

# Predicción de Riesgo de Incendio Forestal en España

Aprendizaje Automático Avanzado: Ejercicio final

Autor: Julen Ercibengoa

julenertzisa@gmail.com, jercibengoa001@ikasle.ehu.eus, julen.ercibengoa@tekniker.es

## 1. Contexto

Actualmente estoy trabajando en la empresa **Tekniker** la cual está involucrada en el proyecto GAIA. Este es un proyecto que pretende dar apoyo a los servicios de extinción de incendios forestales mediante herramientas de prevención y detección temprana utilizando la Inteligencia Artificial.

Mi trabajo consiste en crear un modelo de Aprendizaje Automático que prediga el riesgo de incendio forestal a nivel nacional diariamente. Es decir, el objetivo final del trabajo (el cual es mi Trabajo de Fin de Máster), es generar un modelo que con los datos disponibles hasta hoy, genere un mapa de mañana del riesgo de incendio forestal en toda España.

Para conseguir ese objetivo hacen falta varios pasos, y este trabajo será uno de ellos. En este trabajo haremos una primera prueba y generaremos un predictor de riesgo de incendio forestal que prediga el riesgo de incendio forestal mensual. Además, haremos varios análisis para decidir qué variables son relevantes para el modelo.

## 2. Bases de Datos

Para predecir el riesgo de incendio forestal hay que generar una base de datos con la cual entrenar un modelo de Inteligencia Artificial. En el estado del arte ([Ver el análisis del estado del arte](#)) se sigue una metodología común que vamos a utilizar: se divide la región a analizar en celdas espacio-temporales y cada celda tiene asignadas variables tabulares. Para generar dicha base de datos, vamos a utilizar las siguientes fuentes de datos:

- [Datos históricos de fuegos](#): base de datos de **EFFIS**
- [Datos del tipo de terreno](#): base de datos **Corine Land Cover**
- [Datos DEM \(Digital Elevation Model\)](#): base de datos de **EU DEM**
- [Distancia a carreteras, distancia a ríos y densidad poblacional](#): **WorldPop**
- [Distancia a vías de tren](#): **OpenStreetMap**
- [Proporción de naturaleza2000](#): **Gobierno de España**

Aunque en las visualizaciones de los datos que vamos a hacer a continuación no aparezca, el área que vamos a analizar será toda España, incluyendo las Islas Canarias.

### 2.1 Datos históricos de fuegos: EFFIS

El primer conjunto de datos que vamos a utilizar serán los datos históricos de EFFIS (European Forest Fire Information System). Este conjunto de datos guarda la información de cuándo inició el fuego, cuándo finalizó, el área que quemó y la forma que tiene junto a dónde ocurrió. Los datos que tienen recogidos van desde 2008 hasta la actualidad y usaremos solo los incendios de más de 5 hectáreas, con un total de 5862 fuegos.

Para visualizar el conjunto de datos utilizaremos el paquete `geopandas`, que permite visualizar los datos con información geográfica.

```
In [101]: import geopandas as gpd
gdf_fires = gpd.read_file("../data/Variables/EFFIS_historical_postprocessed.json")
gdf_fires.head()
```

```
Out[101]:   initialdate    finaldate  area_ha          geometry
0  2008-04-27  2008-04-27     419  MULTIPOLYGON (((1652130.767 1048788.051, 16513...
1  2008-06-17  2008-06-17     104  MULTIPOLYGON (((3252074.625 1640344.173, 32521...
2  2008-06-19  2008-06-19     523  MULTIPOLYGON (((2906844.761 1591397.039, 29068...
3  2008-07-01  2008-07-01     253  MULTIPOLYGON (((3016594.759 1855823.362, 30161...
4  2008-07-07  2008-07-07      58  MULTIPOLYGON (((3115823.664 1790836.728, 31160...
```

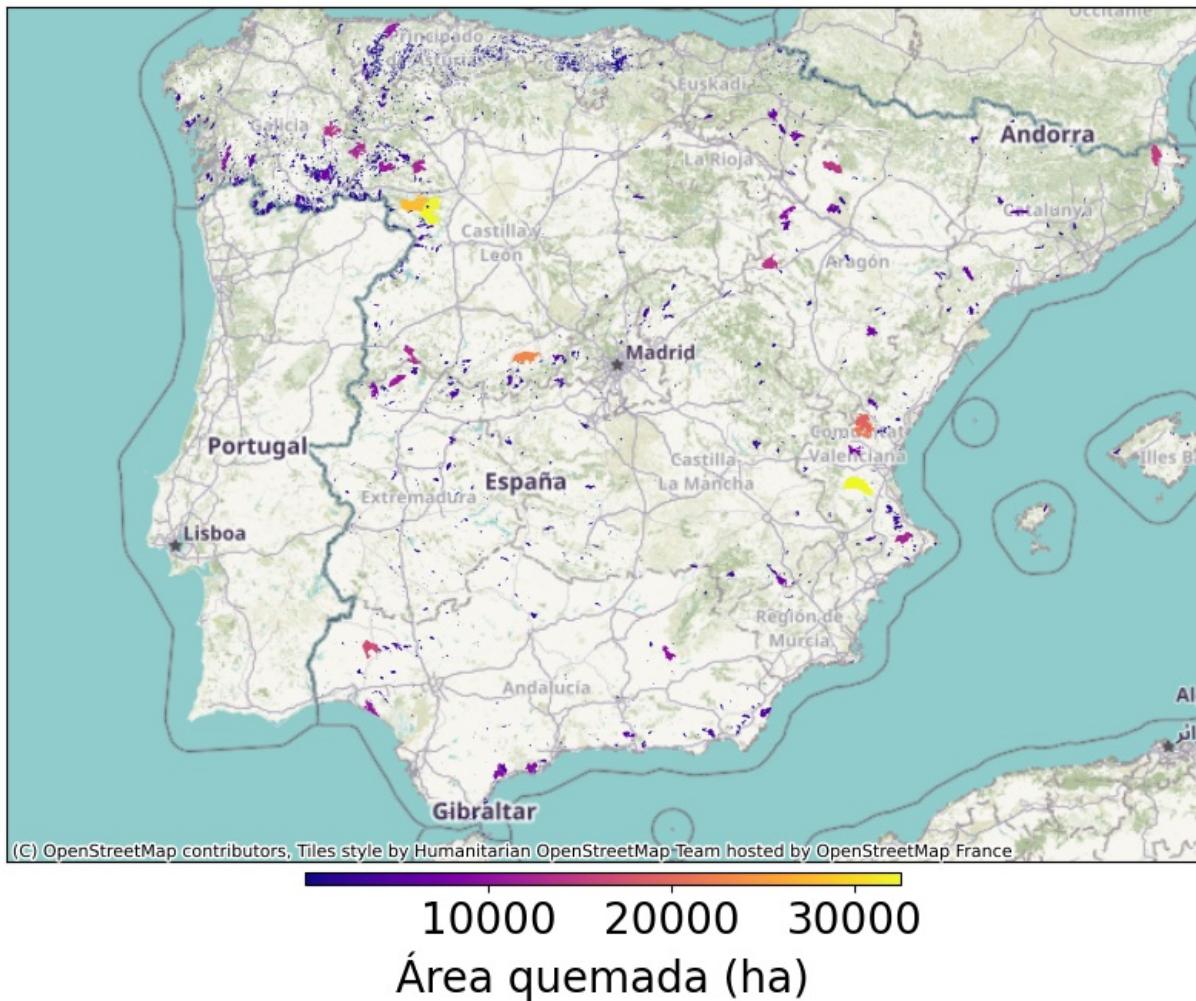
```
In [151]: import contextily as ctx
import matplotlib.pyplot as plt
gdf_fires = gdf_fires.to_crs(epsg=3857)
minx, miny, maxx, maxy = gdf_fires.total_bounds
ax = gdf_fires.plot(column="area_ha", legend=True, figsize=(10,10), cmap="plasma", legend_kwds={'label': "Área de incendio"})
```

```

ax.set_xlim([-1.25*10**6, maxx])
ax.set_ylim([4.25*10**6, maxy])
ctx.add_basemap(ax, crs=gdf_fires.crs.to_string())
ax.set_xticks([])
ax.set_yticks([])
ax.set_title("Área quemada en España (2008-2024)", fontsize=17, pad=20)
plt.show()

```

## Área quemada en España (2008-2024)



## 2.2 Datos del tipo de terreno: Corine Land Cover

El conjunto de datos Corine Land Cover divide a toda Europa en celdas de 100m × 100m y asigna a cada celda una clase de las 44 clases que tienen definidas. Las clases son las siguientes:

Class	LABEL1	LABEL2	LABEL3
1	Artificial surfaces	Urban fabric	Continuous urban fabric
2	Artificial surfaces	Urban fabric	Discontinuous urban fabric
3	Artificial surfaces	Industrial, commercial and transport units	Industrial or commercial units
4	Artificial surfaces	Industrial, commercial and transport units	Road and rail networks and associated land
5	Artificial surfaces	Industrial, commercial and transport units	Port areas
6	Artificial surfaces	Industrial, commercial and transport units	Airports
7	Artificial surfaces	Mine, dump and construction sites	Mineral extraction sites
8	Artificial surfaces	Mine, dump and construction sites	Dump sites
9	Artificial surfaces	Mine, dump and construction sites	Construction sites
10	Artificial surfaces	Artificial, non-agricultural vegetated areas	Green urban areas
11	Artificial surfaces	Artificial, non-agricultural vegetated areas	Sport and leisure facilities
12	Agricultural areas	Arable land	Non-irrigated arable land
13	Agricultural areas	Arable land	Permanently irrigated land
14	Agricultural areas	Arable land	Rice fields
15	Agricultural areas	Permanent crops	Vineyards
16	Agricultural areas	Permanent crops	Fruit trees and berry plantations
17	Agricultural areas	Permanent crops	Olive groves
18	Agricultural areas	Pastures	Pastures
19	Agricultural areas	Heterogeneous agricultural areas	Permanent crops*
20	Agricultural areas	Heterogeneous agricultural areas	Complex cultivation patterns
21	Agricultural areas	Heterogeneous agricultural areas	Land principally occupied by agriculture*
22	Agricultural areas	Heterogeneous agricultural areas	Agro-forestry areas
23	Forest and semi natural areas	Forests	Broad-leaved forest
24	Forest and semi natural areas	Forests	Coniferous forest
25	Forest and semi natural areas	Forests	Mixed forest
26	Forest and semi natural areas	Scrub and/or herbaceous vegetation associations	Natural grasslands
27	Forest and semi natural areas	Scrub and/or herbaceous vegetation associations	Moors and heathland
28	Forest and semi natural areas	Scrub and/or herbaceous vegetation associations	Sclerophyllous vegetation
29	Forest and semi natural areas	Scrub and/or herbaceous vegetation associations	Transitional woodland-shrub
30	Forest and semi natural areas	Open spaces with little or no vegetation	Beaches, dunes, sands
31	Forest and semi natural areas	Open spaces with little or no vegetation	Bare rocks
32	Forest and semi natural areas	Open spaces with little or no vegetation	Sparsely vegetated areas
33	Forest and semi natural areas	Open spaces with little or no vegetation	Burnt areas
34	Forest and semi natural areas	Open spaces with little or no vegetation	Glaciers and perpetual snow
35	Wetlands	Inland wetlands	Inland marshes
36	Wetlands	Inland wetlands	Peat bogs
37	Wetlands	Maritime wetlands	Salt marshes
38	Wetlands	Maritime wetlands	Salines
39	Wetlands	Maritime wetlands	Intertidal flats
40	Water bodies	Inland waters	Water courses
41	Water bodies	Inland waters	Water bodies
42	Water bodies	Marine waters	Coastal lagoons
43	Water bodies	Marine waters	Estuaries
44	Water bodies	Marine waters	Sea and ocean

\*Nombre original modificado

El conjunto de datos se actualiza cada 6 años, por lo que en nuestro caso, como tenemos datos históricos de fuegos desde 2008, utilizaremos los conjuntos de datos de Corine Land Cover de 2006, 2012 y 2018. El siguiente gráfico es una visualización de los datos de 2018.

```
In [106]: import rasterio
import numpy as np
import pickle
import matplotlib.patches as mpatches

ruta_tiff = "../data/Variables/U2018_CLC2018_V2020_20u1.tif"
custom_colors = pickle.load(open("../data/figures/custom_colors.pkl", "rb"))

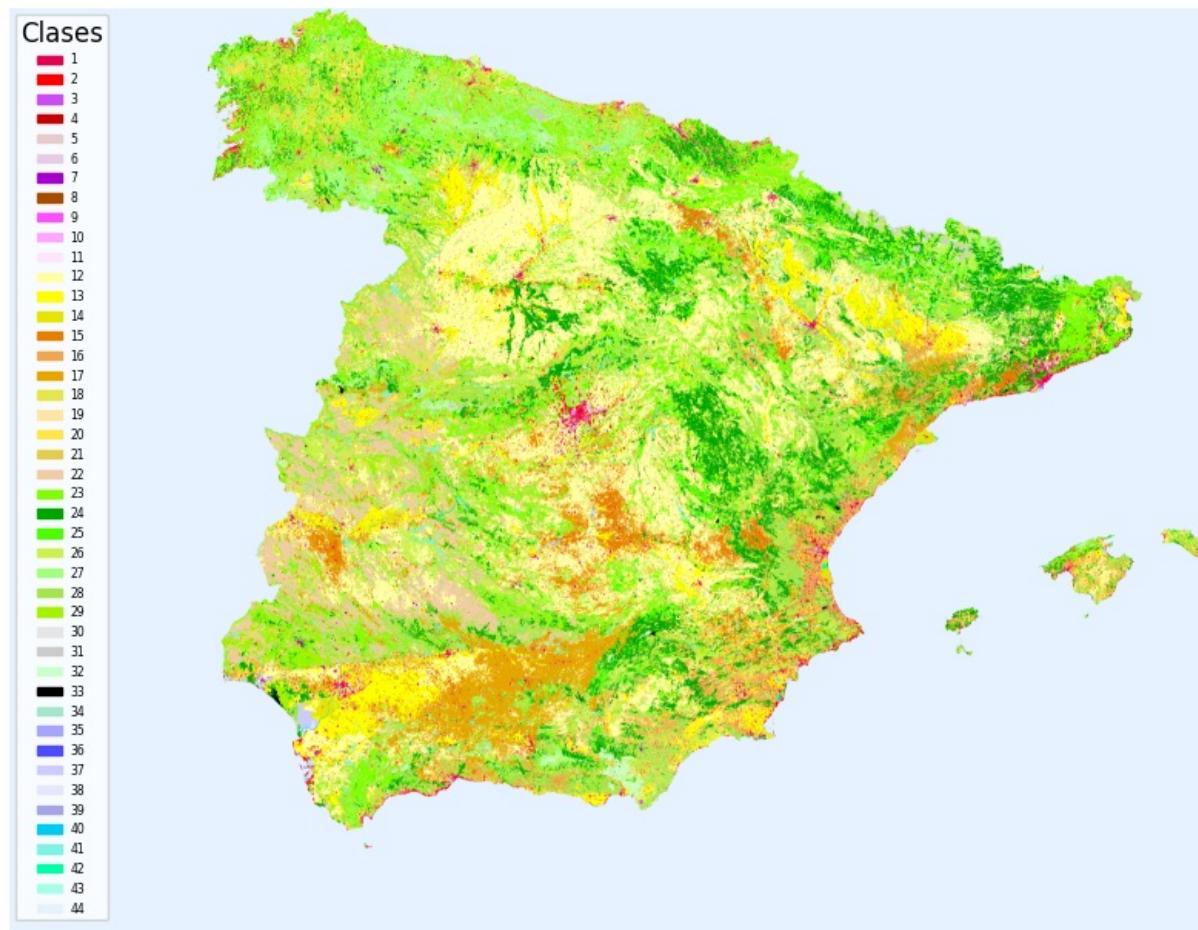
with rasterio.open(ruta_tiff) as dataset:
    datos = dataset.read(1)
    datos = datos[:10000, 10000:]
rgb_image = np.zeros((datos.shape[0], datos.shape[1], 3), dtype=np.uint8)
for key, color in custom_colors.items():
    rgb_image[datos == key] = color
plt.figure(figsize=(10, 10))
plt.imshow(rgb_image)
plt.axis('off')
plt.title("Corine Land Cover año 2018", fontsize=18, pad = 20)
```

```

legend_patches = [mpatches.Patch(color=np.array(color)/255, label=str(key)) for key, color in custom_colors.items()]
legend_patches.pop(-1)
plt.legend(handles=legend_patches,
           loc='upper left',
           title="Clases",
           fontsize=5.5)
plt.show()

```

## Corine Land Cover año 2018



### 2.3 Datos DEM

En cuanto a los datos DEM (Digital Elevation Model), usaremos tres fuentes de datos: la elevación del terreno (metros), la orientación del terreno (grados) y la pendiente (grados). Los datos los obtuvieron la "European Space Agency" y Copernicus, a una resolución de 30m × 30m. Asumiremos que estos datos son atemporales, ya que tienen un cambio mínimo a lo largo del tiempo. En la siguiente imagen se pueden observar estas tres fuentes de datos graficados en Donostia.

```

In [110]: import rasterio
import matplotlib.pyplot as plt
from rasterio.plot import show
import matplotlib.colors as mcolors

dem_file = "../data/Variables/DEM_donostia.tif"
aspect_file = "../data/Variables/Aspect_donostia.tif"
slope_file = "../data/Variables/Slope_donostia.tif"

fig, ax = plt.subplots(3, 1, figsize=(8, 13))

# Open the DEM file using rasterio
with rasterio.open(dem_file) as src:
    dem_data = src.read(1) # Read the first band
    dem_extent = (src.bounds.left, src.bounds.right, src.bounds.bottom, src.bounds.top)
    im0 = ax[0].imshow(dem_data, cmap='terrain', extent=dem_extent)
    ax[0].set_title("Modelo Digital de Elevación (DEM)")
    ax[0].set_xticks([])
    ax[0].set_yticks([])
    ax[0].set_ylim(src.bounds.bottom + 0.003, src.bounds.top)
    # Add colorbar for the DEM
    cbar0 = fig.colorbar(im0, ax=ax[0], orientation='vertical', fraction=0.03, pad=0.04)
    cbar0.set_label("Elevation (metros)")

with rasterio.open(aspect_file) as src:
    im1 = show(src, ax=ax[1], title="Orientación", cmap="viridis")

```

```

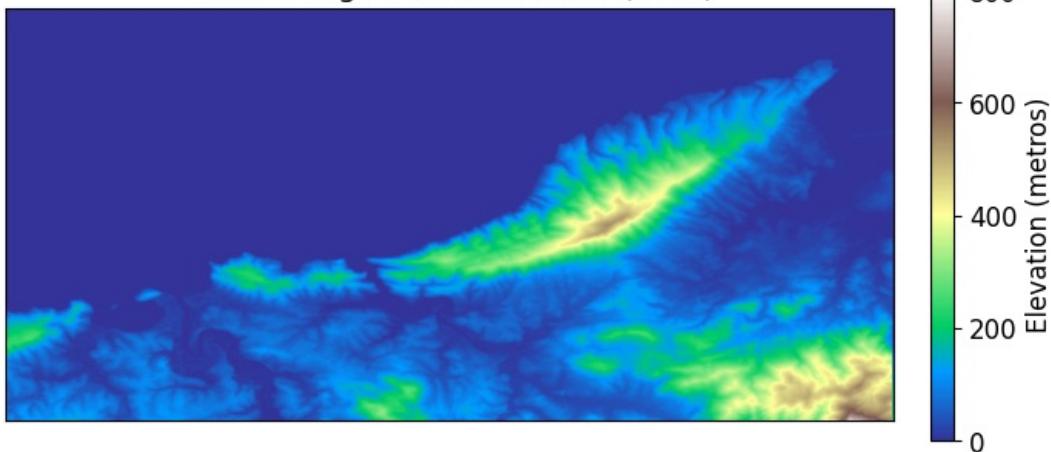
ax[1].set_ylim(src.bounds.bottom + 0.003, src.bounds.top)
ax[1].set_xticks([])
ax[1].set_yticks([])
# Add colorbar for the Aspect
norm = mcolors.Normalize(vmin=src.read(1).min(), vmax=src.read(1).max())
cbar1 = fig.colorbar(im1.get_images()[0], ax=ax[1], orientation='vertical', fraction=0.03, pad=0.04)
cbar1.set_label("Orientación (grados)")

with rasterio.open(slope_file) as src:
    im2 = show(src, ax=ax[2], title="Pendiente", cmap = "terrain")
    ax[2].set_ylim(src.bounds.bottom + 0.003, src.bounds.top)
    ax[2].set_xticks([])
    ax[2].set_yticks([])
# Add colorbar for the Slope
norm = mcolors.Normalize(vmin=src.read(1).min(), vmax=src.read(1).max())
cbar2 = fig.colorbar(im2.get_images()[0], ax=ax[2], orientation='vertical', fraction=0.03, pad=0.04)
cbar2.set_label("Pendiente (grados)")

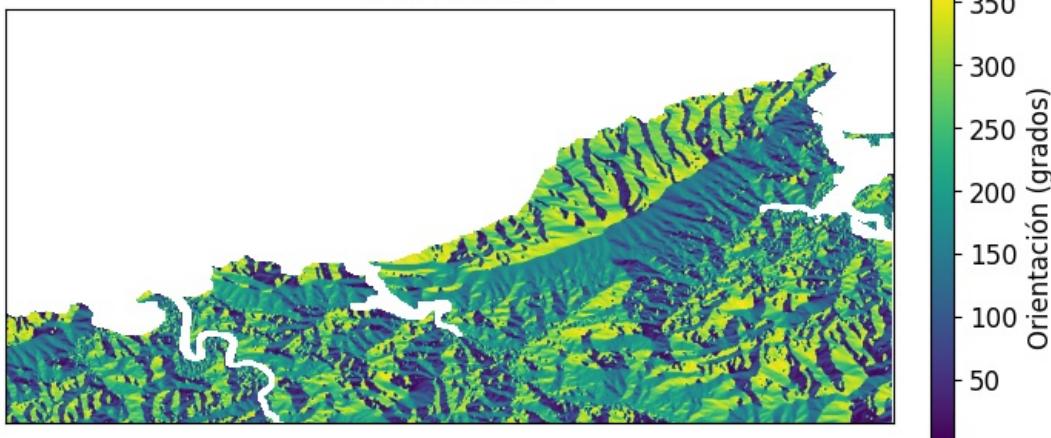
plt.show()

```

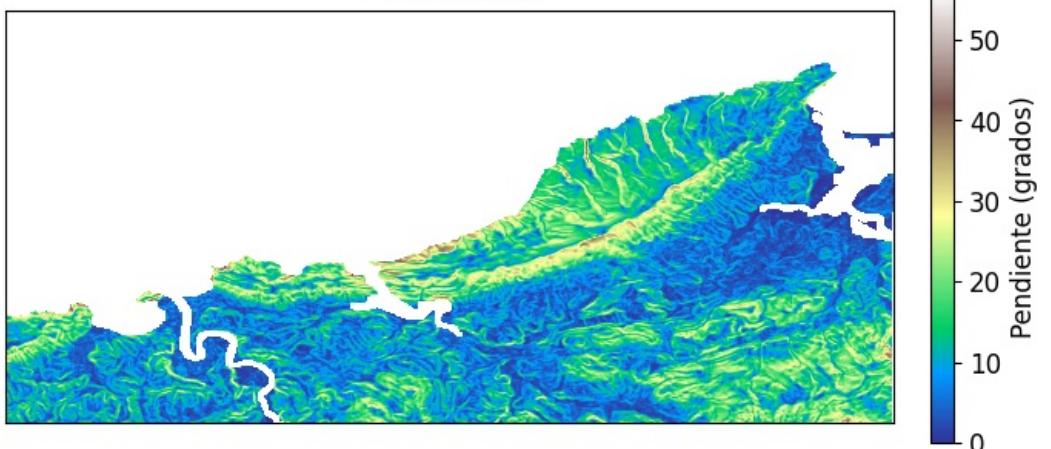
Modelo Digital de Elevación (DEM)



Orientación



Pendiente



Para introducir el factor humano usaremos estas cuatro variables, algunas creadas mediante el software QGIS, que sirve para analizar y transformar variables espaciales:

- Distancia a carreteras
- Distancia a ríos
- Distancia a vías de tren
- Densidad poblacional

Esta última cambia con el tiempo y la base de datos de [WorldPop](#) ofrece datos anuales desde 2008 hasta 2020. Como a partir de 2020 no tenemos datos, asumiremos que la densidad poblacional desde 2021 hasta 2024 es la misma que la de 2020.

```
In [114]: import rasterio
import matplotlib.pyplot as plt
from rasterio.plot import show
import matplotlib.colors as mcolors

railways_file = "../data/Variables/Dist_to_railways_SPAIN.tif"
roads_file = "../data/Variables/Dist_to_roads_SPAIN.tif"
waterways_file = "../data/Variables/Dist_to_waterways_SPAIN.tif"
popdens_file = "../data/Variables/Pop_dens_SPAIN.tif"

fig, ax = plt.subplots(2, 2, figsize=(15, 10))

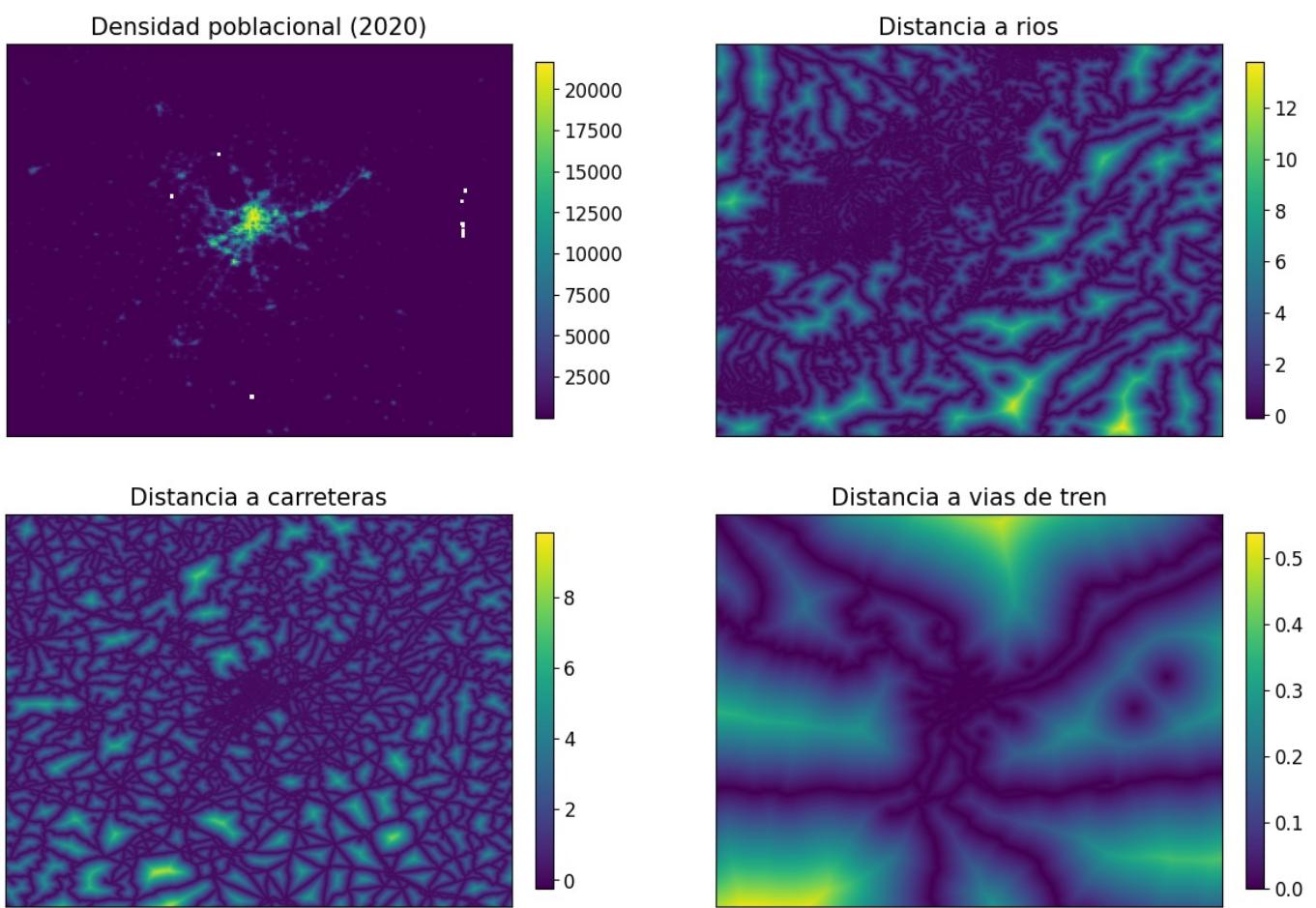
# Open the DEM file using rasterio
with rasterio.open(popdens_file) as src:
    im1 = show(src, ax=ax[0,0], title="Densidad poblacional (2020)", cmap="viridis")
    ax[0,0].set_ylim(src.bounds.bottom + 0.05, src.bounds.top-0.05)
    ax[0,0].set_xlim(src.bounds.left + 0.05, src.bounds.right-0.05)
    ax[0,0].set_xticks([])
    ax[0,0].set_yticks([])
    ax[0,0].set_title("Densidad poblacional (2020)", fontsize=15)
    norm = mcolors.Normalize(vmin=src.read(1).min(), vmax=src.read(1).max())
    cbar1 = fig.colorbar(im1.get_images()[0], ax=ax[0,0], orientation='vertical', fraction=0.03, pad=0.04)

with rasterio.open(roads_file) as src:
    im1 = show(src, ax=ax[1,0], title="Distancia a carreteras", cmap="viridis")
    ax[1,0].set_ylim(src.bounds.bottom + 0.05, src.bounds.top-0.05)
    ax[1,0].set_xlim(src.bounds.left + 0.05, src.bounds.right-0.05)
    ax[1,0].set_xticks([])
    ax[1,0].set_yticks([])
    ax[1,0].set_title("Distancia a carreteras", fontsize=15)
    norm = mcolors.Normalize(vmin=src.read(1).min(), vmax=src.read(1).max())
    cbar1 = fig.colorbar(im1.get_images()[0], ax=ax[1,0], orientation='vertical', fraction=0.03, pad=0.04)

with rasterio.open(waterways_file) as src:
    im2 = show(src, ax=ax[0,1], title="Distancia a ríos", cmap="viridis")
    ax[0,1].set_ylim(src.bounds.bottom + 0.05, src.bounds.top-0.05)
    ax[0,1].set_xlim(src.bounds.left + 0.05, src.bounds.right-0.05)
    ax[0,1].set_xticks([])
    ax[0,1].set_yticks([])
    ax[0,1].set_title("Distancia a ríos", fontsize=15)
    norm = mcolors.Normalize(vmin=src.read(1).min(), vmax=src.read(1).max())
    cbar2 = fig.colorbar(im2.get_images()[0], ax=ax[0,1], orientation='vertical', fraction=0.03, pad=0.04)

with rasterio.open(railways_file) as src:
    im2 = show(src, ax=ax[1,1], title="Distancia a vías de tren", cmap="viridis")
    ax[1,1].set_ylim(src.bounds.bottom + 0.05, src.bounds.top-0.05)
    ax[1,1].set_xlim(src.bounds.left + 0.05, src.bounds.right-0.05)
    ax[1,1].set_xticks([])
    ax[1,1].set_yticks([])
    ax[1,1].set_title("Distancia a vías de tren", fontsize=15)
    norm = mcolors.Normalize(vmin=src.read(1).min(), vmax=src.read(1).max())
    cbar2 = fig.colorbar(im2.get_images()[0], ax=ax[1,1], orientation='vertical', fraction=0.03, pad=0.04)

plt.show()
```

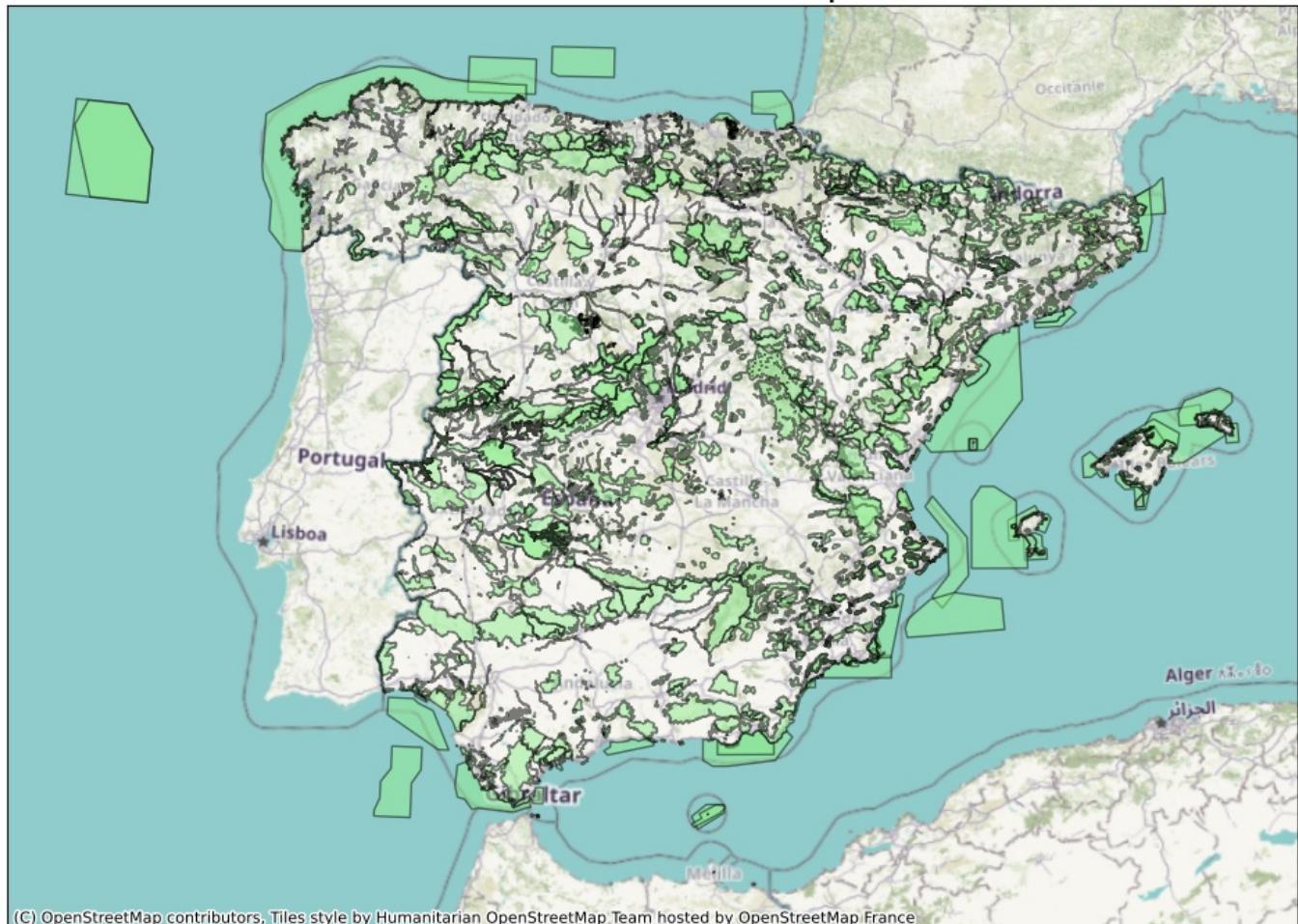


## 2.5 Natura2000

La red Natura 2000 es una red ecológica europea de áreas de conservación de la biodiversidad. Introduciremos esta variable porque el conjunto de datos de EFFIS original también la tiene.

```
In [116]:  
import geopandas as gpd  
import contextily as ctx  
  
Natura2000 = gpd.read_file("../data/Variables/N2000_2.json")  
ax = Natura2000.plot(figsize=(12, 12), alpha=0.5, edgecolor='k', color='lightgreen')  
ax.set_title("Red Natura 2000 en España", fontsize=20)  
ax.set_xticks([])  
ax.set_yticks([])  
ctx.add_basemap(ax, crs=Natura2000.crs.to_string())  
  
plt.show()
```

# Red Natura 2000 en España



## 3. Homogeneización de los datos

En esta sección vamos a explicar brevemente cómo se han homogeneizado los datos.

### 3.1 Instancias del dataset

Para transformar las bases de datos que hemos presentado en un dataset con variables tabulares, vamos a utilizar celdas espacio-temporales, las cuales serán nuestras instancias que utilizaremos para entrenar los algoritmos. Para ello, hemos creado celdas espaciales que cubren toda España con una resolución espacial de 1km × 1km.

Además, las celdas espaciales también tendrán dimensión temporal diaria. Es decir, cada celda espacial generará una instancia por día y cada instancia (celda espacio-temporal) tendrá asignadas varias variables que expliquen las características de esa celda en ese día concreto. Sin embargo, como en este ejercicio no estamos considerando variables que cambian diariamente, no vamos a poder diferenciar dichas instancias con temporalidad diaria. Todo esto lo analizaremos en más profundidad en secciones posteriores, cuando generemos las instancias.

Nota: Estas celdas espaciales las hemos creado utilizando el software QGIS y las hemos guardado en archivos `json`, las cuales se pueden visualizar mediante el paquete `geopandas`. También hemos creado celdas con resolución espacial de 20km × 20km para visualizarlas.

```
In [118]: boxes_visual_20 = gpd.read_file("../data/Variables/Spain_boxes_20km.json")
boxes_visual_1 = gpd.read_file("../data/Variables/Spain_boxes_1km.json")
```

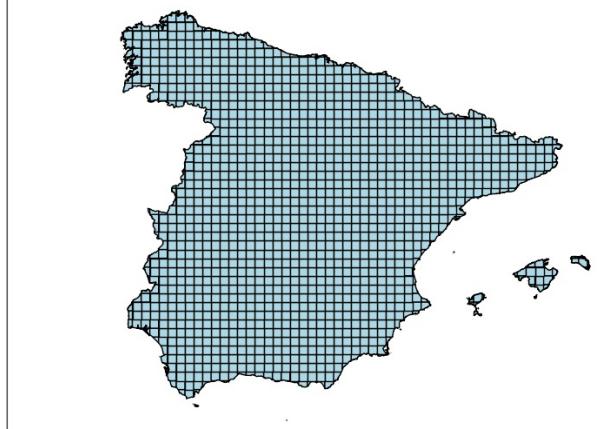
```
In [119]: fig, ax = plt.subplots(1, 2, figsize=(20, 10))
boxes_visual_20.plot(ax=ax[0], edgecolor='k', color='lightblue')
ax[0].set_title("Celdas de 20km x 20km (para visualizar)", fontsize=20)
ax[0].set_xlim([2.5*10**6, 3.9*10**6])
ax[0].set_ylim([1.5*10**6, 2.5*10**6])
ax[0].set_xticks([])
ax[0].set_yticks([])

boxes_visual_1.plot(ax= ax[1], edgecolor='k', color='red', figsize=(10, 10), alpha=0.1)
ax[1].set_title("Celdas de 1km x 1km", fontsize=20)
ax[1].set_xlim([3.345*10**6, 3.37*10**6])
ax[1].set_ylim([2.31*10**6, 2.33*10**6])
ax[1].set_xticks([])
ax[1].set_yticks([])

ctx.add_basemap(ax[1], crs=boxes_visual_1.crs.to_string())
```

```
plt.show()
```

Celdas de 20km x 20km (para visualizar)



Celdas de 1km x 1km



Variables tabulares:

Cada celda espacio-temporal de 1km × 1km tendrá asignadas varias variables tabulares:

- Corine Land Cover: para cada clase, generaremos una variable que será la proporción de casillas de 100m × 100m de esa clase que hay en la celda. Por lo que el dataset CLC nos dará 44 variables.
- DEM:
  - Elevación: Para cada celda espacial, calcularemos las siguientes variables de elevación: suma, media , mediana, desviación estandar, mínimo, máximo y rango.
  - Orientación: Dividiremos 360° en 8 clases: (0-45), (45-90), (90-135), (135-180), (180-225), (225-270), (270-315), y (315-360). Para cada clase de orientación, se calculará la proporción de celdas de 30m × 30m que pertenecen a esa clase dentro de la celda de 1km × 1km. Esto nos dará 8 variables adicionales asociadas a la orientación.
  - Pendiente: Para cada celda espacial, calcularemos las siguientes variables de pendiente: suma, media , mediana, desviación estandar, mínimo, máximo y rango.
- Densidad poblacional: Calcularemos el mínimo, el máximo y la media de cada celda espacio-temporal.
- Distancia a carreteras: Calcularemos la media, la mediana, el mínimo y el máximo.
- Distancia a ríos: Calcularemos la media, la mediana, el mínimo y el máximo.
- Distancia a vías de tren: Calcularemos la media, la mediana, el mínimo y el máximo.
- Natura 2000: Calcularemos la proporción de terreno de Natura 2000 que hay en cada celda.

Es decir, de estos conjuntos de datos obtenemos 82 variables tabulares. Además, vamos a añadir las siguientes variables:

- Coordenada\_x (número entero)
- Coordenada\_y (número entero)
- Mes: Mes en el que se sitúa la instancia.

Las coordenadas sirven para identificar y enumerar las celdas (son índices), es decir, la tupla (x,y) identifica a una única celda. Sin embargo, también sirven como coordenadas espaciales, ya que están ordenadas. Más adelante, en la sección de detección de "outliers", las utilizaremos para visualizar los datos.

**Nota:** La homogeneización de las variables y generación del "datacube" (el conjunto de datos con todas las celdas y sus características) se ha hecho mediante el software QGIS: una herramienta que sirve para manejar datos espaciales.

## 3.2 Clase a predecir

### 3.2.1 Instancias positivas

Una vez que tenemos bien definidas las instancias, la clase a predecir será si hay fuego en una celda un día concreto o no. Utilizando la última base de datos que no hemos usado, EFFIS, generamos las instancias positivas (en las que ha habido un incendio). Esto lo hemos hecho nuevamente utilizando el software QGIS y calculando la intersección entre las celdas espaciales y los incendios que ha habido. El resultado de esa intersección son celdas espaciales en las cuales ha habido un incendio.

Tras un preprocesso de datos en el cual asignamos a cada incendio el conjunto adecuado de Corine Land Cover y densidad poblacional (dependiendo de la fecha en la que ocurrió el incendio), obtenemos el siguiente conjunto de datos:

```
In [68]: import pandas as pd
fire_instances = pd.read_csv("../data/FinalDataset/Fire_instances_expandedbydate.csv")

len(fire_instances)
fire_instances.columns
```

```
Out[68]: Index(['initialdate', 'finaldate', 'x_index', 'y_index', 'elevation_sum',
       'elevation_mean', 'elevation_median', 'elevation_stdev',
       'elevation_min', 'elevation_max', 'elevation_range', 'slope_sum',
       'slope_mean', 'slope_median', 'slope_stdev', 'slope_min', 'slope_max',
       'slope_range', 'aspect_NODATA', 'aspect_0_45', 'aspect_45_90',
       'aspect_90_135', 'aspect_135_180', 'aspect_180_225', 'aspect_225_270',
       'aspect_270_315', 'aspect_315_360', 'dist_to_road_mean',
       'dist_to_road_median', 'dist_to_road_min', 'dist_to_road_max',
       'dist_to_waterway_mean', 'dist_to_waterway_median',
       'dist_to_waterway_min', 'dist_to_waterway_max', 'dist_to_railway_mean',
       'dist_to_railway_median', 'dist_to_railway_min', 'dist_to_railway_max',
       'natura2000_NODATA', 'natura2000_1', 'CLC_1', 'CLC_2', 'CLC_3', 'CLC_4',
       'CLC_5', 'CLC_6', 'CLC_7', 'CLC_8', 'CLC_9', 'CLC_10', 'CLC_11',
       'CLC_12', 'CLC_13', 'CLC_14', 'CLC_15', 'CLC_16', 'CLC_17', 'CLC_18',
       'CLC_19', 'CLC_20', 'CLC_21', 'CLC_22', 'CLC_23', 'CLC_24', 'CLC_25',
       'CLC_26', 'CLC_27', 'CLC_28', 'CLC_29', 'CLC_30', 'CLC_31', 'CLC_32',
       'CLC_33', 'CLC_34', 'CLC_35', 'CLC_36', 'CLC_37', 'CLC_38', 'CLC_39',
       'CLC_40', 'CLC_41', 'CLC_42', 'CLC_43', 'CLC_44', 'CLC_NODATA',
       'population_density_mean', 'population_density_min',
       'population_density_max', 'month'],
      dtype='object')
```

Nota: Al calcular la cantidad de area de cierta clase que hay en cada celda, QGIS también añade una columna "NODATA", que significa que en ese espacio específico no había información de pertenecer a ninguna clase. Por ejemplo, en la red Natura 2000 la clase NODATA significa la cantidad de area que no pertenece a Natura 2000 en la celda. Más adelante las usaremos para obtener las proporciones.

### 3.2.2 Instancias negativas

El conjunto de datos real debería de tener todas las instancias negativas que ha habido, es decir, todas las celdas durante todos los días que no ha habido fuegos. Sin embargo, esto supondría tener alrededor de 100.000 instancias negativas por cada instancia positiva, lo cual es inviable. Por lo que para este trabajo tomaremos instancias negativas aleatoriamente hasta tener una cierta cantidad deseada.

Entonces, para generar las instancias negativas tomaremos días aleatorios en celdas aleatorias. No nos interesa mantener la distribución espacial que siguen las instancias de fuego, ya que queremos que el modelo aprenda patrones en los cuales los fuegos son más propensos. Por eso tomar fechas y celdas aleatorias es la matodología adecuada a seguir.

Como la cantidad de instancias positivas es muy pequeña comparado con la cantidad total de instancias negativas (alrededor de 500000 celdas × 16 años × 365 días), no vamos a introducir un verificador de colisiones que asegure que no introducimos una instancia positiva como instancia negativa (al generar aleatoriamente las instancias negativas podríamos obtener una instancia de `fuego`). Si verificásemos las colisiones, el código tardaría mucho en generar las instancias negativas y no obtendríamos demasiados beneficios. De hecho, ya hicimos un test de este procedimiento y obtuvimos 5 colisiones al generar 20000 instancias negativas, por lo que no merece la pena.

Ahora vamos a generar 2 veces más instancias negativas de las que tenemos positivas. Hemos seleccionado esta proporción basandonos en trabajos similares del estado del arte, los cuales utilizan proporciones balanceadas o un poco desbalanceadas tanto para entrenar como para testar. Sin embargo, para asegurar la eficacia de los modelos a la hora de predecir el riesgo de incendio forestal real, se deberían de tomar todas las instancias para testar los modelos, pero eso lo haremos en el TFM, no en este trabajo.

```
In [69]: datacube = gpd.read_file("../data/FinalDataset/Datacube_1.gpkg")
```

```
In [70]: import random
from datetime import datetime, timedelta
import numpy as np
import warnings

random.seed(42)

def GenerateBox(date, row):
    """
    Receives as inputs a date and a row from the datacube,
    and returns the values of the row filtered by the selected date.
    """

    year = date.year
    month = date.month

    selected_columns = ["initialdate", "finaldate", "x_index", "y_index",
                        "elevation_sum", "elevation_mean", "elevation_median", "elevation_stdev", "elevation_min",
                        "slope_sum", "slope_mean", "slope_median", "slope_stdev", "slope_min", "slope_max", "slope_range",
                        "aspect_NODATA", "aspect_0_45", "aspect_45_90", "aspect_90_135", "aspect_135_180", "aspect_180_225",
                        "aspect_225_270", "aspect_270_315", "aspect_315_360", "dist_to_road_mean", "dist_to_road_median",
                        "dist_to_road_min", "dist_to_road_max", "dist_to_waterway_mean", "dist_to_waterway_median",
                        "dist_to_waterway_min", "dist_to_waterway_max", "dist_to_railway_mean", "dist_to_railway_median",
                        "dist_to_railway_min", "dist_to_railway_max", "natura2000_NODATA", "natura2000_1"]
```

```

# Corine Land Cover
if year < 2012:
    year_clc = 2006
elif year < 2018:
    year_clc = 2012
else:
    year_clc = 2018
clc_columns = [f"CLC_{year_clc}_{i}" for i in range(1,45)]
selected_columns = selected_columns + clc_columns + [f"CLC_{year_clc}_NODATA"]

# Population Density
if year > 2020: # We dont have data for 2021-2024
    year_popdens = 2020
else:
    year_popdens = year
popdens_columns = [f"population_density_{year_popdens}_{case}" for case in ["mean", "min", "max"]]
selected_columns += popdens_columns

filtered_row_values = row[selected_columns].values
filtered_row_values = np.append(filtered_row_values, [int(month)])

return filtered_row_values

def generate_random_date(start_date, end_date):
"""
Generates a random date between the start and end date.
"""
delta = end_date - start_date
random_days = random.randint(0, delta.days)
return start_date + timedelta(days=random_days)

```

In [71]:

```

random.seed(43)

n_nofire_instances = 2 * len(fire_instances)

nofire_instances = np.zeros((n_nofire_instances, len(fire_instances.columns)), dtype=object)
warnings.simplefilter(action='ignore', category=pd.errors.SettingWithCopyWarning)

# Generatenofire instances:
i = 0
while i < len(nofire_instances):
    i_random = random.randint(0, len(datacube)-1) # Select random box
    random_box = datacube.iloc[i_random]
    random_date = generate_random_date(start_date=datetime(2008,1,1),
                                         end_date=datetime(2024,12,31))
    random_box["initialdate"] = random_date.strftime('%Y-%m-%d')
    random_box["finaldate"] = random_date.strftime('%Y-%m-%d')

    random_box_values = GenerateBox(random_date, random_box)
    nofire_instances[i] = random_box_values
    i += 1

    if i % 10000 == 0:
        print(f"Instancias generadas = {i} / {n_nofire_instances}")

warnings.simplefilter(action='default', category=pd.errors.SettingWithCopyWarning)

nofire_instances = pd.DataFrame(nofire_instances, columns=fire_instances.columns)

```

```

Instancias generadas = 10000 / 157084
Instancias generadas = 20000 / 157084
Instancias generadas = 30000 / 157084
Instancias generadas = 40000 / 157084
Instancias generadas = 50000 / 157084
Instancias generadas = 60000 / 157084
Instancias generadas = 70000 / 157084
Instancias generadas = 80000 / 157084
Instancias generadas = 90000 / 157084
Instancias generadas = 100000 / 157084
Instancias generadas = 110000 / 157084
Instancias generadas = 120000 / 157084
Instancias generadas = 130000 / 157084
Instancias generadas = 140000 / 157084
Instancias generadas = 150000 / 157084

```

### 3.2.3 Juntar instancias positivas y negativas

Una vez que tenemos tanto las instancias positivas como negativas, vamos a juntar ambos conjuntos de datos y obtener el dataset final con el que vamos a hacer los entrenamientos:

In [72]:

```

fire_instances["is_fire"] = 1
nofire_instances["is_fire"] = 0

```

```
final_dataset = pd.concat([fire_instances, nofire_instances], axis=0)
final_dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 235626 entries, 0 to 157083
Data columns (total 91 columns):
 #   Column           Non-Null Count  Dtype  
 ---  -- 
 0   initialdate      235626 non-null   object 
 1   finaldate        235626 non-null   object 
 2   x_index          235626 non-null   object 
 3   y_index          235626 non-null   object 
 4   elevation_sum    235626 non-null   object 
 5   elevation_mean   235626 non-null   object 
 6   elevation_median 235626 non-null   object 
 7   elevation_stdev  235626 non-null   object 
 8   elevation_min    235626 non-null   object 
 9   elevation_max    235626 non-null   object 
 10  elevation_range  235626 non-null   object 
 11  slope_sum        235626 non-null   object 
 12  slope_mean       235479 non-null   object 
 13  slope_median     235479 non-null   object 
 14  slope_stdev      235479 non-null   object 
 15  slope_min        235479 non-null   object 
 16  slope_max        235479 non-null   object 
 17  slope_range       235479 non-null   object 
 18  aspect_NODATA    235626 non-null   object 
 19  aspect_0_45       235626 non-null   object 
 20  aspect_45_90      235626 non-null   object 
 21  aspect_90_135     235626 non-null   object 
 22  aspect_135_180    235626 non-null   object 
 23  aspect_180_225    235626 non-null   object 
 24  aspect_225_270    235626 non-null   object 
 25  aspect_270_315    235626 non-null   object 
 26  aspect_315_360    235626 non-null   object 
 27  dist_to_road_mean 235611 non-null   object 
 28  dist_to_road_median 235611 non-null   object 
 29  dist_to_road_min  235611 non-null   object 
 30  dist_to_road_max  235611 non-null   object 
 31  dist_to_waterway_mean 235611 non-null   object 
 32  dist_to_waterway_median 235611 non-null   object 
 33  dist_to_waterway_min 235611 non-null   object 
 34  dist_to_waterway_max 235611 non-null   object 
 35  dist_to_railway_mean 235626 non-null   object 
 36  dist_to_railway_median 235626 non-null   object 
 37  dist_to_railway_min 235626 non-null   object 
 38  dist_to_railway_max 235626 non-null   object 
 39  natura2000_NODATA 235626 non-null   object 
 40  natura2000_1       235626 non-null   object 
 41  CLC_1             235626 non-null   object 
 42  CLC_2             235626 non-null   object 
 43  CLC_3             235626 non-null   object 
 44  CLC_4             235626 non-null   object 
 45  CLC_5             235626 non-null   object 
 46  CLC_6             235626 non-null   object 
 47  CLC_7             235626 non-null   object 
 48  CLC_8             235626 non-null   object 
 49  CLC_9             235626 non-null   object 
 50  CLC_10            235626 non-null   object 
 51  CLC_11            235626 non-null   object 
 52  CLC_12            235626 non-null   object 
 53  CLC_13            235626 non-null   object 
 54  CLC_14            235626 non-null   object 
 55  CLC_15            235626 non-null   object 
 56  CLC_16            235626 non-null   object 
 57  CLC_17            235626 non-null   object 
 58  CLC_18            235626 non-null   object 
 59  CLC_19            235626 non-null   object 
 60  CLC_20            235626 non-null   object 
 61  CLC_21            235626 non-null   object 
 62  CLC_22            235626 non-null   object 
 63  CLC_23            235626 non-null   object 
 64  CLC_24            235626 non-null   object 
 65  CLC_25            235626 non-null   object 
 66  CLC_26            235626 non-null   object 
 67  CLC_27            235626 non-null   object 
 68  CLC_28            235626 non-null   object 
 69  CLC_29            235626 non-null   object 
 70  CLC_30            235626 non-null   object 
 71  CLC_31            235626 non-null   object 
 72  CLC_32            235626 non-null   object 
 73  CLC_33            235626 non-null   object 
 74  CLC_34            235626 non-null   object
```

```

75 CLC_35           235626 non-null object
76 CLC_36           235626 non-null object
77 CLC_37           235626 non-null object
78 CLC_38           235626 non-null object
79 CLC_39           235626 non-null object
80 CLC_40           235626 non-null object
81 CLC_41           235626 non-null object
82 CLC_42           235626 non-null object
83 CLC_43           235626 non-null object
84 CLC_44           235626 non-null object
85 CLC_NODATA       235626 non-null object
86 population_density_mean 235433 non-null object
87 population_density_min   235433 non-null object
88 population_density_max   235433 non-null object
89 month             235626 non-null object
90 is_fire           235626 non-null int64
dtypes: int64(1), object(90)
memory usage: 165.4+ MB

```

### 3.2.4 Limpieza de datos

Todo este proceso ha hecho que los datos se guarden como objetos y no como valores numéricos. Ahora vamos a convertirlos en valores numéricos y a prepararlos para guardarlos como conjunto de datos final.

```
In [73]: numeric_columns = ["x_index", "y_index",
                      "elevation_sum", "elevation_mean", "elevation_median", "elevation_stdev", "elevation_min",
                      "slope_sum", "slope_mean", "slope_median", "slope_stdev", "slope_min", "slope_max", "slope_180",
                      "aspect_NODATA", "aspect_0_45", "aspect_45_90", "aspect_90_135", "aspect_135_180", "aspect_180_225",
                      "dist_to_road_mean", "dist_to_road_median", "dist_to_road_min", "dist_to_road_max",
                      "dist_to_waterway_mean", "dist_to_waterway_median", "dist_to_waterway_min", "dist_to_waterway_max",
                      "dist_to_railway_mean", "dist_to_railway_median", "dist_to_railway_min", "dist_to_railway_max",
                      "natura2000_NODATA", "natura2000_1"]
]

numeric_columns += [f"CLC_{i}" for i in range(1,45)] + [f"CLC_NODATA"] + [f"population_density_{categ}" for categ in categories]
for col in numeric_columns:
    final_dataset[col] = pd.to_numeric(final_dataset[col], errors='coerce')
```

Por cómo hemos hecho los cálculos en el software de QGIS, las variables Corine Land Cover, Aspect (orientación) y Natura 2000 no cuentan la proporción de casillas de ese tipo que hay en cada celda, por lo que ahora vamos a convertirlas en proporciones:

```
In [74]: # Calculate the Corine Land Cover proportions:
clc_columns = [f"CLC_{i}" for i in range(1,45)] + [f"CLC_NODATA"]
final_dataset['clc_sums'] = final_dataset[clc_columns].sum(axis=1)
for col in clc_columns:
    final_dataset[col] = final_dataset[col] / final_dataset['clc_sums']
final_dataset.drop(columns=['clc_sums'], inplace=True)

# Calculate the Aspect proportions:
aspect_columns = ["aspect_NODATA", "aspect_0_45", "aspect_45_90", "aspect_90_135", "aspect_135_180", "aspect_180_225"]
final_dataset['aspect_sums'] = final_dataset[aspect_columns].sum(axis=1)
for col in aspect_columns:
    final_dataset[col] = final_dataset[col] / final_dataset['aspect_sums']
final_dataset.drop(columns=['aspect_sums'], inplace=True)

# Calculate the Natura2000 proportions:
final_dataset["natura2000_1"] = final_dataset["natura2000_1"] / (final_dataset["natura2000_1"] + final_dataset["natura2000_NODATA"])
final_dataset.drop(columns=[f'natura2000_NODATA'], inplace=True)
final_dataset.drop(columns=[f'finaldate'], inplace=True)
```

Ya tenemos el conjunto de datos final que guardaremos para hacer predicciones:

```
In [75]: csv_file = "../data/FinalDataset/FinalDataset.csv"
final_dataset.to_csv(csv_file, index=False)
```

Estas son las variables que tiene el conjunto de datos:

```
In [76]: df = final_dataset
df.columns
```

```
Out[76]: Index(['initialdate', 'x_index', 'y_index', 'elevation_sum', 'elevation_mean',  
       'elevation_median', 'elevation_stdev', 'elevation_min', 'elevation_max',  
       'elevation_range', 'slope_sum', 'slope_mean', 'slope_median',  
       'slope_stdev', 'slope_min', 'slope_max', 'slope_range', 'aspect_NODATA',  
       'aspect_0_45', 'aspect_45_90', 'aspect_90_135', 'aspect_135_180',  
       'aspect_180_225', 'aspect_225_270', 'aspect_270_315', 'aspect_315_360',  
       'dist_to_road_mean', 'dist_to_road_median', 'dist_to_road_min',  
       'dist_to_road_max', 'dist_to_waterway_mean', 'dist_to_waterway_median',  
       'dist_to_waterway_min', 'dist_to_waterway_max', 'dist_to_railway_mean',  
       'dist_to_railway_median', 'dist_to_railway_min', 'dist_to_railway_max',  
       'natura2000_1', 'CLC_1', 'CLC_2', 'CLC_3', 'CLC_4', 'CLC_5', 'CLC_6',  
       'CLC_7', 'CLC_8', 'CLC_9', 'CLC_10', 'CLC_11', 'CLC_12', 'CLC_13',  
       'CLC_14', 'CLC_15', 'CLC_16', 'CLC_17', 'CLC_18', 'CLC_19', 'CLC_20',  
       'CLC_21', 'CLC_22', 'CLC_23', 'CLC_24', 'CLC_25', 'CLC_26', 'CLC_27',  
       'CLC_28', 'CLC_29', 'CLC_30', 'CLC_31', 'CLC_32', 'CLC_33', 'CLC_34',  
       'CLC_35', 'CLC_36', 'CLC_37', 'CLC_38', 'CLC_39', 'CLC_40', 'CLC_41',  
       'CLC_42', 'CLC_43', 'CLC_44', 'CLC_NODATA', 'population_density_mean',  
       'population_density_min', 'population_density_max', 'month', 'is_fire'],  
      dtype='object')
```

Esta es la proporción final de instancias de `fuego` y `no-fuego`:

```
In [2]: from collections import Counter  
Counter(df["is_fire"])
```

```
Out[2]: Counter({0: 157084, 1: 78542})
```

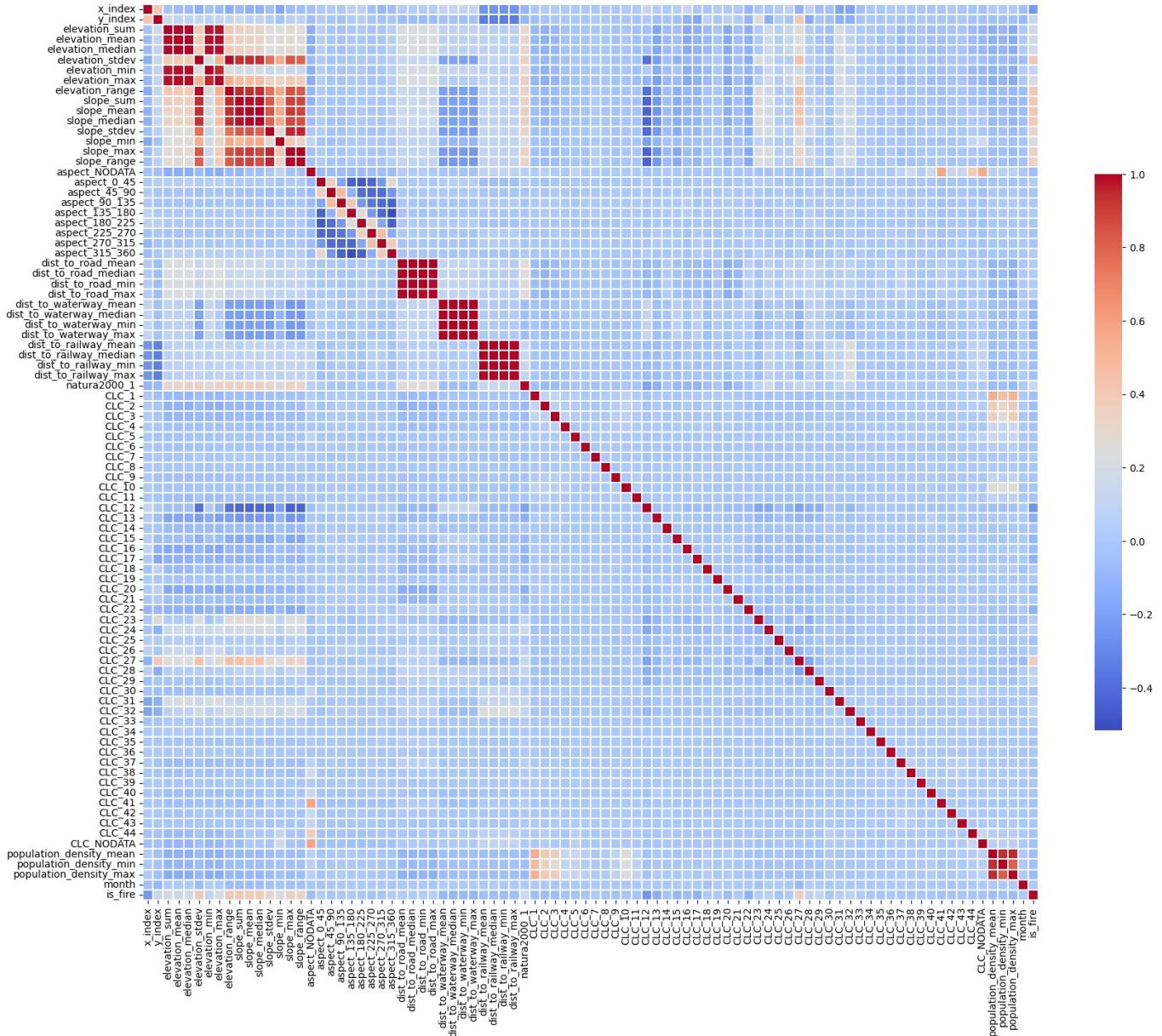
## 4. Visualización de los datos

Una vez que hemos creado el conjunto de datos vamos a visualizar las relaciones que hay entre las variables. Para ello lo primero que vamos a hacer es visualizar las correlaciones entre todas las variables, incluida la variable a predecir.

```
In [1]: import pandas as pd  
df = pd.read_csv("../data/FinalDataset/FinalDataset.csv")
```

```
In [2]: import seaborn as sns  
import matplotlib.pyplot as plt  
  
aux = df.drop(columns=['initialdate']) # Delete the non-numeric columns  
correlation_matrix = aux.corr()  
  
# Visualize the correlation matrix  
plt.figure(figsize=(20, 20))  
sns.heatmap(  
    correlation_matrix,  
    annot=False,  
    cmap='coolwarm',  
    fmt='.2f',  
    cbar=True,  
    square=True,  
    linewidths=0.1,  
    cbar_kws={'shrink': 0.5},  
)  
plt.title('Matriz de correlaciones', fontsize=30)  
plt.xticks(fontsize=10)  
plt.yticks(fontsize=10)  
  
plt.show()
```

# Matriz de correlaciones



En la matriz de correlaciones vemos como las mayores correlaciones se obtienen entre las variables obtenidas de las mismas fuentes como por ejemplo la elevación media, mediana , mínima, máxima y la suma de elevaciones. Por ahora vamos a dejar dichas variables y no vamos a eliminar las redundantes.

```
In [3]: abs(correlation_matrix["is_fire"]).sort_values(ascending=False)[1:20]
```

```
Out[3]: elevation_range      0.406904
slope_sum          0.406793
slope_median       0.401126
slope_mean         0.400350
elevation_stdev    0.395481
CLC_27            0.366918
slope_max          0.346648
slope_range        0.338408
slope_stdev        0.309623
CLC_12            0.252082
x_index           0.238055
elevation_max      0.222899
slope_min          0.216587
elevation_sum       0.172225
elevation_median    0.171649
elevation_mean      0.171503
natura2000_1        0.156645
CLC_13            0.142565
y_index           0.141098
Name: is_fire, dtype: float64
```

Por otro lado, si nos fijamos en la variable predictora, las variables que más correlación tienen con ella son las siguientes (entre otras):

- elevation\_stdev
- elevation\_range
- elevation\_mean

- slope\_sum
- slope\_mean
- slope\_median
- slope\_stdev
- slope\_max
- slope\_range
- dist\_to\_waterway\_mean
- CLC\_12
- CLC\_13
- CLC\_27

Teniendo en cuenta que algunas de las variables están correlacionadas entre si, vamos a visualizar los histogramas de densidad en base a la clase de "is\_fire" de las siguientes variables:

- elevation\_range
- elevation\_mean
- slope\_mean
- slope\_min
- dist\_to\_waterway\_mean
- CLC\_12
- CLC\_13
- CLC\_27

```
In [4]: import seaborn as sns
import matplotlib.pyplot as plt

# List of column names
columns_to_plot = [
    'elevation_range',
    'elevation_mean',
    'slope_mean',
    'slope_min',
    'dist_to_waterway_mean',
    'CLC_12',
    'CLC_13',
    'CLC_27'
]

plt.rc('font', size=12)

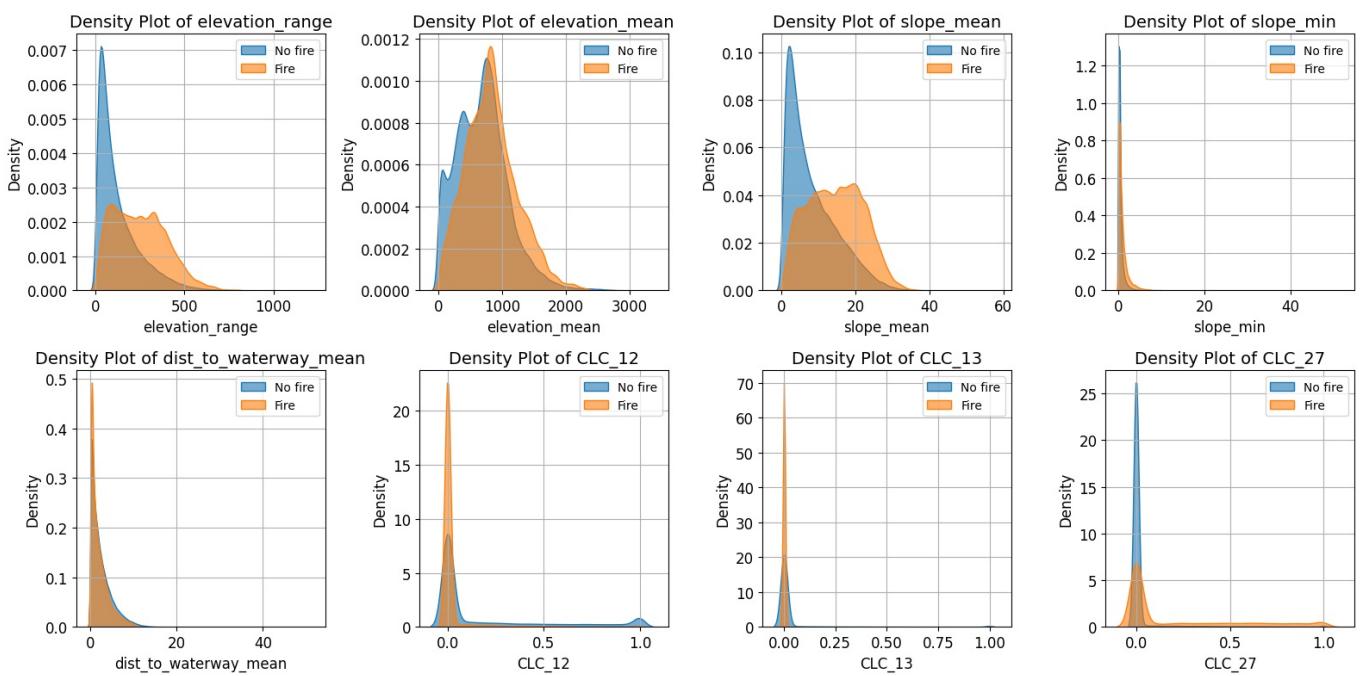
fig, axes = plt.subplots(2, 4, figsize=(16, 8))
axes = axes.flatten()

# Iterate through columns and axes
for i, column_name in enumerate(columns_to_plot):
    var_fire = df[df['is_fire'] == 1][column_name].values
    var_no_fire = df[df['is_fire'] == 0][column_name].values

    sns.kdeplot(var_no_fire, label="No fire", fill=True, alpha=0.6, ax=axes[i])
    sns.kdeplot(var_fire, label="Fire", fill=True, alpha=0.6, ax=axes[i])

    axes[i].set_title(f'Density Plot of {column_name}', fontsize=14)
    axes[i].set_xlabel(column_name, fontsize=12)
    axes[i].set_ylabel('Density', fontsize=12)
    axes[i].legend(fontsize=10)
    axes[i].grid(True)

# Adjust layout for better appearance
plt.tight_layout()
plt.show()
```



De estos gráficos de densidades podemos concluir que para ciertas variables, como `elevation_range`, `elevation_mean` y `slope_mean`, existe una superposición significativa entre las distribuciones de "Fire" y "No Fire". Aunque estas variables tengan una tendencia a ser mayores para el caso de "Fire", como la superposición de las distribuciones es muy grande puede que no sean particularmente útiles para diferenciar entre áreas propensas a incendios.

Sin embargo, en las variables `CLC_12`, `CLC_13` y `CLC_27`, se observan diferencias notables en las densidades entre los incendios y las áreas sin incendios. Esto sugiere que ciertos tipos de cobertura terrestre están más asociados con la ocurrencia de incendios. Por ejemplo, las clases 12 y 13 de Corine Land Cover son poco propensas a tener incendios, mientras que la clase 27 es más propensa a tenerlos.

Por último, en cuanto al impacto de la distancia a ríos, es difícil sacar conclusiones ya que ambas distribuciones son muy parecidas. Esto indica que puede que la variable no sea muy útil a la hora de predecir los incendios.

## Valores NA

Como hemos usado diferentes fuentes de datos para crear el conjunto de datos final, en el proceso de creación han aparecido algunos valores perdidos. Esto puede deberse a la falta de exactitud en algunos mapas, o a que en algunas celdas espaciales no haya información sobre algunas variables por razones desconocidas. Analizemos ahora cuántos valores perdidos tenemos.

```
In [5]: # Columns with no missing values
missing_columns = [col for col in df.columns if not df[col].notna().all()]
print("Columnas con valores perdidos:", missing_columns)
```

Columnas con valores perdidos: ['slope\_mean', 'slope\_median', 'slope\_stdev', 'slope\_min', 'slope\_max', 'slope\_range', 'dist\_to\_road\_mean', 'dist\_to\_road\_median', 'dist\_to\_road\_min', 'dist\_to\_road\_max', 'dist\_to\_waterway\_mean', 'dist\_to\_waterway\_median', 'dist\_to\_waterway\_min', 'dist\_to\_waterway\_max', 'population\_density\_mean', 'population\_density\_min', 'population\_density\_max']

```
In [6]: aux = df[missing_columns]
missing_counts = aux.isna().sum()
missing_percentage = (aux.isna().sum() / len(aux)) * 100

missing_summary = pd.DataFrame({
    'Missing Count': missing_counts,
    'Missing Percentage': missing_percentage
}).sort_values(by='Missing Count', ascending=False)

missing_summary
```

Out[6]:

	Missing Count	Missing Percentage
population_density_max	193	0.081909
population_density_min	193	0.081909
population_density_mean	193	0.081909
slope_range	147	0.062387
slope_median	147	0.062387
slope_mean	147	0.062387
slope_max	147	0.062387
slope_min	147	0.062387
slope_stdev	147	0.062387
dist_to_road_mean	15	0.006366
dist_to_road_median	15	0.006366
dist_to_road_max	15	0.006366
dist_to_waterway_mean	15	0.006366
dist_to_waterway_median	15	0.006366
dist_to_waterway_min	15	0.006366
dist_to_waterway_max	15	0.006366
dist_to_road_min	15	0.006366

Tras contar los valores perdidos en cada variable, vemos que son muy pocos, menos del 0.1% por variable. De hecho, si eliminamos las instancias que contienen estos valores perdidos, se eliminarían 338 instancias de 235626, quedandonos con 235288 instancias.

In [7]: `len(df), len(df.dropna()), len(df) - len(df.dropna())`

Out[7]: (235626, 235288, 338)

In [8]: `instancias_con_na = df[df.isna().any(axis=1)]  
instancias_con_na["is_fire"].value_counts()`Out[8]: `is_fire  
0 252  
1 86  
Name: count, dtype: int64`

En nuestro caso esto es lo que vamos ha hacer, vamos a eliminar todos los valores perdidos que tenemos. Si introdujesemos en los valores perdidos las medias de las variables (o usasemos otra técnica para añadir otros valores), estaríamos añadiendo características que desconocemos si son las correctas en esas celdas espaciales. Es decir, podríamos introducir valores topográficos (como la pendiente) muy incorrectos para esas celdas, lo cual haría que entrenaesemos el algoritmo en instancias que nunca van a existir. Además, como la cantidad de valores perdidos es mínima, y de los 338 valores perdidos, 252 podríamos generarlos de nuevo y obtener instancias de "no-fuego" nuevas sin valores perdidos, eliminarlos no nos hace prácticamente ningún daño.

In [9]: `df.dropna(inplace=True)`

## 5. "Feature Selection"

En esta sección vamos a eliminar algunas variables mediante técnicas de selección de variables. Para ello, vamos a eliminar las variables altamente correlacionadas y a aplicar tests estadísticos univariados para analizar qué variables podríamos eliminar.

### Matriz de correlaciones

Como hemos dicho, vamos a eliminar las columnas con correlación alta, es decir, eliminaremos las columnas con correlación absoluta mayor a 0.98. Este valor es muy alto ya que al ser una tarea muy compleja con muchas relaciones multivariadas, no queremos perder información que podría ser útil al poner un límite más bajo.

```
In [10]: import numpy as np  
upper_triangle = correlation_matrix.where(np.triu(np.ones(correlation_matrix.shape), k=1).astype(bool))  
  
columns_to_drop = [column for column in upper_triangle.columns if any(abs(upper_triangle[column]) > 0.98)]  
  
print("Deleted columns: ")  
print("-----", end = "\n\n")  
for col in columns_to_drop:  
    print(col)  
df.drop(columns=columns_to_drop, inplace=True)
```

Deleted columns:

```
-----  
elevation_mean  
elevation_median  
elevation_min  
elevation_max  
elevation_range  
slope_mean  
slope_median  
slope_range  
dist_to_road_median  
dist_to_road_min  
dist_to_road_max  
dist_to_waterway_median  
dist_to_waterway_min  
dist_to_waterway_max  
dist_to_railway_median  
dist_to_railway_min  
dist_to_railway_max
```

### Tests estadísticos univariados

Otra técnica que vamos a utilizar son los tests estadísticos univariados. Estos tests toman variables de uno en uno, por lo que no toman en cuenta las complejas relaciones que pueden haber entre diferentes variables. Aún así, vamos a hacer dichos tests para analizar qué variables podrían ser más importantes y cuáles no.

```
In [11]: from sklearn.feature_selection import SelectKBest, f_classif  
import pandas as pd  
  
X = df.drop(columns=["is_fire", "initialdate"])  
y = df["is_fire"]  
  
# Use SelectKBest with ANOVA F-statistic  
selector = SelectKBest(score_func=f_classif, k=1)  
X_selected = selector.fit_transform(X, y)  
  
scores = selector.scores_  
feature_scores = pd.DataFrame({  
    "Feature": X.columns,  
    "Score": scores  
})  
  
feature_scores.sort_values(by="Score", ascending=False)
```

	Feature	Score
4	slope_sum	46719.673732
3	elevation_stdev	43613.578962
47	CLC_27	36557.095130
7	slope_max	32166.299836
5	slope_stdev	24970.013250
...	...	...
69	month	1.577093
54	CLC_34	1.469503
39	CLC_19	1.037899
11	aspect_90_135	0.485102
59	CLC_39	0.285541

70 rows × 2 columns

Vemos como la variable "mes" está como una de las menos importantes, sin embargo, está claro que es uno de los factores que más afecta en el riesgo de incendio. Por ello, no vamos a hacer caso a las conclusiones obtenidas mediante este análisis.

## 6. Detección de "outliers"

En esta sección vamos a usar un algoritmo de detección de outliers para encontrarlos en el conjunto de datos. Una vez lo hayamos hecho, vamos a intentar descubrir por qué son outliers y si tiene sentido eliminarlos, ya que por cómo hemos generado el dataset, tener outliers significa que hay celdas espaciales que son "extrañas" o "poco comunes" en el territorio español.

El algoritmo que vamos a utilizar será el algoritmo de "Isolation Forest", el cual mide el "outlierness" en base a la cantidad de hiperplanos que necesita para aislar una instancia. Utilizando ese algoritmo vamos a dibujar la distribución de los valores de "outlierness" que tienen

nuestros datos. Para ello, vamos a introducirle al algoritmo que la proporción de instancias que son outliers es el 1%.

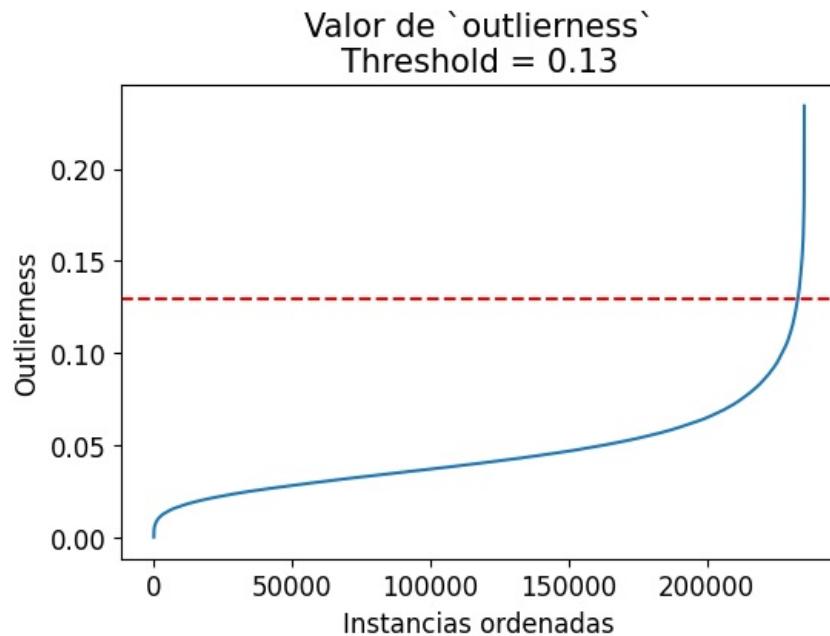
```
In [12]: from sklearn.ensemble import IsolationForest
import numpy as np

isolation_forest = IsolationForest(n_estimators=100,
                                    max_samples='auto',
                                    contamination=0.01,
                                    random_state=42)

y_pred = isolation_forest.fit_predict(df.drop(columns=['initialdate', 'is_fire']))
outlier_scores = -isolation_forest.decision_function(df.drop(columns=['initialdate', 'is_fire'])) # Grado de outlier

# Grafico de outlierness ordenado, con el threshold marcado:
threshold = np.percentile(outlier_scores, 100 * (1 - isolation_forest.contamination))
sorted = np.sort(outlier_scores)
min_val = sorted[0]
sorted = sorted + abs(min_val)
threshold = threshold + abs(min_val)
outlier_scores = outlier_scores + abs(min_val)
colors = ['blue' if value < threshold else 'red' for value in sorted]
# Plot the sorted data with colors
plt.figure(figsize=(6, 4))
plt.plot(range(len(sorted)), sorted)
plt.xlabel("Instancias ordenadas")
plt.ylabel("Outlierness")
plt.title(f"Valor de `outlierness`\nThreshold = {np.round(threshold,3)}", fontsize=15)
plt.axhline(y=threshold, color='red', linestyle='--')

plt.show()
```



```
In [13]: outliers = df[outlier_scores > threshold]
```

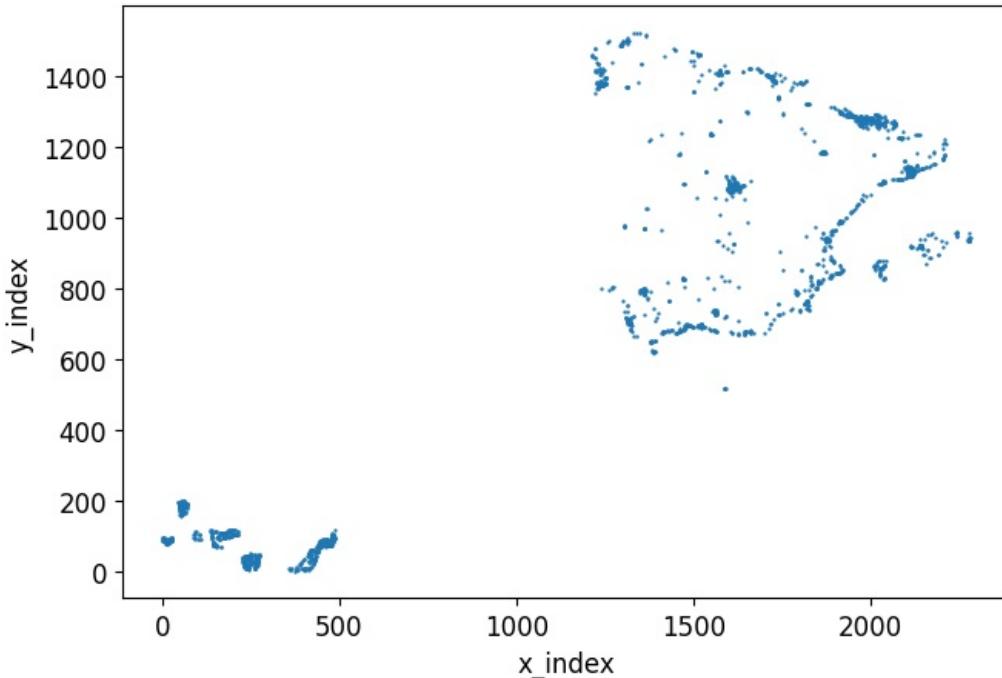
## 6.1 ¿Por qué son outliers?

Ahora que hemos identificado los outliers, vamos a encontrar la razón por la cual son outliers. Para ello, lo primero que vamos a hacer es, ya que tenemos información geográfica, visualizar dónde están los outliers.

```
In [14]: fig, ax = plt.subplots(1, 1, figsize=(8, 5))
scatter = ax.scatter(
    outliers["x_index"],
    outliers["y_index"],
    s=0.5
)
ax.set_aspect('equal', 'box')
ax.set_xlabel("x_index")
ax.set_ylabel("y_index")
ax.set_title("Outliers en el dataset", fontsize=18)

plt.show()
```

## Outliers en el dataset



Vemos que la mayoría de outliers se encuentran en los pirineos, en Madrid y en las Islas Canarias. Esto puede darnos una pista del porqué esas instancias son outliers: puede ser por la elevación del terreno (en el caso de los pirineos por ejemplo). Con esa información en mente hemos dibujado gráficos de densidad de los outliers y los hemos comparado con los gráficos de densidad de todo el conjunto de datos. Las variables con gráficos más diferentes que hemos encontrado son los siguientes:

- Elevación
- Distancia a vías de tren
- Densidad poblacional

Además, también hemos observado que los outliers que hemos obtenido no siguen la distribución de "fuego" y "no-fuego" que tienen los datos: hay muchos mas outliers "no-fuego" que outliers "fuego". La proporción es de alrededor de 5 instancias de "no-fuego" por cada instancia de "fuego", mientras que en el conjunto de datos entero la proporción es de 2 a 1.

```
In [15]: outliers["is_fire"].value_counts()
```

```
Out[15]: is_fire
0    1939
1     414
Name: count, dtype: int64
```

Visualicemos las distribuciones de las variables "elevación", "densidad poblacional" y "distancia a vías de tren". Para ello graficaremos las distribuciones de los datos para el conjunto de datos entero y para los outliers. Además, visualizaremos los datos geográficos y dibujaremos el color de cada punto geográfico en base al valor de las variables en dicho punto:

```
In [16]: fig, axes = plt.subplots(2, 3, figsize=(20, 10))

columns = ["elevation_sum", "population_density_mean", "dist_to_railway_mean"]

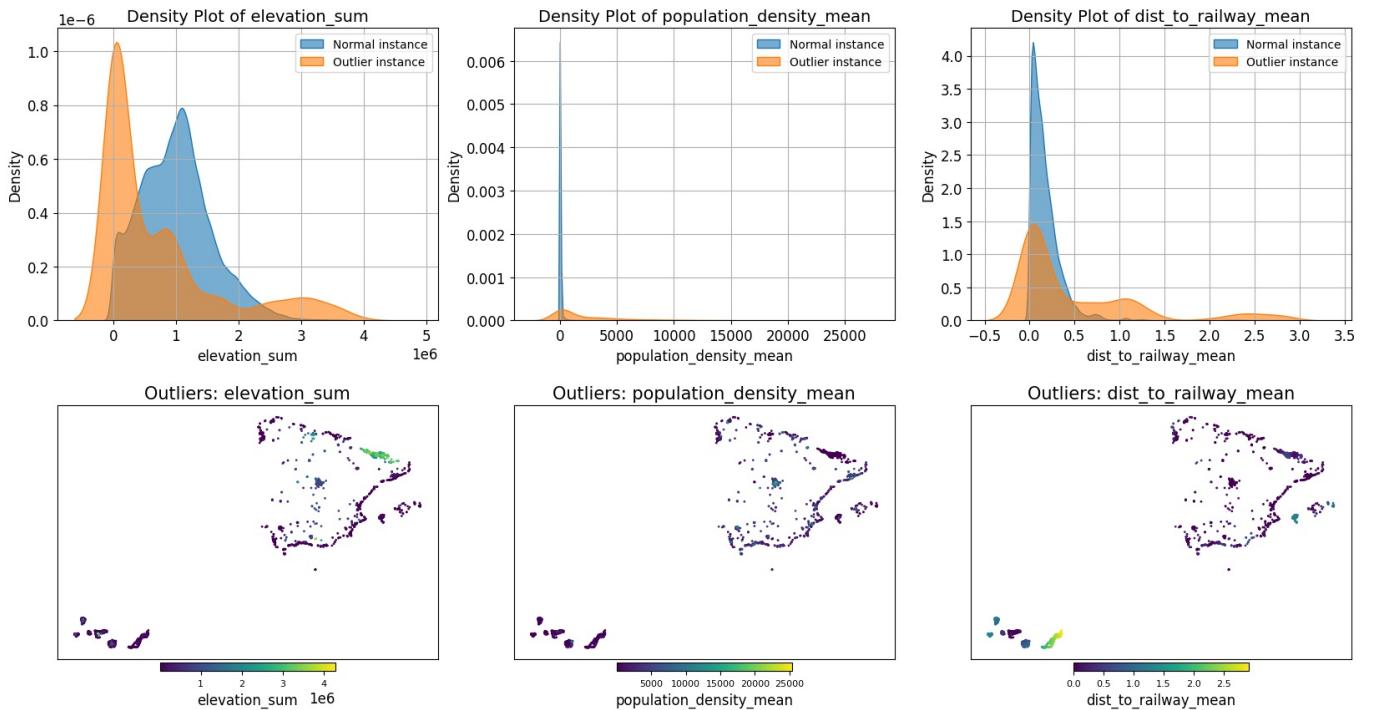
for i, col in enumerate(columns):

    # Density plot
    var_normals = df[col].values
    var_outliers = outliers[col].values
    sns.kdeplot(var_normals, label="Normal instance", fill=True, alpha=0.6, ax=axes[0,i])
    sns.kdeplot(var_outliers, label="Outlier instance", fill=True, alpha=0.6, ax=axes[0,i])

    axes[0,i].set_title(f'Density Plot of {col}', fontsize=14)
    axes[0,i].set_xlabel(col, fontsize=12)
    axes[0,i].set_ylabel('Density', fontsize=12)
    axes[0,i].legend(fontsize=10)
    axes[0,i].grid(True)

    # Geographic plot
    scatter = axes[1,i].scatter(
        outliers["x_index"],
        outliers["y_index"],
        c=outliers[col],
        cmap='viridis', # Choose a colormap, e.g., 'viridis', 'plasma', etc.
        s=1 # Adjust marker size
    )
```

```
# Add a colorbar for reference
cbar = plt.colorbar(scatter, ax=axes[1,i], label=col, orientation='horizontal', fraction=0.03, pad=0.01)
cbar.ax.tick_params(labelsize=8)
axes[1,i].set_aspect('equal', 'box')
axes[1,i].set_xticks([])
axes[1,i].set_yticks([])
axes[1,i].set_title("Outliers: " + col, fontsize=15)
```



Vemos como claramente los outliers que se encuentran en los pirineos son detectados como outliers porque tienen una elevación mucho más alta que el resto del territorio que estamos analizando. De hecho, si nos fijamos en la media de elevación, observamos como en los outliers por un lado tenemos elevaciones muy bajas (muchos puntos en las partes costeras de España) y elevaciones muy altas (los pirineos).

Por otro lado, también se ve claramente que los outliers tienen mayor densidad poblacional que la distribución general que siguen los datos. Esto se ve claramente en Madrid, ya que es con diferencia la región con mayor densidad poblacional de España.

Por último, en cuanto a distancia a vías de tren, lo más destacable son las Islas Canarias. Por los datos de Open Street Map que hemos utilizado para crear la capa de distancia a carreteras, en algunas islas no hay vías de tren por lo que si añadimos la distancia entre las islas, obtenemos valores muy altos. Esto puede indicarnos que debamos añadir un máximo en el valor de esa variable.

Como vemos, los outliers que obtenemos son localizaciones con propiedades poco comunes, sin embargo eliminarlos del conjunto de datos no nos interesa, ya que también queremos calcular el riesgo de incendio en esas localizaciones "extrañas". Si las eliminásemos el algoritmo no sabría qué hacer en esas instancias a la hora de predecir riesgo de incendio en todo el territorio. Por todo ello, no las vamos a eliminar.

## 7. PCA y t-SNE

En esta sección vamos a utilizar dos herramientas de visualización de datos: "Principal Component Analysis" y "t-SNE". Éstos son métodos de reducción de dimensionado y pueden utilizarse para visualizar datos multivariados en dos dimensiones. Para ello, introduciremos todas las variables numéricas de nuestro conjunto de datos y las estandarizaremos antes de introducirnos a los algoritmos.

```
In [61]: from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler

# Standardize the data
X = df.drop(columns=['initialdate', 'is_fire'])

X_scaled = StandardScaler().fit_transform(X)
```

### PCA

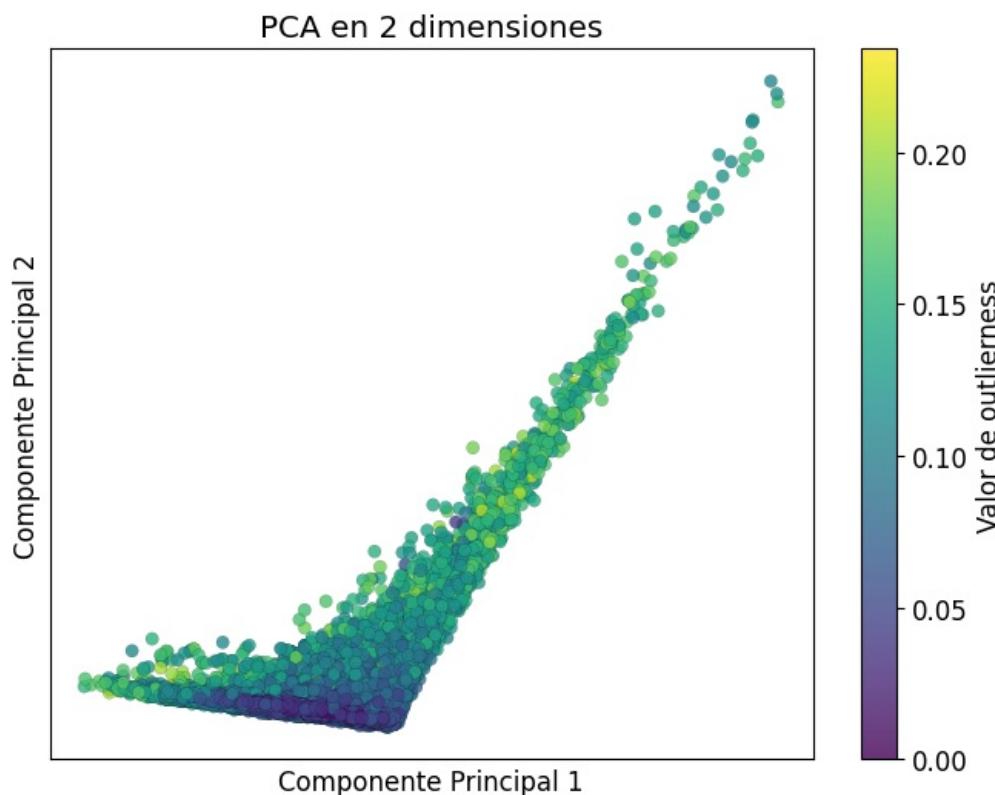
Primero vamos a hacer el análisis de componentes principales, el cual obtiene las componentes que maximizan la varianza explicada en los datos. Un aspecto importante del PCA es la variabilidad explicada, ya que si no es muy grande con la cantidad de dimensiones escogida, el análisis puede no representar lo suficientemente bien los datos. Un valor de referencia es el 80%, si la varianza explicada es menor que eso, hay que tener cuidado con las conclusiones.

```
In [62]: # PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
pca.explained_variance_
```

```
Out[62]: array([5.27881361, 3.59116964])
```

Si visualizamos las primeras dos componentes principales y dibujamos el color en base al valor de "outlierness" obtenido en la sección previa obtenemos el siguiente gráfico:

```
In [63]: plt.figure(figsize=(8, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=outlier_scores, cmap= "viridis", alpha = .8, edgecolors="black", linewidth=1)
colorbar = plt.colorbar()
colorbar.set_label('Valor de outlierness')
plt.title('PCA en 2 dimensiones')
plt.xlabel('Componente Principal 1')
plt.ylabel('Componente Principal 2')
plt.xticks([])
plt.yticks([])
plt.show()
```



Vemos como al hacer el análisis de PCA los datos están todos juntos, lo cual puede significar que la mayoría de instancias son muy parecidas y de que no hay clusters bien definidos. Además, vemos como claramente las instancias que están más "centradas" tienen menor valor de outlierness, lo cual significa que son instancias más "comunes", mientras que las instancias más alejadas del centro tienen, en general, mayor valor de "outlierness".

Sin embargo, si nos fijamos en la varianza explicada, vemos que es menos del 8% entre las dos variables, por lo que no debemos hacer mucho caso al análisis de PCA.

## t-SNE

El t-SNE (t-Distributed Stochastic Neighbor Embedding) es una técnica de reducción de dimensionalidad no lineal diseñada para visualizar datos de alta dimensión en un espacio de dos o tres dimensiones, preservando las relaciones locales entre puntos. Para ello se calculan las similitudes entre puntos en el espacio original basándose en distribuciones de probabilidad. El t-SNE es más costoso computacionalmente que el PCA, por lo que hemos guardado los resultados en un archivo `pickle`.

```
In [64]: calculate_tsne = False
```

```
In [65]: import pickle

if calculate_tsne: # The t-SNE calculation takes a long time
    tsne = TSNE(n_components=2,
                 random_state=42,
                 n_jobs=-1,
                 perplexity=40)
    X_tsne = tsne.fit_transform(X_scaled)
    pickle.dump(X_tsne, open("../data/models/X_tsne.pkl", "wb"))
```

```
pickle.dump(tsne, open("../data/models/tsne.pkl", "wb"))
```

Una vez ejecutado el algoritmo vamos a visualizar los resultados obtenidos en dos dimensiones coloreando las instancias en base a diferentes variables. Las variables que vamos a dibujar serán las siguientes:

- "Outlierness"
- Artificial Surfaces: La suma de las proporciones de las clases 1-11 de Corine Land Cover, tal y como se dice en la tabla de explicación de las clases.
- Agricultural areas: La suma de las proporciones de las clases 12-22 de Corine Land Cover
- Forest areas: La suma de las proporciones de las clases 23-34 de Corine Land Cover
- "x\_index"
- "y\_index"

Vamos a dibujar estas variables porque son las que mejor dividen las dimensiones de t-SNE (tras probar con todas las variables).

```
In [66]: import pickle

X_tsne = pickle.load(open("../data/models/X_tsne.pkl", "rb"))
tsne = pickle.load(open("../data/models/tsne.pkl", "rb"))

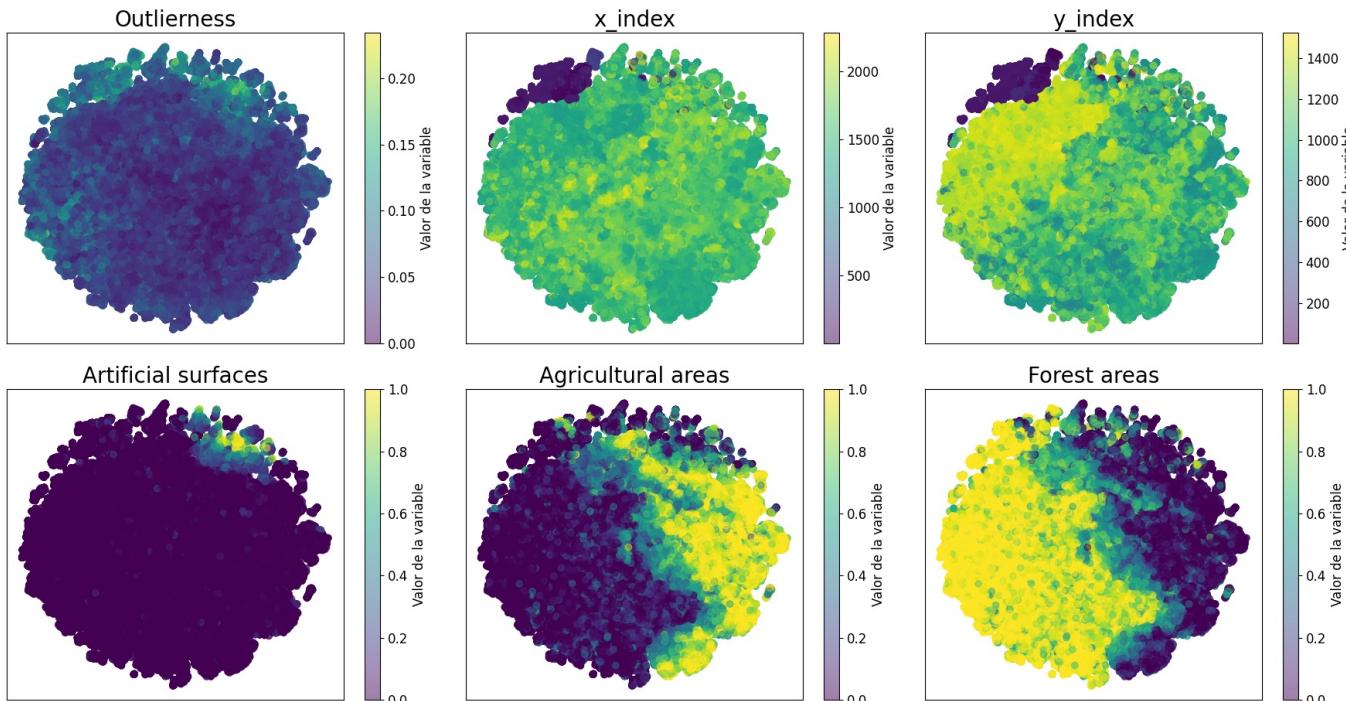
artificial_surfaces = df[["CLC_" + str(i) for i in range(1,12)]].sum(axis=1)
agricultural_areas = df[["CLC_" + str(i) for i in range(12,23)]].sum(axis=1)
forest_areas = df[["CLC_" + str(i) for i in range(23,35)]].sum(axis=1)

columns = [outlier_scores, df["x_index"], df["y_index"], artificial_surfaces, agricultural_areas, forest_areas]
titles = ["Outlierness", "x_index", "y_index", "Artificial surfaces", "Agricultural areas", "Forest areas"]

fig, axes = plt.subplots(2, 3, figsize=(18, 10), sharex=True, sharey=True)
for i, (col, title) in enumerate(zip(columns, titles)):
    row, col_idx = divmod(i, 3) # Convertir el índice lineal en fila y columna
    scatter = axes[row, col_idx].scatter(X_tsne[:, 0], X_tsne[:, 1], c=col, cmap="viridis", alpha=0.5)
    colorbar = fig.colorbar(scatter, ax=axes[row, col_idx])
    colorbar.set_label('Valor de la variable')
    axes[row, col_idx].set_title(title, fontsize=20)
    axes[row, col_idx].set_xticks([])
    axes[row, col_idx].set_yticks([])

fig.suptitle("Visualización de t-SNE mediante varias variables", fontsize=30, y=1.02)
plt.tight_layout()
plt.show()
```

## Visualización de t-SNE mediante varias variables



Por un lado, en cuanto al "outlierness" vemos como hay una cierta tendencia a que los valores del centro tengan menor valor de "outlierness" que los de fuera. Sin embargo, esa tendencia es mucho menor que comparandolo con el análisis de PCA.

Por otro, vemos como las instancias con menor valor en los índices "x\_index" e "y\_index" están claramente separados. Esas instancias son las Islas Canarias que, como hemos dicho anteriormente, también son outliers debido a factores como distancia a vías de tren.

Por último, también se ve que las instancias del conjunto de datos están "agrupadas" en base a los tres componentes que hemos

definido: zonas artificiales, zonas agrícolas y zonas forestales. Esto nos indica que añadir esas tres variables al conjunto de datos puede ser interesante.

## 8. Preprocesos de datos previo a los entrenamientos

Antes de pasar a dividir el conjunto de datos en conjunto de entrenamiento y test, vamos a hablar del preproceso de datos que vamos a aplicar. Ya hemos eliminado los "NA values" o valores perdidos, por lo que vamos a pasar a hablar sobre el escalado de los datos.

Como tenemos muchas variables, las escalas de los mismos varían mucho. Por ejemplo, la elevación llega a tener valores de hasta 3000 metros mientras que "CLC\_1" tiene valores entre 0 y 1. Como tenemos muchas variables del tipo "proporción", es decir, variables con valores entre 0 y 1, vamos a aplicar un escalado del tipo "minmax" que asegure que los datos de entrenamiento estén entre 0 y 1. Esto lo haremos para los datos de entrenamiento y usaremos esos mínimos y máximos para transformar los datos de test, así evitaremos la contaminación de datos ("data leakage").

Aplicaremos la transformación a las siguientes variables:

- x\_index, y\_index
- elevation (todas)
- slope (todas)
- dist\_to\_road (todas)
- dist\_to\_waterway (todas)
- dist\_to\_railway (todas)
- population\_density (todas)

Las otras variables ya tienen los valores en el rango entre 0 y 1, ya que son proporciones. Dichas variables son:

- aspect (todas)
- Natura 2000
- CLC (todas)

Por otro lado, debido al análisis de instancias "outlier" que hemos hecho, antes de aplicar esta transformación añadiremos un valor máximo en la variable "dist\_to\_railway". También añadiremos las variables "artificial\_percent", "agricultural\_percent" y "forest\_percent", explicadas en la sección anterior al hacer el análisis de t-SNE.

Por último, la variable "month" la añadiremos como variable categórica mediante "one-hot-encoding".

```
In [67]: df[["artificial_percent"]] = df[["CLC_" + str(i) for i in range(1,12)]].sum(axis=1)
df[["agricultural_percent"]] = df[["CLC_" + str(i) for i in range(12,23)]].sum(axis=1)
df[["forest_percent"]] = df[["CLC_" + str(i) for i in range(23,35)]].sum(axis=1)

df[["dist_to_railway_mean"]] = df[["dist_to_railway_mean"]].clip(upper=1.5)

df[["month"]] = df[["month"]].astype("category")
df = pd.get_dummies(df, columns=["month"], prefix="month")
```

Nota: El escalado de los datos lo haremos en la siguiente sección tras dividir el conjunto de datos en conjunto de entrenamiento y de test, para evitar "data leakage".

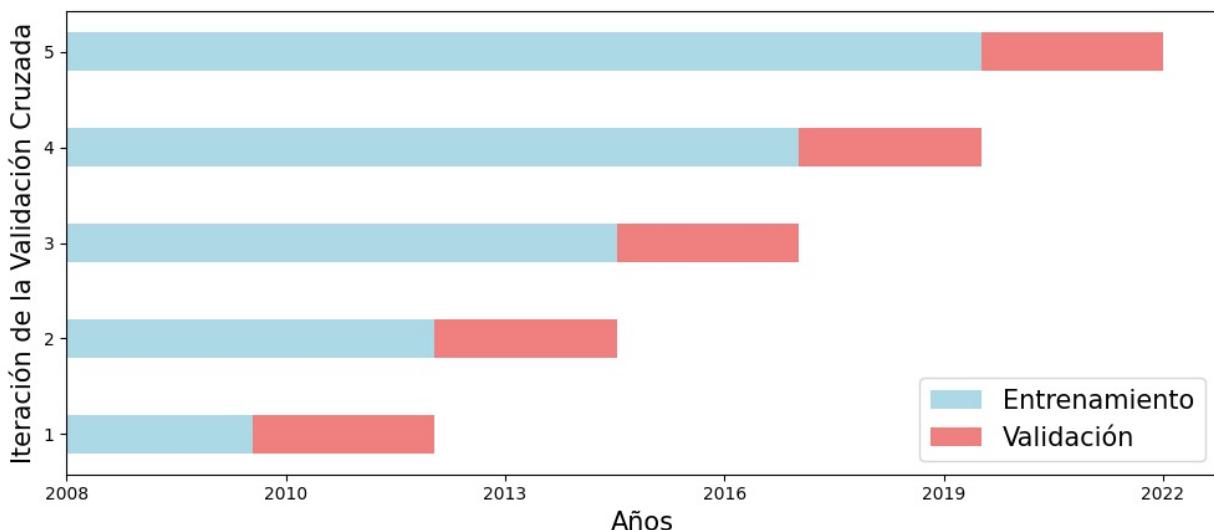
## 9. Partición de entrenamiento, validación y test

Por la forma de generar las instancias, un mismo incendio puede generar varias instancias de "fuego" en el conjunto de datos. Si dividiéramos el conjunto de datos aleatoriamente en subconjuntos de entrenamiento y test, sería muy probable que entrenasemos y testasemos en instancias generadas por el mismo "fuego". Es decir, testaríamos en instancias muy parecidas con las cuales se ha entrenado. Esto haría que sobreestimásemos las capacidades reales del modelo. Por ello, vamos a utilizar validación de series temporales.

La validación cruzada en series temporales se hace manteniendo la temporalidad de los datos. Se entrena en datos del pasado y se valida en datos del futuro, además, la ventana de entrenamiento se va aumentando en cada etapa de la validación cruzada, a medida que se añaden instancias temporales.

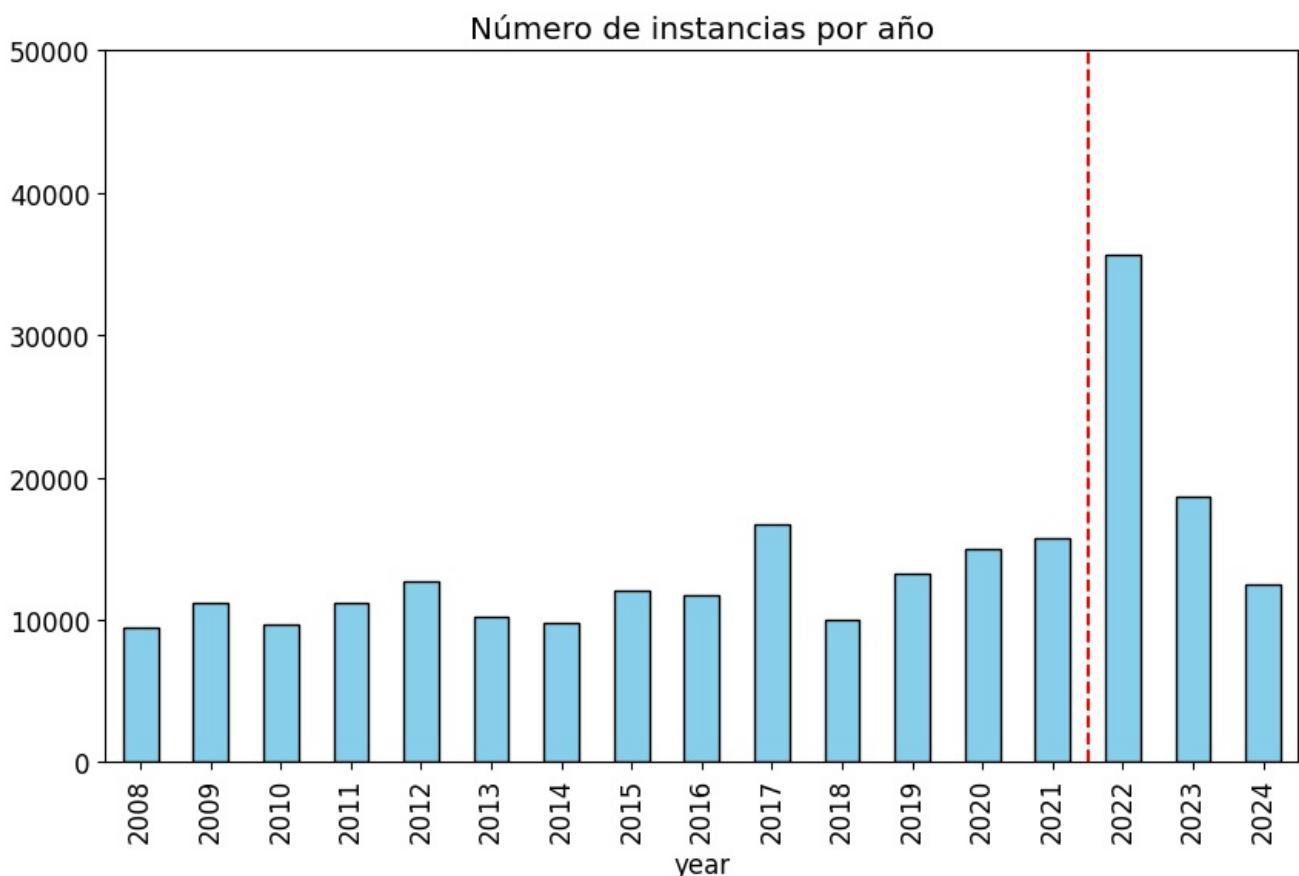
Por ejemplo, 5 etapas de validación cruzada en series temporales se harían así:

## Validación Cruzada en Series Temporales ( $k = 5$ )



Para ello, vamos a añadir la columna "año" en nuestro conjunto de datos, la cual usaremos para dividir el conjunto de datos en datos de entrenamiento y datos de test. Para asegurarnos de que hay una cantidad suficiente para entrenar y testar, visualizemos la cantidad de instancias que tenemos en base al año:

```
In [68]: df["year"] = pd.to_datetime(df["initialdate"]).dt.year
df["year"].value_counts().sort_index().plot(kind='bar', figsize=(10, 6),
                                              color='skyblue',
                                              edgecolor='black',
                                              title='Número de instancias por año')
plt.vlines(x=13.5, ymin=0, ymax=50000, color='red', linestyle='--', label='Año de corte')
plt.ylim(0, 50000)
plt.show()
```



Vemos como en general tenemos una misma cantidad de datos en todos los años menos en 2022, que hay el doble de lo normal. Si tomásemos los datos de entrenamiento los años 2008-2021, tendríamos un 71% de datos para entrenar. Si tomásemos los datos de 2008-2022, tendríamos un 86% de datos para entrenar. Creemos que entrenar con alrededor del 70% de los datos es suficiente, por lo que nos vamos a quedar con esta opción. Además, como el año 2022 es un año anómalo, entrenar en esos datos podría hacer que el modelo aprendiese patrones que no van a replicarse en el futuro.

```
In [69]: h = df["year"].value_counts()
h[[i for i in range(2008,2022)]].sum() / h.sum(), h[[i for i in range(2008,2023)]].sum() / h.sum()
```

```
Out[69]: (0.7162498724966849, 0.8675325558464521)
```

Vamos a ordenar las instancias del conjunto de datos en base a la fecha de la celda espacio-temporal.

```
In [70]: df = df.sort_values(by='initialdate').reset_index(drop=True) # The old index is not saved as a new column
```

## 9.1 Partición de entrenamiento y test:

Ahora estamos listos para dividir el conjunto de entrenamiento en conjunto de "train" y conjunto de "test". Como hemos dicho antes, vamos a tomar para entrenar los años 2008-2021 y para testar los años 2022-2024. Además, no vamos a introducir la variable "año" ni "fecha inicial" como variables del algoritmo.

Como el año 2022 fue un año con muchísima más área quemada de lo normal, vamos a dividir el conjunto de test en dos subconjuntos más para poder testar en un conjunto "normal" (años 2023-2024) y un conjunto "poco normal" (año 2022).

```
In [71]: X_train = df[df["year"] < 2022].drop(columns=['initialdate', 'is_fire', 'year'])
y_train = df[df["year"] < 2022]['is_fire']

X_test = df[df["year"] >= 2022].drop(columns=['initialdate', 'is_fire', 'year'])
y_test = df[df["year"] >= 2022]['is_fire']

X_test_2022 = df[df["year"] == 2022].drop(columns=['initialdate', 'is_fire', 'year'])
y_test_2022 = df[df["year"] == 2022]['is_fire']

X_test_2023_24 = df[df["year"] >= 2023].drop(columns=['initialdate', 'is_fire', 'year'])
y_test_2023_24 = df[df["year"] >= 2023]['is_fire']
```

## 9.2 Data scaling

Ahora que tenemos los conjuntos de entrenamiento y de test, vamos a escalar los datos. Para ello, como hemos dicho en la sección anterior, vamos a aplicar el escalado "minmax" en el conjunto de entrenamiento, con el cual nos aseguraremos de tener valores entre 0 y 1 en todas las columnas que no tengan esa escala. Luego, utilizaremos esas mismas transformaciones (con los mismos valores máximos y mínimos del conjunto de entrenamiento), para transformar el conjunto de testeo. Esto lo hacemos así para no contaminar los datos de entrenamiento con los datos de test.

Si usásemos los valores máximos y mínimos de todo el conjunto de datos (incluyendo el conjunto de test), y luego hiciéramos la partición de entrenamiento y test, los datos de entrenamiento tendrían información del conjunto de test, ya que podrían haber sido transformados utilizando valores máximos o mínimos que se encuentran en dicho conjunto de datos.

```
In [72]: from sklearn.preprocessing import MinMaxScaler
import pickle

scaler = MinMaxScaler()

columns_to_scale = ["x_index", "y_index",
                     "elevation_sum", "elevation_stdev",
                     "slope_sum", "slope_stdev", "slope_min", "slope_max",
                     "dist_to_road_mean",
                     "dist_to_waterway_mean",
                     "dist_to_railway_mean",
                     "population_density_mean", "population_density_min", "population_density_max"]
X_train[columns_to_scale] = scaler.fit_transform(X_train[columns_to_scale])

# We transform the test set with the scaler fitted to the train set (no data leakage)
X_test[columns_to_scale] = scaler.transform(X_test[columns_to_scale])
X_test_2022[columns_to_scale] = scaler.transform(X_test_2022[columns_to_scale])
X_test_2023_24[columns_to_scale] = scaler.transform(X_test_2023_24[columns_to_scale])
pickle.dump(scaler, open("../data/models/scaler.pkl", "wb"))
```

## 9.3 Validación

Como hemos dicho antes, vamos a utilizar la validación cruzada de series temporales. Para ello, vamos a usar la función `TimeSeriesSplit` del paquete de `sklearn` con `n=5` particiones.

```
In [73]: from sklearn.model_selection import TimeSeriesSplit

n_splits = 5
tscv = TimeSeriesSplit(n_splits=n_splits)
```

# 10. Entrenamiento y testeо

Ahora que hemos preparado todos los datos y que hemos decidido el método de validación que usaremos, vamos a entrenar algunos clasificadores. Para eso, como nos interesa predecir el riesgo de incendio forestal, el riesgo lo representaremos como la probabilidad de pertenencia a la clase "fuegos". Es decir, tenemos que utilizar clasificadores probabilistas, por lo que basandonos en la literatura sobre

predicciones de riesgo de incendio forestal, usaremos los siguientes:

- Regresión Logística
- Random Forest
- XGBoost

Como hemos dicho anteriormente, este problema es un problema muy desbalanceado y, para nuestro caso, el desbalanceo es de 2 casos de "no-fuego" por cada caso de "fuego". Por ello no sería adecuado usar la métrica de "accuracy", ya que un clasificador que siempre predice "no-fuego" tendría un porcentaje de acierto alto, pero sería inútil. Por ello, vamos a utilizar las siguientes métricas de evaluación:

- **Sensibilidad:** mide la proporción de casos positivos reales que fueron correctamente identificados como positivos por el modelo:

$$\text{Sensitivity} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

- **Especificidad:** mide la proporción de negativos reales que fueron correctamente identificados como negativos por el modelo:

$$\text{Specificity} = \frac{\text{True Negatives (TN)}}{\text{True Negatives (TN)} + \text{False Positives (FP)}}$$

- **ROC-AUC (Área bajo la curva ROC):** representa el área bajo la curva ROC (*Receiver Operating Characteristic*), que es una gráfica que compara el *true positive rate* (sensibilidad) con el *false positive rate* (1 - especificidad) a través de diferentes umbrales de decisión. El valor de AUC se calcula como:

$$\text{AUC} = \int_0^1 \text{ROC}(\theta) d\theta$$

siendo  $\theta \in [0, 1]$  el umbral de decisión para clasificar como clase 1 o clase 0.

En el **ROC-AUC**:

- El eje X es el **false positive rate** (1 - especificidad).
- El eje Y es el **true positive rate** (sensibilidad).

y la curva se construye moviendo el umbral de decisión ( $\theta$ ).

Como queremos asegurar que nuestro modelo no solo obtenga buenos resultados para predecir los casos de fuego (sensibilidad), sino que también sea capaz de no predecir riesgo de incendio cuando no lo hay (especificidad), utilizaremos como métrica de evaluación el **ROC-AUC**, que considera ambas métricas conjuntamente. Lo usaremos para seleccionar el modelo a la hora de hacer la validación y para testar los modelos en el conjunto de test.

A la hora de hacer la validación, para cada modelo seleccionaremos algunos valores posibles para los hiperparámetros y haremos una búsqueda aleatoria en la cross validación ("Randomized Search"). Además, una vez obtenidos los mejores hiperparámetros, vamos a reentrenar el algoritmo con todo el conjunto de entrenamiento con esos hiperparámetros. Esto lo haremos utilizando la función `RandomizedSearchCV` del paquete `sklearn`.

Para decidir qué hiperparámetros seleccionar, hemos mirado los que toma por defecto el paquete `sklearn` y hemos añadido esos junto a algunos valores cercanos.

Nota: Como los entrenamientos tardan mucho, vamos a guardar los mejores modelos y los mejores hiperparámetros mediante el paquete `pickle`.

## 10.1 Regresión Logística

### Entrenamiento

Vamos a crear una función genérica que entrene y guarde los algoritmos con la metodología que hemos explicado hasta ahora. Para guardar los algoritmos utilizaremos el paquete `pickle` que permite guardar objetos de Python en forma binaria, para luego cargarlos en otro momento. Esto es muy útil ya que podemos no entrenar el algoritmo cada vez que necesitemos utilizarlo.

Además, vamos a entrenar los algoritmos en paralelo para acelerar el proceso de entrenamiento. Esto lo haremos seleccionando la opción `n_jobs = -1` en la función `RandomizedSearchCV`, que paraleliza el proceso de validación.

```
In [74]: from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import RandomizedSearchCV
import pickle

def train_and_save(model, X, y, model_name, param_distributions, iterations, save_model = False):
    random_search = RandomizedSearchCV(
        estimator=model,
        param_distributions=param_distributions,
        n_iter=iterations,      # number of random combinations
        scoring="roc_auc",      # metric to optimize
        cv=tscv,
```

```

        random_state=42,
        verbose=1,
        n_jobs=-1
    )

random_search.fit(X, y)

# Best hyperparameters and best cross-validation score
print("-----")
print("Best parameters:", random_search.best_params_)
print("Best cross-validation score (ROC AUC):", random_search.best_score_)

# Save the best model and hyperparameters
best_model_logreg = random_search.best_estimator_
best_hyperparams_logreg = random_search.best_params_

if save_model:
    pickle.dump(best_model_logreg, open("../data/models/model_" + model_name + ".pkl", "wb"))
    pickle.dump(best_hyperparams_logreg, open("../data/models/hyperparameters_" + model_name + ".pkl", "wb"))
    print(f"Model {model_name} saved successfully")

```

Ahora que tenemos la función de entrenamiento, vamos a entrenar el modelo de regresión logística:

```
In [75]: param_distributions_logreg = {
    'C': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear', 'saga'],
    'max_iter': [75, 50, 100, 200, 300, 500]
}

logreg_model = LogisticRegression()

train_logreg = False

if train_logreg:
    train_and_save(model=logreg_model,
                   X=X_train,
                   y=y_train,
                   model_name="LOGREG",
                   param_distributions=param_distributions_logreg,
                   iterations=100,
                   save_model=True)
```

## Testeo

Crearemos una función que muestre información de los resultados del modelo, introduciendo como "input" las probabilidades predichas por el modelo. Así, podemos usar la función para testar en diferentes modelos sin repetir código.

La función obtendrá el umbral de decisión que maximiza el índice de Youden, es decir, el que obtiene mayor sensibilidad y especificidad conjuntamente. Además, la función también dibujará la curva ROC y dará información de la sensibilidad y especificidad para diferentes umbrales de decisión ( $\theta = 0.5$ ).

Más adelante analizaremos los resultados obtenidos en estos entrenamientos.

```
In [76]: import pickle
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_auc_score, roc_curve, confusion_matrix

def display_test_results(probabilities, y_test, print_information=True, print_50_threshold_info=True):
    # ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(y_test, probabilities)
    roc_auc = roc_auc_score(y_test, probabilities)

    # optimal threshold based on Youden's J statistic = TPR - FPR
    youden_j = tpr - fpr
    best_idx = np.argmax(youden_j)
    best_threshold = thresholds[best_idx]

    # Make binary predictions using the optimal threshold
    optimal_test_predictions = (probabilities >= best_threshold).astype(int)

    # Calculate Sensitivity, specificity at that threshold
    tn, fp, fn, tp = confusion_matrix(y_test, optimal_test_predictions).ravel()
    sensitivity = tp / (tp + fn) # also known as Recall or TPR
    specificity = tn / (tn + fp) # also known as TNR

    if print_information:
        print(f"ROC AUC: {roc_auc:.6f}")
        print(f"Optimal Threshold (Youden's J): {best_threshold:.6f}")
```

```

print(f"Sensitivity (TPR) at Optimal Threshold: {sensitivity:.6f}")
print(f"Specificity (TNR) at Optimal Threshold: {specificity:.6f}")

if print_50_threshold_info:
    # -- ADD THRESHOLD = 0.5 POINT --
    # Find the index of the threshold closest to 0.5
    idx_05 = np.argmin(np.abs(thresholds - 0.5))
    fpr_05 = fpr[idx_05]
    tpr_05 = tpr[idx_05]

    # Make binary predictions using threshold = 0.5
    predictions_05 = (probabilities >= 0.5).astype(int)
    tn_05, fp_05, fn_05, tp_05 = confusion_matrix(y_test, predictions_05).ravel()
    sens_05 = tp_05 / (tp_05 + fn_05)    # TPR at threshold=0.5
    spec_05 = tn_05 / (tn_05 + fp_05)    # TNR at threshold=0.5

    print(f"Sensitivity (TPR) at 0.5 Threshold: {sens_05:.4f}")
    print(f"Specificity (TNR) at 0.5 Threshold: {spec_05:.4f}")

    # Plot the ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='blue', label=f'ROC curve (AUC = {roc_auc:.3f})')
    plt.plot([0, 1], [0, 1], color='red', linestyle='--', label='Random Guess')

    # Mark the best threshold point on the ROC curve
    plt.plot(fpr[best_idx], tpr[best_idx], marker='o', markersize=8, color='black')
    plt.annotate(
        (
            f"Threshold = {best_threshold:.2f}\n"
            f"Sens. = {sensitivity:.2f}\n"
            f"Spec. = {specificity:.2f}"
        ),
        xy=(fpr[best_idx], tpr[best_idx]),
        xytext=(fpr[best_idx] + 0.05, tpr[best_idx] - 0.15),
        arrowprops=dict(arrowstyle="->", connectionstyle="arc3")
    )

if print_50_threshold_info:
    plt.plot(fpr_05, tpr_05, marker='s', markersize=8, color='green')
    plt.annotate(
        (
            f"Threshold = 0.50\n"
            f"Sens. = {sens_05:.2f}\n"
            f"Spec. = {spec_05:.2f}"
        ),
        xy=(fpr_05, tpr_05),
        xytext=(fpr_05 + 0.05, tpr_05 - 0.05),
        arrowprops=dict(arrowstyle="->", connectionstyle="arc3")
    )

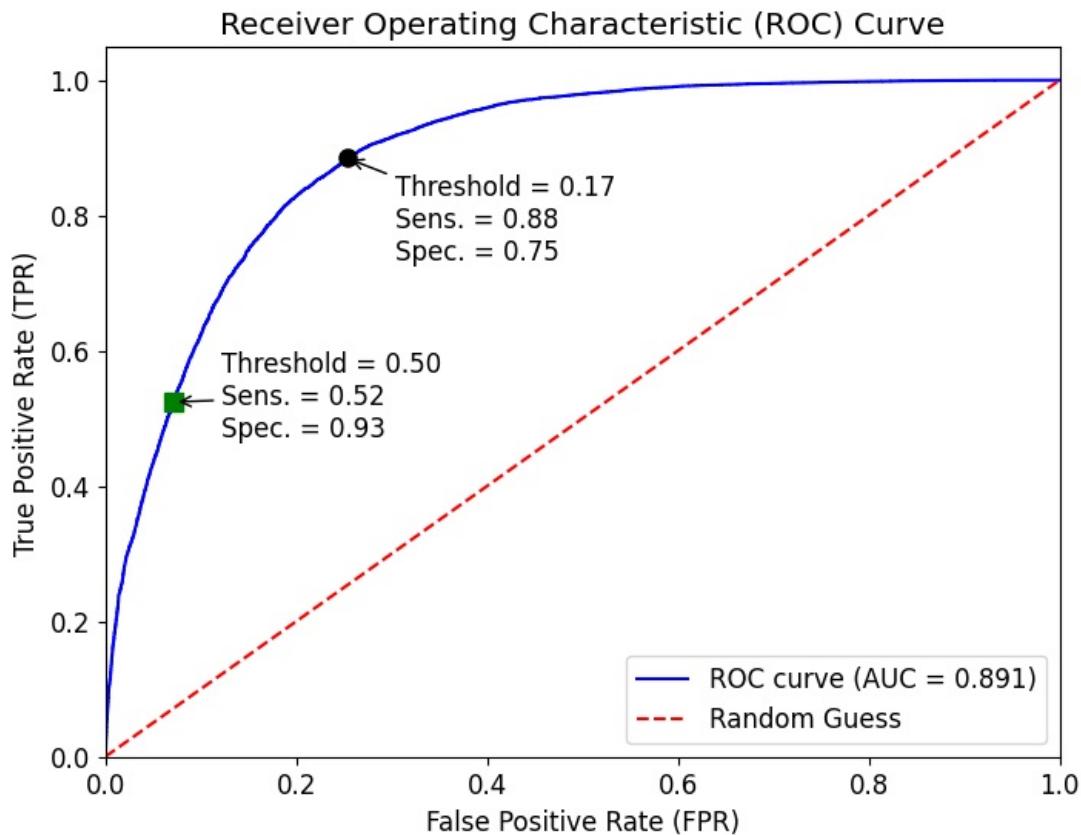
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate (FPR)')
    plt.ylabel('True Positive Rate (TPR)')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc='lower right')
    plt.show()
else:
    return roc_auc

```

```
In [77]: logreg_model = pickle.load(open("../data/models/model_LOGREG.pkl", "rb"))
test_probabilities = logreg_model.predict_proba(X_test)[:, 1]

display_test_results(test_probabilities, y_test)
```

ROC AUC: 0.891253  
Optimal Threshold (Youden's J): 0.171724  
Sensitivity (TPR) at Optimal Threshold: 0.884767  
Specificity (TNR) at Optimal Threshold: 0.746928  
Sensitivity (TPR) at 0.5 Threshold: 0.5243  
Specificity (TNR) at 0.5 Threshold: 0.9279



## 10.2 Random Forest:

Entrenamiento

```
In [78]: from sklearn.ensemble import RandomForestClassifier

param_distributions = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 7, 10],
    'min_samples_leaf': [1, 2, 3, 4, 5, 6],
    'bootstrap': [True, False]
}

rf_model = RandomForestClassifier()

train_rf = False

if train_rf:
    train_and_save(model=rf_model,
                  X=X_train,
                  y=y_train,
                  model_name="RF",
                  param_distributions=param_distributions,
                  iterations=50,
                  save_model=True)
```

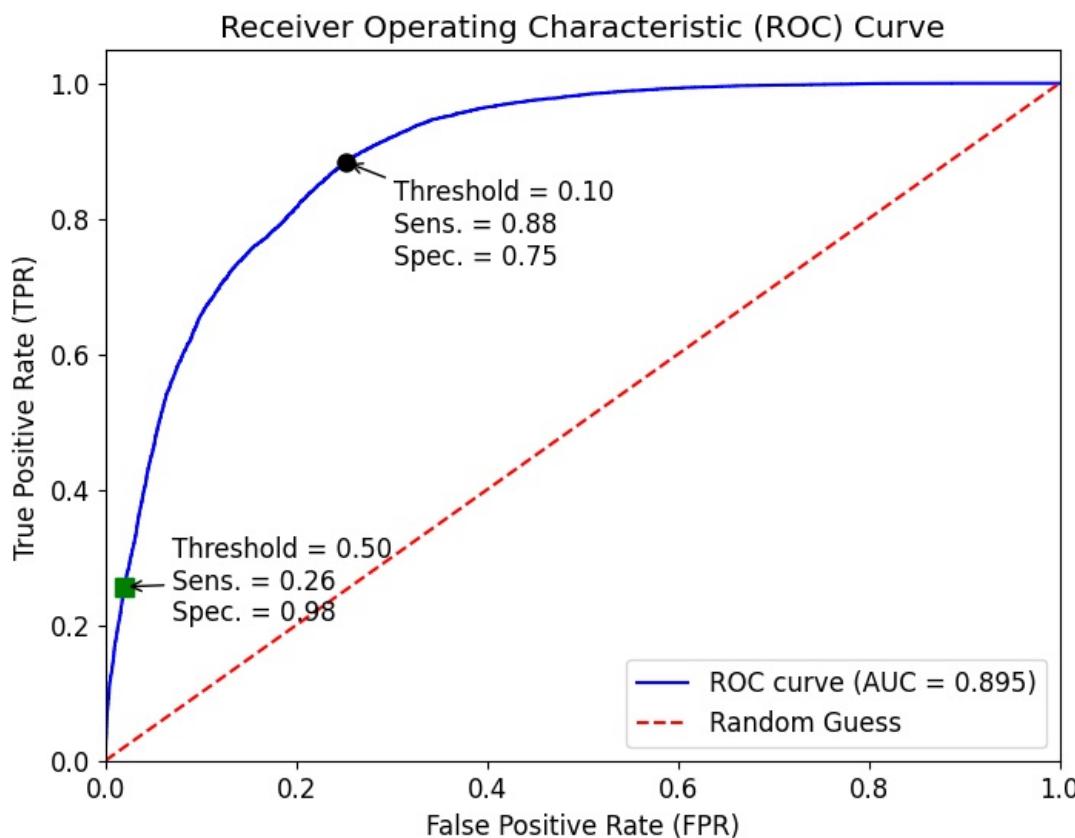
Testeo:

```
In [79]: import pickle

rf_model = pickle.load(open("../data/models/model_RF.pkl", "rb"))
test_probabilities = rf_model.predict_proba(X_test)[:, 1]
```

```
display_test_results(test_probabilities, y_test)
```

ROC AUC: 0.894886  
Optimal Threshold (Youden's J): 0.096295  
Sensitivity (TPR) at Optimal Threshold: 0.883691  
Specificity (TNR) at Optimal Threshold: 0.748009  
Sensitivity (TPR) at 0.5 Threshold: 0.2568  
Specificity (TNR) at 0.5 Threshold: 0.9797



### 10.3 XGBoost model:

Entrenamiento

```
In [80]: from xgboost import XGBClassifier

param_distributions = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [3, 5, 6, 10, 15],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0],
    'gamma': [0, 1, 5, 10],
    'min_child_weight': [1, 3, 5, 7],
    'reg_alpha': [0, 0.01, 0.1, 1],
    'reg_lambda': [1, 5, 10]
}

xgb_model = XGBClassifier(eval_metric="auc")

train_xgb = False

if train_xgb:
    train_and_save(model=xgb_model,
                  X=X_train,
                  y=y_train,
                  model_name="XGB",
                  param_distributions=param_distributions,
                  iterations=50,
                  save_model=True)
```

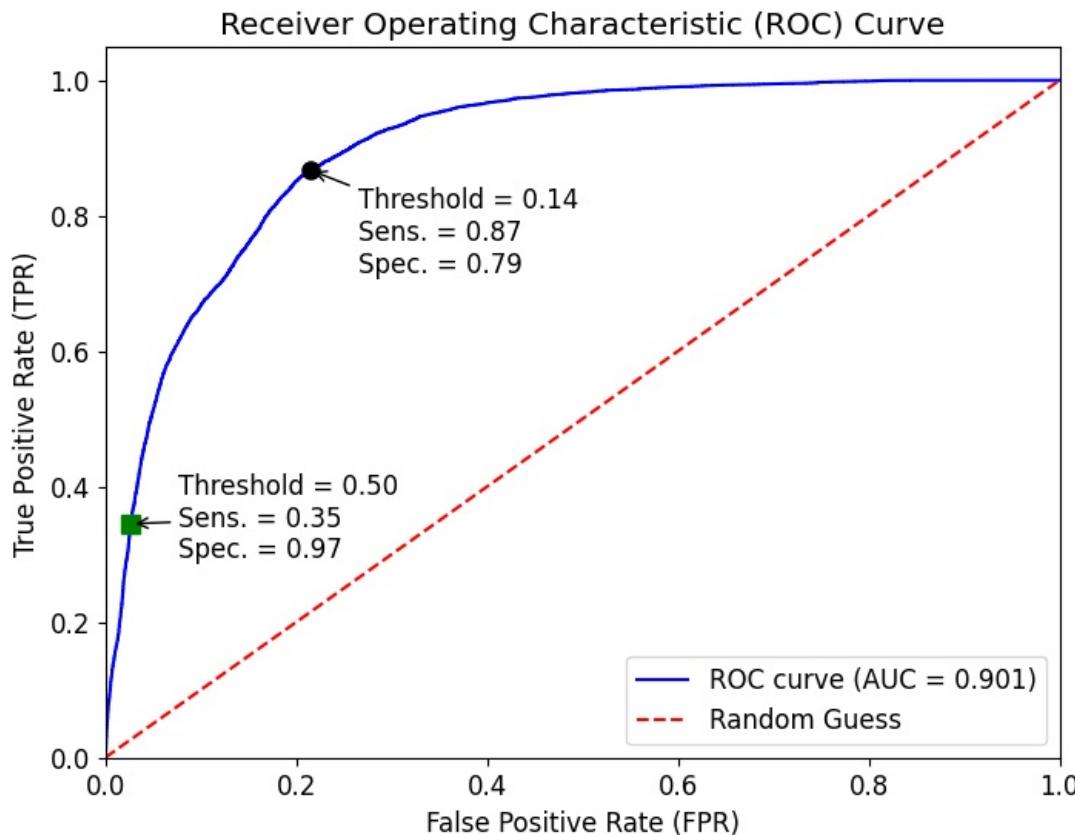
Testeo:

```
In [81]: import pickle

xgb_model = pickle.load(open("../data/models/model_XGB.pkl", "rb"))
test_probabilities = xgb_model.predict_proba(X_test)[:, 1]

display_test_results(test_probabilities, y_test)
```

ROC AUC: 0.900912  
 Optimal Threshold (Youden's J): 0.141179  
 Sensitivity (TPR) at Optimal Threshold: 0.867591  
 Specificity (TNR) at Optimal Threshold: 0.785624  
 Sensitivity (TPR) at 0.5 Threshold: 0.3452  
 Specificity (TNR) at 0.5 Threshold: 0.9730



## 11. Desbalanceo de los datos: SMOTE

Nuestro problema es un problema de clasificación desbalanceado, por lo que vamos a probar a utilizar el algoritmo `SMOTE` ("Synthetic Minority Over-sampling Technique") para sobremuestrear la clase minoritaria en el conjunto de entrenamiento y volver a entrenar los tres algoritmos.

El algoritmo `SMOTE` va a generar instancias de "fuego" sintéticas, las cuales no tendrán significado. Es decir, esas instancias no tendrán una fecha inicial en la que ocurrieron, por lo que no vamos a poder hacer una partición de series temporales correcta y hacer la validación.

Por ello, vamos a intercalar las instancias originales con las nuevas instancias, manteniendo el orden de las instancias originales. Es decir, en el nuevo conjunto de datos, las instancias originales mantendrán el orden temporal, mientras que las instancias sintéticas no.

```
In [82]: from imblearn.over_sampling import SMOTE
from collections import Counter

h = Counter(y_train)
# Ver distribución de clases antes de SMOTE
print("Distribución antes de SMOTE:", h)

# Aplicar SMOTE
smote = SMOTE(random_state=42, sampling_strategy='minority')
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

# Tomar solamente las instancias generadas
generated_amount = h[0] - h[1]
X_fires_synthetic, y_fires_synthetic = X_resampled[-generated_amount:], y_resampled[-generated_amount:]

# Ver distribución de clases después de SMOTE
print("Distribución después de SMOTE", Counter(y_resampled))
```

Distribución antes de SMOTE: Counter({0: 129077, 1: 39448})  
 Distribución después de SMOTE: Counter({0: 129077, 1: 129077})

Intercalemos las instancias originales con las instancias sintéticas:

```
In [83]: import pandas as pd
from itertools import zip_longest
import warnings
```

```

warnings.simplefilter(action='ignore', category=pd.errors.SettingWithCopyWarning)
X_train["is_fire"] = y_train
X_fires_synthetic["is_fire"] = y_fires_synthetic
warnings.simplefilter(action='default', category=pd.errors.SettingWithCopyWarning)

df1 = X_train
df2 = X_fires_synthetic

# Intercalar manteniendo el orden
def intercalar_datasets(df1, df2):
    filas1 = df1.to_dict(orient='records')
    filas2 = df2.to_dict(orient='records')

    # zip_longest para casos de diferentes longitudes
    filas_intercaladas = [fila for pair in zip_longest(filas1, filas2) for fila in pair if fila]

    return pd.DataFrame(filas_intercaladas)

df_intercalado = intercalar_datasets(df1, df2)

X_train_new = df_intercalado.drop(columns=['is_fire'])
y_train_new = df_intercalado["is_fire"]

X_train.drop(columns=["is_fire"], inplace=True)

print("Distribución después de SMOTE tras intercalar los datos:", Counter(y_train_new))

```

Distribución después de SMOTE tras intercalar los datos: Counter({0: 129077, 1: 129077})

## Entrenamientos

Ahora que hemos obtenido el nuevo conjunto de entrenamiento balanceado, vamos a utilizar las funciones que hemos creado en la sección anterior para entrenar y testar los algoritmos. Como hemos dicho anteriormente, más adelante analizaremos los resultados obtenidos.

### Regresión Logística con SMOTE

```
In [84]: param_distributions = {
    'C': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear', 'saga'],
    'max_iter': [75, 50, 100, 200, 300, 500]
}
```

```

logreg_model = LogisticRegression()

train_logreg = False

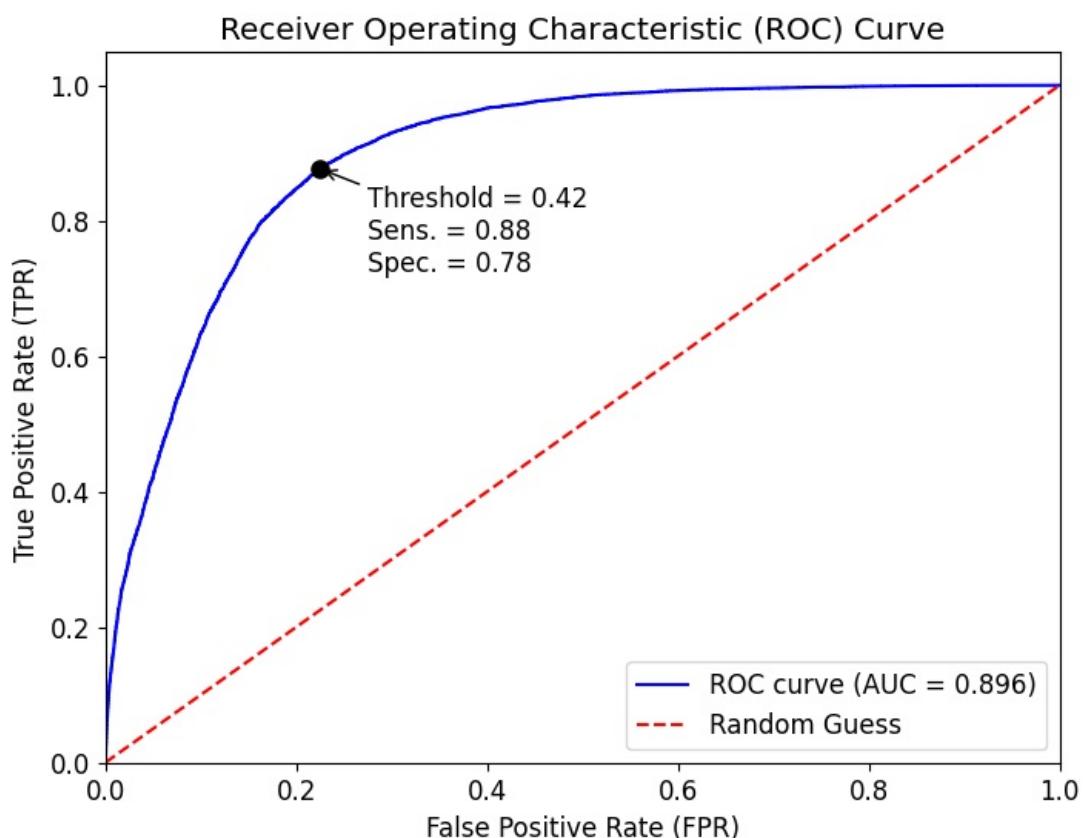
if train_logreg:
    train_and_save(model=logreg_model,
                   X=X_train_new,
                   y=y_train_new,
                   model_name="LOGREG_SMOTE",
                   param_distributions=param_distributions,
                   iterations=100,
                   save_model=True)

```

```
In [85]: logreg_model = pickle.load(open("../data/models/model_LOGREG_SMOTE.pkl", "rb"))
test_probabilities = logreg_model.predict_proba(X_test)[:, 1]

display_test_results(test_probabilities, y_test, print_50_threshold_info=False)
```

ROC AUC: 0.896037  
Optimal Threshold (Youden's J): 0.424491  
Sensitivity (TPR) at Optimal Threshold: 0.876718  
Specificity (TNR) at Optimal Threshold: 0.775212



Random Forest con SMOTE

```
In [86]: param_distributions = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 7, 10],
    'min_samples_leaf': [1, 2, 3, 4, 5, 6],
    'bootstrap': [True, False]
}

rf_model = RandomForestClassifier()

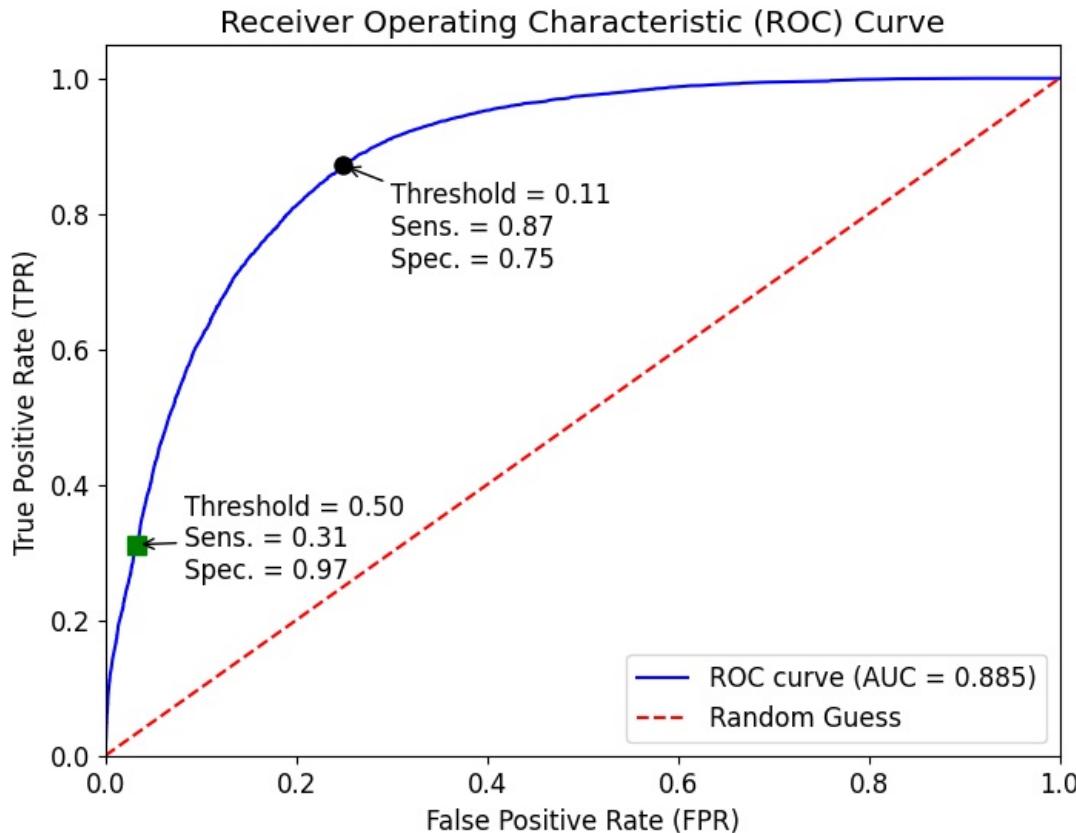
train_rf = False

if train_rf:
    train_and_save(model=rf_model,
                  X=X_train_new,
                  y=y_train_new,
                  model_name="RF_SMOTE",
                  param_distributions=param_distributions,
                  iterations=50,
                  save_model=True)
```

```
In [87]: rf_model = pickle.load(open("../data/models/model_RF_SMOTE.pkl", "rb"))
test_probabilities = rf_model.predict_proba(X_test)[:, 1]

display_test_results(test_probabilities, y_test)
```

ROC AUC: 0.885073  
 Optimal Threshold (Youden's J): 0.114167  
 Sensitivity (TPR) at Optimal Threshold: 0.871308  
 Specificity (TNR) at Optimal Threshold: 0.750640  
 Sensitivity (TPR) at 0.5 Threshold: 0.3117  
 Specificity (TNR) at 0.5 Threshold: 0.9666



### XGBoost con SMOTE

```
In [88]: param_distributions = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [3, 5, 6, 10, 15],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0],
    'gamma': [0, 1, 5, 10],
    'min_child_weight': [1, 3, 5, 7],
    'reg_alpha': [0, 0.01, 0.1, 1],
    'reg_lambda': [1, 5, 10]
}
```

```
xgb_model = XGBClassifier(eval_metric="auc")
```

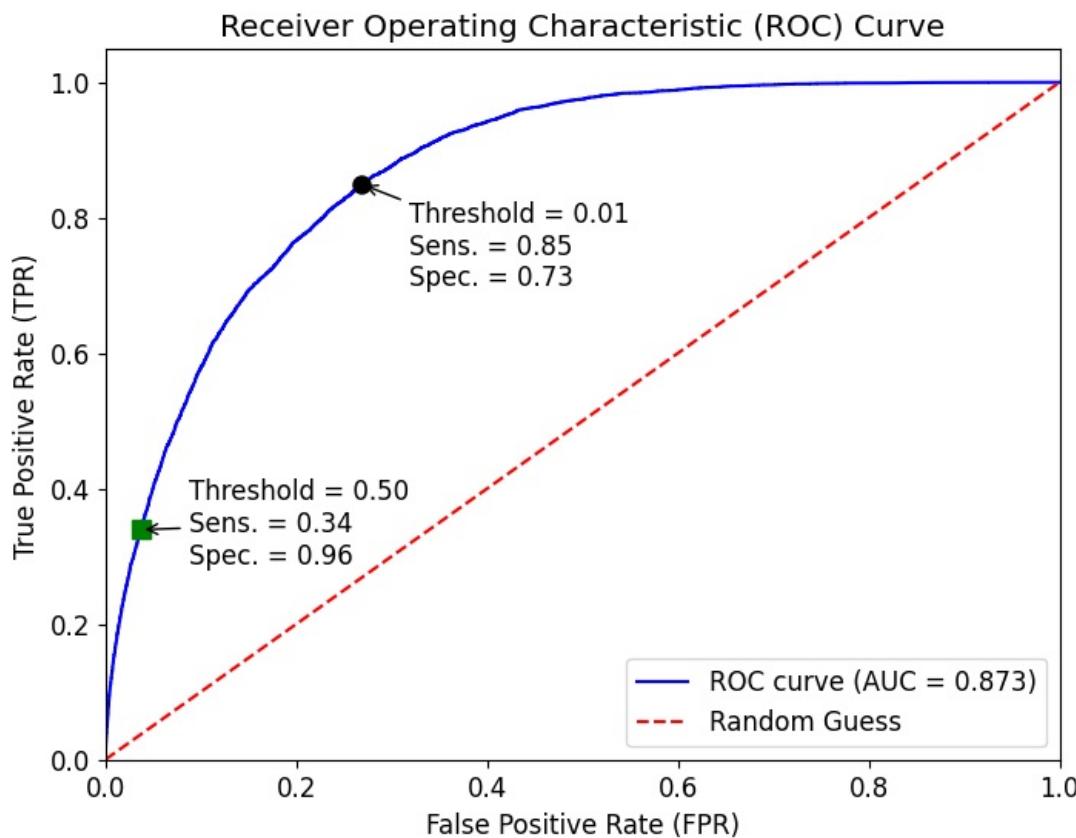
```
train_xgb = False
```

```
if train_xgb:
    train_and_save(model=xgb_model,
                  X=X_train_new,
                  y=y_train_new,
                  model_name="XGB_SMOTE",
                  param_distributions=param_distributions,
                  iterations=50,
                  save_model=True)
```

```
In [89]: xgb_model = pickle.load(open("../data/models/model_XGB_SMOTE.pkl", "rb"))
test_probabilities = xgb_model.predict_proba(X_test)[:, 1]

display_test_results(test_probabilities, y_test)
```

ROC AUC: 0.873264  
 Optimal Threshold (Youden's J): 0.008606  
 Sensitivity (TPR) at Optimal Threshold: 0.849877  
 Specificity (TNR) at Optimal Threshold: 0.732156  
 Sensitivity (TPR) at 0.5 Threshold: 0.3394  
 Specificity (TNR) at 0.5 Threshold: 0.9629



## 12. Comparativa estadística de los modelos

Tras entrenar y validar todos los modelos, tanto los que han sido entrenados con un conjunto de datos desbalanceado como los que lo han sido con instancias "artificiales", vamos a hacer una comparativa estadística para decidir si hay diferencias significativas entre ellos, y en tal caso, cuál sería el que tiene mejor rendimiento.

Para ello, vamos a testar los algoritmos en tres conjuntos de datos: los datos de 2022, los de 2023-2024 y los dos anteriores juntos, 2022-2024. Esto lo haremos porque los datos del año 2022 fueron datos anomalos, por lo que puede que no tengan la misma distribución de probabilidad que los datos de entrenamiento.

La métrica de evaluación que utilizaremos será el ROC-AUC, y haremos el test de Friedman para verificar si hay diferencias significativas entre los modelos. Lo primero que haremos será calcular qué valor de ROC-AUC obtienen los modelos en los tres conjuntos de test.

```
In [90]: model_names = ["LOGREG_SMOTE", "LOGREG", "RF_SMOTE", "RF", "XGB_SMOTE", "XGB"]
dataset_names = ["2022", "2023-2024", "2022-2024"]

auc_scores = np.zeros((3,6))

for i, model_name in enumerate(model_names):
    model = pickle.load(open("../data/models/model_" + model_name + ".pkl", "rb"))
    test_probabilities = model.predict_proba(X_test_2022)[:, 1]
    auc_scores[0,i] = display_test_results(test_probabilities, y_test_2022, print_information=False)

    test_probabilities = model.predict_proba(X_test_2023_24)[:, 1]
    auc_scores[1,i] = display_test_results(test_probabilities, y_test_2023_24, print_information=False)

    test_probabilities = model.predict_proba(X_test)[:, 1]
    auc_scores[2,i] = display_test_results(test_probabilities, y_test, print_information=False)

df_scores = pd.DataFrame(auc_scores, columns=model_names, index=dataset_names)
df_scores
```

	LOGREG_SMOTE	LOGREG	RF_SMOTE	RF	XGB_SMOTE	XGB
2022	0.904070	0.896492	0.877805	0.883346	0.863046	0.889781
2023-2024	0.880719	0.881518	0.902567	0.921211	0.897447	0.926691
2022-2024	0.896037	0.891253	0.885073	0.894886	0.873264	0.900912

```
In [91]: pd.DataFrame([np.mean(auc_scores, axis=0)], columns=model_names, index=[ "Mean scores" ])
```

	LOGREG_SMOTE	LOGREG	RF_SMOTE	RF	XGB_SMOTE	XGB
Mean scores	0.893609	0.889754	0.888481	0.899815	0.877919	0.905795

Vemos como para el conjunto de datos de test de 2022 el modelo que obtiene el mejor resultado es la regresión logística entrenada en datos balanceados mediante el algoritmo SMOTE, mientras que en el conjunto de test de 2023-2024 y 2022-2024 se obtienen los mejores resultados en el modelo XGBoost entrenado en datos desbalanceados.

Si hacemos una media de las métricas obtenidas para cada modelo, obtenemos que el modelo con la mejor media del valor de ROC-AUC es el modelo XGBoost entrenado en datos desbalanceados. Para comprobar si esta diferencia es estadísticamente significativa, vamos a ejecutar el test de Friedman, el cual tiene como hipótesis nula que no hay diferencias significativas entre los modelos.

```
In [92]: from scipy.stats import friedmanchisquare
stat, p_value = friedmanchisquare(*df_scores.T.values) # models as rows and datasets as columns
print(f"Estadístico de Friedman: {stat}")
print(f"Valor p: {p_value}")
```

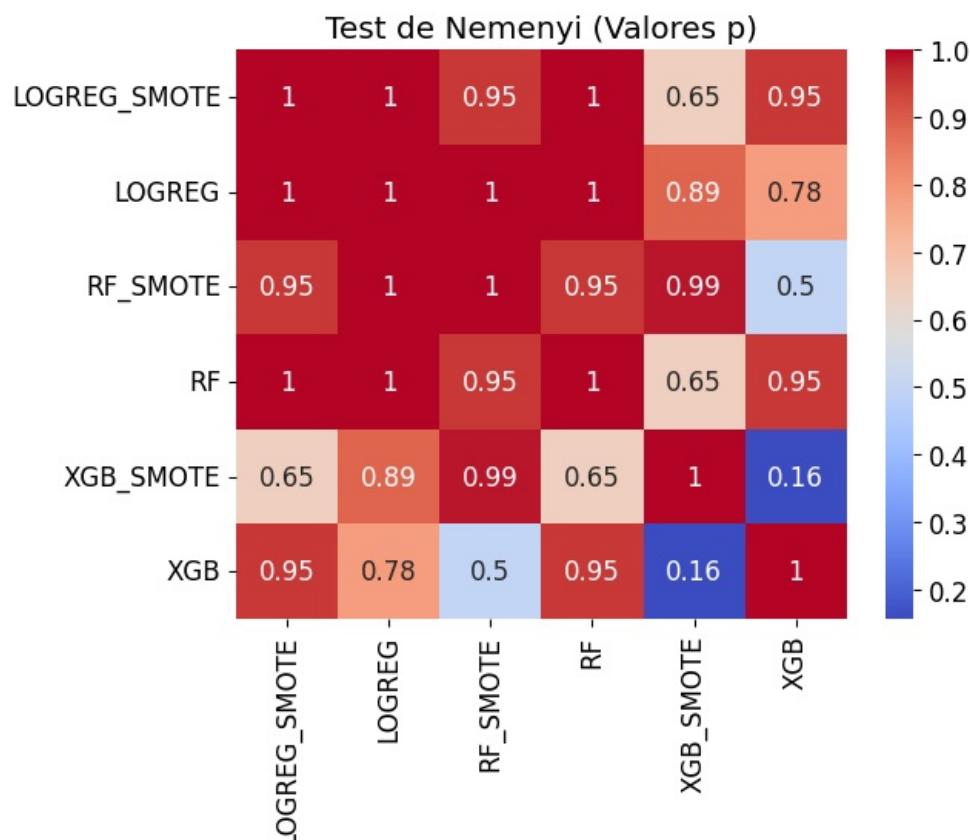
Estadístico de Friedman: 6.80952380952381  
 Valor p: 0.23519605163635354

Vemos que no podemos rechazar la hipótesis nula, por lo que no hay diferencias significativas estadísticamente hablando en cuanto al rendimiento de los modelos. Sin embargo, como el `p-valor` es bastante pequeño, vamos a ejecutar igualmente un "post-hoc bivariate test", mediante el cual comprobaremos entre qué algoritmos hay mayores diferencias.

```
In [93]: from scikit_posthocs import posthoc_nemenyi_friedman
import seaborn as sns

# Post-hoc Nemenyi test
posthoc_results = posthoc_nemenyi_friedman(auc_scores)

sns.heatmap(posthoc_results, annot=True, cmap="coolwarm", xticklabels=model_names, yticklabels=model_names)
plt.title("Test de Nemenyi (Valores p)")
plt.show()
```



En este test vemos como entre los primeros cuatro modelos realmente no hay diferencias significativas entre ellos, ya que el `p-valor`

del test es muy elevado. En los últimos dos modelos se pueden observar `p-valores` más pequeños, aunque siguen sin haber diferencias significativas entre ellos.

A partir de este punto, aunque no haya habido diferencias significativas entre los modelos, utilizaremos el que mejores resultados medios ha obtenido en el ROC-AUC: el `XGBoost` entrenado en datos desbalanceados.

## 13. Feature Importance

Ahora que hemos obtenido un buen predictor, veremos qué variables afectan más en el rendimiento del modelo a la hora de hacer dichas predicciones. Para ello, calcularemos las `feature importances` permutando aleatoriamente una variable a la vez y analizando cuánto decrece la métrica del ROC-AUC prediciendo con el conjunto de datos con la variable alterada. Para ello haremos uso de la función `permutation_importance` del paquete `sklearn`.

```
In [94]: from sklearn.inspection import permutation_importance
from sklearn.metrics import roc_auc_score
import pickle
import matplotlib.pyplot as plt

model = pickle.load(open("../data/models/model_XGB.pkl", "rb"))
test_probabilities = model.predict_proba(X_test)[:, 1]

baseline_roc = roc_auc_score(y_test, test_probabilities)

result = permutation_importance(model, X_test, y_test, n_repeats=20, random_state=42, scoring="roc_auc")

feature_names = [X_test.columns[i] for i in result.importances_mean.argsort()[:-1]]
importances = result.importances_mean[result.importances_mean.argsort()[:-1]]

# Filtered feature names and importances (excluding "month_")
filtered_features = [(name, imp) for name, imp in zip(feature_names, importances) if not name.startswith("month")]
filtered_names, filtered_importances = zip(*filtered_features) if filtered_features else ([], [])

show_n_features = 20

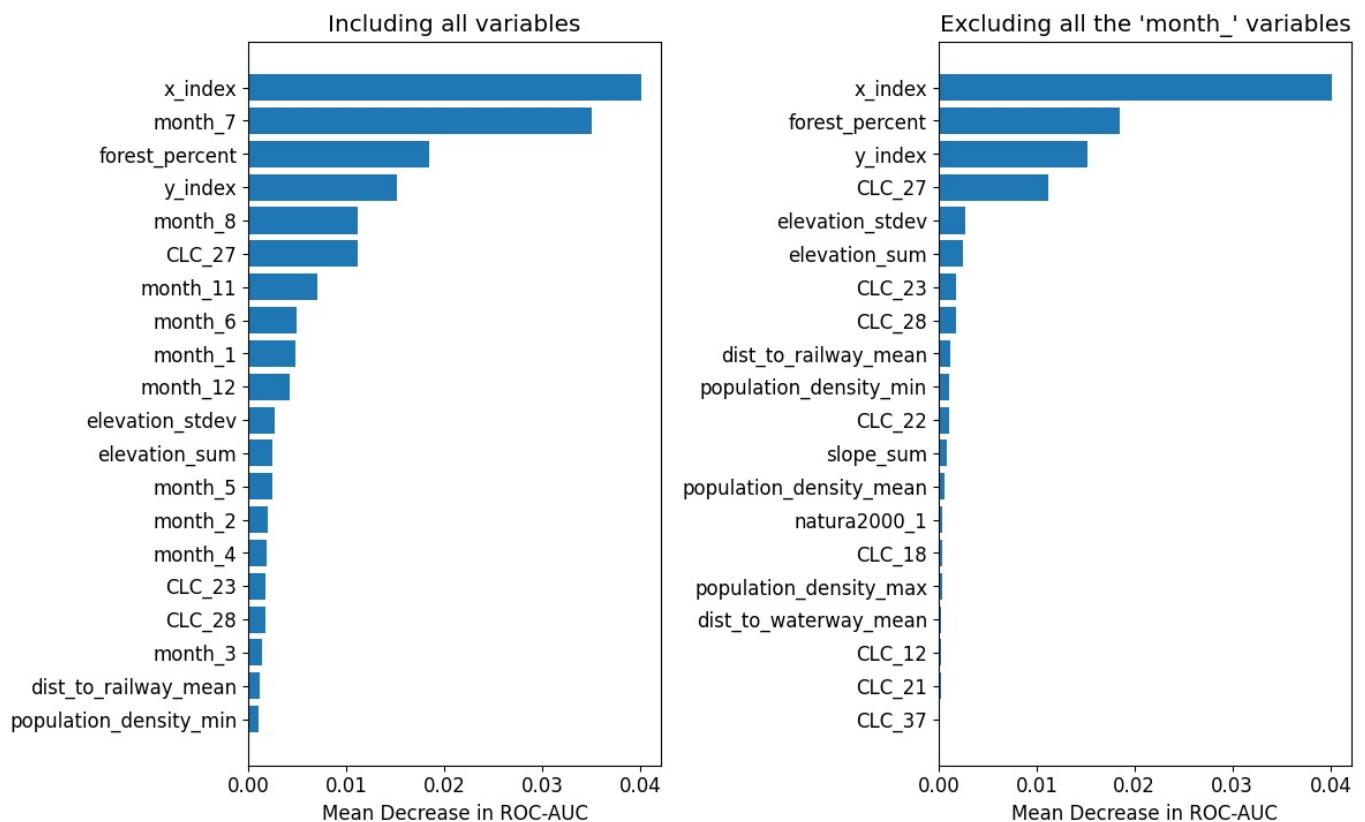
fig, ax = plt.subplots(1, 2, figsize=(12, 8), sharex=False)
fig.suptitle(f"Feature Importances Analysis:\nfirst {show_n_features} most important features out of {len(X_test)}")

ax[0].barh(feature_names[:show_n_features][:-1], importances[:show_n_features][:-1], align="center")
ax[0].set_xlabel("Mean Decrease in ROC-AUC")
ax[0].set_title("Including all variables")

# Excluding columns starting with "month_"
ax[1].barh(filtered_names[:show_n_features][:-1], filtered_importances[:show_n_features][:-1], align="center")
ax[1].set_xlabel("Mean Decrease in ROC-AUC")
ax[1].set_title("Excluding all the 'month_' variables")

plt.tight_layout()
plt.show()
```

## Feature Importances Analysis: first 20 most important features out of 84



Por un lado vemos como las variables `x_index` e `y_index` son muy importantes para hacer las predicciones. Esto puede deberse a que el modelo aprende en qué zonas geográficas ha habido más incendios y en dichas zonas devuelve mayor probabilidad de pertenencia a la clase de `fuego`.

Por otro lado, vemos también que la variable `month` (en general) es muy importante, ya que de las 20 variables más importantes obtenidas en esta sección, 9 vienen de hacer `one-hot-encoding` de la variable `month`. También se observa que los meses con mayor importancia son junio julio y agosto por un lado, y noviembre, diciembre y enero por otro. Esto puede indicar que juntar esos meses en variables como `es_verano` y `es_inviero` podría ser beneficioso al hacer el TFM.

Otra observación interesante es que la variable `forest_percent` que hemos creado, es la tercera variable más importante de acuerdo a este análisis. Por otro lado, una clase importante del conjunto de datos "Corine Land Cover" es la clase 27, que representa el tipo de terreno llamado "**Moors and heathland**", el cual se define como "vegetación baja, dominada por arbustos, matorrales...". Este resultado concuerda con el análisis que hemos hecho en la sección 4 al visualizar las distribuciones de las variables en base a la clase `fuego` o `no-fuego`.

Por último, vemos como la elevación también juega un papel importante a la hora de hacer las predicciones, mientras que las variables relacionadas con la actividad humana no tienen tanto impacto.

## 14. Machine Learning Interpretable: valores SHAP

Otra herramienta que usaremos para interpretar las decisiones del modelo serán los "SHAP values". Basados en la teoría de juegos, los valores SHAP asignan a cada variable de una instancia predicha una contribución específica en la predicción del modelo, proporcionando explicaciones cuantitativas del impacto de cada variable en una predicción dada.

Para clasificación binaria, valores de SHAP altos en una variable significa que esa variable hace que el modelo tienda a predecir la clase 1 (o clase `fuego`), mientras que valores de SHAP bajos hacen que prediga la clase 0 (`no-fuego`).

```
In [95]: import shap
import pickle

train_shap = False # Takes a long time to calculate the shap values

if train_shap:
    model = pickle.load(open("../data/models/model_XGB.pkl", "rb"))
    explainer = shap.Explainer(model, X_train.astype({f'month_{i}': 'int64' for i in range(1,13)})) # The input
    pickle.dump(explainer, open("../data/models/shap_explainer_XGB.pkl", "wb"))

    shap_values = explainer(X_test.astype({f'month_{i}': 'int64' for i in range(1,13)}))
    pickle.dump(shap_values, open("../data/models/shap_values_XGB.pkl", "wb"))
```

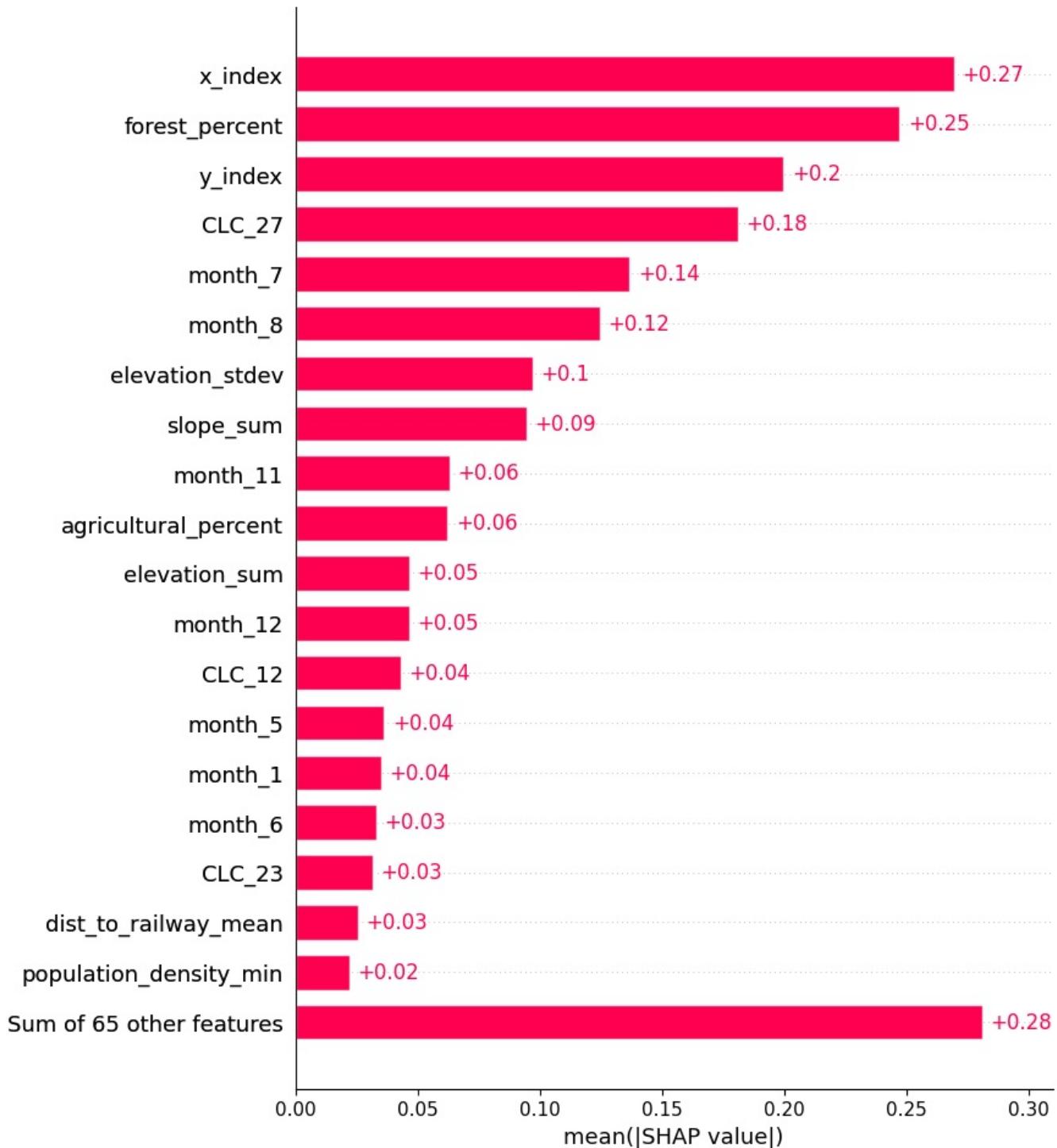
Una vez calculados los valores SHAP de cada instancia, interpretaremos los resultados mediante dos gráficos. Por un lado graficaremos el valor absoluto de SHAP de cada variable, lo cual puede traducirse en la importancia que tiene cada variable en el modelo para hacer predicciones. Por otro lado, dibujaremos cada valor de SHAP del conjunto de datos de test a la vez que los coloreamos en base a si la instancia tiene un valor alto en la variable o no. Esto nos permitirá obtener un entendimiento más profundo del comportamiento del modelo.

Feature Importance mediante SHAP values:

```
In [96]: import pickle
import shap
import matplotlib.pyplot as plt

shap_values = pickle.load(open("../data/models/shap_values_XGB.pkl", "rb"))

shap.plots.bar(shap_values,
               max_display=20)
```

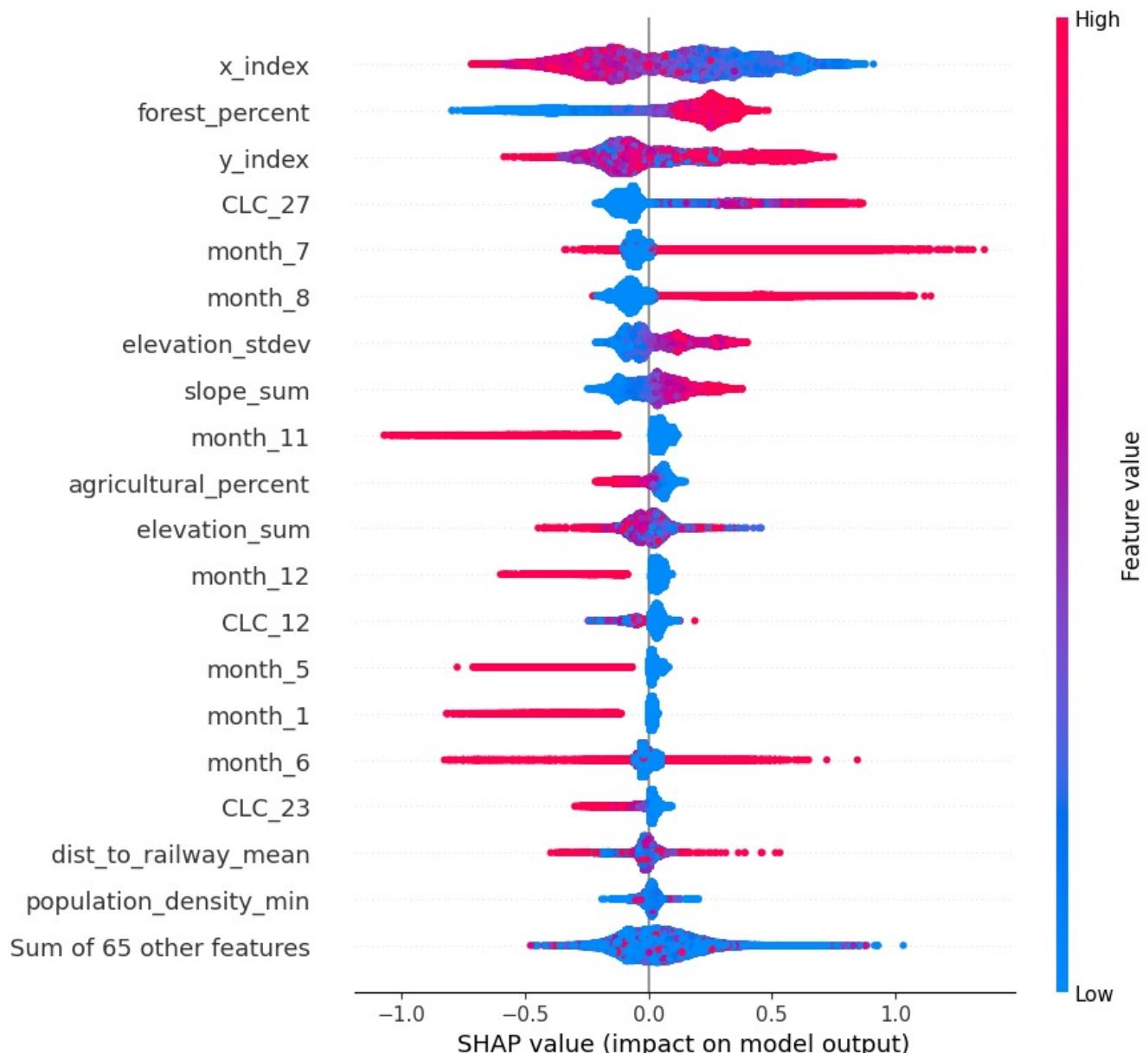


Vemos como las variables más importantes obtenidas mediante este análisis concuerdan en gran medida con las variables más importantes obtenidas en la sección anterior. Sin embargo, hay algunas diferencias; por ejemplo, una variable interesante que aparece en este análisis y que no aparece en el análisis anterior es la variable `agricultural_percent`.

En cuanto a las otras variables, las conclusiones son prácticamente las mismas que en el análisis de la sección anterior.

## Cómo afecta cada variable en la predicción del modelo según los valores SHAP:

```
In [97]: shap.plots.beeswarm(shap_values,  
                           max_display=20)
```



El gráfico anterior sirve para entender cómo afectan en las predicciones los diferentes valores que pueden tomar las variables.

Por un lado tenemos las variables `x_index` e `y_index`. Cuando el primero toma valores altos, el predictor tiende a dar menor riesgo de incendio forestal a la vez que cuando toma valores bajos, el predictor tiende a dar menor riesgo. Sin embargo, cuando `y_index` toma valores altos, el predictor suele dar mayor riesgo de incendio. Esto concuerda claramente con el gráfico que hemos hecho en la sección 2.1 del área quemada en España. En dicho gráfico se ve claramente como para valores altos de `y_index` y valores bajos de `x_index` (zona norte de España: Asturias y Galicia), hay muchos incendios. A su vez, para valores altos de `x_index` (zona este de España) se ve como hay menos incendios.

Por otro lado tenemos la variable `forest_percent`, que cuando tiene valores altos hace que el modelo tienda a predecir mayor riesgo. Esto es del todo lógico, ya que en zonas no forestales es más difícil que un fuego se esparza.

Cuando la variable `CLC_27` toma valores bajos, vemos que el modelo tiende a predecir menor riesgo de incendio forestal. Es decir, cuando el terreno no es del tipo "[Moors and heathland](#)", el modelo tiende a predecir menor riesgo de incendio forestal. Esto concuerda nuevamente con el análisis de las distribuciones de las variables por clase que hemos hecho en la sección 4.

También vemos que cuando la celda espacio-temporal está situada en julio o agosto, la variable tiende a predecir mucho más riesgo de incendio forestal. A su vez, cuando la celda se sitúa en noviembre diciembre o enero, el modelo tiende a predecir menor riesgo de incendio.

En cuanto a las variables topográficas, vemos que cuanto mayor es la desviación estandar de la elevación, mayor riesgo tiende a predecir el modelo. Ocurre lo mismo con la suma de la pendiente. Esto puede deberse a que cuando el terreno tiene mucha pendiente y la elevación cambia bruscamente (valores altos de desviación estandar), el fuego se propaga más: el fuego siempre va hacia arriba. Por otro lado, cuanta mayor altitud, el modelo tiende a predecir menor riesgo de incendio forestal. Esto es completamente lógico puesto que a mayor altitud, menor oxígeno y menos vegetación, por lo que el fuego no puede propagarse tan fácilmente.

## 15. Mapas de riesgo

Ahora que hemos analizado las razones por las cuales el modelo predice el riesgo de incendio forestal, vamos a hacer unas predicciones en todas las celdas espaciales para diferentes meses del año.

Para hacer las predicciones tomaremos los valores más actuales disponibles de las variables. Es decir, tomaremos el conjunto de datos de Corine Land Cover del 2018 (ya que el de 2024 no está disponible) y tomaremos como densidad poblacional el año 2020. Además, generaremos los gráficos para cada mes del año.

La interpretación de estas predicciones no está del todo clara. Como este ejercicio es un test inicial del trabajo, no se han incluido las variables con temporalidad diaria (como la temperatura), por lo que nos es imposible diferenciar dos instancias ocurridas en un mismo mes. Es decir, aunque las instancias del conjunto de entrenamiento y del conjunto de test sean todas distintas en cuanto a significado (porque cada instancia tiene una fecha diaria asociada), muchas de esas instancias no las puede diferenciar el modelo. Por ejemplo, si en un mismo mes ocurrieron dos fuegos diferentes en la misma celda, como no tenemos variables con temporalidad diaria y la fecha del incendio no se incluye en las variables que se le dan al modelo, ambas instancias serán idénticas. Esto es, las predicciones que hace el modelo cambian en el tiempo solamente mediante la variable `mes`, por lo que el modelo predice el mismo riesgo de incendio forestal para todo el mes entero. De alguna forma, el modelo predice el "riesgo medio" de incendio forestal que hay en cada celda para todo el mes.

Como este ejercicio tiene intención de ser un primer test del trabajo final y no un predictor mensual del riesgo de incendio forestal, no hay que hacer demasiado caso a las predicciones que hace el modelo. Sin embargo, es interesante visualizar dónde predice mayor riesgo de incendio forestal el modelo y cómo cambian las predicciones según el mes del año.

Para calcular esas predicciones hace falta procesar el `datacube` al formato de datos que se le ha dado al modelo. Eso lo haremos en las siguientes líneas de código:

Procesar los datos para poder hacer las predicciones:

```
In [98]: import geopandas as gpd
datacube = gpd.read_file("../data/FinalDataset/Datacube_1.gpkg")

In [99]: # Delete columns:
columns_to_drop += [f"CLC_{year}_{clas}" for year in [2006, 2012] for clas in range(1,45)] + [f"CLC_{year}_{clas}" for year in [2008, 2020] for clas in range(1,45)]
columns_to_drop += [f"population_density_{year}_{case}" for year in range(2008, 2020) for case in ["min", "max"]]
datacube = datacube.drop(columns=columns_to_drop)

In [100]: # Process CLC:
datacube.rename(columns={f"CLC_2018_{case}":f"CLC_{case}" for case in range(1,45)}, inplace=True)
datacube.rename(columns={f"CLC_2018_{case}":f"CLC_{case}" for case in ["NODATA"]}, inplace=True)
datacube.rename(columns={f"population_density_2020_{case}":f"population_density_{case}" for case in ["min", "max"]}, inplace=True)

# Calculate the Corine Land Cover proportions:
clc_columns = [f"CLC_{i}" for i in range(1,45)] + ["CLC_NODATA"]
sums = datacube[clc_columns].sum(axis=1)
for col in clc_columns:
    datacube[col] = datacube[col] / sums

# Calculate the Aspect proportions:
aspect_columns = ["aspect_NODATA", "aspect_0_45", "aspect_45_90", "aspect_90_135", "aspect_135_180", "aspect_180_225"]
sums = datacube[aspect_columns].sum(axis=1)
for col in aspect_columns:
    datacube[col] = datacube[col] / sums

# Calculate the Natura2000 proportions:
datacube["natura2000_1"] = datacube["natura2000_1"] / (datacube["natura2000_1"] + datacube["natura2000_NODATA"])
datacube.drop(columns=['natura2000_NODATA'], inplace=True)

In [101]: # Add columns:
datacube["artificial_percent"] = datacube[[f"CLC_{i}" for i in range(1,12)]].sum(axis=1)
datacube["agricultural_percent"] = datacube[[f"CLC_{i}" for i in range(12,23)]].sum(axis=1)
datacube["forest_percent"] = datacube[[f"CLC_{i}" for i in range(23,35)]].sum(axis=1)

datacube["dist_to_railway_mean"] = datacube["dist_to_railway_mean"].clip(upper=1.5)

for i in range(1,13):
    datacube[f"month_{i}"] = 0
```

Hacer las predicciones mensuales:

```
In [102]: import pickle

model = pickle.load(open("../data/models/model_XGB.pkl", "rb"))
scaler = pickle.load(open("../data/models/scaler.pkl", "rb"))
```

```

columns_to_scale = ["x_index", "y_index",
                    "elevation_sum", "elevation_stdev",
                    "slope_sum", "slope_stdev", "slope_min", "slope_max",
                    "dist_to_road_mean",
                    "dist_to_waterway_mean",
                    "dist_to_railway_mean",
                    "population_density_mean", "population_density_min", "population_density_max"]

make_predictions = datacube.drop(columns=["geometry"])
make_predictions = make_predictions[X_train.columns]

for month in range(1,13):
    make_predictions_month = make_predictions.copy()
    make_predictions_month[f"month_{month}"] = 1
    make_predictions_month[columns_to_scale] = scaler.transform(make_predictions_month[columns_to_scale])

    predictions = model.predict_proba(make_predictions_month)[:, 1]

    datacube[f"predictions_month_{month}"] = predictions

```

Graficar las predicciones mensuales:

El modelo que estamos utilizando es el `XGBoost` entrenado en datos balanceados, el cual tiene `0.14` como valor óptimo de umbral de decisión. Es decir, si tomamos como instancia de `fuego` las probabilidades mayores a `0.14` y instancia de `no-fuego` las probabilidades menores a `0.14`, obtenemos las mejores métricas conjuntamente de sensibilidad y especificidad: `0.87` y `0.74` respectivamente. Entonces, tenemos que tener esto en cuenta para graficar correctamente el riesgo de incendio forestal. Si añadimos el riesgo utilizando un gradiante de color homogéneo que asigna el verde al valor `0` y rojo al valor `1`, estaríamos graficando el riesgo como si el modelo óptimo fuese con el umbral `0.5`. Para evitar eso, vamos a transformar las predicciones del modelo `XGBoost` para que el `0.14` se quede como `0.5` de forma lineal mediante la siguiente fórmula:

$$T(p) = \begin{cases} \frac{p}{0.14} \cdot 0.5 & \text{si } p \in [0, 0.14] \\ 0.5 + \frac{p-0.14}{1-0.14} \cdot 0.5 & \text{si } p \in (0.14, 1] \end{cases}$$

```

In [134... threshold = 0.14

def transform_predictions(predictions, threshold=0.14):

    return np.where(
        predictions <= threshold,
        (predictions / threshold) * 0.5,
        0.5 + ((predictions - threshold) / (1 - threshold)) * 0.5
    )

for month in range(1, 13):
    predictions_transformed = transform_predictions(datacube[f"predictions_month_{month}"], threshold=threshold)
    datacube[f"predictions_month_{month}_transformed"] = predictions_transformed

```

Una vez hechas las transformaciones, utilizaremos un gradiante de color normal y corriente.

```

In [138... import matplotlib.pyplot as plt

meses = {
    1: "Enero",
    2: "Febrero",
    3: "Marzo",
    4: "Abril",
    5: "Mayo",
    6: "Junio",
    7: "Julio",
    8: "Agosto",
    9: "Septiembre",
    10: "Octubre",
    11: "Noviembre",
    12: "Diciembre"
}

for month in range(1, 13):

    fig, axes = plt.subplots(1, 2, figsize=(20, 10))

    for ax in axes:
        ax.set_facecolor('black')

    axes[0].set_xlim(-10, 5)
    axes[0].set_ylim(35, 44)
    axes[1].set_xlim(-18.5, -13)

```

```

axes[1].set_ylim(27, 29.5)

# Gráfico para la Península
im1 = datacube.plot(
    f"predictions_month_{month}_transformed",
    cmap="RdYlGn_r",
    ax=axes[0],
    legend=False,
    vmin = 0,
    vmax = 1
)
axes[0].set_title(f"Peninsula y Baleares", fontsize=18)
axes[0].set_xlabel("Longitud")
axes[0].set_ylabel("Latitud")

# Gráfico para las Islas Canarias
im2 = datacube.plot(
    f"predictions_month_{month}_transformed",
    cmap="RdYlGn_r",
    ax=axes[1],
    legend=False,
    vmin = 0,
    vmax = 1
)
axes[1].set_title(f"Islas Canarias", fontsize=18)
axes[1].set_xlabel("Longitud")
axes[1].set_ylabel("Latitud")

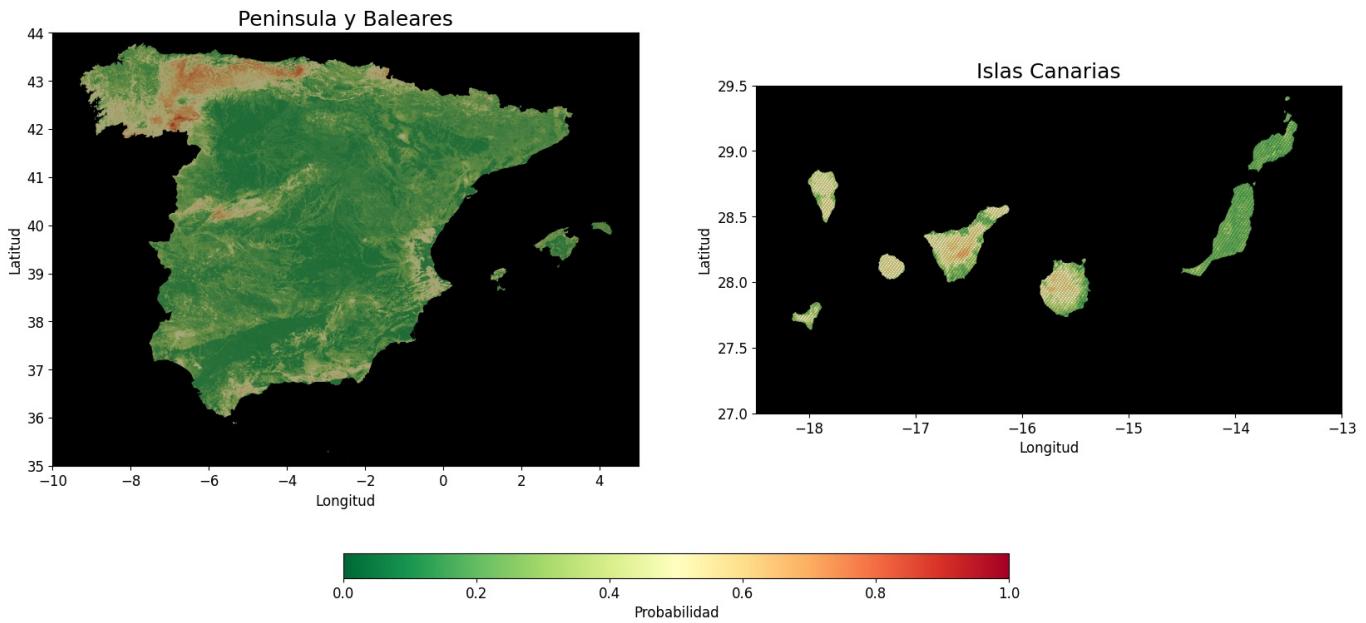
fig.suptitle(
    f"Mapa mensual de riesgo de incendio forestal\nMes: {meses[month]}",
    fontsize=20,
    y=0.88
)
cbar_ax = fig.add_axes([0.3, 0.1, 0.4, 0.03])

cbar = fig.colorbar(im1.get_children()[0], cax=cbar_ax, orientation='horizontal')
cbar.set_label("Probabilidad")

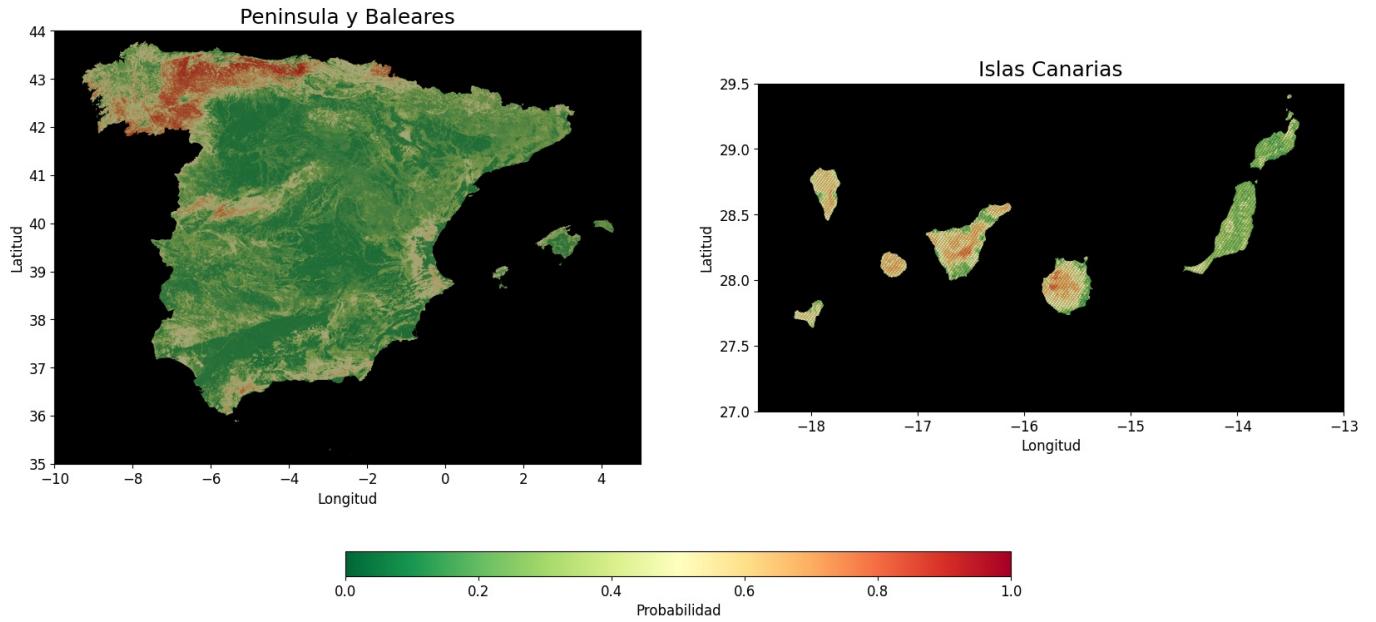
plt.show()

```

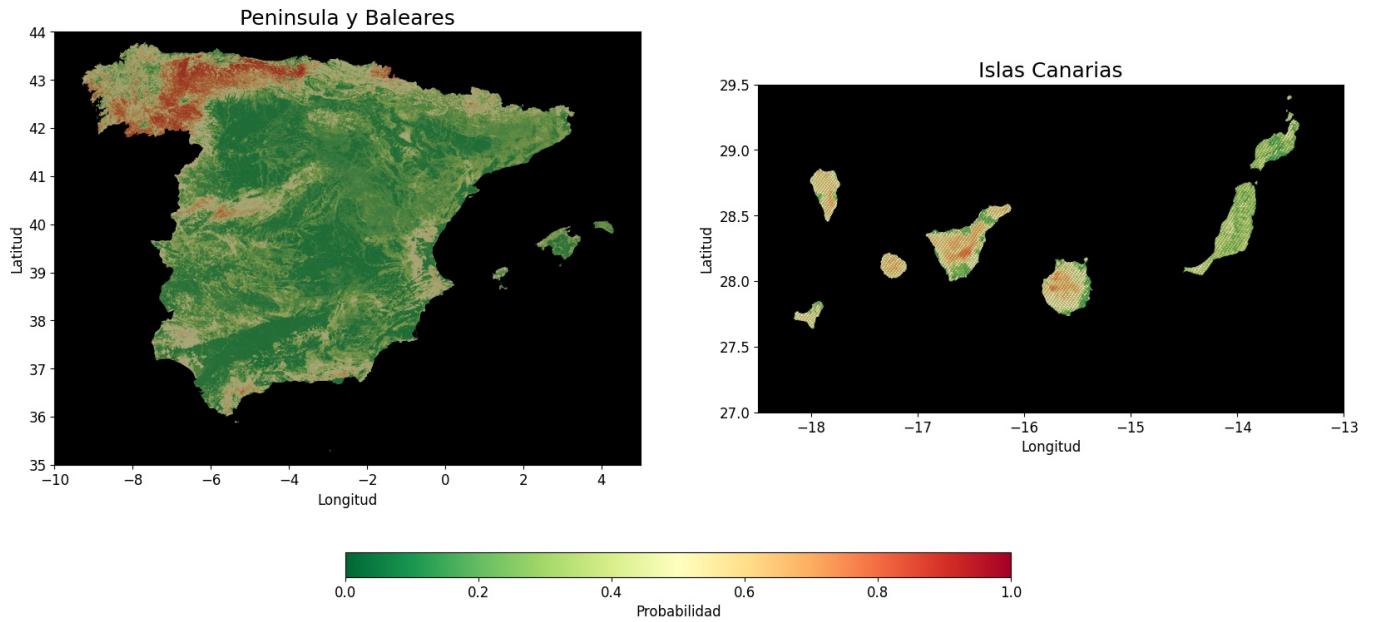
Mapa mensual de riesgo de incendio forestal  
Mes: Enero



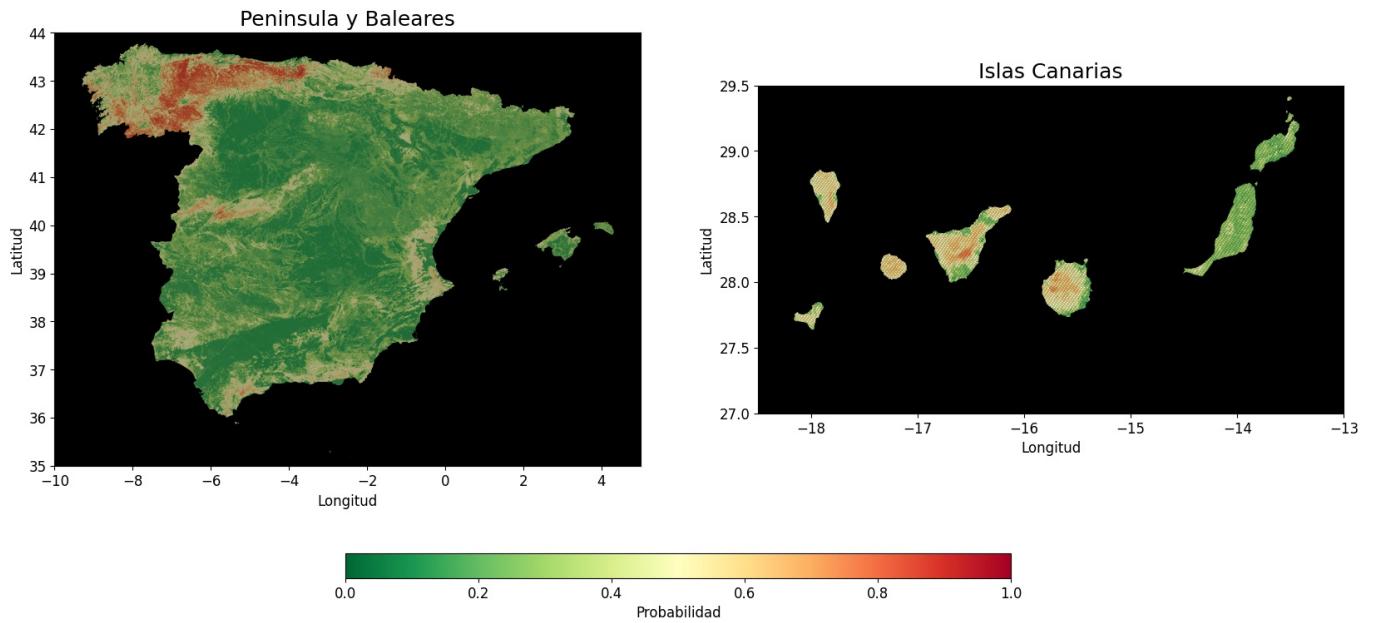
Mapa mensual de riesgo de incendio forestal  
Mes: Febrero



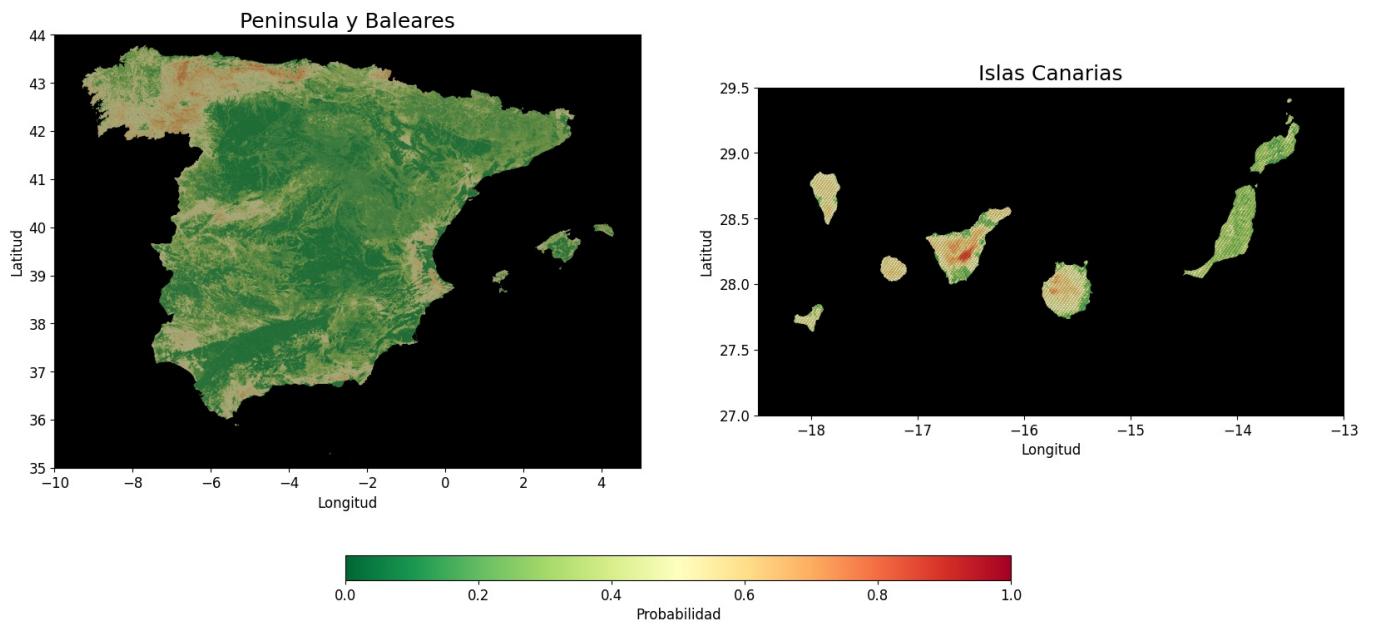
Mapa mensual de riesgo de incendio forestal  
Mes: Marzo



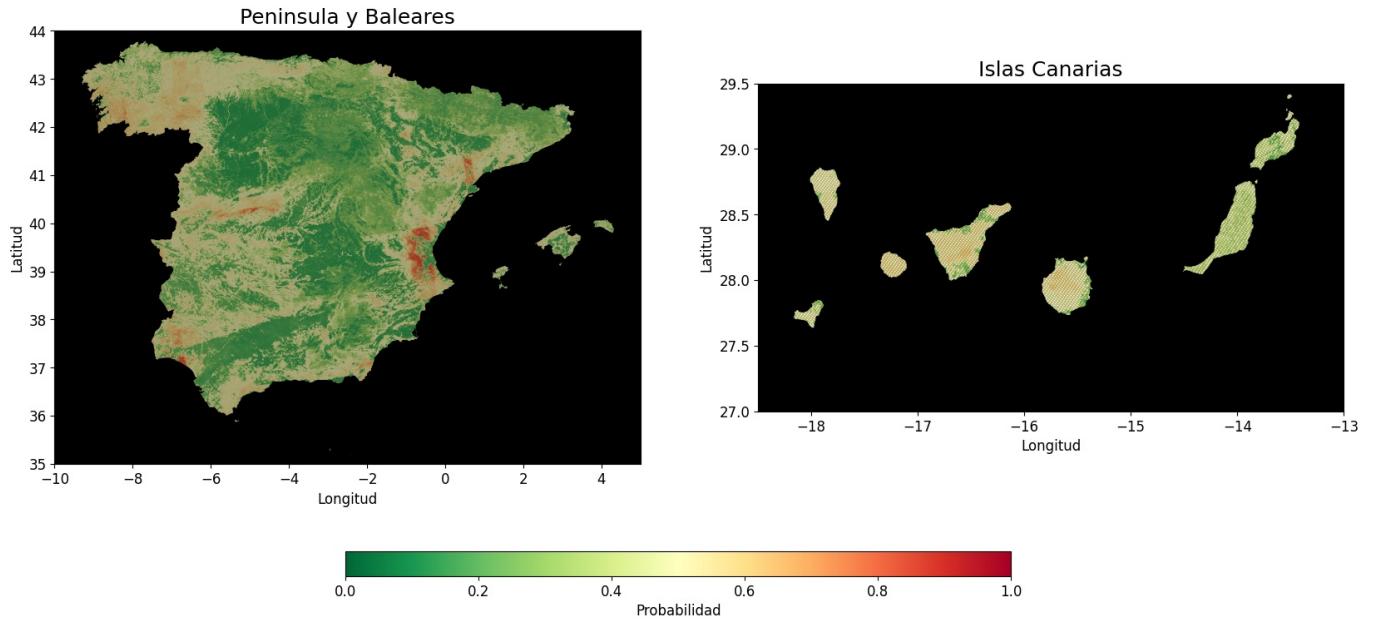
Mapa mensual de riesgo de incendio forestal  
Mes: Abril



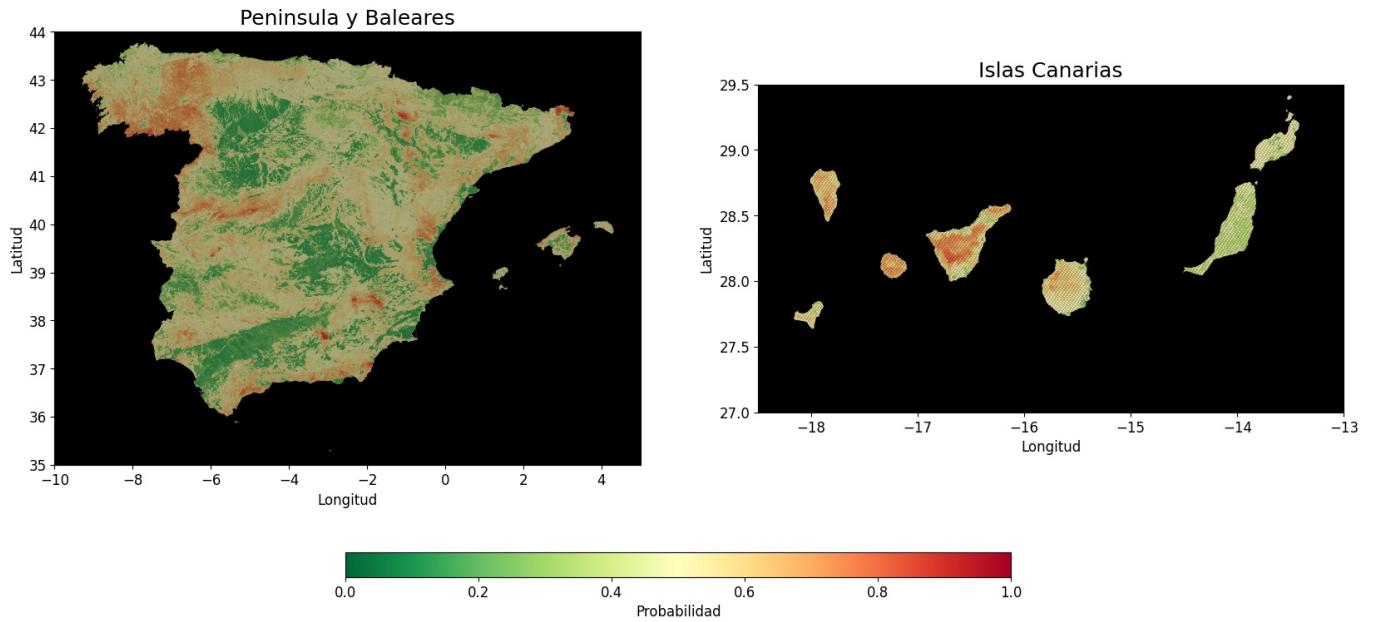
Mapa mensual de riesgo de incendio forestal  
Mes: Mayo



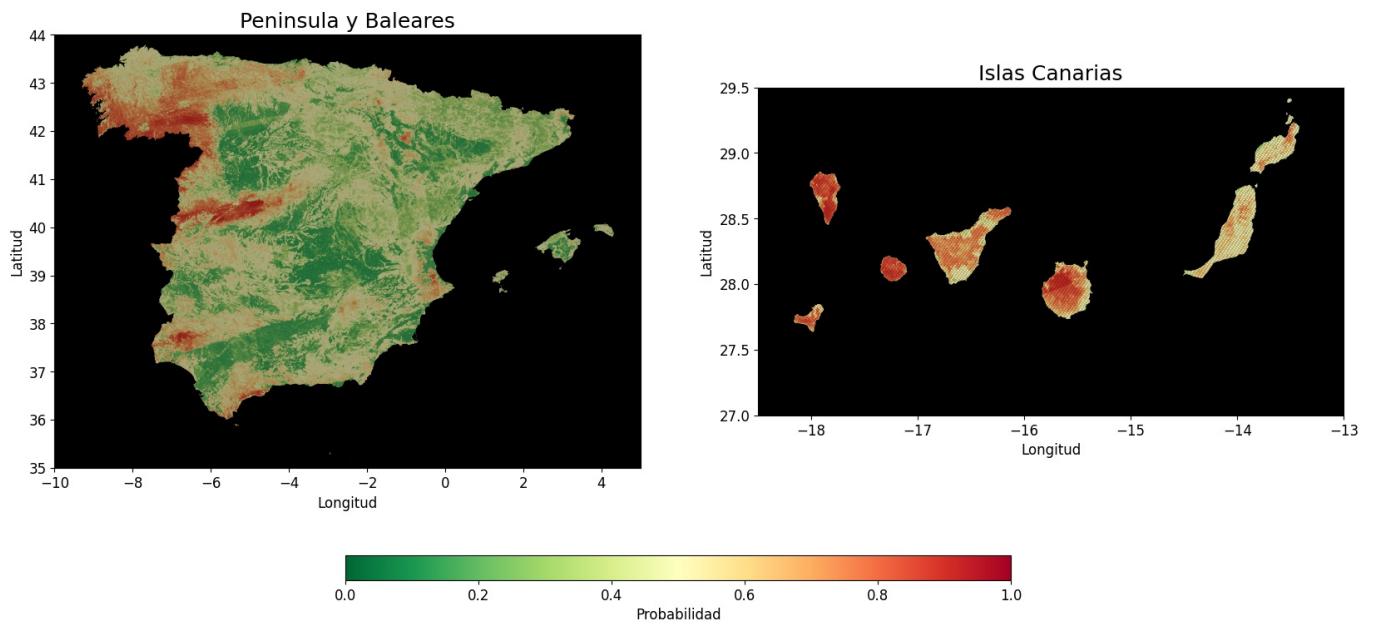
Mapa mensual de riesgo de incendio forestal  
Mes: Junio



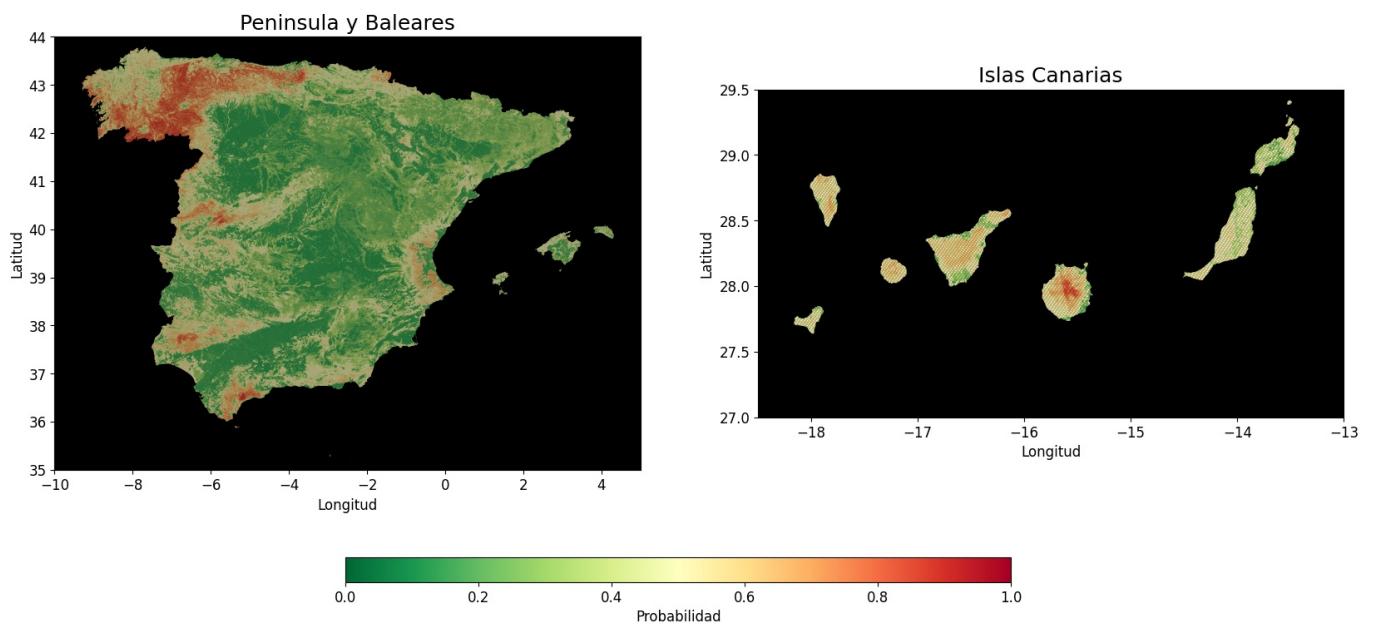
Mapa mensual de riesgo de incendio forestal  
Mes: Julio



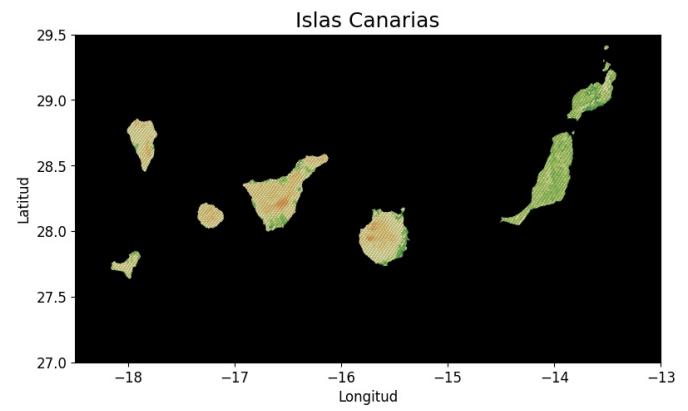
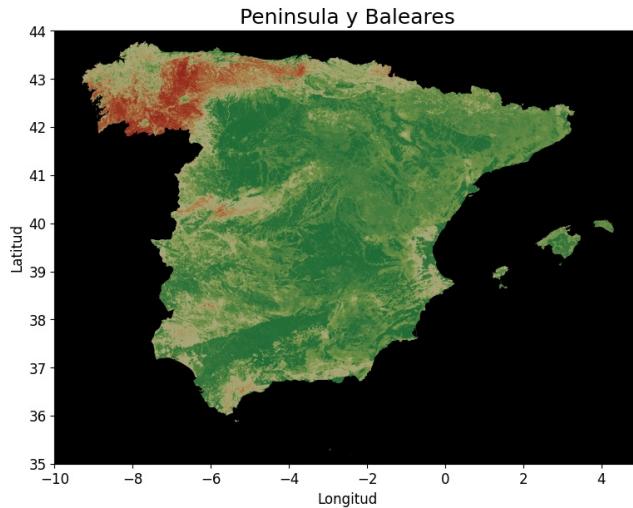
Mapa mensual de riesgo de incendio forestal  
Mes: Agosto



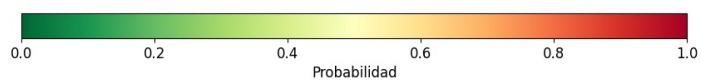
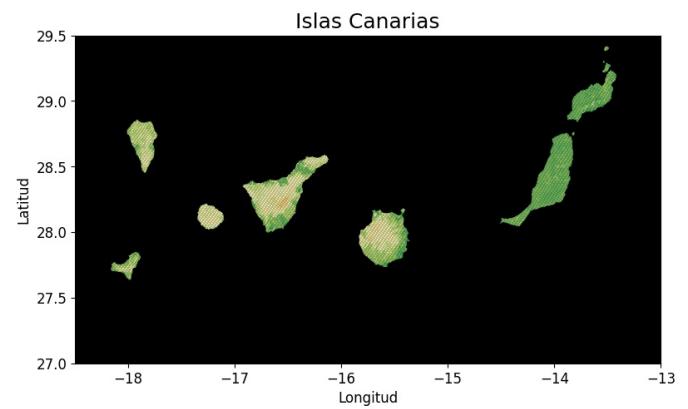
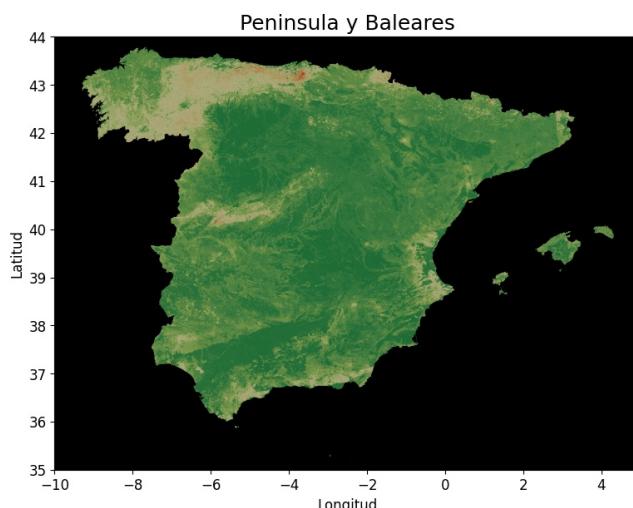
Mapa mensual de riesgo de incendio forestal  
Mes: Septiembre



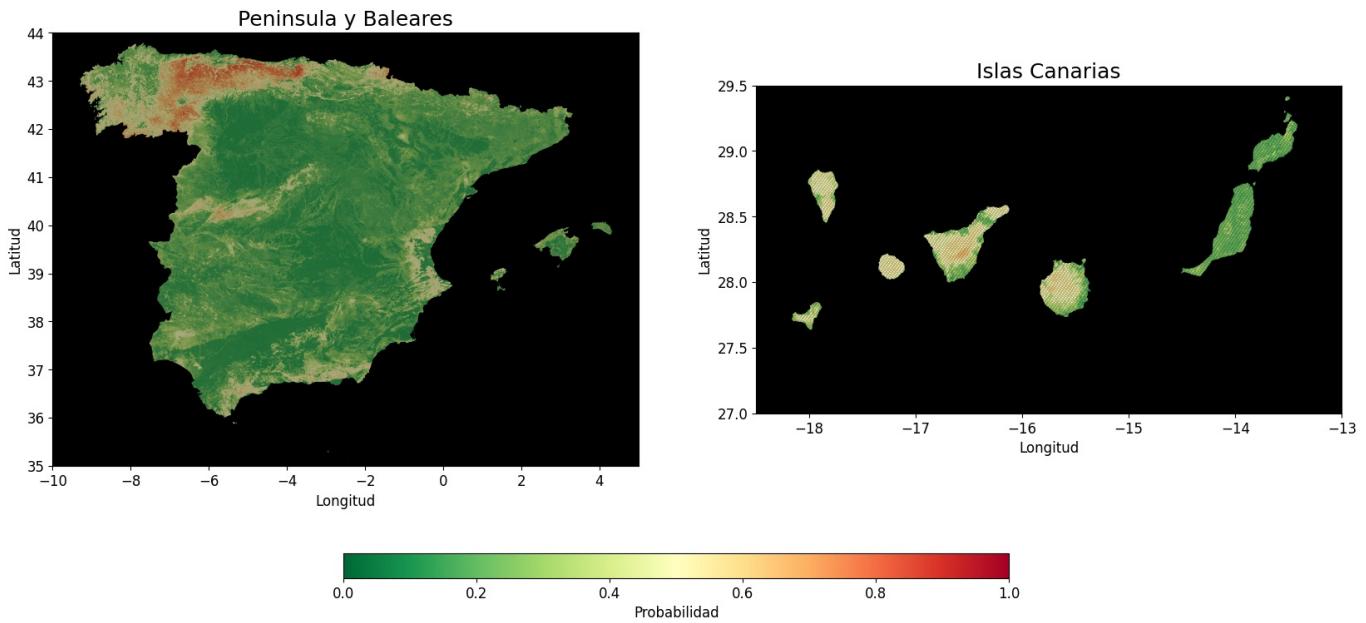
Mapa mensual de riesgo de incendio forestal  
Mes: Octubre



Mapa mensual de riesgo de incendio forestal  
Mes: Noviembre



Mapa mensual de riesgo de incendio forestal  
Mes: Diciembre



Tras hacer las predicciones se puede observar como las conclusiones obtenidas mediante los SHAP values se cumplen: los meses julio y agosto tienen mayor riesgo general de incendio forestal que los meses noviembre y diciembre. Por otro lado, también se puede observar como en la zona del norte de España, en Galicia y Asturias, hay un riesgo alto de incendio durante casi todo el año. Esto también concuerda con los resultados obtenidos mediante los SHAP values.

Cabe recalcar que la variable `forest_percent` es muy importante para el modelo. Como hemos dicho en el análisis de los SHAP values, cuando dicha variable toma valores altos, el modelo tiende a predecir mayor riesgo de incendio forestal. Si visualizamos en el mapa de España dicha variable, vemos claramente como concuerda con las zonas con mayor riesgo de incendio forestal, por lo que una vez mas, el análisis de los SHAP values concuerda con las predicciones hechas por el modelo.

Resumiendo, las predicciones que de riesgo de incendio forestal que hace el modelo son coherentes con el análisis detallado de las variables proporcionado por los SHAP values. Esto refuerza la robustez del modelo y su capacidad para identificar patrones relevantes en los datos.

```
In [139]: import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

for ax in axes:
    ax.set_facecolor('black')

axes[0].set_xlim(-10, 5)
axes[0].set_ylim(35, 44)
axes[1].set_xlim(-18.5, -13)
axes[1].set_ylim(27, 29.5)

# Gráfico para la Península
im1 = datacube.plot(
    f"forest_percent",
    cmap="viridis",
    ax=axes[0],
    legend=False,
    vmin = 0,
    vmax = 1
)
axes[0].set_title(f"Peninsula y Baleares", fontsize=18)
axes[0].set_xlabel("Longitude")
axes[0].set_ylabel("Latitude")
```

```

# Gráfico para las Islas Canarias
im2 = datacube.plot(
    f"forest_percent",
    cmap="viridis",
    ax=axes[1],
    legend=False,
    vmin = 0,
    vmax = 1
)
axes[1].set_title(f"Islands Canarias", fontsize=18)
axes[1].set_xlabel("Longitude")
axes[1].set_ylabel("Latitude")

fig.suptitle(
    f"Variable 'forest percent' en España",
    fontsize=20,
    y=0.88
)

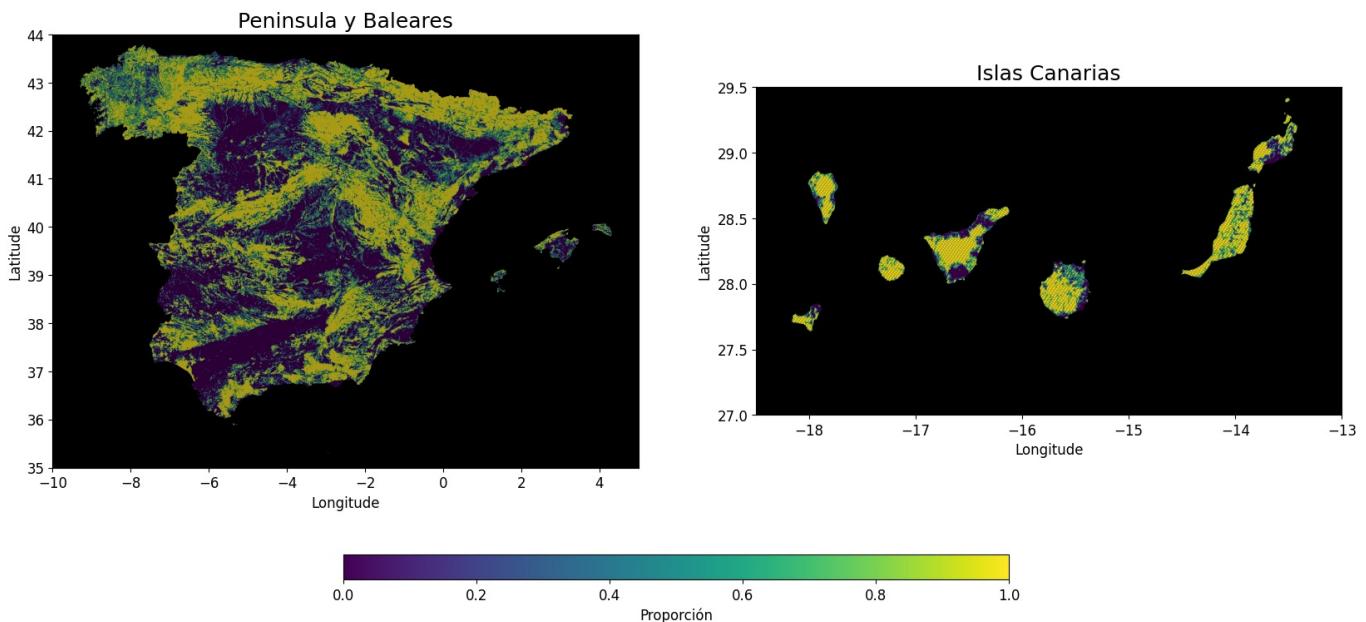
cbar_ax = fig.add_axes([0.3, 0.1, 0.4, 0.03])

cbar = fig.colorbar(im1.get_children()[0], cax=cbar_ax, orientation='horizontal')
cbar.set_label("Proporción")

plt.show()

```

Variable 'forest percent' en España



## 16. Conclusiones

En este trabajo hemos hecho una primera prueba de predicción del riesgo de incendio forestal siguiendo la metodología que se pretende aplicar en el Trabajo final de Fin de Máster. Hemos obtenido muy buenos modelos que han sido capaces de predecir correctamente el 87% de todos los incendios y con un ROC-AUC del 0.9, solamente utilizando datos que cambian muy poco en el tiempo. Sin embargo, cabe recalcar que los modelos se han testado en conjuntos de datos muy balanceados comparando con la distribución real de los datos, cosa que favorece el rendimiento de los modelos. En un escenario real, donde los datos están desbalanceados debido a que los incendios forestales son eventos raros en comparación con la ausencia de incendios, el rendimiento de los modelos se vería muy mermado. Sin embargo, al ser esta una primera prueba que no incluye los factores que más afectan a los incendios (precipitaciones, temperatura y viento) ni pretende ser un buen predictor diario de riesgo de incendio forestal, no hay que preocuparse por testar en un conjunto de datos desbalanceado.

Este ejercicio ha servido para ver que vamos por el buen camino con la metodología seguida y testar qué tanto y cómo afectan en el riesgo de incendio forestal las variables atemporales que se han introducido. Esto se ha hecho mediante los "SHAP values" y "permutation feature importance". Además, el análisis profundo de los datos que se ha seguido en este trabajo ha servido para encontrar nuevas variables muy importantes como la variable `forest_percent`, y encontrar patrones interesantes en los datos al hacer "outlier detection". Se ha descubierto que las Islas Canarias se consideran como "outliers" por lo que en el trabajo final puede ser interesante hacer un predictor para la península y otro para las Islas Canarias.

En resumen, aunque los resultados preliminares son alentadores, queda un camino significativo por recorrer para adaptar y optimizar la metodología a los objetivos que tenemos. Los pasos futuros incluirán la incorporación de variables con temporalidad diaria como la `temperatura`, las `precipitaciones`, índices de vegetación como el `NDVI` y índices de riesgo de incendio forestal como el `FWI`. También se entrenaran modelos más sofisticados de `Deep Learning` introduciendo contexto temporal y espacial mediante los cuales se esperan obtener buenos resultados en conjuntos de test con distribuciones desbalanceadas reales.