buena opción general

#### Umbrales 0.75 (Alto)

- ↓ Sensibilidad
- ↑ Especificidad
- Más FN
- Pocos falsos positivos pero perdemos casos reales

Lo que descubrimos fue que **el umbral más bajo causaba overfitting**: el modelo parecía mejor en métricas locales pero generalizaba peor. En medicina, el trade-off es complicado, pero generalmente preferimos capturar más casos (sacrificar especificidad) que perder enfermos (reducir FN).

### 5.4 Curva ROC y AUC

Generamos la curva ROC (Sensibilidad vs FPR) para visualizar el rendimiento a diferentes umbrales. El AUC (Area Under Curve) nos da un número único: si es 1.0 es perfecto, si es 0.5 es aleatorio, y  $> 0.8$  generalmente es bueno. Comparamos usando tanto probabilidades como binarios convertidos.

## 6 Ajuste de Hiperparámetros (Lab 3.7)

### 6.1 Qué Hicimos

Hasta ahora usábamos `num_round=42` y otros valores "por defecto". En este lab, dejamos que AWS busque automáticamente los mejores valores usando el servicio `HyperparameterTuner`.

### 6.2 Configuración del Tuner

Definimos rangos de búsqueda para 5 hiperparámetros:

Bloque 8: Rangos de búsqueda

```
1 hyperparameter_ranges = {  
2     'alpha': (0, 100),  
3     'min_child_weight': (1, 5),  
4     'subsample': (0.5, 1),  
5     'eta': (0.1, 0.3),  
6     'num_round': (1, 50)  
7 }  
8  
9 max_jobs = 10  
10 max_parallel_jobs = 1  
11
```

#### Significado:

- **alpha:** Regularización L1
- **min\_child\_weight:** Peso mínimo en nodos
- **subsample:** Fracción de datos por árbol
- **eta:** Tasa de aprendizaje
- **num\_round:** Número de árboles

### 6.3 Ejecución

El tuner ejecutó 10 trabajos completos de entrenamiento, cada uno probando una combinación diferente. Tardó aproximadamente 45 minutos en total. La métrica que intentaba minimizar era `validation:error@.40`, que es la tasa de error del conjunto de validación con umbral 0.40.

### 6.4 Desafío: Análisis de Resultados

**La pregunta era:** ¿El modelo ajustado es mejor que el original?

La respuesta no siempre fue sí. Hay varias razones por las que el ajuste puede no mejorar mucho:

- El modelo original ya era bastante bueno
- Con solo 248 datos de training, hay poco margen de mejora
- Probamos solo 10 combinaciones (un tuning real usa 30+)
- Los rangos fueron reducidos para ahorrar tiempo

En un escenario real, con más datos, más trabajos (30-50) y rangos completos de búsqueda, probablemente veríamos mejoras más significativas. Pero este lab demostró cómo SageMaker automatiza completamente la tarea de encontrar los mejores hiperparámetros sin necesidad de hacerlo manualmente.

## 7 Conclusiones

Tras acabar esta práctica, creo que lo más importante que me llevo es entender que hacer un modelo de ML no es solo entrenar y ya. Es todo un proceso donde cada decisión importa: desde cómo explores los datos, hasta qué umbral de clasificación eliges al final.

La exploración de datos fue clave. Sin saber que el `degree_spondylolisthesis` tenía esos valores extremos o que el dataset estaba desbalanceado, hubiera sido difícil hacer decisiones informadas después. Los gráficos y estadísticas nos dijeron mucho de cómo trabajar.

Lo que más me sorprendió fue el impacto del umbral en las métricas. Con umbral 0.25 todo parecía mejor, pero en realidad estábamos siendo demasiado agresivos y probablemente sobreajustando. Con 0.75 al revés. Eso que los notebooks indicaban sobre overfitting cobró sentido cuando vimos los números.

SageMaker simplifica mucho las cosas: entrenar un modelo, desplegarlo, hacer predicciones... todo está integrado. Lo complicado es entender qué datos usar, cómo dividirlos, qué métricas importan. La plataforma es solo una herramienta.

El ajuste automático de hiperparámetros mostró que SageMaker puede hacer búsquedas que manualmente serían imposibles. Aunque en nuestro caso con 248 datos no vimos grandes mejoras, en un dataset real con decenas de miles de registros y rangos más amplios, probablemente sí.

En definitiva, creo que el pipeline completo que implementamos (exploración → codificación → división → entrenamiento → evaluación → ajuste) es lo que realmente funciona en producción. No es un solo paso, es un ciclo donde probablemente tendremos que iterar varias veces hasta llegar a un modelo que satisfaga nuestros requisitos de negocio.

Y en contexto médico especialmente, las métricas y el umbral no son decisiones técnicas puramente, son decisiones que afectan a pacientes reales. Eso es lo que hizo esta práctica más concreta que números en una pantalla.