

# Detección de fallas en sensores IoT mediante Machine Learning

<<Nombre y apellidos – Autor1>><<Nombre y apellidos – Autor2>>  
<<Datos de filiación – Autor 1>><<Datos de filiación – Autor 2>>  
<<Email Autor1>><<Email Autor2>>

**Nota:** Los artículos remitidos para su evaluación estarán sujetos a un proceso de evaluación por pares externos y doble ciego. Por lo tanto, para ser evaluados, no debe contener nombres de autores, filiaciones institucionales o ninguna información que pueda revelar la identidad del autor. Esta información será solicitada en la versión final, de ser aceptado el artículo. En ambas instancias de deberá enviar el trabajo en formato .pdf.

## Resumen

El objetivo de este paper consiste en el planteo de un caso de estudio cuya finalidad es construir un modelo, empleando aprendizaje automático (machine learning), que sea capaz de predecir cuándo se producen fallos o no en sensores IoT. Para ello, evaluaremos dos datasets obtenidos de la página de Kaggle.com, uno de ellos referidos a la [detección de incendios](#) y otro referido a la [detección de ocupantes](#) dentro de un recinto. La magnitud del error introducido en un determinado sensor, y para cada modelo propuesto, se cuantificará como una proporción del desvío estándar  $\hat{\sigma}$  de las mediciones originales de dicho sensor, tanto para los datos de entrenamiento como de validación cruzada. Se plantearán dos escenarios, uno creando un modelo bicategórico (que prediga fallo o no fallo en un sensor en específico) y un segundo escenario usando un único modelo multicategórico, el cual empleará  $p+1$  categorías, una para cada sensor fallado y una categoría aparte para indicar que todos los sensores funcionan correctamente.

## 1. Introducción

### 1.1 Problemática a resolver

Se analizarán dos datasets referidos a sensores utilizados en el ámbito del IoT. Los mismos fueron obtenidos de la página de [Kaggle.com](#), siendo sus títulos respectivos los siguientes: *Room Occupancy detection data (IoT sensor)* [1] y *Smoke Detection Dataset* [2], ver a continuación Figuras 1 y 2 respectivamente.

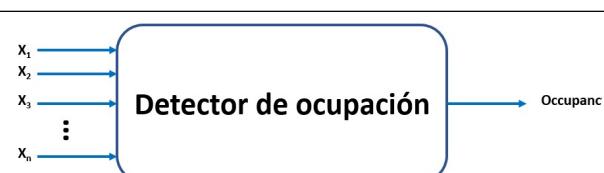


Figura 1. Modelo predictor de ocupación generado por machine learning

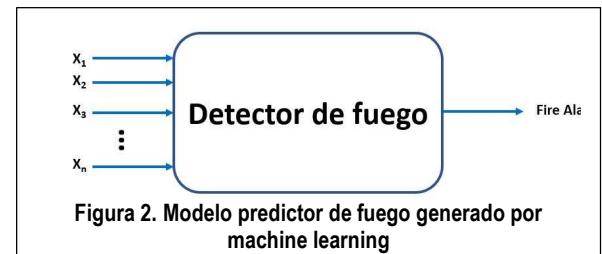


Figura 2. Modelo predictor de fuego generado por machine learning

Ahora bien, a diferencia de estudios anteriores en donde lo que importaba era efectivamente predecir una variable predictora binaria (Fire Alarm o Occupancy), en esta oportunidad, el objeto de estudio radica en la obtención de un modelo en donde sólo se tendrán en cuenta las variables de entrada correspondientes a cada uno de los sensores (por lo tanto, se suprimirán del estudio las columnas *Fire Alarm* y *Occupancy* de ambos datasets, más otras columnas también irrelevantes para el análisis, tales como fechas, índices, etc.). Cada uno de los sensores se los hará corresponder con una categoría, por lo tanto, el primer escenario a estudiar será un modelo bicategórico [3] (que predice o no fallo de un sensor en particular). Se entrenará un modelo en donde a cada sensor se le introducirá a modo de simulación de fallo una proporción del desvío estándar  $\hat{\sigma}$  de las mediciones de dicho sensor en los datos de entrenamiento (en la etapa de desarrollo de la experiencia se dará más detalle). Con posterioridad, en una segunda etapa de nuestro caso de estudio, en vez de detectar las fallas de cada sensor de forma individual haciendo foco en si ese sensor funciona correctamente o no, lo que se hará es entrenar un único modelo más complejo en donde, a partir de  $p$  sensores, se tengan  $p+1$  categorías, una para cada sensor fallado y otra categoría para indicar que todos los sensores funcionan correctamente.

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

A continuación, se presentan dos cuadros representativos de los datasets correspondientes a *Room Occupancy detection data (IoT sensor)* y de *Smoke Detection Dataset*, en donde constan las variables que se han tenido en cuenta para el análisis. Ver a continuación las Figuras 3 y 4 respectivamente.

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

20560 rows × 6 columns

Figura 3. Dataset correspondiente a Room Occupancy detection data (IoT sensor) en donde se han suprimido las columnas Occupancy y date

Temperature[C]	Humidity[%]	TVOC[ppb]	eCO2[ppm]	Raw H2	Raw Ethanol	Pressure[hPa]	PM1.0	PM2.5	NO2.5	NC1.0	NC2.5
20.000	57.36	0	400	12306	18520	939.735	0.00	0.00	0.00	0.00	0.00
20.015	56.67	0	400	12345	18651	939.744	0.00	0.00	0.00	0.00	0.00
20.029	55.98	0	400	12374	18764	939.738	0.00	0.00	0.00	0.00	0.00
20.044	55.28	0	400	12390	18849	939.736	0.00	0.00	0.00	0.00	0.00
20.059	54.69	0	400	12403	18921	939.744	0.00	0.00	0.00	0.00	0.00
...	...	...	...	...	...	...	...	...	...	...	...
18.438	15.79	625	400	13723	20569	936.670	0.63	0.65	4.32	0.673	0.015
18.653	15.87	612	400	13731	20588	936.678	0.61	0.63	4.18	0.652	0.015
18.867	15.84	627	400	13725	20582	936.687	0.57	0.60	3.95	0.617	0.014
19.083	16.04	638	400	13712	20566	936.680	0.57	0.59	3.92	0.611	0.014
19.299	16.52	643	400	13696	20543	936.678	0.57	0.59	3.90	0.607	0.014

62630 rows × 13 columns

Figura 4. Dataset correspondiente a Smoke Detection en donde se han suprimido las columnas Fire Alarm, UTC, CNT y la columna 0

Como puede observarse en la Figura 4, se trata de doce variables de entrada. Las mismas son: temperatura en °C, humedad en %, TVOC (Total Volatile Organic Compound, medido en partes por billón). Los compuestos orgánicos son sustancias químicas que contienen carbono y se encuentran en todos los seres vivos, estos compuestos orgánicos volátiles se convierten fácilmente en vapores o gases. Junto con el carbono, contienen elementos como hidrógeno, oxígeno, flúor, cloro, bromo, azufre o nitrógeno. Luego tenemos eCO2 (dióxido de carbono equivalente, medido en partes por millón), Raw H2 (gas Hidrógeno) y Raw Ethanol (gas Etano); el término Raw menciona una detección de dichos gases en crudo, es decir sin ningún filtrado o procesamiento, luego tenemos la presión en hectopascales; inmediatamente tenemos PM1.0 y PM2.5 que se trata de la detección de partículas materiales de diámetros de 1 y 2,5 micrómetros o más pequeñas aún. Estas partículas provienen de diversas fuentes, incluidos los procesos de combustión, las actividades industriales, las emisiones vehiculares y fuentes naturales como el polvo y el polen. Luego a continuación tenemos NC0.5, NC1.0 y NC2.5 que miden el número de concentración actual de partículas materiales con diámetros iguales o inferiores a 0,5, 1,0 y 2,5 micrómetros respectivamente.

## 2 Explicación de los dos escenarios propuestos y de los modelos a emplear

### 2.1 Primer escenario. Modelo bicategórico

El siguiente procedimiento de introducción de ruido (error del sensor) será igual para ambos datasets y se repetirá para cada sensor:

- Sean  $N$  la cantidad de observaciones de entrenamiento,  $M$  la cantidad de observaciones de validación cruzada y  $T$  la cantidad de observaciones de test.
- Se calculará  $\hat{\sigma}$  el estimador del desvío estándar de los datos de entrenamiento.
- Se realizarán los siguientes pasos para distintos valores de un coeficiente positivo  $\alpha$ :
- Se generarán  $2*N$  datos como ejemplos de observaciones con el sensor fallado. Estos ejemplos consistirán en la observación original  $\pm \alpha * \hat{\sigma}$ . Estas  $2*N$  observaciones, se añadirán a las  $N$  observaciones originales.
- Se entrenará el modelo descripto en el punto anterior para detectar las observaciones manipuladas con respecto a las observaciones originales.
- A continuación, se evaluará el modelo para distintos valores de  $\beta$  que indicarán la magnitud de los errores introducidos en los datos de validación cruzada.
- Para los datos de validación, se tomarán los datos de validación cruzada y nuevamente se generarán  $2*M$  datos como ejemplos de observaciones con el sensor fallado. Estos ejemplos consistirán en la observación original  $\pm \beta * \hat{\sigma}$ .
- Finalmente, se evaluará la capacidad del modelo de detectar los fallidos vs los originales.

Por lo tanto, se espera que al aumentar  $\beta$  sea más fácil para el modelo detectar los casos anómalos.

### 2.2 Segundo escenario. Modelo multicategórico

El siguiente procedimiento de introducción de ruido (error del sensor) será igual para ambos datasets y se repetirá para cada sensor:

- Sean  $N$  la cantidad de observaciones de entrenamiento,  $M$  la cantidad de observaciones de validación cruzada y  $T$  la cantidad de observaciones de test.
- Sea  $p$  la cantidad de sensores
- Se generarán  $2*p*N$  datos como ejemplo de observaciones con un sensor fallado. En cada

- observación se elegirá al azar el sensor que falla y el signo (positivo o negativo) de la falla, y se generará una observación con el valor original + signo \*  $\alpha * \hat{\sigma}_i$  (donde  $i$  es el sensor).
- d. Se le asignará a la observación la categoría  $i$  si se introdujo error en el sensor  $i$  o  $0$  si no se introdujo error.
  - e. Se entrenará un modelo para detectar la categoría de la observación.
  - f. Se realizará la generación de datos de validación cruzada creando  $M$  nuevas observaciones de validación.
  - g. Se evaluará la capacidad del modelo con dos métricas distintas. En la primera métrica, se evaluará únicamente si es correcto distinguir si la observación está fallada o si no lo está. Es decir, si detecta correctamente la categoría 0.
  - h. En la segunda métrica, se evaluará si el modelo es capaz de detectar la categoría correcta, es decir poder atribuir al sensor específico el error.

Nuevamente, se espera que al aumentar  $\beta$  sea más fácil para el modelo detectar los casos anómalos.

## 2.3 Modelos a emplear en el escenario bicategórico

### 2.3.1 Regresión logística simple

Para comparar los modelos se crearán clases que envuelvan el modelo y permitan entrenarlo y evaluarlo de forma sencilla.

En este modelo se plantea:

$$P(Y = 1|X = x) = \text{sigmoide}(\theta^T * x + \beta)$$

Las funciones sigmoides son una familia de funciones que se caracteriza por ser acotadas, diferenciables, con derivada positiva en cada punto y con un único punto de inflexión (raíz de la segunda derivada). En general, se usa una función particular de esta familia que es la función logística  $L(x)$ , de la forma:

$$L(x) = \frac{1}{1 + e^{-x}}$$

En este modelo los parámetros entrenables, es decir que se pueden modificar para adaptar el modelo a los datos de estudio, son  $\theta$  y  $\beta$ .

### 2.3.2 Modelo estadístico de detección de anomalías

Este es un modelo estadístico sencillo. Consiste en simplemente calcular la probabilidad de que la observación

sea generada por la distribución de los datos sin fallas, para lo cual se la considerará como una normal multivariada. Se fijará el umbral que maximice la precisión en los datos de entrenamiento.

Formalmente se tiene:

$$X \sim N(\mu, \Sigma)$$

Donde  $\mu$  y  $\Sigma$  son los estimadores de la media y la matriz de covarianza de los datos de entrenamiento. Se pueden acceder fácilmente a  $\mu$  y  $\Sigma$  con `df.mean()` y `df.cov()` respectivamente.

### 2.3.3 Modelo de Bosque Aleatorio (Random Forest)

Un bosque aleatorio [4] se construye agregando múltiples árboles de decisión construidos con:

- Subconjuntos aleatorios de las columnas (variables predictoras) del dataset.
- Pesos aleatorios asignados a cada una de las observaciones. (Es decir, a cada uno de estos submodelos le importan más algunas predicciones que otras).

El modelo de Bosque Aleatorio funciona agregando las predicciones de cada uno de estos árboles de decisión. En este caso, binario, la decisión es la decisión de la mayoría de los árboles.

## 2.4 Modelos a emplear en el escenario multicategórico

Convenientemente, los modelos antes definidos *Regresión Logística Scikit Learn* [5] y *Random Forest Scikit Learn* [6] pueden ser utilizados para este escenario, ya que soporta múltiples categorías de forma nativa.

### 2.4.1 Regresión logística multinomial

En este caso, para cada una de las categorías se entrena un modelo que estima la probabilidad de pertenecer a esa categoría y se tomará la categoría con mayor probabilidad como la predicción del modelo.

$$f_c(X) = \hat{P}(Y = c|X) = L(\Theta_c * X) = \frac{1}{1 + e^{-\Theta_c * X}}$$

$$\text{pred}(X) = \underset{c}{\operatorname{argmax}} f_c(X)$$

## 2.4.2 Normal multivariada multinomial

En este caso, para cada una de las categorías se entrena un modelo que estima la probabilidad de pertenecer a esa categoría y se tomará la categoría con mayor probabilidad como la predicción del modelo.

$$\text{Likelihood}_c(X) = \frac{1}{(2\pi)^n/2 |\Sigma_c|^{1/2}} e^{-\frac{1}{2}(X-\mu_c)^T \Sigma_c^{-1} (X-\mu_c)}$$

$$\text{pred}(X) = \underset{c}{\operatorname{argmax}} \text{ Likelihood}_c(X)$$

## 2.4.3 Random Forest multinomial

En este caso, al igual que para clasificación binaria, se entrenan múltiples árboles de decisión y se toma la categoría con mayor cantidad de votos como la predicción del modelo.

**Nota:** el Gradient Boosting Multinomial no se utiliza debido a que resulta muy lento para entrenar.

## 3 Desarrollo de la experiencia

### 3.1 Descripción de la plataforma de trabajo a emplear

Se empleará para este caso de estudio el editor de código Vscode. La versión de Python empleada fue la 3.11.9, creándose un entorno virtual para tal fin, con la finalidad de establecer correctamente las dependencias de los paquetes de librería instalados para la versión de python. 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 [7] y WSL (Windows Subsystem Linux) [8]. Se pueden ver más detalles de la implementación en el repositorio del proyecto [9].

### 3.2 Importación de módulos para python 3.11

En este estudio se utilizarán las siguientes librerías de código abierto: pandas [10], numpy [11], matplotlib.pyplot [12], joblib [13], clear\_output, LogisticRegression [14], multivariate\_normal [15], RandomForestClassifier [16] y tqdm [17].

### 3.3 Lectura y limpieza de los datasets. División del dataset en datos de entrenamiento, validación y prueba

A continuación, se indica como se procedió en primer

lugar a la limpieza de los datasets y luego a la separación de los datos en entrenamiento, validación y prueba, observar el detalle en la Figura 5. Como puede observarse se han suprimido las columnas *Fire Alarm* (que es una variable binaria predictora de presencia de fuego), *UTC*, *CNT* y la columna 0 de índices, siendo estas últimas tres irrelevantes para nuestro estudio. Notar que el método *dropna()* en ambos datasets elimina todas las filas que tengan faltantes de datos. Por otro lado, en cuanto a la división de los datos en entrenamiento, validación y prueba, primero el método *sample()* aleatoriza todo el dataset completo, esto se indica con *frac=1*, luego el término *int(.6\*len(humo))*, toma el 60% de los datos para entrenamiento. El segundo parámetro *int(.8\*len(humo))*, toma el 20% del dataset para validación, quedando por lo tanto el 20% final reservado para datos de prueba del modelo.

```
humo = pd.read_csv('smoke_detection_iot.csv')
humo = humo.dropna().drop(columns=[humo.columns[0], "UTC", "fire Alarm", "CNT"])

ocupacion = pd.read_csv('occupancy.csv')
ocupacion = ocupacion.dropna().drop(columns=["date", "Occupancy"])

humo_train, humo_cv, humo_test = np.split(humo.sample(frac=1), [int(.6*len(humo)), int(.8*len(humo))])
ocupacion_train, ocupacion_cv, ocupacion_test = np.split(ocupacion.sample(frac=1), [int(.6*len(ocupacion)), int(.8*len(ocupacion))])
```

Figura 5. Limpieza de los datasets humo y ocupación y su correspondiente división en datos de training, validación y test

```
from sklearn.linear_model import LogisticRegression
class ModeloRegresionLogisticaScikit:
    @staticmethod
    def nombre():
        return "Regresion Logistica"

    def __init__(self, datos_originales):
        self.modelo = LogisticRegression(max_iter=1000)

    def entrenar(self, Xs, ys):
        self.modelo.fit(Xs,ys)

    def predecir(self, Xs):
        return self.modelo.predict(Xs)
```

Figura 6. Entrenamiento, predicción y generación de la clase “Regresión logística”

```
from scipy.stats import multivariate_normal
class RedNeuronalMultivariada:
    @staticmethod
    def nombre():
        return "Red Neuronal Multivariada"

    def __init__(self, datos_originales):
        self.normal = multivariate_normal(mean=datos_originales.mean(), cov=datos_originales.cov(), allow_singular=True)

    def entrenar(self, xs, ys):
        probs = self.normal.pdf(xs)
        observaciones = list(zip(probs, ys))
        observaciones.sort(key=lambda x: x[0])
        umbral = 0
        pres = 1/3
        optimo = 1/3
        N = len(observaciones)
        for i, (prob, y) in enumerate(observaciones):
            if y == 1:
                pres += 1/N
            else:
                pres -= 1/N
            if pres < optimo:
                optimo = pres
                umbral = prob
        self.umbral = umbral
        self.optimo = optimo

    def predecir(self, xs):
        return self.normal.pdf(xs) < self.umbral
```

Figura 7. Entrenamiento, predicción y generación de la clase “Normal Multivariada”

### 3.4 Creación de clases para la comparación de los diferentes modelos bicategóricos

Para comparar los modelos se crearán clases que envuelvan el modelo y permitan entrenarlo y evaluarlo de forma sencilla, observar al respecto las Figuras 6, 7 y 8 respectivamente para mayor detalle.

```
from sklearn.ensemble import RandomForestClassifier
class ModeloRandomForestScikit:
    @staticmethod
    def nombre():
        return "Random Forest"

    def __init__(self, datos_originales):
        self.modelo = RandomForestClassifier(n_estimators=100, max_depth=4, random_state=0)

    def entrenar(self, xs, ys):
        self.modelo.fit(xs, ys)

    def predecir(self, xs):
        return self.modelo.predict(xs)
```

Figura 8. Entrenamiento, predicción y generación de la clase “Random Forest”

### 3.5 Definición de la función de evaluación para modelos bicategóricos

Lo siguiente que corresponde es definir la función que utilizaremos para evaluar los modelos definidos en la sección anterior correspondiente al escenario bicategórico. La función deberá recibir el dataset, la columna a modificar, el modelo a evaluar y las grillas de  $\alpha$  y  $\beta$ . La función devolverá un *diccionario* donde la *clave* será el valor de  $\alpha$  utilizado para el entrenamiento y el *contenido* será un diccionario de vectores con los resultados de evaluación para cada valor de  $\alpha$  y  $\beta$ . Ver a continuación figuras 9 y 10 respectivamente para mayor detalle de lo narrado.

```
def Generar(Xs, columna, delta):
    Xs_suma = Xs.copy()
    Xs_suma[columna] += delta
    Xs_resta = Xs.copy()
    Xs_resta[columna] -= delta
    ys = np.array([0]*len(Xs) + [1]*2*len(Xs))
    return pd.concat([Xs, Xs_suma, Xs_resta], ignore_index=True), ys

def Evaluar(Xs_train, Xs_cv, columna, Modelo, alfas, betas):
    resultados = dict()
    for alfa in alfas:
        modelo = Modelo(Xs_train)
        desvio = np.std(Xs_train[columna])
        Xs_train_alfa, ys_train_alfa = Generar(Xs_train, columna, desvio*alfa)
        modelo.entrenar(Xs_train_alfa, ys_train_alfa)
        resultados_alfa = list()
        for beta in betas:
            Xs_cv_beta, ys_cv_beta = Generar(Xs_cv, columna, desvio*beta)
            preds = modelo.predecir(Xs_cv_beta)
            resultados_alfa.append(np.mean(preds == ys_cv_beta))
        resultados[alfa] = np.array(resultados_alfa)
    return resultados
```

Figura 9. Función de evaluación para modelos bicategóricos (1)

```
from tqdm.notebook import tqdm
from time import time
def Evaluacion_Global(Xs_train, Xs_cv, Modelos, alfas, betas):
    Resultados = list()
    avance = tqdm(total=len(Modelos)*len(Xs_train.columns))
    for Modelo in Modelos:
        resultados_modelo = list()
        for columna in Xs_train.columns:
            t0 = time()
            resultados = Evaluar(Xs_train, Xs_cv, columna, Modelo, alfas, betas)
            t1 = time()
            print(f"Modelo {Modelo.nombre()} con {columna} en {t1-t0:.2f} segundos")
            resultados_modelo.append((resultados, t1-t0))
            avance.update(1)
        Resultados.append(resultados_modelo)
    return Resultados
```

Figura 10. Función de evaluación para modelos bicategóricos (2)

### 3.6 Definición de la función de evaluación para modelos multicategóricos

Para la evaluación de los modelos multicategóricos, se van a generar  $ratio * |X\_train|$  observaciones de entrenamiento con fallas introducidas en los sensores. En cada falla se elegirá un sensor y un signo al azar y se le sumará  $signo * \alpha * \delta_i$  a la  $i$ -ésima categoría de la observación original. Para el entrenamiento utilizamos el ratio 2\*#sensores, y para generar los datos de validación se utilizará el ratio 1\*#sensores. Esto se realiza para equilibrar las categorías. Para mayor detalle observar las figuras 11 y 12 respectivamente.

```
def Generar_Multicategoría(Xs, stds, alfa, ratio):
    array_X = [Xs]
    array_Y = [np.array([0]*len(Xs))]
    for i in range(ratio):
        Xs_i = Xs.copy()
        random = np.random.randint(0, len(Xs.columns), len(Xs))
        signo = np.random.choice([-1, 1], len(Xs))
        Xs_i.iloc[:, random] += stds[random]*alfa*signo
        array_X.append(Xs_i)
        array_Y.append((random))
    return pd.concat(array_X, ignore_index=True), np.concatenate(array_Y)

def Evaluar_Multicategoría(Xs_train, Xs_cv, Modelo, alfas, betas, ratio = 1, tqdm_bar = None):
    resultados = dict()
    for alfa in alfas:
        modelo = Modelo(Xs_train)
        desvio = Xs_train.std()
        Xs_train_alfa, ys_train_alfa = Generar_Multicategoría(Xs_train, desvio, alfa, ratio)
        modelo.entrenar(Xs_train_alfa, ys_train_alfa)
        deteccion = []
        atribucion = []
        for beta in betas:
            Xs_cv_beta, ys_cv_beta = Generar_Multicategoría(Xs_cv, desvio, beta, len(Xs_train.columns))
            preds = modelo.predecir(Xs_cv_beta)
            deteccion.append(np.mean(preds == (ys_cv_beta == 0)))
            atribucion.append(np.mean(preds == ys_cv_beta))
        resultados[alfa] = np.array((deteccion, atribucion))
    if tqdm_bar is not None:
        tqdm_bar.update(1)
    return resultados
```

Figura 11. Función de evaluación para modelos multicategóricos (1)

```
def Evaluacion_Global_Multicategoría(Xs_train, Xs_cv, Modelos, alfas, betas, ratio = 1):
    Resultados = list()
    avance = tqdm(total=len(Modelos) * len(alfas))
    for Modelo in Modelos:
        to = 0
        resultados = Evaluar_Multicategoría(Xs_train, Xs_cv, Modelo, alfas, betas, ratio = ratio, tqdm_bar = avance)
        to += 1
        print(f"Modelo {Modelo.nombre()} en {to:.2f} segundos")
        Resultados.append((resultados, to))
    return Resultados
```

Figura 12. Función de evaluación para modelos multicategóricos (2)

## 4 Generación de resultados

### 4.1 Resultados para modelos bicategóricos

Vamos a utilizar las funciones antes definidas para generar los resultados para los modelos propuestos. Como esta operación es muy costosa (pudiendo tomar horas de cómputo), vamos a almacenar los resultados mediante la librería *joblib* en el directorio apuntado por la variable *dir\_variables* para posteriormente poder recuperarlas fácilmente sin necesidad de volver a computar los resultados.

### 4.2 Gráficos para modelos bicategóricos

La función *Graficar* genera un gráfico para cada combinación de modelo y sensor, donde se muestra una serie distinta para cada valor de  $\alpha$  utilizado en el entrenamiento. En estos gráficos, el eje x son los valores de  $\beta$  utilizados en la evaluación y el eje y es la precisión del modelo en la detección de fallas. En síntesis, cada gráfico representa un modelo y un sensor. Cada serie representa un valor de  $\alpha$  utilizado en el entrenamiento y por último cada punto de la serie representa la precisión del modelo en la detección de fallas para un valor de  $\beta$  en la evaluación. Ver a continuación la Figura 13.

```
Resultados_Ocupacion_2 = joblib.load(dir_variables + "Resultados_Ocupacion.pkl")
fig, axs = Graficar("Detección de Ocupación",
                     Resultados_Ocupacion_2, Modelos, ocupacion_train,
                     alfas, betas)
fig.savefig(dir_graficos + "Resultados_Ocupacion.png")

Resultados_Humo_2 = joblib.load(dir_variables + "Resultados_Humo.pkl")
fig, axs = Graficar("Detección de Humo",
                     Resultados_Humo_2, Modelos, humo_train,
                     alfas, betas)
fig.savefig(dir_graficos + "Resultados_Humo.png")
```

Figura 13. Guardado de los gráficos en la carpeta *dir\_graficos*

### 4.3 Resultados para modelos multicategóricos

A la hora de generar los gráficos para la comparación multicategórica vamos a generar tanto gráficos para la detección de fallas como para la detección de sensores fallados. En ambos pondremos, como referencia, el resultado que obtendría un modelo que responda aleatoriamente basándose únicamente en la distribución de las categorías (es decir, tenga una probabilidad X de responder la categoría cuya proporción de la población es X).

### 4.4 Gráficos para modelos multicategóricos

Se harán dos tipos de gráficos. Uno similar al caso binario, donde cada gráfico es un modelo, cada serie es un

valor de  $\alpha$  y cada punto es la precisión del modelo en la detección de fallas para un valor de  $\beta$ . En el segundo gráfico global, cada gráfico corresponde a un valor de  $\alpha$  y cada serie corresponde a un modelo. Cada punto de la serie es la precisión del modelo en la detección de fallas para un valor de  $\beta$ . En ambos casos se hará un gráfico para la detección de fallas y otro para la atribución de las mismas.

## 5 Resultados obtenidos

A continuación, se muestran los resultados obtenidos correspondientes a:

1. Detección de ocupación para el escenario bicategórico. Ver Figura 14.
2. Detección de humo para el escenario bicategórico. Ver Figura 15.
3. Detección de ocupación para el escenario multicategórico, indicando detección y atribución para los tres modelos de análisis correspondientes a: Regresión Logística, Normal Multivariada y Random Forest. Ver Figura 16.
4. Detección de ocupación para el escenario multicategórico, indicando detección y atribución para distintos valores de  $\alpha$ . Ver Figura 17.
5. Detección de humo para el escenario multicategórico, indicando detección y atribución para los tres modelos de análisis correspondientes a: Regresión Logística, Normal Multivariada y Random Forest. Ver Figura 18.
6. Detección de ocupación para el escenario multicategórico, indicando detección y atribución para distintos valores de  $\alpha$ . Ver Figura 19.

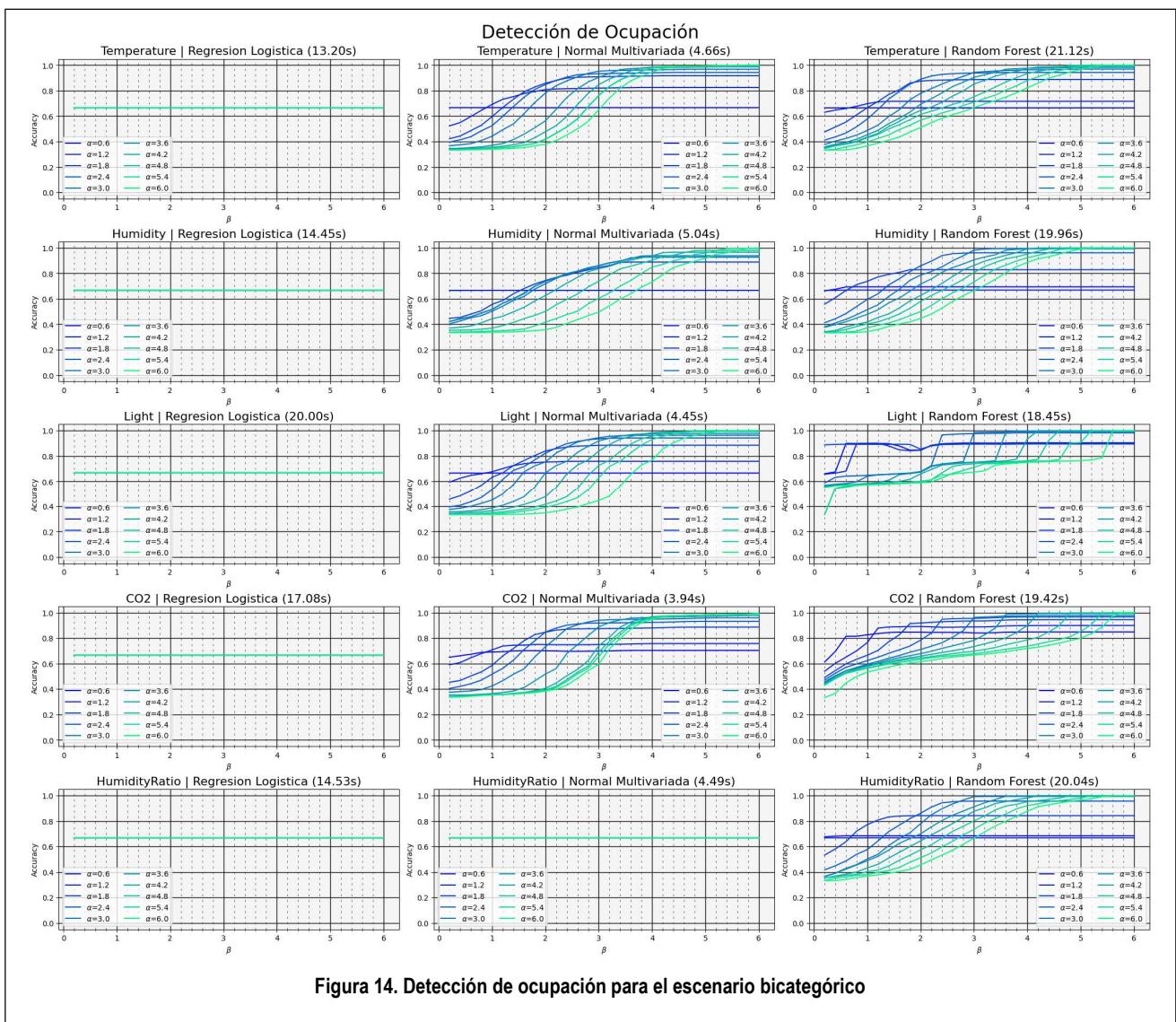


Figura 14. Detección de ocupación para el escenario bicategórico

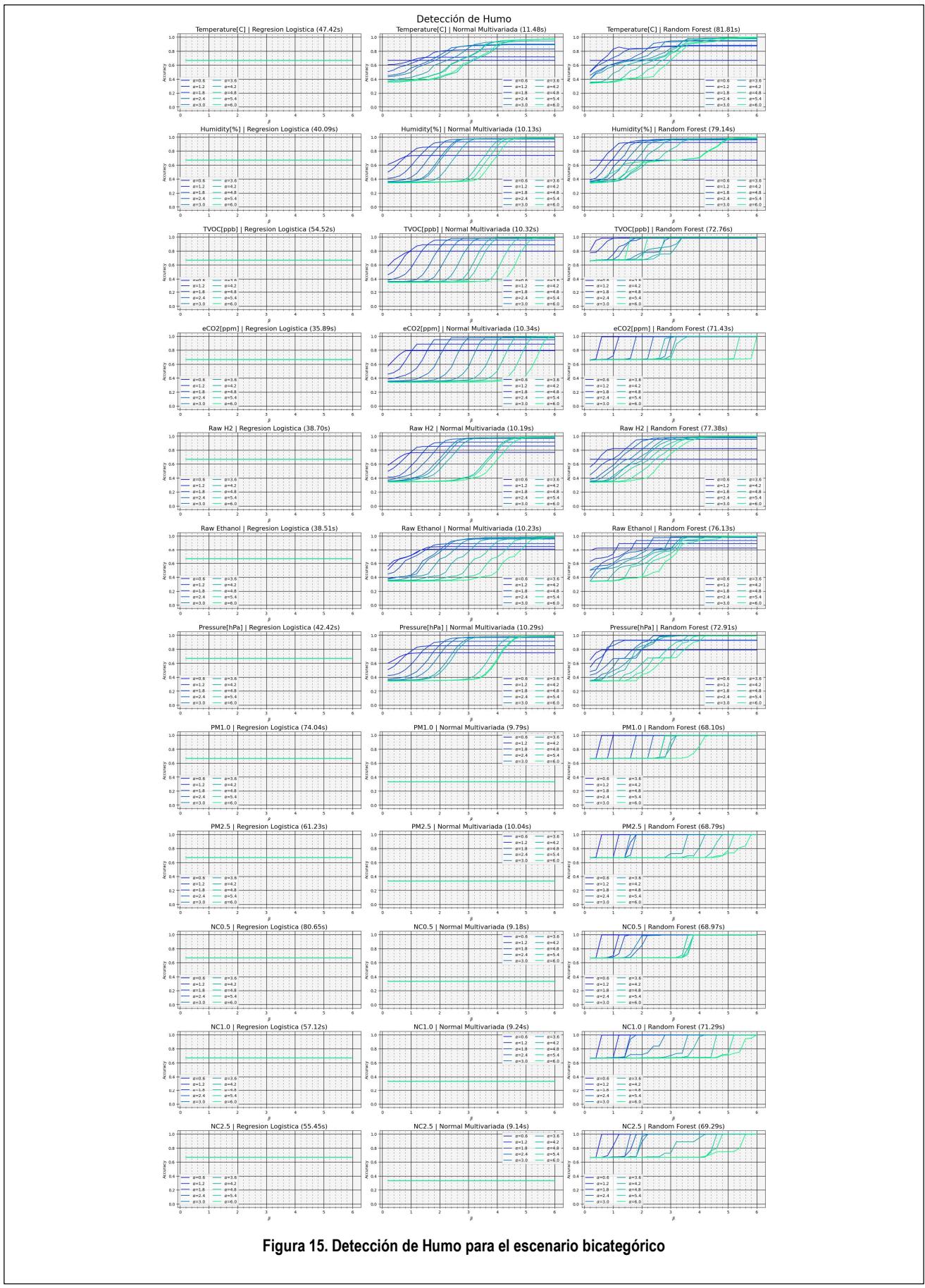
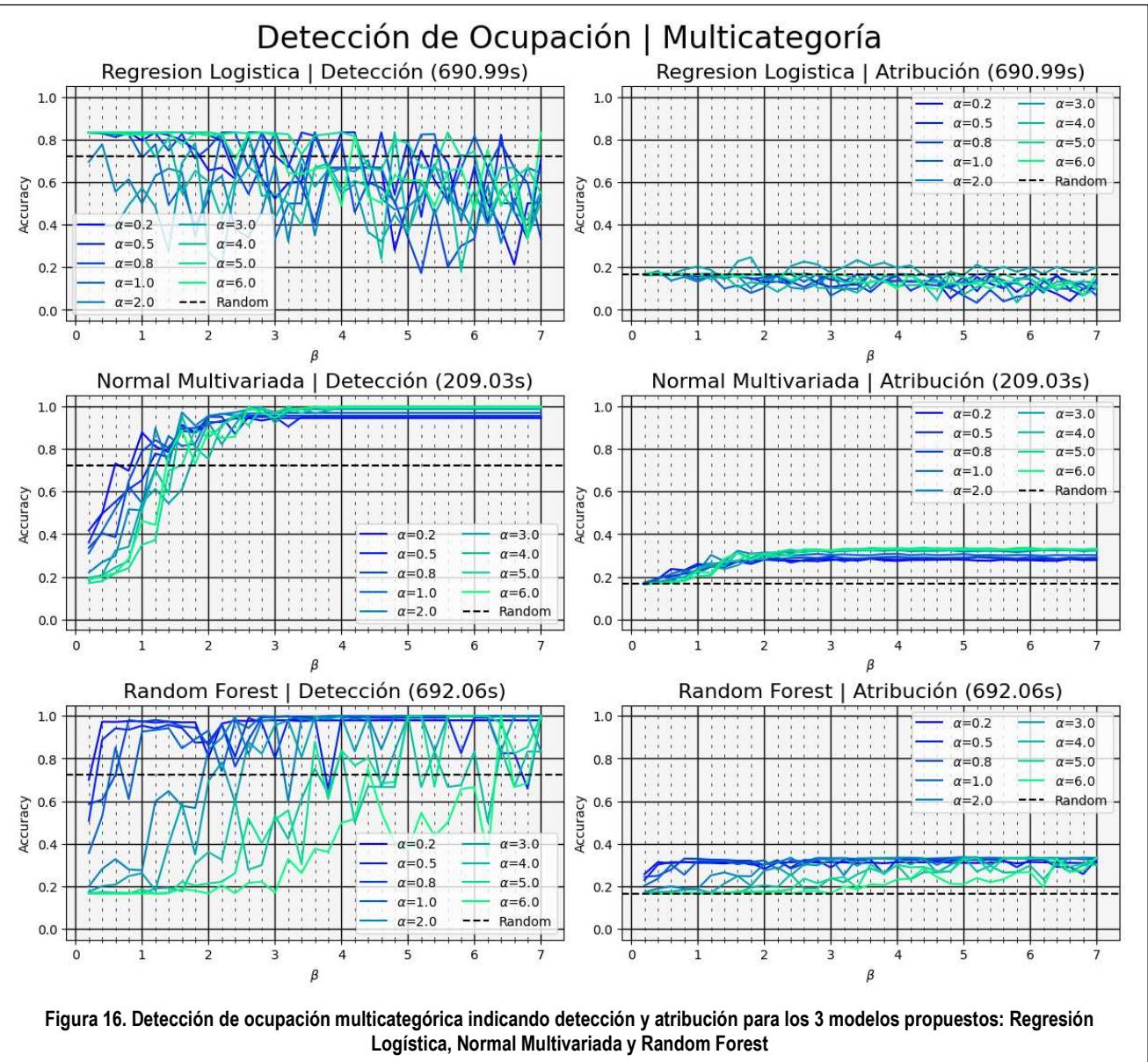
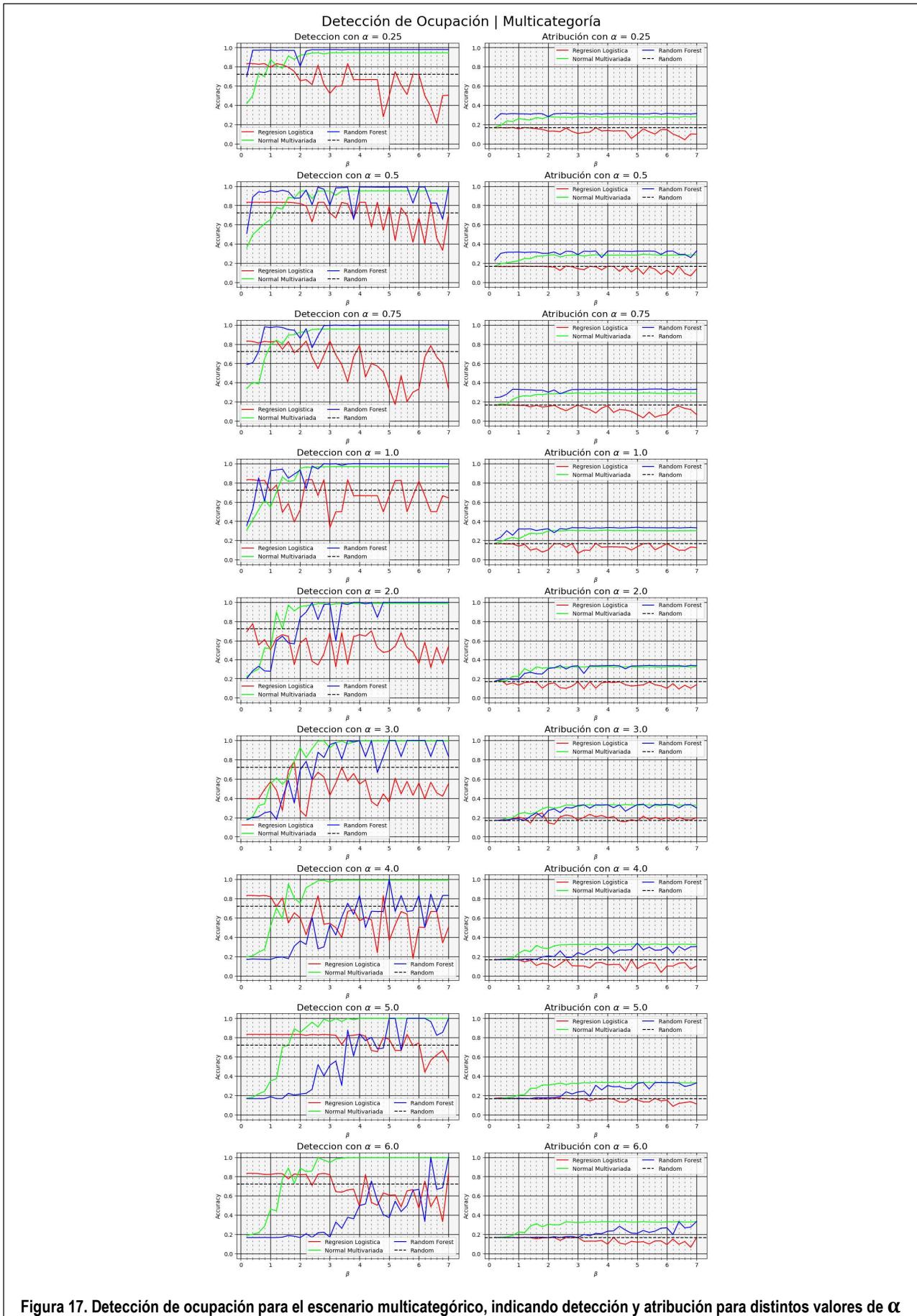


Figura 15. Detección de Humo para el escenario bicategórico



**Figura 16.** Detección de ocupación multicategórica indicando detección y atribución para los 3 modelos propuestos: Regresión Logística, Normal Multivariada y Random Forest



**Figura 17. Detección de ocupación para el escenario multicategórico, indicando detección y atribución para distintos valores de  $\alpha$**

## Detección de Humo | Multicategoría

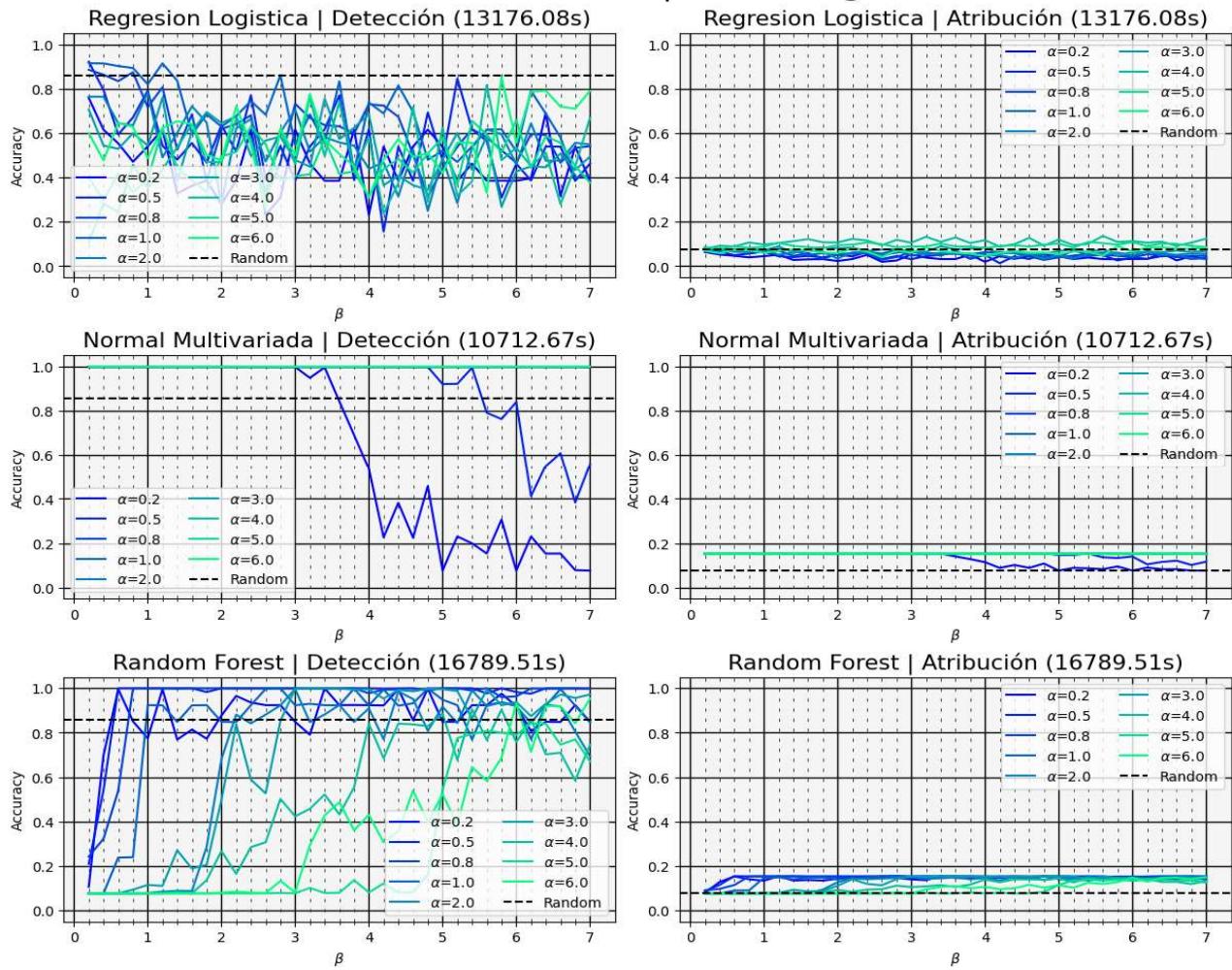
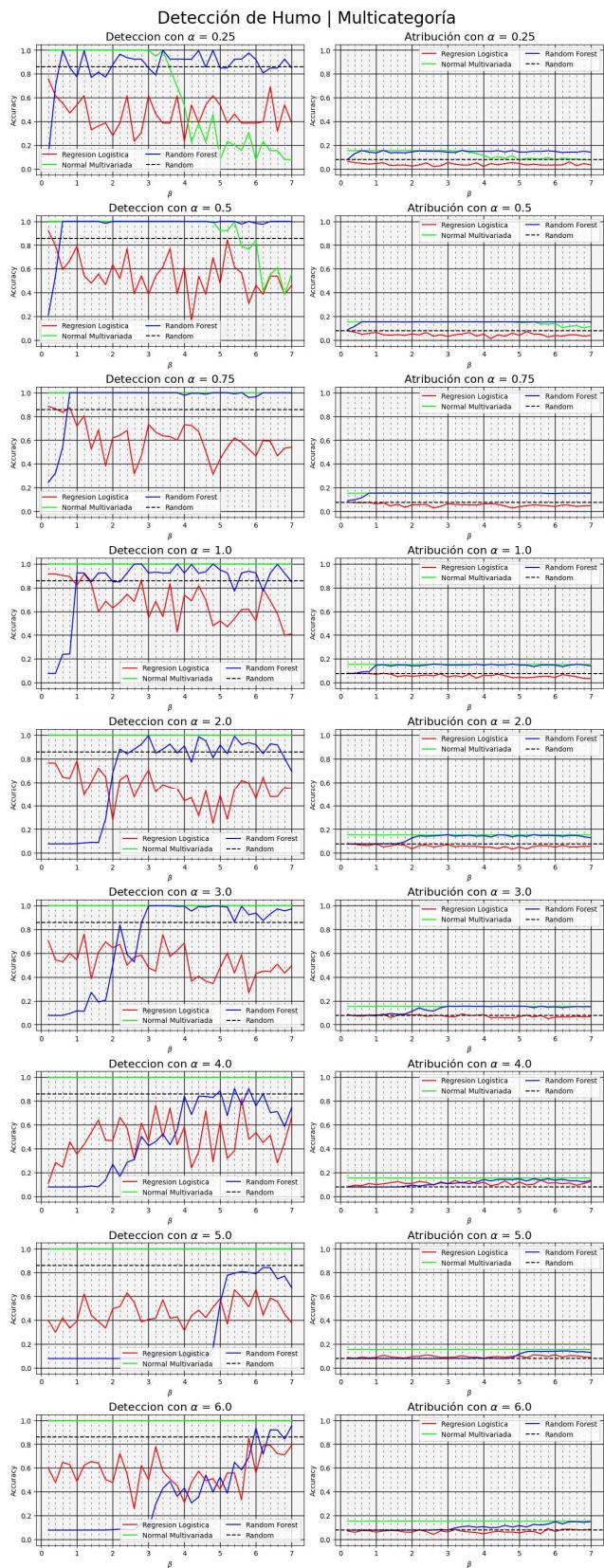


Figura 18. Detección de Humo multicategoría indicando detección y atribución para los 3 modelos propuestos: Regresión Logística, Normal Multivariada y Random Forest



**Figura 19. Detección de Humo para el escenario multicategórico, indicando detección y atribución para distintos valores de  $\alpha$**

## 7 Conclusiones

En el gráfico de la Figura 14 podemos observar que la regresión logística es incapaz de aprender patrones no lineales y por ende se desempeña de forma pobre, como cabe esperar, respondiendo de forma constante que el sensor está fallado (acertando 2/3 de las veces). Eso también pasa en un sensor en el caso de la normal multivariada, pero en el caso del resto se comporta adecuadamente. Tanto Normal Multivariada como Random Forest son capaces de detectar con certeza errores en los sensores para valores altos de  $\beta$ , pero Random Forest parece desempeñarse mejor para valores bajos de  $\beta$ , además de ser eficaz para todos los sensores.

En el gráfico de la Figura 15 se ve nuevamente que la regresión logística es incapaz de aprender patrones no lineales, pero además vemos que hay más sensores donde la normal multivariada se comporta de forma pobre, incluso a veces acertando sólo 1/3 de las veces. En cambio, el Random Forest tiene un buen desempeño en todos los sensores. Se ve en todos los casos que la eficacia es peor para valores más altos de  $\alpha$  y más bajos de  $\beta$ .

En el gráfico de la Figura 16 no es completamente eficaz para hacer observaciones precisas sobre los datos, pero podemos ver que la regresión logística es peor que el modelo aleatorio, mientras que los otros dos modelos le superan. En este caso, la normal multivariada se desempeña mejor para valores altos de  $\alpha$ , mientras que el random forest es mejor para valores bajos de  $\alpha$ . Estas conclusiones son válidas tanto para la detección como para la atribución de los fallos.

En el gráfico de la Figura 17 se confirma que para valores altos de  $\alpha$  la normal multivariada es superior al random forest, y se puede apreciar que para ambas tareas (detección y atribución de fallos) el random forest entrenado con el mejor  $\alpha$  de la muestra es la mejor de todas las opciones para prácticamente todos los valores de  $\beta$ .

En el gráfico de la Figura 18 podemos ver como la normal multivariada es mucho más consistente que los demás modelos en ambas tareas para distintos valores de  $\alpha$  y de  $\beta$ . Nuevamente la regresión logística es peor que el modelo aleatorio, mientras que los otros dos modelos le superan. En este caso, la normal multivariada se desempeña mejor para valores altos de  $\alpha$ , mientras que el random forest es mejor para valores bajos de  $\alpha$ . Estas conclusiones son válidas tanto para la detección como para la atribución de los fallos.

Finalmente, en el gráfico de la Figura 19 podemos observar como la normal multivariada entrenada con valores no muy bajos de  $\alpha$  logra precisión de aproximadamente el 100% en el caso de la detección de fallos y una precisión mejor a los demás modelos en el caso de atribución de estos fallos.

## 8 Referencias

- [1] <https://www.kaggle.com/datasets/deepcontractor/smoke-detection-dataset>
- [2] <https://www.kaggle.com/datasets/kukuroo3/room-occupancy-detection-data-iot-sensor>
- [3] Agresti, A. (2013). Categorical Data Analysis (3rd ed.). Wiley
- [4] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An Introduction to Statistical Learning: with Applications in R. Springer.
- [5] [https://scikit-learn.org/stable/api/sklearn.linear\\_model.html](https://scikit-learn.org/stable/api/sklearn.linear_model.html)
- [6] <https://scikit-learn.org/stable/modules/tree.html>
- [7] <https://jupyter.org/>
- [8] <https://ubuntu.com/desktop/vsl>
- [9]
- [10] <https://pandas.pydata.org/>
- [11] <https://numpy.org/>
- [12] [https://matplotlib.org/3.5.3/api/\\_as\\_gen/matplotlib.pyplot.html](https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html)
- [13] <https://joblib.readthedocs.io/en/stable/>
- [14] Harrell, F. E. (2015). Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis (2nd ed.). Springer.
- [15] [https://en.wikipedia.org/wiki/Multivariate\\_normal\\_distribution](https://en.wikipedia.org/wiki/Multivariate_normal_distribution)
- [16] <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- [17] <https://pypi.org/project/tqdm/>