Estudio de tolerancia a errores en sensores empleados para detección de ocupación de habitantes de un recinto empleando técnicas de aprendizaje automático

Carlos Alberto Binker[12], Hugo Tantignone[12], Lautaro Lasorsa[12]
Guillermo Buranits[12], Eliseo Zurdo[12], Maximiliano Frattini[12]

[1]Departamento de Ingeniería e Investigaciones Tecnológicas
[2]Universidad Nacional de La Matanza, Florencio Varela 1903 (B1754JEC) -- San Justo, Provincia de Buenos Aires, Argentina
{cbinker, htantignone, laulasorsa, gburanits, eazurdo, mfrattini }@unlam.edu.ar

Resumen

En este paper se consideró un caso de estudio cuyo resultado consistió en la obtención de un modelo predictivo que determina la ocupación de habitantes de un recinto en función de ciertas variables de entrada, como ser, temperatura, humedad, luz, CO2 y humedad relativa, obtenidas a partir de un dataset externo de Kaggle.com. El caso de estudio pretendió determinar qué bien tolera el modelo la falla de los sensores que representan a cada una de las variables predictoras indicadas, sin que el modelo infiera la existencia de dicha falla. A tal efecto se plantearon dos escenarios. En el primer escenario, se consideró que un sensor falla de manera abrupta, es decir que esa variable que representa dicho sensor en realidad queda deshabilitada, o que está ausente. Por otro lado, en el segundo escenario, a cada sensor que falla en vez de anularlo, se le introdujo un error (un ruido), y se observó como este ruido se traslada en la predicción de la ocupación de habitantes de un recinto.

1. Introducción

1.1 Problemática a resolver

La idea inicial de este trabajo consistió en la obtención de un modelo predictivo (empleando técnicas de machine learning [1]) que, a partir de un conjunto de datos, el cual está conformado por valores de temperatura, dióxido de carbono, humedad, luz y humedad relativa que están asociados con una serie de sensores específicos para tal fin, sea capaz de predecir la ocupación o no de personas dentro de un recinto. Por lo tanto, este modelo predictivo poseerá un conjunto de variables de entrada predictoras y una variable de salida dependiente binaria que dará cuenta de la indicación de la existencia o no de personas, tal como se observa en la Figura 1.

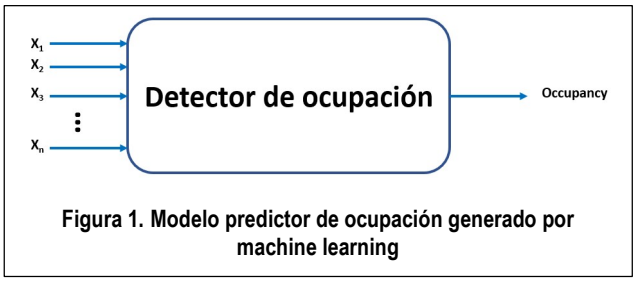

Figura 1. Modelo predictor de ocupación generado por machine learning

En este caso de estudio se plantearon dos escenarios bien diferenciados. En el primero se consideró que un sensor falla en su totalidad, es decir podemos suponer que dicha variable predictora está ausente, y por lo tanto por ser 5 (cinco) las variables predictoras, se consideró el entrenamiento de un modelo para 31 subconjuntos de dichas variables predictoras, excluyendo el conjunto vacío. Dado que el número de variables predictoras del dataset bajo estudio es relativamente pequeño, el costo computacional para este primer escenario resultó relativamente bajo y por ende perfectamente realizable. A este escenario lo bautizamos con el nombre de ***Falla Total.***

Luego en el segundo escenario, se consideró la situación en donde cada sensor no fallaba abruptamente, sino que estando el mismo presente, se le introdujo un ruido que simuló constituir un error en su comportamiento. A este escenario lo bautizamos con el nombre de ***Falla Oculta.*** Dicho ruido se introdujo sumándole a los datos originales un múltiplo de una normal con media 0 (cero) y la varianza de los datos de entrenamiento.

A nivel práctico se empleó Python [2] por ser un lenguaje altamente desarrollado para su empleo en machine learning.

1.2 Descripción de las variables de entrada y de los sensores asociados

Se analizó un dataset referido a sensores utilizados en el ámbito del IoT. El mismo fue obtenido de la página de Kaggle.com, siendo su título respectivo el siguiente: *Room Occupancy detection data (IoT sensor)* [3]. Kaggle.com es un sitio web de uso público que reúne cientos de datasets de diversas temáticas.

En nuestro caso elegimos una temática relacionada con el IoT, porque este estudio constituyó el paso preliminar para la puesta práctica del modelo obtenido en un dispositivo de borde, como ser un ESP32 o una Raspberry Pi, etc.

2. Plan de trabajo propuesto

El plan de trabajo del presente paper consistió en dos etapas, a saber:

- Falla total de los sensores
- Falla oculta de los sensores

2.1. Escenario de Falla total de los sensores

En este primer escenario, se trabajó únicamente con un subconjunto de los sensores habilitados, asumiendo que los demás sensores han fallado completamente y el que sistema es consciente de ello y por lo tanto utiliza modelos entrenados específicamente para el conjunto de sensores restantes. Esto tiene similitudes con un paper ya existente ubicado en IEEE International Conference on Machine Learning and Applications (ICMLA 2022) [4], al cual tomamos de referencia como trabajo previo de la temática planteada, pero en este caso se trabajó con modelos ligeros de tal forma que se pueda tener un modelo acorde a cada subconjunto de sensores, aprovechando que el caso de prueba es particularmente amigable a este enfoque.

2.2. Escenario de Falla oculta de los sensores

En este escenario se modeló la falla introduciendo ruido en los sensores, pero sin que el sistema se percatara de dicha situación, siendo esto algo no contemplado en el trabajo citado en el punto anterior y constituye por lo tanto *nuestro aporte al caso de estudio en cuestión*. Se buscó evaluar la robustez de los modelos originales frente a la introducción de ruido en los datos de validación.

3. Modelos estadísticos a emplear

3.1. Comparativa de los modelos

Se compararán diversos modelos de aprendizaje automático disponibles en la biblioteca *sklearn* [5]. Los mismos son:

- Regresión Logística [6]
- Random Forest [7]
- SVM (Support Vector Machines) [8]
- Gradient Boosting [9]

3.2. Explicación sucinta de los modelos

3.2.1. Regresión Logística

La regresión logística consiste en modelar la probabilidad de que una observación con variables predictoras **X** pertenezca a la clase positiva (**Y=1**) como:

$$P(Y = 1|X) = sigmoide(\theta * X) = \frac{1}{1 + e^{-\theta * X}}$$

Utilizando la función de costo de entropía cruzada:

$$Costo(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log\left(h_\theta(x^{(i)})\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta(x^{(i)})\right) + \sum_{j=1}^{n} \theta_j^2$$

Donde

$$h_\theta(x) = \frac{1}{1 + e^{-\theta * X}}$$

Notar que el término $\sum_{j=1}^{n} \theta_j^2$ es un término de regularización que se agrega para evitar el sobreajuste. El peso relativo de ambas sumatorias se puede controlar con el parámetro **C** de la regresión logística, que por defecto es 1 en *sklearn*, lo que implica que ambos términos tienen el mismo peso.

Al entrenar el modelo, θ se ajusta de tal forma que minimiza el costo. Una forma de hacerlo es utilizando el método de descenso de gradiente [10]. Para predecir, predecirá **1** si $h_\theta(x) > 0.5$, ó 0 en caso contrario.

3.2.2. Random Forest

Un Bosque Aleatorio consiste en entrenar un gran conjunto de árboles de decisión, cada uno entrenado con una muestra aleatoria de los datos y de las variables predictoras. Para las predicciones se elige la categoría votada por la mayoría de los árboles. Un árbol de decisión realiza sucesivas divisiones de los datos, tomando en cada paso una decisión basada en una de las variables predictoras buscando minimizar la impureza de los nodos resultantes. La impureza se mide, por defecto en la implementación de *sklearn*, con el índice de Gini [11], que se define como:

$$Gini = 1 - \sum_{i=1}^{C} p_i^2$$

Donde **C** es la cantidad de clases y $\boldsymbol{p_i}$ es la proporción de observaciones de la clase **i** en el nodo. Notar que para este caso **C=2**. El modelo además pone cotas a la profundidad máxima de los árboles.

3.2.3. SVM (Support Vector Machines)

Algo importante de la SVM de *sklearn* es que a la clase negativa no le asigna la etiqueta **0** sino la etiqueta **-1**. Esto es importante para la formulación de la función de costo y de la predicción.

La implementación de SVM en *sklearn* utiliza la formulación de *C-Support Vector Classification* [12] que busca minimizar la función de costo:

$$\frac{1}{2}\sum_{i}^{N}\sum_{j}^{N}\alpha_i\alpha_j y_i y_j K(x_i, x_j) - \sum_{i}^{N}\alpha_i$$

Sujeto a:

$$\sum_{i}^{N}\alpha_i y_i = 0\,, \qquad con\ 0 \le \alpha_i \le C$$

Los parámetros $\boldsymbol{\alpha_i}$ son los entrenables.

Donde **K** es la función de *Kernel*, que en el caso de *sklearn* es por defecto una función de kernel radial (*Radial Basis Function* o **RBF** [13]).

$$K(x_i, x_j) = \exp\left(-\gamma \left|\left|x_i - x_j\right|\right|^2\right)$$

Notar que por defecto $C = \gamma = 1$ en *sklearn*.

3.2.4. Gradient Boosting

Gradient Boosting es, al igual que Random Forest, un método de ensamble que combina varios modelos más simples para formar uno más complejo. La diferencia es que, en vez de entrenar todos los modelos a la vez, se entrena uno a la vez, y cada modelo subsiguiente se entrena para corregir los errores del modelo anterior. Al igual que Random Forest, los modelos base son árboles de decisión con una profundidad máxima fija y que utilizan el índice de Gini para medir la impureza.

El modelo de Gradient Boosting entrena el primer modelo dando el mismo peso a todas las observaciones. Luego, para el *i-ésimo* modelo, se priorizan las observaciones que fueron mal clasificadas por el acumulado de los modelos anteriores.

4. Desarrollo de la experiencia

Se empleó para este caso de estudio el editor de código Vscode [14]. La versión de Python empleada fue la 3.11.9, creándose un *entorno virtual* [15] para tal fin, con la finalidad de establecer correctamente las dependencias de los paquetes de librería instalados para la versión de python en cuestión. Se utilizó una laptop con 64 GB de memoria RAM, micro Intel core I7 de décima tercera generación. Las extensiones principales necesarias a instalar fueron jupyter notebook [16] y WSL (Windows Subsystem Linux) [17]. Se pueden ver más detalles de la implementación en el repositorio del paper [18].

4.1. Importación de módulos para Python 3.11

En nuestro caso de estudio se utilizaron las siguientes librerías de código abierto: pandas [19], numpy [20], tqdm [21], matplotlib.pyplot [22], seaborn [23], mpl_toolkits.axes_grid1.inset_locator [24], Python.display [25], random [26], y joblib [27] .

Una vez fijadas las dependencias, antes de comenzar con la importación de los datos, se fijó una semilla para que todas las ejecuciones del notebook sean iguales. A tal efecto observar la figura 2.

```
seed = 2024

# Establecer la semilla para la generación de números aleatorios en Python
random.seed(seed)

# Establecer la semilla para numpy
np.random.seed(seed)
```

Figura 2. Fijación de una semilla

4.2. Importación de los datos

A continuación, se presenta un cuadro representativo del dataset donde constan las variables de entrada y la salida a predecir. Como puede observarse se trata de 5 (cinco) variables de entrada. Las mismas son: temperatura, humedad, Luz, dióxido de carbono y humedad relativa. Por lo tanto, la salida de nuestro modelo predictivo será una variable booleana a la que denominaremos *Occupancy*, siguiendo la nomenclatura brindada por el dataset.

Tal como puede observarse en la Figura 3, se ha eliminado el *timestamp* ya que su valor no aporta información al modelo y podría causar una fuga de información entre los datos de validación y los datos de entrenamiento.

```
data = pd.read_csv("Occupancy.csv")
# Elimino el timestamp porque no me interesa usar esta información
data = data.drop(columns=["date"])
data
```

[22]

	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
0	23.7000	26.2720	585.200000	749.200000	0.004764	1
1	23.7180	26.2900	578.400000	760.400000	0.004773	1
2	23.7300	26.2300	572.666667	769.666667	0.004765	1
3	23.7225	26.1250	493.750000	774.750000	0.004744	1
4	23.7540	26.2000	488.600000	779.000000	0.004767	1
...	...	...	...	...	...	...
20555	20.8150	27.7175	429.750000	1505.250000	0.004213	1
20556	20.8650	27.7450	423.500000	1514.500000	0.004230	1
20557	20.8900	27.7450	423.500000	1521.500000	0.004237	1
20558	20.8900	28.0225	418.750000	1632.000000	0.004279	1
20559	21.0000	28.1000	409.000000	1864.000000	0.004321	1

20560 rows × 6 columns

Figura 3. Vista del dataset a emplear en nuestro modelo predictor de ocupación de habitantes

4.3. División del dataset

Se dividirán los datos de entrada en tres subconjuntos elegidos aleatoriamente:

- *Datos de entrenamiento*: 60% del dataset original (12336 observaciones)
- *Datos de validación*: 20% del dataset original (4112 observaciones)
- *Datos de prueba*: 20% del dataset original (4112 observaciones)

Nota: Los datos de prueba fueron separados preventivamente, aunque finalmente no fue necesario su utilización en nuestro caso de estudio.

4.4. Análisis exploratorio de los datos

La primera etapa, antes de definir un modelo, es realizar un análisis exploratorio de los datos. A continuación, se muestra un gráfico muy importante que toma los datos de entrenamiento y permite ver la distribución de cada variable para los casos positivos (valor 1) y los negativos (valor 0). Ver Figura 4 para más detalle.

De cada variable se observa lo siguiente:

1. En cuanto a la *temperatura* parece que está más cálido en los momentos en los que hay gente.
2. La *humedad* no muestra grandes diferencias
3. La *luz* tiene distribuciones completamente distintas en ambos casos. Hay mucha más gente en los momentos que hay luz.
4. El *CO2* también tiene una aparente correlación positiva con la presencia de personas.
5. El *ratio de humedad* tampoco muestra diferencias tan grandes, aunque más notorias que en el nivel absoluto de humedad.

El problema que puede haber es que varios de estos factores en vez de ser consecuencia y algo que permita detectar si hay personas, sean en realidad una causa de que la gente asista o una consecuencia de una causa de que la gente asista, por ejemplo de la hora (hay más calor y luz de día que de noche).

4.5. Generación de resultados para el caso de Falla total

En esta primera parte se tomó un subconjunto de las variables predictoras. La primera prueba comparativa que se hizo consistió en tomar un subconjunto de dichas variables y utilizar modelos que sólo trabajan con esas variables predictoras. Por ende, se generaron 31 subconjuntos, donde cada subconjunto arroja la precisión (Accuracy) de la variable predictora para cada uno de los modelos propuestos en el estudio. Ver figuras 5 y 6 respectivamente.

4.6. Consideración gráfica acerca de los resultados para el caso de Falla total

Dada la complejidad de los escenarios y la cantidad de modelos a comparar, es un desafío confeccionar un gráfico que sintetice correctamente los resultados. Optaremos por utilizar una *métrica* que sintetice los resultados obtenidos por los distintos modelos en base a la distribución de la cantidad de sensores fallados. Por lo tanto, se considerarán las siguientes dos distribuciones: Binomial [28] y Poisson [29], con el fin de ilustrar con mayor claridad los resultados.

- $Binomial(p, \lambda)$: la probabilidad de que cada sensor falle es λ y hay **p** sensores. De esta forma, si los sensores activos son **p'**, la probabilidad de esa observación dado λ es $\lambda^{p'}(1-\lambda)^{p-p'}$.

- $Poisson(\lambda)$: la probabilidad de que en un período de tiempo (al momento de la medición) se distribuye como una Poisson de parámetro λ.

Por otro lado, con la finalidad de generar estas gráficas en base a estas dos distribuciones indicadas es que se han implementado dos clases, una correspondiente al caso binomial y la otra clase para el caso de la distribución de Poisson. Observar a tal efecto las figuras 7 y 8 respectivamente.

```python
class Binomial:
    def __init__(self, n, p0 = 0, p1 = (1 - 1e-4), n_points = 1000):
        self.n = n
        self.x = np.linspace(p0,p1,n_points)

    def get_x(self):
        return self.x

    def get_nombre(self):
        return "Binomial"

    def get_x_label(self):
        return "Probabilidad de fallo en un sensor"

    def get_y(self, rsubconjuntos, i):
        y = []
        tams = np.array([len(r[0]) for r in rsubconjuntos])
        resi = np.array([r[1][i][1] for r in rsubconjuntos])
        for l in self.x:
            pesos = (1-l) ** tams * l ** (self.n - tams)
            pesos = pesos / np.sum(pesos)
            y.append(np.sum(resi * pesos))
          #  y.append(np.sum(resi * (1-l) ** tams * l ** (self.n - tams)))
        return np.array(y)
```

Figura 7. Implementación de la clase binomial

```python
import math
class Poisson:
    def __init__(self, l0 = 0, l1 = 1, n_points = 1000):
        self.x = np.linspace(l0,l1,n_points)

    def get_x(self):
        return self.x

    def get_nombre(self):
        return "Poisson"

    def get_x_label(self):
        return "Esperanza de cantidad de fallos en un periodo"

    def get_y(self, rsubconjuntos, i):
        y = []
        tams = np.array([len(rsubconjuntos[-1][0]) - len(r[0]) for r in rsubconjuntos])
        resi = np.array([r[1][i][1] for r in rsubconjuntos])
        for l in self.x:
            pesos = np.exp(-l) * l ** tams / np.array([math.factorial(tam) for tam in tams])
            pesos /= np.sum(pesos)
            y.append(np.sum(resi * pesos))
        return np.array(y)
```

Figura 8. Implementación de la clase Poisson

4.7. Análisis de la precisión con un único sensor activo para el caso de Falla total

Como se ponderan igual los desempeños de escenarios con la misma cantidad de sensores funcionando, cabe la pregunta de si no habrá sensores que aporten una gran cantidad de información y que sean suficientes al modelo para predecir la presencia de personas, y por lo tanto suban significativamente el promedio de la precisión, incluso entre los escenarios con un solo sensor activo.

4.8. Generación de resultados para el caso de Falla oculta (incorporación de ruido)

Recordemos que en este escenario comparamos los modelos, pero en vez de deshabilitar algunos sensores (de una forma que es conocida por los modelos), lo que hicimos fue introducir errores en un subconjunto de los sensores (sin que los modelos lo adviertan). Para evaluar estos modelos tenemos el inconveniente de que hay 2 probabilidades distintas con las que hay que trabajar:

- La probabilidad de que un sensor falle
- La distribución de las probabilidades de los errores.

Para esto, generaremos:

- Para cada sensor, un vector de tamaño igual a la cantidad de datos de validación de valores con distribución $\boldsymbol{N(0, \sigma_e^2)}$, donde $\boldsymbol{\sigma_e^2}$ es la varianza de ese sensor en los datos de entrenamiento.
- Para distintos valores de un parámetro **s**, se evaluará al modelo con los datos de validación $\boldsymbol{X + s \cdot E}$, donde **E** es el vector de errores generados.
- Finalmente, a cada escenario se le asignará además una ponderación dependiente de la cantidad de sensores con fallas usando las mismas distribuciones que antes.

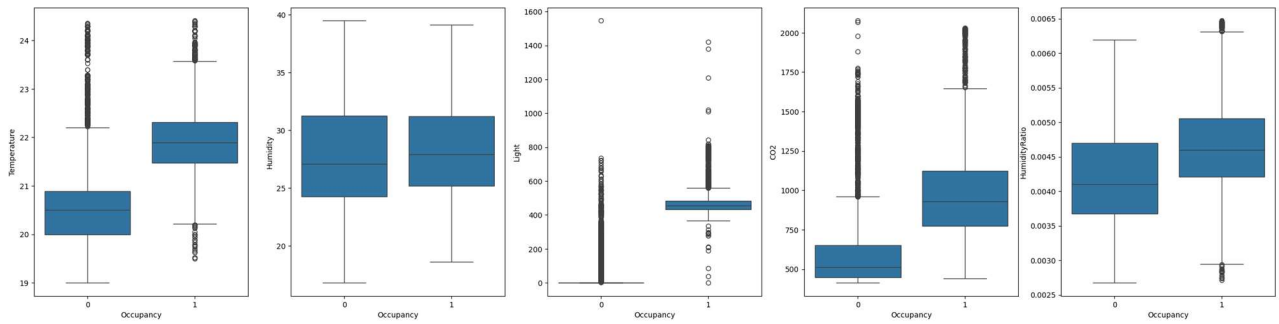

Figura 4. Distribución de las variables predictoras tomando los datos de entrenamiento

```
[(Index(['Temperature'], dtype='object'),
  [('Regresión Logística', 0.8139591439688716),
   ('Random Forest', 0.8443579766536965),
   ('SVM', 0.8523832684824902),
   ('Gradient Boosting', 0.853112840466926)]),
 (Index(['Humidity'], dtype='object'),
  [('Regresión Logística', 0.7592412451361867),
   ('Random Forest', 0.7903696498054474),
   ('SVM', 0.7592412451361867),
   ('Gradient Boosting', 0.7728599221789884)]),
 (Index(['Temperature', 'Humidity'], dtype='object'),
  [('Regresión Logística', 0.8314688715953308),
   ('Random Forest', 0.935068093385214),
   ('SVM', 0.8686770428015564),
   ('Gradient Boosting', 0.8942120622568094)]),
 (Index(['Light'], dtype='object'),
  [('Regresión Logística', 0.9854085603112841),
   ('Random Forest', 0.9824902723735408),
   ('SVM', 0.9868677042801557),
   ('Gradient Boosting', 0.9861381322957199)]),
 (Index(['Temperature', 'Light'], dtype='object'),
  [('Regresión Logística', 0.9871108949416343),
   ('Random Forest', 0.9878404669260701),
   ('SVM', 0.9868677042801557),
   ('Gradient Boosting', 0.9868677042801557)]),
 (Index(['Humidity', 'Light'], dtype='object'),
  [('Regresión Logística', 0.9854085603112841),
   ('Random Forest', 0.9878404669260701),
   ('SVM', 0.9868677042801557),
   ('Gradient Boosting', 0.9873540856031129)]),
 (Index(['Temperature', 'Humidity', 'Light'], dtype='object'),
  [('Regresión Logística', 0.9863813229571985),
   ('Random Forest', 0.9902723735408561),
   ('SVM', 0.9868677042801557),
   ('Gradient Boosting', 0.9878404669260701)]),
 (Index(['CO2'], dtype='object'),
  [('Regresión Logística', 0.7803988326848249),
   ('Random Forest', 0.8210116731517509),
   ('SVM', 0.8472762645914397),
   ('Gradient Boosting', 0.8501945525291829)]),
 (Index(['Temperature', 'CO2'], dtype='object'),
  [('Regresión Logística', 0.8183365758754864),
   ('Random Forest', 0.9382295719844358),
   ('SVM', 0.8394941634241245),
   ('Gradient Boosting', 0.915612840466926)]),
 (Index(['Humidity', 'CO2'], dtype='object'),
  [('Regresión Logística', 0.7869649805447471),
   ('Random Forest', 0.9467412451361867),
   ('SVM', 0.8385214007782101),
   ('Gradient Boosting', 0.9015077821011673)]),
 (Index(['Temperature', 'Humidity', 'CO2'], dtype='object'),
  [('Regresión Logística', 0.8180933852140078),
   ('Random Forest', 0.9744649805447471),
   ('SVM', 0.8409533073929961),
   ('Gradient Boosting', 0.9287451361867705)]),
 (Index(['Light', 'CO2'], dtype='object'),
  [('Regresión Logística', 0.9871108949416343),
   ('Random Forest', 0.9878404669260701),
   ('SVM', 0.9873540856031129),
   ('Gradient Boosting', 0.9875972762645915)]),
 (Index(['Temperature', 'Light', 'CO2'], dtype='object'),
  [('Regresión Logística', 0.9873540856031129),
   ('Random Forest', 0.9892996108949417),
   ('SVM', 0.9873540856031129),
   ('Gradient Boosting', 0.9875972762645915)]),
 (Index(['Humidity', 'Light', 'CO2'], dtype='object'),
  [('Regresión Logística', 0.9861381322957199),
   ('Random Forest', 0.9895428015564203),
   ('SVM', 0.9873540856031129),
   ('Gradient Boosting', 0.9875972762645915)]),
 (Index(['Temperature', 'Humidity', 'Light', 'CO2'], dtype='object'),
  [('Regresión Logística', 0.9875972762645915),
   ('Random Forest', 0.9902723735408561),
   ('SVM', 0.9873540856031129),
   ('Gradient Boosting', 0.9866245136186771)]),
 (Index(['HumidityRatio'], dtype='object'),
  [('Regresión Logística', 0.7592412451361867),
   ('Random Forest', 0.8139591439688716),
   ('SVM', 0.7760214007782101),
   ('Gradient Boosting', 0.7830739299610895)]),
```

Figura 5. Precisión de los distintos subconjuntos de variables predictoras para cada modelo – Subconjuntos 1 a 16

```
(Index(['Temperature', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.8139591439688716),
  ('Random Forest', 0.9345817120622568),
  ('SVM', 0.8312568093385214),
  ('Gradient Boosting', 0.8912937743190662)]),
(Index(['Humidity', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.7592412451361867),
  ('Random Forest', 0.932636186770428),
  ('SVM', 0.7592412451361867),
  ('Gradient Boosting', 0.8895914396887159)]),
(Index(['Temperature', 'Humidity', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.8314688715953308),
  ('Random Forest', 0.9396887159533074),
  ('SVM', 0.8416828793774319),
  ('Gradient Boosting', 0.8985894941634242)]),
(Index(['Light', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.9854085603112841),
  ('Random Forest', 0.9880836575875487),
  ('SVM', 0.9868677042801557),
  ('Gradient Boosting', 0.9858949416342413)]),
(Index(['Temperature', 'Light', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.9871108949416343),
  ('Random Forest', 0.9902723735408561),
  ('SVM', 0.9868677042801557),
  ('Gradient Boosting', 0.9871108949416343)]),
(Index(['Humidity', 'Light', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.9854085603112841),
  ('Random Forest', 0.9897859922178989),
  ('SVM', 0.9868677042801557),
  ('Gradient Boosting', 0.9866245136186771)]),
(Index(['Temperature', 'Humidity', 'Light', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.9863813229571985),
  ('Random Forest', 0.9902723735408561),
  ('SVM', 0.9868677042801557),
  ('Gradient Boosting', 0.9878404669260701)]),
(Index(['CO2', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.7803988326848249),
  ('Random Forest', 0.9384727626459144),
  ('SVM', 0.8385214007782101),
  ('Gradient Boosting', 0.8951848249027238)]),
(Index(['Temperature', 'CO2', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.8183365758754864),
  ('Random Forest', 0.97568093385214),
  ('SVM', 0.8402237354085603),
  ('Gradient Boosting', 0.9233949416342413)]),
(Index(['Humidity', 'CO2', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.7869649805447471),
  ('Random Forest', 0.9725194552529183),
  ('SVM', 0.8402237354085603),
  ('Gradient Boosting', 0.9260700389105059)]),
(Index(['Temperature', 'Humidity', 'CO2', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.8180933852140078),
  ('Random Forest', 0.9771400778210116),
  ('SVM', 0.8411964980544747),
  ('Gradient Boosting', 0.929717898832684 9)]),
(Index(['Light', 'CO2', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.9871108949416343),
  ('Random Forest', 0.9900291828793775),
  ('SVM', 0.9873540856031129),
  ('Gradient Boosting', 0.9871108949416343)]),
(Index(['Temperature', 'Light', 'CO2', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.9873540856031129),
  ('Random Forest', 0.9895428015564203),
  ('SVM', 0.9873540856031129),
  ('Gradient Boosting', 0.9868677042801557)]),
(Index(['Humidity', 'Light', 'CO2', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.9861381322957199),
  ('Random Forest', 0.9907587548638133),
  ('SVM', 0.9873540856031129),
  ('Gradient Boosting', 0.9868677042801557)]),
(Index(['Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio'], dtype='object'),
 [('Regresión Logística', 0.9875972762645915),
  ('Random Forest', 0.9902723735408561),
  ('SVM', 0.9873540856031129),
  ('Gradient Boosting', 0.9871108949416343)])]
```

Figura 6. Precisión de los distintos subconjuntos de variables predictoras para cada modelo – Subconjuntos 17 a 31

5. Resultados obtenidos

A continuación, se muestran los resultados de las gráficas obtenidas para el escenario de *Falla Total* (figuras 7 y 8 respectivamente), mientras que los resultados de las gráficas obtenidas para el escenario de *Falla oculta* se puedan observar en las figuras 9, 10 y 11 respectivamente. Las conclusiones acerca de los resultados obtenidos se vierten en el punto 6.

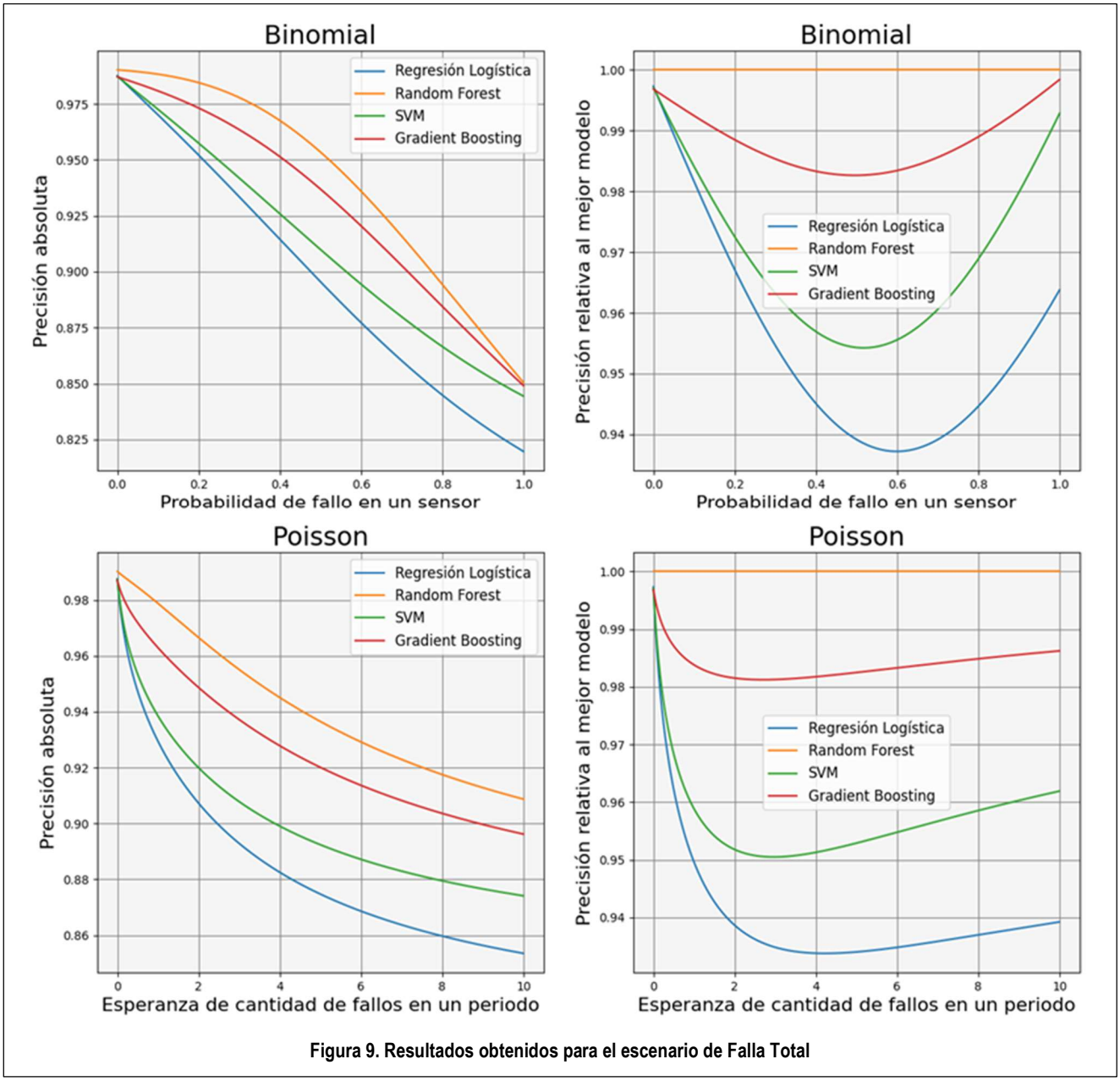

Figura 9. Resultados obtenidos para el escenario de Falla Total

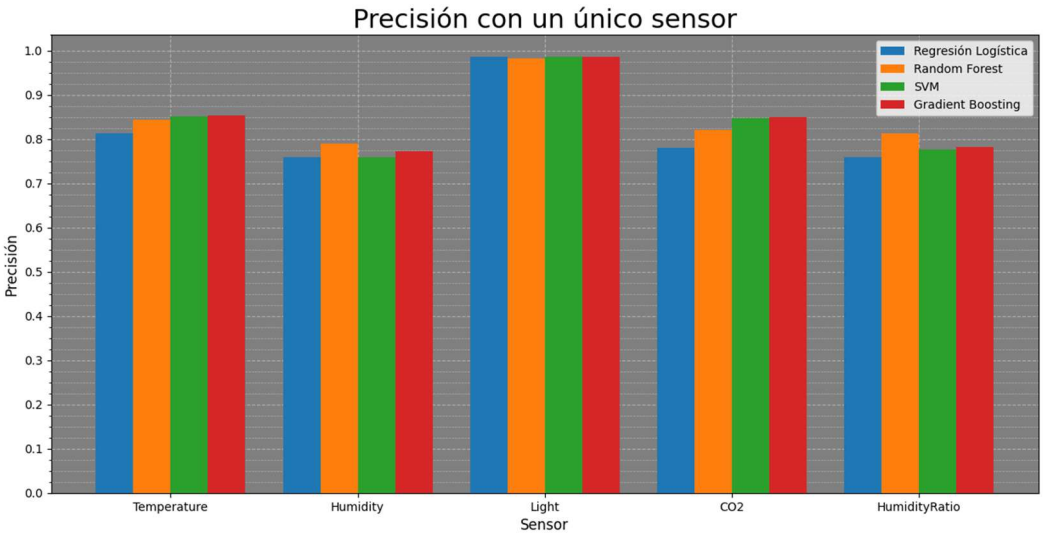

Figura 10. Resultados obtenidos para el escenario de Falla Total para un único sensor activo

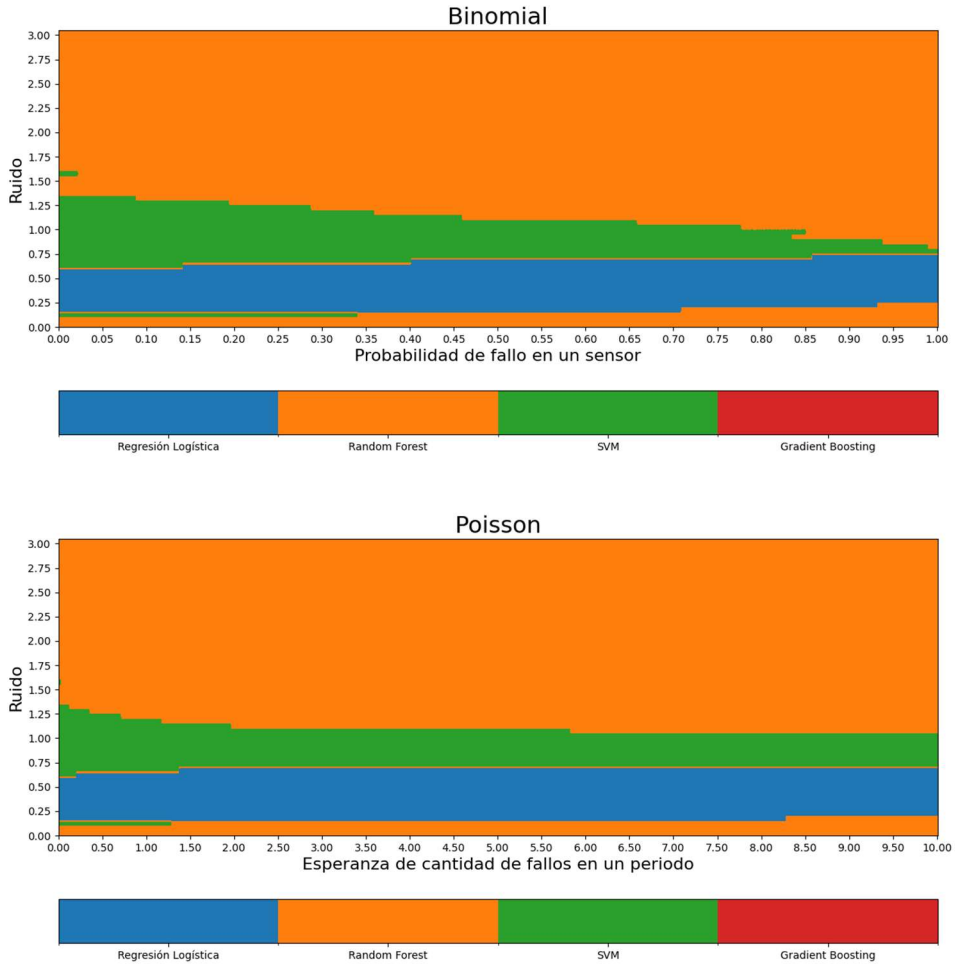

Figura 11. Resultados obtenidos para el escenario de Falla oculta considerando la falla de un único sensor

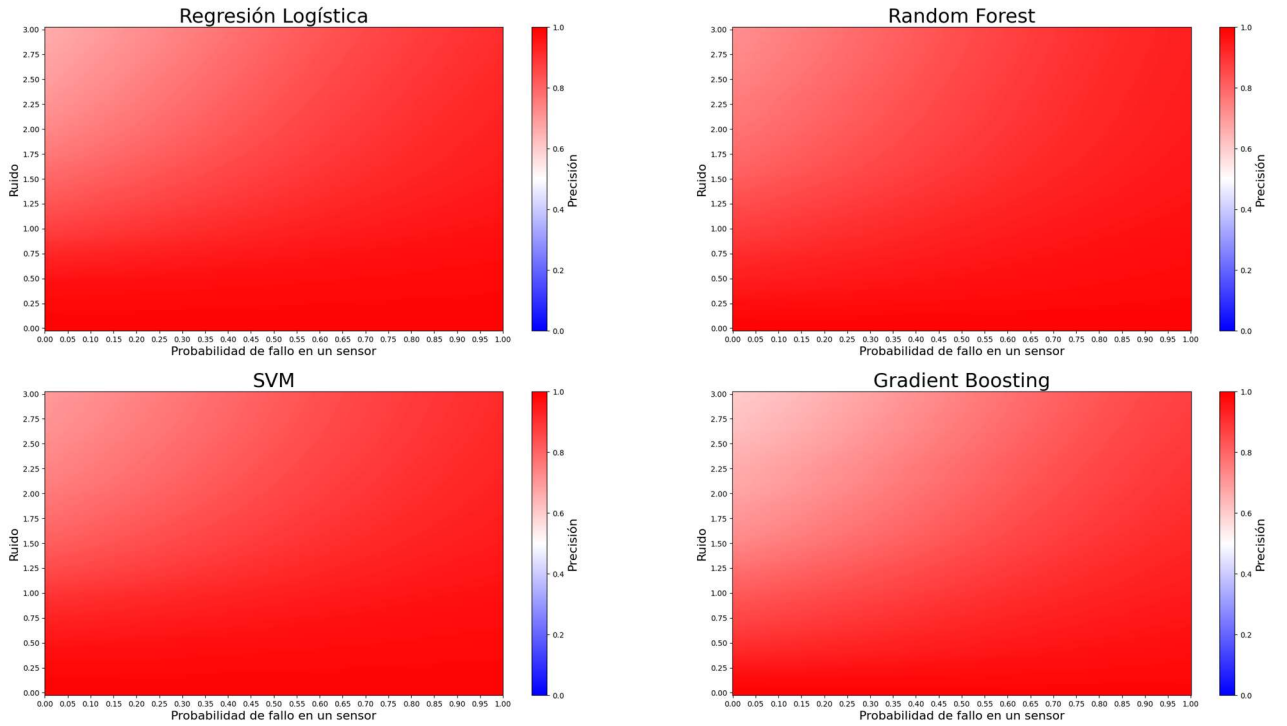

Figura 12. Resultados obtenidos en el escenario de Falla oculta para distribución Binomial

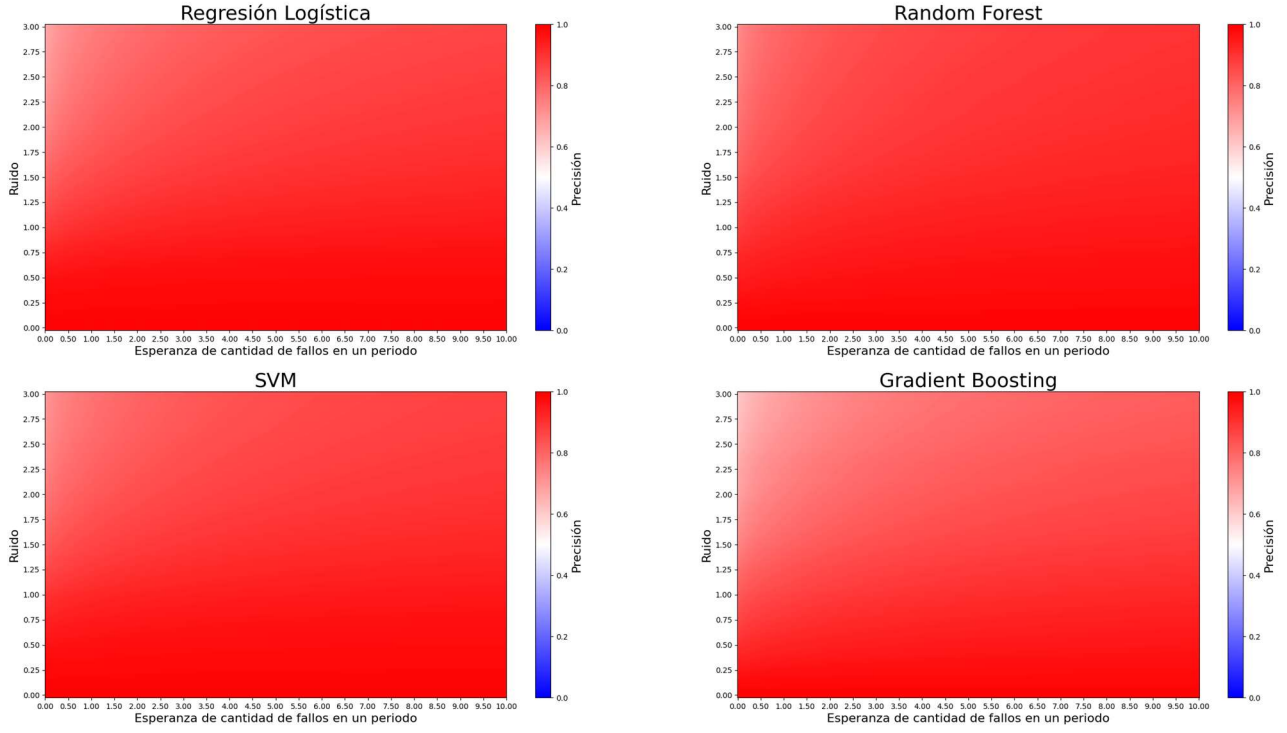

Figura 13. Resultados obtenidos en el escenario de Falla oculta para distribución Poisson

6. Conclusiones

- En el gráfico de la Figura 9, podemos observar cómo incluso cuando consideramos las fallas muy probables (siempre ignorando el caso donde fallan todos los sensores) la precisión ponderada de los modelos cae poco. Con ambas distribuciones y para cualquier valor del parámetro, el orden de los modelos, de mejor a peor, es: Random Forest, Gradient Boosting, SVM, Regresión Logística.

- En el gráfico de la Figura 10, podemos ver por qué incluso cuando el error es muy probable, los modelos se comportan bien. Notemos que, con un solo sensor, en el peor de los casos, el Random Forest está cerca del 80% de precisión, y que el sensor Light por sí solo permite a todos los modelos una precisión mayor al 97%. Aunque haya varios sensores, cada uno por separado aporta mucha información sobre la variable a predecir.

- En el gráfico de la Figura 11, podemos ver una situación que se da tanto usando la distribución Binomial como la Poisson. Para la mayoría de los valores de probabilidad de fallo y de la magnitud del ruido, el Random Forest es superior a los demás modelos. Sin embargo, hay una franja en torno a los errores bajos donde la regresión logística es llamativamente superior, y en una franja inmediatamente posterior el dominante es la SVM. No parece depender significativamente de la probabilidad de que los sensores fallen.

- Finalmente, para las gráficas de las figuras 12 y 13, podemos observar que para cualquier valor de la magnitud del ruido y de la probabilidad de error todos los valores de la precisión de los distintos modelos están entre el 75% y el 100%, lo que se ve por un color rojo sólido. Es llamativo ver que Gradient Boosting es el único que en una región tiene una precisión del orden de 60% a 80% (un rojo claro), y que es para errores poco probables, pero de gran magnitud.

7. Referencias

[1] Warden, P., & Situnayake, D. (2019). TinyML: Machine Learning with Tensorflow Lite on Arduino and Ultra-Low-Power Microcontrollers. O'Reilly Media. D.

[2] Mark Lutz. O' REILLY. Programming Python: Powerful Object-Oriented Programming 4th Edición D.

[3] https://www.kaggle.com/datasets/kukuroo3/room-occupancy-detection-data-iot-sensor

[4] https://arxiv.org/pdf/2210.02144

[5] https://scikit-learn.org/stable/

[6] An Introduction to Statistical Learning: with Applications in R" by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani.

[7] Tree-based Machine Learning Algorithms: Decision Trees, Random Forests, and Boosting Edición Kindle D.

[8] Cristianini, N., & Shawe-Taylor, J. (2000). An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods. Cambridge University Press.

[9] Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd ed.). Springer.

[10] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. ISBN: 978-0262035613.

[11] Cowell, F. A. (2011). Measuring Inequality (3rd Edition). Oxford University Press. ISBN: 978-0199594030.

[12] Chang, C. C., & Lin, C. J. (2011). LIBSVM: A Library for Support Vector Machines. ACM Transactions on Intelligent Systems and Technology (TIST), 2(3), 1-27.

[13] Buhmann, M. D. (2003). Radial Basis Functions: Theory and Implementations. Cambridge University Press.

[14] April Speight. Visual Studio Code for Python Programmers. 2020. Wiley.

[15] J. Crick and T. Mitchener, "Python Virtual Environments: A Practical Overview for Researchers and Developers," Journal of Open Research Software, vol. 7, no. 1, p. 30, 2019.

[16] https://jupyter.org/

[17] https://ubuntu.com/desktop/wsl

[18] https://github.com/carlucho1/CONAIISI-2024-2

[19] https://pandas.pydata.org/

[20] https://numpy.org/

[21] https://tqdm.github.io/

[22] https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html

[23] https://seaborn.pydata.org/

[24] https://matplotlib.org/stable/api/_as_gen/mpl_toolkits.axes_grid1.inset_locator.inset_axes.html

[25] https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.html

[26] https://docs.python.org/es/3/library/random.html

[27] https://joblib.readthedocs.io/en/stable/

[28] Blitzstein, J. K., & Hwang, J. (2014). Introduction to Probability. Chapman & Hall/CRC.

[29] Ross, S. M. (2014). Introduction to Probability Models (11th ed.). Academic Press.