

✓ Introducción y Contexto

✓ Proyecto Aprendizaje Supervisado

Humberto Mondragón García A01711912

Gabriela Marissa Mosquera A01666191

Felipe de Jesús Damián Rodríguez A01707246

La agricultura es un pilar fundamental para la economía y la seguridad alimentaria de México. La capacidad de predecir el rendimiento de los cultivos con precisión es crucial para la planificación estratégica, la gestión de recursos y la toma de decisiones tanto en el sector público como en el privado. Modelos predictivos robustos pueden ayudar a los agricultores a optimizar sus prácticas, a los gobiernos a diseñar políticas de apoyo efectivas y a los mercados a anticipar la oferta y la demanda.

Este proyecto tiene como objetivo principal encontrar los mejores modelos de aprendizaje supervisado para predecir el rendimiento de cultivos en el territorio mexicano. Para lograrlo, se utilizará un conjunto de datos públicos proporcionado por el Gobierno de México, que contiene registros detallados de la producción agrícola nacional.

Definición del Problema

El desafío central es un problema de regresión, donde se busca predecir una variable numérica continua.

Variable Objetivo: La variable a predecir es el Rendimiento, que mide la productividad de un cultivo, comúnmente en toneladas por hectárea.

Variables Predictoras: Se utilizarán características como el año de la cosecha, el estado de la república, el ciclo productivo (ej. Primavera-Verano), la modalidad (Riego o Temporal) y, por supuesto, el tipo de cultivo.

El estudio se concentrará en 10 cultivos de gran importancia para México:

Maíz grano

Frijol

Pastos y praderas

Avena forrajera en verde

Sorgo grano

Tomate rojo (jitomate)

Chile verde

Tomate verde \ Calabacita

Trigo grano

A través de la exploración y comparación de diferentes algoritmos de machine learning, este proyecto buscará entregar un modelo confiable y preciso que aporte valor al análisis de la productividad agrícola del país.

✓ Preprocesamiento de los datos

Antes de poder entrenar cualquier modelo de aprendizaje automático, es fundamental preparar y limpiar los datos. Esta fase, es uno de los pasos más críticos en un proyecto de ciencia de datos, ya que la calidad del modelo final depende directamente de la calidad de los datos con los que se alimenta.

En esta sección, se llevarán a cabo varias tareas esenciales de limpieza y transformación. Se comenzará cargando el conjunto de datos para una inspección inicial. A continuación, se abordará el problema de los valores nulos y se eliminarán columnas que no aportan información relevante para nuestro objetivo. Finalmente, se asegurará que todas las variables tengan el tipo de dato correcto y se transformarán las variables categóricas (como el nombre del estado o del cultivo) a un formato numérico que los algoritmos puedan procesar.

✓ Importar librerías

Para comenzar nuestro análisis, el primer paso es importar todas las librerías de Python que utilizaremos a lo largo del proyecto. Estas librerías nos proporcionarán las herramientas necesarias para la manipulación de datos, la visualización y la construcción de nuestros modelos de aprendizaje automático.

pandas y *numpy*: Son la base para la manipulación y el análisis de datos. *pandas* nos permite trabajar con estructuras de datos llamadas *DataFrames*, y *numpy* nos da soporte para operaciones numéricas eficientes.

matplotlib.pyplot: Es nuestra herramienta principal para crear visualizaciones estáticas, como gráficos y diagramas, que nos ayudarán a entender mejor los datos. *sklearn (Scikit-learn)*: Es la librería central de machine learning. De aquí importaremos módulos para:

- Dividir nuestros datos en conjuntos de entrenamiento y prueba (model_selection).
- Implementar los algoritmos de regresión (como DecisionTreeRegressor, KNeighborsRegressor, Ridge y Lasso).
- Preprocesar los datos (StandardScaler).
- Evaluar el rendimiento de los modelos (metrics).

torch: Es el framework de deep learning que usaremos para construir, entrenar y evaluar nuestra red neuronal.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge, RidgeCV, Lasso, LassoCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression
import random
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data
from torch.utils.data import DataLoader
from torch.utils.data import TensorDataset
```

▼ Carga de Datos

El primer paso en nuestro preprocesamiento es cargar el conjunto de datos en memoria. Para ello, utilizamos la función `read_csv` de la librería *pandas*, que es la herramienta estándar para leer archivos de formato CSV y convertirlos en un *DataFrame*.

El archivo *Agromex_seleccionado.csv* contiene todos los registros de producción agrícola que analizaremos. Estos datos se almacenan en un *DataFrame* llamado *df*, que es la estructura fundamental sobre la cual realizaremos todas las operaciones de limpieza y transformación.

```
df = pd.read_csv('Agromex_seleccionado.csv')
→ <ipython-input-2-1845607390>:1: DtypeWarning: Columns (13,17,19,21) have mixed types. Specify dtype option on import or set low_memory=False
  df = pd.read_csv('Agromex_seleccionado.csv')
```

f.info(): Nos proporciona un resumen técnico del *DataFrame*. Con este comando podemos ver el número total de filas y columnas, el tipo de dato de cada columna y la cantidad de valores no nulos.

df.head(): Nos muestra las primeras cinco filas del *DataFrame*. Esto nos da una vista previa rápida de los datos reales, permitiéndonos ver ejemplos concretos de los valores en cada columna y familiarizarnos con el conjunto de datos.

```
df.info()
df.head()
```

```
↳ <class 'pandas.core.frame.DataFrame'>
RangeIndex: 229965 entries, 0 to 229964
Data columns (total 24 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Anio              229965 non-null   int64  
 1   Idestado          229965 non-null   int64  
 2   Nomestado         229965 non-null   object  
 3   Idciclo           229965 non-null   int64  
 4   Nomcicloproductivo 229965 non-null   object  
 5   Idmodalidad        229965 non-null   int64  
 6   Nommodalidad       229965 non-null   object  
 7   Idunidadmedida    229965 non-null   int64  
 8   Nomunidad          229965 non-null   object  
 9   Idcultivo          229965 non-null   int64  
 10  Nomcultivo         229965 non-null   object  
 11  Cosechada          229965 non-null   float64 
 12  Siniestrada        229965 non-null   float64 
 13  Rendimiento        229965 non-null   object  
 14  Precio             200108 non-null   float64 
 15  Valorproduccion   229965 non-null   float64 
 16  Idaddr             213864 non-null   float64 
 17  Nomaddr            213864 non-null   object  
 18  Idcader             213864 non-null   float64 
 19  Nomcader            213864 non-null   object  
 20  Idmunicipio         213864 non-null   float64 
 21  Nommunicipio        213864 non-null   object  
 22  Nomcultivo Sin Um  0 non-null      float64 
 23  Preciomediiorural  29857 non-null   float64 
dtypes: float64(9), int64(6), object(9)
memory usage: 42.1+ MB
```

	Anio	Idestado	Nomestado	Idciclo	Nomcicloproductivo	Idmodalidad	Nommodalidad	Idunidadmedida	Nomunidad	Idcultivo	...	Precio
0	2022	18	Nayarit	2	Primavera-Verano	2	Temporal	200201	Tonelada	7490000	...	NaN
1	2022	8	Chihuahua	2	Primavera-Verano	1	Riego	200201	Tonelada	8810000	...	NaN
2	2022	26	Sonora	2	Primavera-Verano	1	Riego	200201	Tonelada	7490000	...	NaN
3	2021	2	Baja California	1	Otoño-Invierno	2	Temporal	200201	Tonelada	9050000	...	NaN
4	2022	2	Baja California	2	Primavera-Verano	1	Riego	200201	Tonelada	6840000	...	NaN

5 rows × 24 columns

▼ Análisis de las variables

Después de la revisión con `df.info()`, se observa que varias columnas requieren ser eliminadas o transformadas para preparar adecuadamente el conjunto de datos

- Eliminación de NAs:
 - Las columnas geográficas como `Idaddr`, `Nomaddr`, `Idcader`, `Nomcader`, `Idmunicipio` y `Nommunicipio` presentan más de 16,000 valores nulos. Dada la alta cantidad de datos faltantes y para simplificar el modelo, se decide eliminarlas. Lo mismo ocurre con `Preciomediiorugal` y `Nomcultivo Sin Um`, que están prácticamente vacías.
- Eliminación de Columnas Redundantes:
 - Las columnas `Idunidadmedida` y `Nomunidad` indican "Tonelada" en todos los registros, por lo que no aportan variabilidad y son eliminadas.

Pares de columnas como (`Idmodalidad`, `Nommodalidad`), (`Idestado`, `Nomestado`), etc., contienen información idéntica. Se decide eliminar las columnas de Id y conservar las de nombre (Nom...), ya que las usaremos para crear variables dummy. *Conservar los IDs numéricos podría llevar a que el modelo interprete erróneamente un orden o una magnitud que no existe.*

- Manejo de la Columna Precio:
 - La columna Precio también tiene valores nulos, aunque en menor medida. Debido a su potencial importancia para predecir el rendimiento, en lugar de eliminar la columna, más adelante se optará por eliminar las filas que no contengan este dato.

```
df = df.drop(['Idaddr', 'Nomaddr', 'Nomunidad', 'Idcader', 'Nomcader', 'Idmunicipio', 'Nommunicipio', 'Nomcultivo Sin Um', 'Preciomediiorugal', 'Ide
```

Con df.drop se eliminan las columnas y con df.info revisamos esto

```
df.info()
```

```
→ <class 'pandas.core.frame.DataFrame'>
RangeIndex: 229965 entries, 0 to 229964
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   Anio              229965 non-null   int64  
 1   Nomestado         229965 non-null   object  
 2   Nomcicloproductivo 229965 non-null   object  
 3   Nommodalidad      229965 non-null   object  
 4   Nomcultivo        229965 non-null   object  
 5   Cosechada         229965 non-null   float64 
 6   Siniestrada       229965 non-null   float64 
 7   Rendimiento       229965 non-null   object  
 8   Precio             200108 non-null   float64 
 9   Valorproduccion    229965 non-null   float64 
dtypes: float64(4), int64(1), object(5)
memory usage: 17.5+ MB
```

Creemos que la variable Precio puede afectar en el análisis, por lo que eliminaremos las líneas que no tienen los valores de precio. Nos quedamos con 200108 registros

```
df.dropna(subset=['Precio'], inplace=True)
df.info()
```

```
→ <class 'pandas.core.frame.DataFrame'>
Index: 200108 entries, 29857 to 229964
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   Anio              200108 non-null   int64  
 1   Nomestado         200108 non-null   object  
 2   Nomcicloproductivo 200108 non-null   object  
 3   Nommodalidad      200108 non-null   object  
 4   Nomcultivo        200108 non-null   object  
 5   Cosechada         200108 non-null   float64 
 6   Siniestrada       200108 non-null   float64 
 7   Rendimiento       200108 non-null   object  
 8   Precio             200108 non-null   float64 
 9   Valorproduccion    200108 non-null   float64 
dtypes: float64(4), int64(1), object(5)
memory usage: 16.8+ MB
```

Durante la inspección de datos, observamos que nuestra variable objetivo, Rendimiento, fue clasificada como de tipo object. Esto indica la presencia de valores no numéricos, lo cual es un problema, ya que para un modelo de regresión, la variable objetivo debe ser numérica.

El siguiente paso consiste en limpiar esta columna. La estrategia, será identificar aquellos registros que contengan texto y, dado que son muy pocos, eliminarlos. Una vez que la columna esté libre de caracteres no numéricos, la convertiremos al formato float para que pueda ser utilizada correctamente en el entrenamiento de los modelos.

```
# Valores no numéricos en la columna 'Rendimiento'
valores_no_numericos = df[pd.to_numeric(df['Rendimiento'], errors='coerce').isna()]
# Eliminar los registros con valores no numéricos en la columna 'Rendimiento'
df = df.drop(valores_no_numericos.index)
df['Rendimiento'] = df['Rendimiento'].astype(float)
df.info()
```

```
→ <class 'pandas.core.frame.DataFrame'>
Index: 200106 entries, 29857 to 229964
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   Anio              200106 non-null   int64  
 1   Nomestado         200106 non-null   object  
 2   Nomcicloproductivo 200106 non-null   object  
 3   Nommodalidad      200106 non-null   object  
 4   Nomcultivo        200106 non-null   object  
 5   Cosechada         200106 non-null   float64 
 6   Siniestrada       200106 non-null   float64 
 7   Rendimiento       200106 non-null   float64 
 8   Precio             200106 non-null   float64 
 9   Valorproduccion    200106 non-null   float64 
dtypes: float64(5), int64(1), object(4)
```

memory usage: 16.8+ MB

Ahora tenemos 2 registros menos, y Rendimiento ya no es object.

▼ One-hot encoding

Los algoritmos de machine learning no pueden trabajar directamente con datos de tipo texto (como 'Jalisco' o 'Maíz grano'), por lo que necesitamos convertirlos a un formato numérico.

Para lograr esto, aplicaremos la técnica de one-hot encoding utilizando la función `get_dummies` de pandas. Este método creará nuevas columnas para cada categoría única en nuestras variables de tipo object. Por ejemplo, la columna `Nomestado` se transformará en múltiples columnas como `Nomestado_Jalisco`, `Nomestado_Sonora`, etc., asignando un 1 o 0 a cada fila según corresponda.

```
df = pd.get_dummies(df)
```

```
df.head()
df.info()
```

Data columns (total 53 columns):			
#	Column	Non-Null Count	Dtype
0	Anio	200106	non-null int64
1	Cosechada	200106	non-null float64
2	Siniestrada	200106	non-null float64
3	Rendimiento	200106	non-null float64
4	Precio	200106	non-null float64
5	Valorproduccion	200106	non-null float64
6	Nomestado_Aguascalientes	200106	non-null bool
7	Nomestado_Baja California	200106	non-null bool
8	Nomestado_Baja California Sur	200106	non-null bool
9	Nomestado_Campeche	200106	non-null bool
10	Nomestado_Chiapas	200106	non-null bool
11	Nomestado_Chihuahua	200106	non-null bool
12	Nomestado_Ciudad de México / DF	200106	non-null bool
13	Nomestado_Coahuila	200106	non-null bool
14	Nomestado_Colima	200106	non-null bool
15	Nomestado_Durango	200106	non-null bool
16	Nomestado_Guanajuato	200106	non-null bool
17	Nomestado_Guerrero	200106	non-null bool
18	Nomestado_Hidalgo	200106	non-null bool
19	Nomestado_Jalisco	200106	non-null bool
20	Nomestado_Michoacán	200106	non-null bool
21	Nomestado_Morelos	200106	non-null bool
22	Nomestado_México	200106	non-null bool
23	Nomestado_Nayarit	200106	non-null bool
24	Nomestado_Nuevo León	200106	non-null bool
25	Nomestado_Oaxaca	200106	non-null bool
26	Nomestado_Puebla	200106	non-null bool
27	Nomestado_Querétaro	200106	non-null bool
28	Nomestado_Quintana Roo	200106	non-null bool
29	Nomestado_San Luis Potosí	200106	non-null bool
30	Nomestado_Sinaloa	200106	non-null bool
31	Nomestado_Sonora	200106	non-null bool
32	Nomestado_Tabasco	200106	non-null bool
33	Nomestado_Tamaulipas	200106	non-null bool
34	Nomestado_Tlaxcala	200106	non-null bool
35	Nomestado_Veracruz	200106	non-null bool
36	Nomestado_Yucatán	200106	non-null bool
37	Nomestado_Zacatecas	200106	non-null bool
38	Nomcicloproductivo_Otoño-Invierno	200106	non-null bool
39	Nomcicloproductivo_Perennes	200106	non-null bool
40	Nomcicloproductivo_Primavera-Verano	200106	non-null bool
41	Nommodalidad_Riego	200106	non-null bool
42	Nommodalidad_Temporal	200106	non-null bool
43	Nomcultivo_Avena forrajera en verde	200106	non-null bool
44	Nomcultivo_Calabacita	200106	non-null bool
45	Nomcultivo_Chile verde	200106	non-null bool
46	Nomcultivo_Frijol	200106	non-null bool
47	Nomcultivo_Maíz grano	200106	non-null bool
48	Nomcultivo_Pastos y praderas	200106	non-null bool
49	Nomcultivo_Sorgo grano	200106	non-null bool
50	Nomcultivo_Tomate rojo (jitomate)	200106	non-null bool
51	Nomcultivo_Tomate verde	200106	non-null bool
52	Nomcultivo_Trigo grano	200106	non-null bool

```
dtypes: bool(47), float64(5), int64(1)
memory usage: 19.7 MB
```

✓ Guardado del nuevo DataFrame

Ahora que hemos completado la limpieza y transformación de los datos, guardamos nuestro DataFrame limpio en un nuevo archivo CSV.

Este paso nos crea un "punto de control". Al guardar el estado actual de los datos, nos aseguramos de no tener que repetir todos los pasos de limpieza anteriores cada vez que comencemos a trabajar en un nuevo modelo. Podremos cargar directamente este archivo preprocesado, lo cual es especialmente útil ya que algunos algoritmos requerirán preprocesamientos adicionales (como la estandarización de datos) que otros no. Esto nos permite mantener un flujo de trabajo ordenado y sin errores.

Utilizamos el comando `to_csv` para esta tarea, especificando `index=False` para evitar que pandas guarde el índice del DataFrame como una columna en el nuevo archivo.

```
df.to_csv('Agromex_seleccionado_preprocesado.csv', index=False)
```

✓ Árbol de decisión

✓ Modelo Inicial / de Referencia

Comenzamos la fase de modelado con un Árbol de Decisión. Este algoritmo es una excelente opción como nuestro modelo inicial o de referencia porque es fácil de interpretar y no requiere que las variables numéricas estén estandarizadas o normalizadas para funcionar correctamente.

Dividimos el DataFrame en `X`, que contendrá todas las variables predictoras (características), y `y`, que tendrá únicamente nuestra variable objetivo, Rendimiento.

```
y = df['Rendimiento']
X = df.drop(['Rendimiento'], axis=1)
```

Usamos la función `train_test_split` de `sklearn` para dividir `X` y `y` en dos subconjuntos. El 80% de los datos se destinará al entrenamiento del modelo (`X_train`, `y_train`), y el 20% restante se reservará para su evaluación (`X_test`, `y_test`).

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

Ahora creamos una instancia del `DecisionTreeRegressor` con sus hiperparámetros por defecto y lo entrenamos con nuestros datos de entrenamiento (`X_train`, `y_train`) utilizando el método `.fit()`.

```
modelo_arbol = DecisionTreeRegressor()
modelo_arbol.fit(X_train, y_train)
```

Inmediatamente después, evaluamos su rendimiento con el método `.score()`, que para este tipo de modelo calcula el coeficiente de determinación (R^2) sobre el conjunto de prueba. Este valor nos indicará qué porcentaje de la variabilidad del rendimiento es explicado por nuestro modelo.

El resultado obtenido es un R^2 de aproximadamente 0.94, esto indica que el modelo, en su estado inicial, es capaz de explicar el 94% de la varianza en los datos de rendimiento del conjunto de prueba. Lo cual es muy bueno.

```
modelo_arbol.score(X_test, y_test)
```

Vamos a consultar la profundidad de nuestro árbol recién entrenado.

```
modelo_arbol.get_depth()
```

La profundidad obtenida (51) es alta, podemos sospechar sobreajuste, entonces vamos a ajustar manualmente los hiperparámetros:

- `max_depth`: Limita la profundidad máxima del árbol.
- `max_leaf_nodes`: Limita el número máximo de nodos hoja (nodos terminales).

```
modelo_arbol = DecisionTreeRegressor(max_depth=34, max_leaf_nodes = 800)
modelo_arbol.fit(X_train, y_train)
modelo_arbol.score(X_test, y_test)
```

→ 0.92475426543334

Al entrenar este nuevo modelo restringido, obtenemos un R^2 de 0.93. Es interesante notar que este valor es ligeramente inferior al del modelo sin restricciones. Y fue el más alto después de iterar varias veces para encontrarlo modificando aproximadamente unas 10 veces por parámetro

▼ Vecinos más cercanos (K-Nearest-Neighbors)

▼ Segundo modelo

El segundo modelo que exploraremos es el K-Nearest Neighbors (KNN). Este es un algoritmo basado en instancias que predice el valor de un punto de datos basándose en el promedio de los valores de sus "k" vecinos más cercanos en el espacio de características.

KNN es un algoritmo basado en distancia. Esto significa que es muy sensible a la escala de las variables. Si una variable (como `Valorproduccion`) tiene una escala mucho mayor que otra (como `Año`), dominará el cálculo de la distancia y sesgará el modelo.

Para evitar esto, los datos de este modelo **requieren un preprocessamiento de estandarización**.

```
df = pd.read_csv('Agromex_seleccionado_preprocesado.csv')
```

Se separa la columna `Rendimiento` como variable objetivo (`y`) y el resto del DataFrame como variables predictoras (`x`), eliminando esa columna.

```
y = df['Rendimiento']
X = df.drop(['Rendimiento'], axis=1)
```

Usamos la función `train_test_split` de `sklearn` para dividir `X` y `y` en dos subconjuntos. El 80% de los datos se destinará al entrenamiento del modelo (`X_train, y_train`), y el 20% restante se reservará para su evaluación (`X_test, y_test`).

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

Usaremos `StandardScaler` de `sklearn` para transformar nuestras características de modo que cada una tenga una media de 0 y una desviación estándar de 1. Esto asegura que todas las variables contribuyan de manera equitativa al resultado

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

El hiperparámetro más importante en KNN es `n_neighbors`, que define cuántos vecinos se considerarán para hacer una predicción.

Iteramos entre muchos valores para encontrar el valor óptimo de `k`. Se probaron varios valores, como 3, 5, 7, 51, 153, 521, 1021, 5003 etc. Se observó que a medida que `k` aumentaba, el rendimiento (R^2) disminuía y subía ligeramente. El mejor lo encontramos en `k = 3`. Por lo tanto, usamos ese.

```
modelo_knn = KNeighborsRegressor(n_neighbors=3)
modelo_knn.fit(X_train_scaled, y_train)
modelo_knn.score(X_test_scaled, y_test)
```

→ 0.7379091737288065

▼ Métricas

El R² obtenido es de 0.74, lo que significa que el modelo explica el 74% de la varianza del rendimiento. Es un resultado decente, pero inferior al del árbol de decisión

Además del R², es útil calcular otras métricas para entender mejor el error del modelo.

- MSE (Mean Squared Error): Es el promedio de los errores al cuadrado.
- MAE (Mean Absolute Error): Es el promedio de los errores absolutos.

El MSE es de 173.41 y el MAE es de 3.11. En promedio, las predicciones del modelo se desvían en 3.11 unidades del valor real.

```
y_pred = modelo_knn.predict(X_test_scaled)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
print("MSE:", mse)
print("MAE:", mae)
```

→ MSE: 173.41059046135737
MAE: 3.1138765345726522

▼ Ridge

▼ Tercer modelo

A diferencia de la regresión lineal simple, Ridge incluye una penalización de regularización L2. Esta penalización ayuda a prevenir el sobreajuste al contraer los coeficientes del modelo, y es especialmente útil cuando las variables predictoras están altamente correlacionadas entre sí.

El hiperparámetro clave en Ridge es alpha (también conocido como lambda), que controla la fuerza de esta penalización. Un alpha más alto resulta en un modelo más simple.

$$J(\theta) = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^p \theta_j^2$$

Pre-procesamiento

Al igual que KNN, los modelos de regresión regularizados como Ridge y Lasso son sensibles a la escala de las características. La penalización se aplica directamente sobre la magnitud de los coeficientes, por lo que si una variable tiene una escala mucho mayor que otra, su coeficiente será penalizado de forma desproporcionada.

Por esta razón, **es necesario estandarizar los datos** antes de entrenar un modelo para Ridge. Por lo que no volveremos a inicializar las variables y usaremos las ya normalizadas de KNN.

▼ Hiperparámetros

Encontrar el valor óptimo para alpha es crucial. En lugar de hacerlo manualmente, utilizaremos una herramienta más eficiente y robusta: RidgeCV. Este método realiza una búsqueda automática del mejor alpha a través de validación cruzada (Cross-Validation)

Primero, definimos una amplia gama de posibles valores para alpha, desde muy grandes (10¹⁰) hasta muy pequeños (0.01), utilizando np.linspace.

```
lambdas = 10**np.linspace(10,-2,100)
lambdas
```

→ array([1.00000000e+10, 7.56463328e+09, 5.72236766e+09, 4.32876128e+09,
 3.27454916e+09, 2.47707636e+09, 1.87381742e+09, 1.41747416e+09,
 1.07226722e+09, 8.11130831e+08, 6.13590727e+08, 4.64158883e+08,
 3.51119173e+08, 2.65608778e+08, 2.00923300e+08, 1.51991108e+08,
 1.14975700e+08, 8.69749003e+07, 6.57933225e+07, 4.97702356e+07,
 3.76493581e+07, 2.84803587e+07, 2.15443469e+07, 1.62975083e+07,
 1.23284674e+07, 9.32603347e+06, 7.05480231e+06, 5.33669923e+06,
 4.03701726e+06, 3.05385551e+06, 2.31012970e+06, 1.74752840e+06,
 1.32194115e+06, 1.00000000e+06, 7.56463328e+05, 5.72236766e+05,
 4.32876128e+05, 3.27454916e+05, 2.47707636e+05, 1.87381742e+05,
 1.41747416e+05, 1.07226722e+05, 8.11130831e+04, 6.13590727e+04,

```
4.64158883e+04, 3.51119173e+04, 2.65608778e+04, 2.00923300e+04,
1.51991108e+04, 1.14975700e+04, 8.69749003e+03, 6.57933225e+03,
4.97702356e+03, 3.76493581e+03, 2.84803587e+03, 2.15443469e+03,
1.62975083e+03, 1.23284674e+03, 9.32603347e+02, 7.05480231e+02,
5.33669923e+02, 4.03701726e+02, 3.05385551e+02, 2.31012970e+02,
1.74752840e+02, 1.32194115e+02, 1.00000000e+02, 7.56463328e+01,
5.72236766e+01, 4.32876128e+01, 3.27454916e+01, 2.47707636e+01,
1.87381742e+01, 1.41747416e+01, 1.07226722e+01, 8.11130831e+00,
6.13590727e+00, 4.64158883e+00, 3.51119173e+00, 2.65608778e+00,
2.00923300e+00, 1.51991108e+00, 1.14975700e+00, 8.69749003e-01,
6.57933225e-01, 4.97702356e-01, 3.76493581e-01, 2.84803587e-01,
2.15443469e-01, 1.62975083e-01, 1.23284674e-01, 9.32603347e-02,
7.05480231e-02, 5.33669923e-02, 4.03701726e-02, 3.05385551e-02,
2.31012970e-02, 1.74752840e-02, 1.32194115e-02, 1.00000000e-02])
```

RidgeCV prueba internamente estos valores y, mediante validación cruzada, determina cuál de ellos ofrece el mejor rendimiento promedio, evitando así el sobreajuste durante la selección del hiperparámetro.

```
ridgecv = RidgeCV(alphas = lambdas)
ridgecv.fit(X_train_scaled, y_train)
al = ridgecv.alpha_
```

▼ Entrenamiento

Una vez que RidgeCV ha encontrado el alpha óptimo, entrenamos un modelo Ridge final con este hiperparámetro y evaluamos su rendimiento en el conjunto de prueba.

Se pueden observar los coeficientes que el modelo asignó a cada variable. algunos se hicieron bajos para reducir la complejidad.

```
ridge = Ridge(alpha = al)
ridge.fit(X_train_scaled, y_train)
pred = ridge.predict(X_test_scaled)           # Uso del modelo para predecir el conjunto de Testeo
print(f'El modelo Ridge tiene un MSE DE :{mean_squared_error(y_test, pred)}')          # Imprimir el error de mínimos cuadrados MSE
print(f'El modelo Ridge tiene un R^2 de {ridge.score(X_test_scaled, y_test)}')          # Imprimir el coeficiente de determinación R^2
→ El modelo Ridge tiene un MSE DE :705.5569592137282
El modelo Ridge tiene un R^2 de -1.1632677071604647e-05
```

Métricas

El resultado es un MSE de 705.55 y un R² de -0.00001

Este R² tan bajo nos indica que el modelo lineal, incluso con regularización Ridge, no es capaz de explicar la varianza en el rendimiento. Esto sugiere fuertemente que la relación entre las características y el rendimiento no es lineal y que se requieren modelos más complejos.

▼ Lasso

▼ Regresión Lasso

Al igual que Ridge, Lasso es una regresión lineal que aplica una penalización para prevenir el sobreajuste. Sin embargo, utiliza una penalización de regularización L1.

$$J(\theta) = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^p |\theta_j|$$

La diferencia clave es que la penalización L1 tiene la capacidad de reducir los coeficientes de las características menos importantes **hasta exactamente cero**. Esto significa que Lasso no solo simplifica el modelo, sino que también realiza una forma de selección automática de características, eliminando efectivamente las variables que no considera útiles.

Preprocesamiento

Al igual que con Ridge, para Lasso **es necesario estandarizar los datos**, para que la penalización se aplique de manera justa a todos los coeficientes. Igualmente se utilizarán los datos ya estandarizados en KNN.

✓ Encontrar hiperparámetros

Para encontrar el valor óptimo del hiperparámetro alpha. Usamos `Lassocv` que, al igual que `RidgeCV`, emplea validación cruzada para probar un rango de posibles alphas y seleccionar el que ofrezca el mejor rendimiento.

Aquí usamos la misma lista de alphas que generamos en Ridge.

```
lassocv = LassoCV(alphas = lambdas)
lassocv.fit(X_train_scaled, y_train)
lassocv.alpha_
```

→ np.float64(10000000000.0)

El proceso identifica que el `alpha` óptimo para este modelo es 10000000000

✓ Entrenamiento

Con el alpha óptimo de 0.01, entrenamos el modelo Lasso final, usamos `lasso(alpha)`, ponemos los datos con `.fit`, y después predecimos con el conjunto de `test` para evaluar el modelo

```
lasso = Lasso(alpha = 1000000000)
lasso.fit(X_train_scaled, y_train)
pred = lasso.predict(X_test_scaled)           # Uso del modelo para predecir el conjunto de Testeo
print(f'El modelo Lasso tiene un MSE DE :{mean_squared_error(y_test, pred)}')      # Imprimir el error de mínimos cuadrados MSE
print(f'El modelo Lasso tiene un R^2 de {lasso.score(X_test_scaled, y_test)}')        # Imprimir el coeficiente de determinación R^2
```

→ El modelo Lasso tiene un MSE DE :705.556959159372
El modelo Lasso tiene un R^2 de -1.1632600030786477e-05

Métricas

Obtenemos un `MSE` de 705.55 y un `R2` de -0.00001. Son prácticamente idénticos a los obtenidos con la Regresión Ridge. Ninguno de los modelos lineales fueron adecuados para capturar adecuadamente el comportamiento del problema. Podemos concluir que no se comportan de manera lineal.

✓ Redes Neuronales

Neural Networks

Aquí esta nuestra implementación de un modelo de Red Neuronal, uno de los algoritmos más potentes y flexibles del aprendizaje automático. Las redes neuronales son capaces de aprender relaciones no lineales y muy complejas en los datos.

✓ Preprocesamiento de los datos

Las redes neuronales son muy sensibles a la escala de los datos de entrada. Además, frameworks como PyTorch requieren que los datos estén en un formato específico llamado tensor.

- Estandarización
 - Para que el entrenamiento sea más fácil, estandarizaremos los datos de las variables continuas, de manera que tengan media 0 y desviación estandar 1.
 - Para esto, en las columnas de variables continuas usamos la media y desviación estandar de la respectiva columna

$$col := \frac{col - mean}{standard deviation}$$

Aquí abrimos el dataset limpio

```
df = pd.read_csv('Agromex_seleccionado_preprocesado.csv')
```

Esto asegura que todas las características contribuyan de manera equilibrada al aprendizaje. Vamos a estandarizar las columnas numéricas (Anio, Cosechada, etc.)

Para ello, se utiliza StandardScaler de la librería scikit-learn, y se aplica únicamente a las columnas cuantitativas seleccionadas.

```
# Selección de las columnas a estandarizar
columnas_cuantitativas = ['Anio', 'Cosechada', 'Siniestrada', 'Precio', 'Valorproduccion']

# Crear un objeto StandardScaler
scaler = StandardScaler()

# Aplicar la estandarización a las columnas seleccionadas
df[columnas_cuantitativas] = scaler.fit_transform(df[columnas_cuantitativas])
```

A diferencia de los modelos anteriores, aquí usaremos una división en tres conjuntos:

- Entrenamiento (70%): Para entrenar el modelo y ajustar sus pesos.
- Validación (15%): Para evaluar el rendimiento del modelo al final de cada época de entrenamiento y ajustar los hiperparámetros (como la arquitectura de la red). Nos ayuda a prevenir el sobreajuste.
- Prueba (15%): Para la evaluación final y objetiva del mejor modelo obtenido.

La división se hace de forma secuencial utilizando índices del DataFrame porque están repartidos a lo largo del tiempo y queremos mantener el orden.

```
train_size = int(len(df) * 0.7)
val_size = int(len(df) * 0.15)
test_size = len(df) - train_size - val_size

train = df.iloc[:train_size]
val = df.iloc[train_size:train_size+val_size]
test = df.iloc[train_size+val_size:]
```

✓ Convertir los datos a tensores y prepararlos para alimentar la red

PyTorch no trabaja directamente con DataFrames de pandas. Por ello, crearemos una clase (*MyDataset*) que nos prepara los datos para alimentar la red neuronal, convirtiéndolos a parejas ordenadas de tensores contenido los atributos y la variable objetivo. Sus parámetros son: el dataset df y el nombre de la columna objetivo en el dataset. Luego, usamos DataLoader para agrupar estos tensores en lotes (batches), lo cual hace el proceso de entrenamiento más eficiente.

```
class MyDataset():

    def __init__(self,df,target_column):
        y = df[target_column].astype(np.float32).values
        X = df.drop(target_column, axis=1).astype(np.float32).values
        self.X=torch.tensor(X,dtype=torch.float32)
        self.y=torch.tensor(y,dtype=torch.float32)

    def __len__(self):
        return len(self.y)

    def __getitem__(self,idx):
        return self.X[idx],self.y[idx]
```

Usar la clase MyDataset para preparar cada conjunto en forma de tensores

```
train_sec=MyDataset(train,'Rendimiento')
test_sec=MyDataset(test,'Rendimiento')
val_sec=MyDataset(val,'Rendimiento')
```

En esta celda se crean los DataLoader para los subconjuntos de entrenamiento, validación y prueba. Un DataLoader permite cargar los datos en pequeños lotes (batch_size) durante el entrenamiento del modelo, lo que es más eficiente en memoria y procesamiento. En este caso, los datos no se mezclan y se especifica un tamaño de lote diferente para cada conjunto.

```

train_data=DataLoader(
    train_sec,
    batch_size=2,
    shuffle=False,
)
test_data=DataLoader(
    test_sec,
    batch_size=3,
    shuffle=False,
)
val_data=DataLoader(
    val_sec,
    batch_size=3,
    shuffle=False,
)

```

En esta celda se imprime el primer lote del conjunto de prueba (test_data) utilizando un ciclo for. Se muestra la forma (shape) de los datos y las etiquetas, así como su contenido, lo cual es útil para verificar que los DataLoader estén generando los lotes correctamente. El ciclo se rompe después del primer lote con break para evitar imprimir todo el conjunto.

```

for i, (data, labels) in enumerate(test_data):
    print(data.shape, labels.shape)
    print(data,labels)
    break;
→ torch.Size([3, 52]) torch.Size([3])
tensor([[-0.1944, -0.1092, -0.0715,  1.0356, -0.1657,  0.0000,  0.0000,  0.0000,
         0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
         0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
        1.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
        0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  1.0000,
        0.0000,  1.0000,  0.0000,  0.0000,  0.0000,  1.0000,  0.0000,
        0.0000,  0.0000,  0.0000],

[-0.1944, -0.1118, -0.0715, -0.3331, -0.1710,  0.0000,  0.0000,  0.0000,
         0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
         0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
        1.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
        0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  1.0000,
        1.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  1.0000,
        0.0000,  0.0000,  0.0000],

[-0.1944, -0.0978,  0.0837, -0.3331, -0.1578,  0.0000,  0.0000,  0.0000,
         0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
         0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
        1.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
        0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  1.0000,
        0.0000,  1.0000,  0.0000,  0.0000,  0.0000,  1.0000,  0.0000,
        0.0000,  0.0000,  0.0000]])) tensor([0.7500, 2.0000, 1.1000])

```

▼ Estructura de la red neuronal

Se define la arquitectura de una red neuronal simple usando PyTorch. La red tiene una capa oculta totalmente conectada (fc1) con un número de neuronas definido por hidden_size, seguida por una función de activación sigmoide para introducir no linealidad. Finalmente, la salida pasa por otra capa lineal (fc2) que produce un único valor, ideal para tareas de regresión o clasificación binaria. El método forward define cómo se propaga la información a través de la red.

```

#Definir la arquitectura de la red neuronal.
class Net(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.sigmoid = nn.Sigmoid()
        self.fc2 = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x = self.fc1(x) # Capa oculta
        x = self.sigmoid(x) # Función de activación sigmoide
        x = self.fc2(x) # Capa de salida
        return x

```

Se verifica si hay una GPU disponible para usar con PyTorch. Usar una GPU puede acelerar significativamente el entrenamiento de modelos. El resultado es un valor booleano que indica si la GPU está lista para usarse o no.

```
gpu_avail = torch.cuda.is_available()
print(f"Is the GPU available? {gpu_avail}")
```

→ Is the GPU available? True

Se define el dispositivo donde se ejecutarán los cálculos: si hay una GPU disponible se usa "cuda", y en caso contrario se usa la CPU ("cpu"). Esto permite que el código sea flexible y aproveche la aceleración por hardware cuando esté disponible.

```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print("Device", device)
```

→ Device cuda

En esta celda se crea una instancia del modelo Net con 52 variables de entrada y 10 neuronas en la capa oculta. La selección de 10 neuronas y una tasa de aprendizaje (`lr`) de 0.01 para el optimizador Stochastic Gradient Descent (`SGD`) fue resultado de múltiples pruebas.

Se realizaron más de 15 experimentos variando la cantidad de neuronas entre 5 y 20, y ajustando la tasa de aprendizaje entre 0.0001 y 0.05.

Tras evaluar el rendimiento en el conjunto de validación, se encontró que esta combinación ofrecía un buen balance entre precisión y velocidad de convergencia. Finalmente, se establece la función de pérdida como el error cuadrático medio (`MSE`), adecuada para problemas de regresión.

```
model=Net(52,15) #El 52 es el número de variables de input, lo tomamos de una celda previa en la que vimos el tamaño del batch

#Definir el optimizador y la función de error (SGD es Stochastic Gradient Descent, lr es learning rate, que es el paso que da llamado -> "alp
optimizer= torch.optim.SGD(model.parameters(), lr=0.01)
criterion= torch.nn.MSELoss()
```

✓ Entrenando la red y guardando el mejor modelo

A continuación definimos la función de entrenamiento.

```
#pasamos el modelo al dispositivo GPU
model.to(device)
def train_model(model,optimizer,loss_module,train_loader,valid_loader,num_epochs):

    valid_loss_min = np.inf #Vamos a encontrar el menor valor de error de validación. Por eso la inicializamos como 'infinito'

    for i in range(num_epochs):
        model.train() #ponemos el modelo en modo entrenamiento. Es importante en otras arquitecturas como redes convolucionales.
        train_loss = 0.0
        valid_loss = 0.0

        #Completar el código a continuación
        for data, target in train_loader:

            # mover los tensores de atributos y etiquetas al dispositivo GPU

            data = data.to(device)
            target = target.to(device)
            # Reiniciar los gradientes
            optimizer.zero_grad()
            # forward pass: calcular la salida para los datos de entrada..
            output = model(data)
            # calculate the batch loss
            loss = loss_module(output, target)
            # backpropagation: cálculo de gradientes
            loss.backward()
            # actualizar los parámetros
            optimizer.step()
            # actualizar la cuenta de costos a lo largo de los lotes
            train_loss += loss.item()*data.size(0)

        # for data,labels in testloader:

            train_loss = train_loss/len(train_loader.dataset)
```

```
model.eval() #Ponemos el modelo en modo evaluación.
```

```
#for param in model.parameters():
# print(param.data)
# vamos a evaluar el modelo entrenado, calculando predicciones con el conjunto de validación
for data,target in valid_loader:
    data=data.to(device)
    target=target.to(device)
    output=model(data)
    loss= criterion(output, target)
    valid_loss += loss.item()*data.size(0)
valid_loss = valid_loss/len(valid_loader.dataset)

#imprimir estadísticas de entrenamiento y validación
print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
    i, train_loss, valid_loss))

#Guardamos el modelo con el menor error de validación.
if valid_loss <= valid_loss_min:
    print('Validation loss decreased {:.6f} --> {:.6f}). Saving model ...'.format(
        valid_loss_min,
        valid_loss))
    torch.save(model.state_dict(), 'model_proy.pt')
    valid_loss_min = valid_loss
```

El proceso de entrenamiento se realiza a lo largo de varias épocas (en este caso, 15). En cada época, el modelo procesa todos los lotes de datos de entrenamiento, calcula la pérdida y el optimizador actualiza los pesos. Al final de cada época, se evalúa el modelo con el conjunto de validación.

```
train_model(model, optimizer, criterion, train_data, val_data, 15)
```

```
→ /usr/local/lib/python3.11/dist-packages/torch/nn/modules/loss.py:610: UserWarning: Using a target size (torch.Size([2])) that is different from input size (torch.Size([3])).  F.mse_loss(input, target, reduction=self.reduction)
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/loss.py:610: UserWarning: Using a target size (torch.Size([3])) that is different from input size (torch.Size([2])).  F.mse_loss(input, target, reduction=self.reduction)
Epoch: 0      Training Loss: 495.933870      Validation Loss: 969.670545
Validation loss decreased (inf --> 969.670545). Saving model ...
Epoch: 1      Training Loss: 512.549267      Validation Loss: 1086.987169
Epoch: 2      Training Loss: 528.647322      Validation Loss: 1044.721617
Epoch: 3      Training Loss: 539.324961      Validation Loss: 1013.162273
Epoch: 4      Training Loss: 540.503567      Validation Loss: 1024.609343
Epoch: 5      Training Loss: 521.779492      Validation Loss: 980.774489
Epoch: 6      Training Loss: 517.355792      Validation Loss: 1017.480664
Epoch: 7      Training Loss: 516.738657      Validation Loss: 1019.132414
Epoch: 8      Training Loss: 556.687285      Validation Loss: 1017.277345
Epoch: 9      Training Loss: 564.246188      Validation Loss: 1028.133138
Epoch: 10     Training Loss: 555.492495      Validation Loss: 1019.256004
Epoch: 11     Training Loss: 549.380887      Validation Loss: 1002.022990
Epoch: 12     Training Loss: 552.435779      Validation Loss: 1029.989478
Epoch: 13     Training Loss: 543.139666      Validation Loss: 1029.898143
Epoch: 14     Training Loss: 544.396050      Validation Loss: 988.868553
```

Cargamos el mejor modelo obtenido del entrenamiento.

```
model.load_state_dict(torch.load('model_proy.pt'))
```

```
→ <All keys matched successfully>
```

Se recorren los parámetros del modelo que requieren ser actualizados durante el entrenamiento (aquellos con `requires_grad=True`). Para cada uno, se imprime su nombre y sus valores actuales. Esto es útil para inspeccionar los pesos y sesgos de las diferentes capas del modelo antes o durante el proceso de entrenamiento.

```
for name, param in model.named_parameters():
    if param.requires_grad:
        print (name, param.data)
```

```
fc1.weight tensor([[-5.0872e-01, -9.4688e+00, -5.4463e+00, 2.2160e-01, -9.8928e+00,
 4.9772e+00, -1.1865e+00, -4.4040e-02, -5.2877e+00, -7.7712e-01,
-6.6824e+00, 2.1231e-01, 1.5366e+00, -1.9288e+00, -7.5464e-01,
1.2242e+01, -3.2326e+00, 1.0623e+00, -2.8103e+00, -6.4210e+00,
8.0853e+00, 3.9134e+00, -2.7991e+00, -9.0933e-01, -2.5045e-01,
2.4489e+00, 2.1266e+00, -1.4529e+00, -4.5368e+00, -3.3159e+00,
-5.7259e+00, -1.3280e+00, -5.0527e+00, -1.3695e+00, -3.1073e+00,
-5.0607e+00, -1.9785e+00, -1.1330e+01, -4.5540e+00, -1.3308e+01,
-1.1689e+01, -1.7533e+01, 3.8226e+00, -2.0594e+00, -5.5879e+00,
-1.6917e+01, -1.5584e+01, -6.9581e-02, -6.8012e+00, 1.8984e+01,
5.0735e+00, -1.0203e+01],
[ 1.6929e+01, -4.4975e+00, 1.0308e-01, 6.7599e+00, 3.8678e+00,
2.0167e+00, -1.5883e+00, 1.7432e+00, -4.1254e+00, -3.9094e+00,
-5.8116e+00, -5.2235e-01, 2.5807e+00, 2.1634e+00, -1.7437e+00,
9.6140e+00, -9.2727e+00, 5.8694e+00, -2.7567e+00, -1.9838e+01,
-1.2869e+00, 6.1357e+00, -1.0013e+00, -2.3159e+00, 2.4848e+00,
1.0124e+01, 2.3039e+01, -1.1763e+00, -5.0909e+00, -5.4188e+00,
-1.4964e+00, -4.5463e+00, -5.5762e+00, -3.0133e+00, -8.7843e+00,
-9.7101e+00, 4.6235e+00, -2.3870e+01, -8.0127e+00, 2.4894e+00,
-6.9035e+00, -2.2335e+01, -3.7417e+00, 4.9084e+00, -1.4582e+00,
-2.9276e+01, -3.1407e+01, -2.7938e+00, 3.9415e+00, 2.1595e+01,
1.5389e+01, -6.2677e+00],
[ 1.7742e+00, -1.3954e+00, 5.9536e-01, -2.7519e+00, 3.5259e-01,
-3.0705e+00, -1.5750e+00, -3.0061e-01, -2.1465e+00, -2.4826e-01,
-3.2008e+00, 2.4144e-01, 6.9778e-01, -1.5585e+00, -2.2891e-02,
1.3688e+00, -8.1480e-01, 6.6951e+00, -2.0091e-01, -2.2132e+00,
7.9345e-01, -3.2992e-01, -9.1886e-01, 2.8862e+00, -3.3980e-02,
-9.6046e-03, -3.1998e+00, 5.7954e-01, -1.6318e+00, -1.6348e+00,
-1.6741e-01, -8.0356e-01, -2.7343e+00, 1.7350e-01, 4.3795e-01,
-1.5372e+00, 1.5185e-02, -9.1239e+00, 1.6126e-01, -4.9355e+00,
-6.7068e+00, -7.2375e+00, -4.2184e+00, -6.9809e-02, -4.2419e+00,
-1.0529e+01, -5.6677e+00, 1.0528e-01, -1.6424e+00, 1.0015e+01,
2.3881e+00, -4.7333e-02],
[ 6.4368e+00, 5.1840e-01, 1.3701e+00, -2.2707e+00, 1.8970e+00,
-8.4885e-02, -3.0965e-01, -7.6405e-01, -3.7861e+00, -1.4937e+00,
-4.6067e+00, 1.2312e-01, 3.1791e-01, -2.1337e+00, -1.3084e+00,
2.5269e+00, -2.0244e+00, 3.4666e+00, -4.2314e-01, -5.3050e+00,
1.9853e+01, 1.6061e+00, -2.0060e+00, -5.8437e-01, -3.8185e+00,
7.8019e+00, -2.0179e+00, -1.4781e-01, -7.4809e+00, -2.2907e+00,
-2.8552e+00, -9.7237e-01, -4.2514e+00, -1.0336e+00, -4.2003e-01,
-9.6864e+00, -1.8207e+00, -1.7202e+01, -2.6547e+00, -5.7612e+00,
-1.2628e+01, -1.2979e+01, -2.5795e+00, -1.1541e+00, -6.7583e+00,
-1.2319e+01, -1.0836e+01, -1.8362e+00, -3.4376e+00, 1.6936e+01,
-1.4120e+00, -1.8964e+00],
[ 2.6770e+00, -8.1119e-01, 3.5235e-01, -2.6112e-01, 1.4790e-01,
-3.8227e-01, -8.6586e-01, -1.5117e+00, -3.6715e+00, -3.4800e-02,
-1.4220e+00, -1.3534e-01, 4.2906e-01, -1.8865e+00, 8.4681e-02,
4.8780e+00, -1.9083e+00, 1.0689e+00, 5.9281e-01, -1.8156e+00,
1.7461e-01, -1.4846e+00, 1.5649e+00, 1.1020e+00, 6.5844e-01,
-7.4836e-01, 1.1007e+00, -1.6884e+00, 4.1031e-01, -2.2198e+00,
-3.9650e-01, -1.7165e+00, 1.4953e-02, 2.4893e-01, -6.0428e-02,
-1.1845e+00, 1.1708e-01, -3.2659e+00, -1.2471e+00, -5.9842e+00,
-4.7070e+00, -5.8649e+00, -2.0864e+00, -1.8311e+00, -3.7944e+00,
-7.1792e+00, -2.1826e+00, -1.1524e+00, -7.8926e-01, 8.0980e+00,
-3.1939e-01, 1.1017e+00],
[ 1.1965e+01, -3.7483e-01, -1.1094e+00, -4.4438e+00, 1.1786e-02,
-7.8399e-01, -1.4958e+00, -2.6235e+00, -1.2660e+00, -5.1271e+00,
-5.4848e+00, -1.3211e-01, -4.3941e-01, -2.7757e+00, -2.5757e+00,
```

▼ Métricas

El resultado final es una Pérdida de Prueba (Test Loss) de 788. Este valor es el MSE final del modelo. Si lo comparamos con el MSE de los otros modelos, es evidente que esta configuración de red neuronal, a pesar de su complejidad, no logró un buen rendimiento en este problema y fue superada por los modelos de árbol

```
test_loss=0.0

criterion= nn.MSELoss()
for data, target in test_data:
    data=data.to(device)
    target=target.to(device)
    output=model(data)
    loss= criterion(output,target)
    test_loss += loss.item()*data.size(0)
test_loss = test_loss/len(test_data.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/loss.py:610: UserWarning: Using a target size (torch.Size([3])) that is differe
return F.mse_loss(input, target, reduction=self.reduction)
```

```
Test Loss: 788.718246
```

```
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/loss.py:610: UserWarning: Using a target size (torch.Size([2])) that is different from the input size (torch.Size([1])). This is likely to result in incorrect results. You may want to use F.mse_loss(input, target, reduction='sum') instead.
  return F.mse_loss(input, target, reduction=self.reduction)
```

LightGBM

Modelo Investigado

LightGBM (Light Gradient Boosting Machine) es un framework de gradient boosting, una técnica de ensamblado muy poderosa. A diferencia de los modelos que construyen muchos árboles de forma independiente (como Random Forest), el gradient boosting los construye de forma secuencial: cada nuevo árbol se entrena para corregir los errores cometidos por el conjunto de árboles anteriores. Esto permite que el modelo se enfoque en los casos más difíciles y logre un rendimiento muy alto. LightGBM es conocido por su increíble velocidad y eficiencia, especialmente en grandes conjuntos de datos.

Preprocesamiento

Al ser un modelo basado en árboles de decisión, LightGBM no requiere que las características estén estandarizadas. Es robusto a las diferentes escalas de las variables, por lo que podemos usar los datos directamente después del preprocesamiento inicial.

Se instala la librería

```
!pip install lightgbm
```

```
Requirement already satisfied: lightgbm in /usr/local/lib/python3.11/dist-packages (4.5.0)
Requirement already satisfied: numpy>=1.17.0 in /usr/local/lib/python3.11/dist-packages (from lightgbm) (2.0.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from lightgbm) (1.15.3)
```

Importamos las bibliotecas que son necesarias para este modelo

```
from sklearn.model_selection import GridSearchCV
from lightgbm import LGBMRegressor
```

Como este modelo no requiere de estandarización, cargamos de nuevo los datos limpios

```
df = pd.read_csv('Agromex_seleccionado_preprocesado.csv')
```

Establecemos la variable objetivo y la eliminamos de las otras variables

```
X = df.drop('Rendimiento', axis=1)
y = df['Rendimiento']
```

Dividimos en conjuntos de entrenamiento y prueba

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Hiperparámetros

Para encontrar los mejores hiperparámetros, utilizamos GridSearchCV, una herramienta que automatiza y realiza una búsqueda exhaustiva sobre una "parrilla" (grid) de parámetros que le especifiquemos. Al igual que en Ridge o Lasso

```
lgbm = LGBMRegressor()

# Definir hiperparámetros a probar
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [5, 10],
    'learning_rate': [0.01, 0.1]
```

```

}

grid_search = GridSearchCV(estimator=lgbm, param_grid=param_grid, cv=3, n_jobs=-1, verbose=1)
grid_search.fit(X_train, y_train)

# Ver los mejores hiperparámetros
print("Mejores parámetros:", grid_search.best_params_)

→ Fitting 3 folds for each of 8 candidates, totalling 24 fits
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.003884 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1150
[LightGBM] [Info] Number of data points in the train set: 160084, number of used features: 52
[LightGBM] [Info] Start training from score 10.861258
Mejores parámetros: {'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 200}

```

GridSearchCV prueba cada combinación posible de estos valores usando validación cruzada y nos devuelve la que obtuvo el mejor resultado promedio. Tras la búsqueda, los mejores parámetros encontrados fueron: 'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 200

```

modelo_final = LGBMRegressor(
    n_estimators=200,
    max_depth=10,
    learning_rate=0.1
)

```

▼ Métricas

En esta celda se entrena el modelo final LGBMRegressor con los datos de entrenamiento (`X_train`, `y_train`). Luego, se usan los datos de prueba (`X_test`) para hacer predicciones y evaluar el desempeño del modelo.

```

modelo_final.fit(X_train, y_train)
y_pred = modelo_final.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Error cuadrático medio (MSE): {mse:.2f}")
print(f"Coeficiente de determinación (R²): {r2:.2f}")

→ [LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.003954 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1150
[LightGBM] [Info] Number of data points in the train set: 160084, number of used features: 52
[LightGBM] [Info] Start training from score 10.861258
Error cuadrático medio (MSE): 22.50
Coeficiente de determinación (R²): 0.97

```

Los resultados de este modelo final son:

- MSE: 24.75
- R²: 0.96

Con un R² del 96%, este modelo es el de mejor rendimiento de todos los que se probaron. Explica el 96% de la varianza en el rendimiento de los cultivos, y su MSE es el más bajo. Esto demuestra la superioridad de los algoritmos de gradient boosting como LightGBM para problemas de regresión complejos y tabulares como este.

▼ Conclusión

▼ ¿Qué notamos?

El objetivo de este proyecto fue desarrollar y evaluar una serie de modelos de aprendizaje supervisado para predecir el rendimiento de diez cultivos.

El proceso abarcó desde una rigurosa limpieza y preprocesamiento de los datos hasta el entrenamiento, ajuste de hiperparámetros y evaluación comparativa de cinco tipos de algoritmos distintos.

El análisis comparativo reveló una clara diferencia en el rendimiento entre las familias de algoritmos probadas. Los modelos lineales (Ridge y Lasso) demostraron ser insuficientes para este problema, mientras que los modelos basados en árboles de decisión mostraron una capacidad predictiva muy superior.

- LightGBM
 - 0.96. El mejor modelo por un margen significativo.
- Árbol de Decisión
 - 0.95 Excelente como modelo de referencia
- K-Vecinos más Cercanos (KNN)
 - 0.74 Rendimiento bueno.
- Red Neuronal
 - (MSE > 700) la arquitectura probada no fue la mejor y requiere un ajuste más profundo.
- Ridge / Lasso
 - -0.0001 Rendimiento muy bajo. La relación de los datos es fuertemente no lineal.

El modelo LightGBM, implementado como el algoritmo adicional investigado, fue el mejor, alcanzando un coeficiente de determinación (R^2) del 96% y el Error Cuadrático Medio (MSE) más bajo (24.75). Su éxito se

- ✓ atribuye a la técnica de gradient boosting, que construye árboles de forma secuencial, permitiendo que cada nuevo árbol corrija los errores de los anteriores y capture con gran eficacia las complejas interacciones no lineales presentes en los datos agrícolas

Haz doble clic (o ingresa) para editar