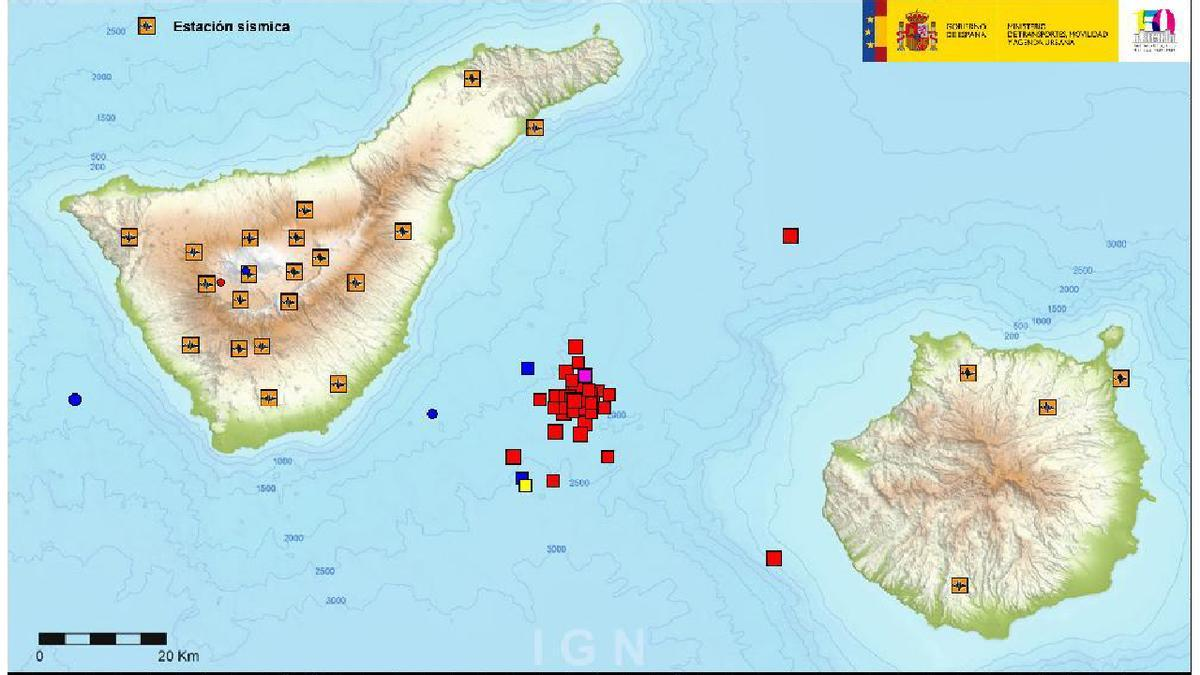
Predicción de Riesgo de derrumbamiento Terremotos



Adrián Yared Armas de la Nuez

**Contenido**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[**1. Objetivo 2**](#_2c35568x6wg8)

[**2. Resolución 2**](#_qgsn3mjaq1no)

[**2.1 Imports e instalación 2**](#_t8cwbo6n4466)

[**2.1.1 Comando 2**](#_aig92oe7k2m0)

[**2.1.1 Instalación Lazypredict 3**](#_e9uqlcl56mu)

[**2.2 Dataset 3**](#_f7btayu18wps)

[**2.2.1 Descarga 3**](#_8kjl2s31xjnh)

[**2.2.2 Carga del dataset 3**](#_eplnnfk0eef8)

[**2.2.2.1 Código 3**](#_ylk7qd2kfd7l)

[**2.2.2.2 Ejecución 4**](#_mrl2whb6xw4f)

[**2.3 Análisis de datos 4**](#_j3l1fukwutuw)

[**2.3.1 Código 4**](#_bl47kpfittu7)

[**2.3.2 Resultado 6**](#_gc4zhxsjfhaf)

[**2.3.2.1 Distribución de clases de daño 6**](#_vv5lb9sh2k2n)

[**2.3.2.2 Nivel de daño 1 7**](#_z1tv12l855km)

[**2.3.2.3 Nivel de daño 2 7**](#_cjtyv7l3kl15)

[**2.3.2.4 Nivel de daño 3 7**](#_1axhxs77vnl)

[**2.3.2.5 Count floors 8**](#_qo5u13hfm9a5)

[**2.3.2.6 Edad 8**](#_z4xwbgk7t1lt)

[**2.3.2.7 Porcentaje por área 8**](#_mg3lsl24nro)

[**2.3.2.8 Porcentaje por altura 9**](#_wbjhdh3plb4s)

[**2.3.2.9 Matriz de correlación 9**](#_93b2nogtbzzo)

[**2.3.2.10 Proporción de clases por condición del suelo 9**](#_gz7s0dytuwhq)

[**2.3.2.11 Proporción de clases por tipo de cimentación 10**](#_9ltqkxjbr7wn)

[**2.3.2.12 Proporción de clases por tipo de tejado 10**](#_5mv7crishh9m)

[**2.3.2.13 Proporción de clases por tipo de suelo 10**](#_fkoh75oeeeu7)

[**2.3.2.14 Proporción de clases por otro tipo de suelo 10**](#_3rthojryigh6)

[**2.3.2.15 Tabla de datos 11**](#_67nzxptqvp32)

[**2.4 Selección de características 11**](#_svzcty9dvuh0)

[**2.4.1 Sin dendrogramas 11**](#_ncxrdis0km3x)

[**2.4.1.1 Código 11**](#_q5r9izk1ohwx)

[**2.4.1.2 Ejecución 15**](#_as29aydht6t7)

[**2.4.1.2.1 Tabla F (Anova) 15**](#_y64xj1jgulhg)

[**2.4.1.2.2 Tabla F (Anova) visual 15**](#_8z502atv5zck)

[**2.4.1.2.3 Tabla información mutua 16**](#_j65ffhf35s48)

[**2.4.1.2.4 Tabla información mutua visual 16**](#_djdcdhvv7131)

[**2.4.1.2.5 Tabla RandomForest 17**](#_kxxv3hi1z36u)

[**2.4.1.2.6 RandomForest visual 17**](#_oyp8r86broq3)

[**2.4.1.2.7 Varianza acumulada vs número de componentes 18**](#_jf7gz9wp21ih)

[**2.4.1.2.8 Características comunes 18**](#_ak4rvd705m6g)

[**2.4.1.2.9 Características seleccionadas 18**](#_1s7sa7h5qd9t)

[**2.4.2 Con dendrogramas 18**](#_g2onicudthng)

[**2.4.2.1 Código 19**](#_34hily7n4s67)

[**2.4.2.2 Ejecución 20**](#_m10q0czg0h5g)

[**2.4.2.2.1 Dendograma de características numéricas 20**](#_pkg8kztypoo1)

[**2.4.2.2.2 Matriz de correlación 21**](#_7t4a5kbdklun)

[**2.4.2.2.3 Características seleccionadas 21**](#_b8fx8a3jg4ke)

[**2.5 Preprocesamiento de datos y selección de muestra 21**](#_8gswnn2qh0z9)

[**2.5.1 Comando 21**](#_wywhfclzhret)

[**2.5.2 Resultado 24**](#_w2lyqel1xuq7)

[**2.6 Lazy Predict 24**](#_35bngn16czk7)

[**2.6.1 Código 24**](#_f9wa6jrqlf90)

[**2.6.2 Resultado 26**](#_4leeh03o5xup)

[**2.6.2.1 Tabla comparativa 26**](#_gzdiegbv1sc0)

[**2.6.2.2 Comparativa visual 27**](#_2yu903ehadc1)

[**2.6.2.2.1 Comparativa visual (Precisión) 27**](#_ui0ncj8sr4sv)

[**2.6.2.2.2 Comparativa visual (F1) 27**](#_oxjkg5zem0mb)

[**2.7 Modelos 28**](#_icchvu77o55e)

[**2.7.1 Modelos de árbol 28**](#_6e0meao0vsjg)

[**2.7.1.1 Randomforest 28**](#_jdhk8bo3com)

[**2.7.1.1.1 Código 28**](#_y5xhwbw6lyn)

[**2.7.1.1.2 Resultado 31**](#_hdgsaea1cv1o)

[**2.7.1.2 BeggingClassifier 31**](#_fqnxgsr6huwg)

[**2.7.1.2.1 Código 31**](#_sjd6264383i5)

[**2.7.1.2.2 Resultado 35**](#_evq5a8eu2dn)

[**2.7.1.3 LGBMClassifier 36**](#_cdb5eoghqjer)

[**2.7.1.3.1 Código 36**](#_8c9hpkbjc1fc)

[**2.7.1.3.2 Resultado 40**](#_e0k1rl8it5if)

[**2.7.4 SVG 40**](#_nxyvq8227504)

[**2.7.4.1 Código 40**](#_4kgjn24rpew0)

[**2.7.4.2 Resultado 43**](#_hx5fmgdkr2jb)

[**2.7.5 Comparación de los modelos 44**](#_b9f6ibjf8bgl)

[**2.7.5.1 Código 44**](#_8m90d45coxvl)

[**2.7.5.2 Resultado 46**](#_swos11lsbql4)

[**2.7.5.2.1 Modelos F1 46**](#_wpqtjr8h8cli)

[**2.7.5.2.2 Comparación Gridsearch vs randomizedSearch 47**](#_9bykra4wgimv)

[**2.7.6 Aplicación de RandomizedSearchCV para LGBMClassifier 47**](#_5day44hvzwkg)

[**2.7.6.1 Código 47**](#_9k8mqh3tmwg2)

[**2.7.6.2 Resultado 51**](#_dyj0pxowo11d)

[**2.8 Predicción y csv 51**](#_j7kplgd7iw)

[**2.8.1 Código 51**](#_bwqgjyhqs0b)

[**2.8.2 Resultado 54**](#_shjfe42m3q0a)

[**2.8.1.1 Resultado del archivo 54**](#_5pe5fxbyi8n6)

[**3. Resultado 54**](#_qdzwbc3pmmxi)

[**4. Problemas encontrados 55**](#_3otegx9aao5p)

[**5. Github y Colab 55**](#_lmdngf8dlin3)

## 

## 

## 

## 

## **1. Objetivo**

El objeto de esta actividad es participar en la competición de ofrecida de la web de DrivenData denominada: Richter's Predictor: Modeling Earthquake Damage.

## **2. Resolución**

### **2.1 Imports e instalación**

#### **2.1.1 Comando**

# Importación de librerías necesarias

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from scipy.cluster import hierarchy

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV, StratifiedKFold

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.metrics import f1\_score, confusion\_matrix, classification\_report

from sklearn.model\_selection import ParameterSampler

from sklearn.ensemble import RandomForestClassifier, BaggingClassifier

from sklearn.svm import SVC

from lightgbm import LGBMClassifier

import lazypredict

from lazypredict.Supervised import LazyClassifier

import pickle

from sklearn.metrics import classification\_report

# Importación de bibliotecas específicas para selección de características

from sklearn.feature\_selection import SelectKBest, f\_classif, mutual\_info\_classif, SelectFromModel

from sklearn.ensemble import RandomForestClassifier

from sklearn.decomposition import PCA

from tqdm.notebook import tqdm # Para barras de progreso en notebook

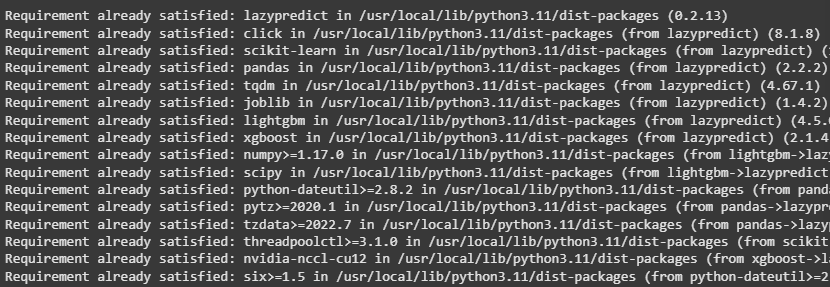
# Si estás usando script y no notebook, usa:

# from tqdm import tqdm

import warnings

#### **2.1.1 Instalación Lazypredict**

!pip install lazypredict



### **2.2 Dataset**

#### **2.2.1 Descarga**

# Download dataset from github

train\_values\_url = "https://raw.githubusercontent.com/AdrianYArmas/IaBigData/refs/heads/main/SNS/3%20%20-%20Algoritmos%20y%20herramientas%20para%20el%20aprendizaje%20supervisado%20/3.7%20%20Predicci%C3%B3n%20de%20Riesgo%20de%20derrumbamiento\_Terremotos/dataset/train\_values.csv"

train\_labels\_url = "https://raw.githubusercontent.com/AdrianYArmas/IaBigData/refs/heads/main/SNS/3%20%20-%20Algoritmos%20y%20herramientas%20para%20el%20aprendizaje%20supervisado%20/3.7%20%20Predicci%C3%B3n%20de%20Riesgo%20de%20derrumbamiento\_Terremotos/dataset/train\_labels.csv"

test\_values\_url = "https://raw.githubusercontent.com/AdrianYArmas/IaBigData/refs/heads/main/SNS/3%20%20-%20Algoritmos%20y%20herramientas%20para%20el%20aprendizaje%20supervisado%20/3.7%20%20Predicci%C3%B3n%20de%20Riesgo%20de%20derrumbamiento\_Terremotos/dataset/test\_values.csv"

#### **2.2.2 Carga del dataset**

##### **2.2.2.1 Código**

Carga de los datos y muestra de las tres primeras líneas para comprobar que o ha dado errores o está null:

# Load datasets

train\_values = pd.read\_csv(train\_values\_url)

train\_labels = pd.read\_csv(train\_labels\_url)

test\_values = pd.read\_csv(test\_values\_url)

print("Dimensions of training dataset (features):", train\_values.shape)

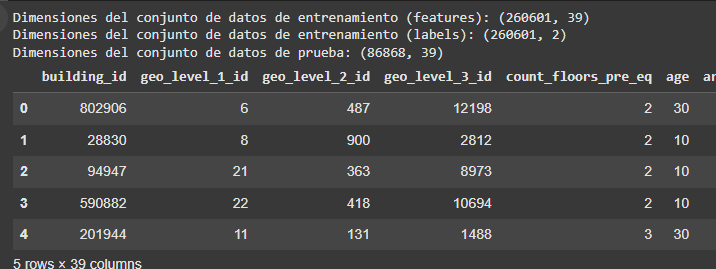
print("Dimensions of training dataset (labels):", train\_labels.shape)

print("Dimensions of test dataset:", test\_values.shape)

# Display the first records

train\_values.head()

##### **2.2.2.2 Ejecución**



### **2.3 Análisis de datos**

Este código realiza un análisis exploratorio de datos (EDA) sobre un conjunto de datos de entrenamiento: combina los datos y etiquetas, visualiza la distribución de las clases de daño, explora las características numéricas y categóricas, calcula correlaciones y analiza las columnas binarias, todo con el fin de entender mejor las relaciones entre las variables y el nivel de daño de los edificios.

#### **2.3.1 Código**

# Merge training data and labels for analysis

train\_data = pd.merge(train\_values, train\_labels, on="building\_id")

# Explore target variable distribution

plt.figure(figsize=(10, 6))

damage\_counts = train\_data['damage\_grade'].value\_counts().sort\_index()

damage\_counts.plot(kind='bar', color=['lightgreen', 'orange', 'red'])

plt.title('Damage Class Distribution', fontsize=15)

plt.xlabel('Damage Level', fontsize=12)

plt.ylabel('Number of Buildings', fontsize=12)

plt.xticks([0, 1, 2], ['Low (1)', 'Medium (2)', 'High (3)'])

# Add values above bars

for i, v in enumerate(damage\_counts):

plt.text(i, v + 50, str(v), ha='center', fontsize=10)

plt.tight\_layout()

plt.show()

# Explore numerical features

numerical\_features = ['geo\_level\_1\_id', 'geo\_level\_2\_id', 'geo\_level\_3\_id', 'count\_floors\_pre\_eq', 'age', 'area\_percentage', 'height\_percentage']

fig, axes = plt.subplots(len(numerical\_features), 1, figsize=(12, 4\*len(numerical\_features)))

for i, feature in enumerate(numerical\_features):

sns.boxplot(x='damage\_grade', y=feature, data=train\_data, ax=axes[i])

axes[i].set\_title(f'Distribution of {feature} by Damage Level', fontsize=14)

axes[i].set\_xlabel('Damage Level', fontsize=12)

axes[i].set\_ylabel(feature, fontsize=12)

plt.tight\_layout()

plt.show()

# Correlation matrix of numerical features

plt.figure(figsize=(10, 8))

sns.heatmap(train\_data[numerical\_features + ['damage\_grade']].corr(), annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)

plt.title('Correlation Matrix of Numerical Features', fontsize=15)

plt.tight\_layout()

plt.show()

# Analysis of categorical features

categorical\_features = ['land\_surface\_condition', 'foundation\_type', 'roof\_type', 'ground\_floor\_type', 'other\_floor\_type']

fig, axes = plt.subplots(len(categorical\_features), 1, figsize=(14, 4\*len(categorical\_features)))

for i, feature in enumerate(categorical\_features):

cat\_proportions = pd.crosstab(train\_data[feature], train\_data['damage\_grade'], normalize='index') \* 100

cat\_proportions.plot(kind='bar', stacked=True, ax=axes[i], color=['lightgreen', 'orange', 'red'])

axes[i].set\_title(f'Class Proportion by {feature}', fontsize=14)

axes[i].set\_xlabel(feature, fontsize=12)

axes[i].set\_ylabel('Percentage (%)', fontsize=12)

axes[i].legend(title='Damage Level', labels=['Low (1)', 'Medium (2)', 'High (3)'])

plt.tight\_layout()

plt.show()

# Identify binary columns (encoded as 0-1)

binary\_columns = [col for col in train\_values.columns if set(train\_values[col].unique()) == {0, 1}]

print(f"{len(binary\_columns)} binary columns identified")

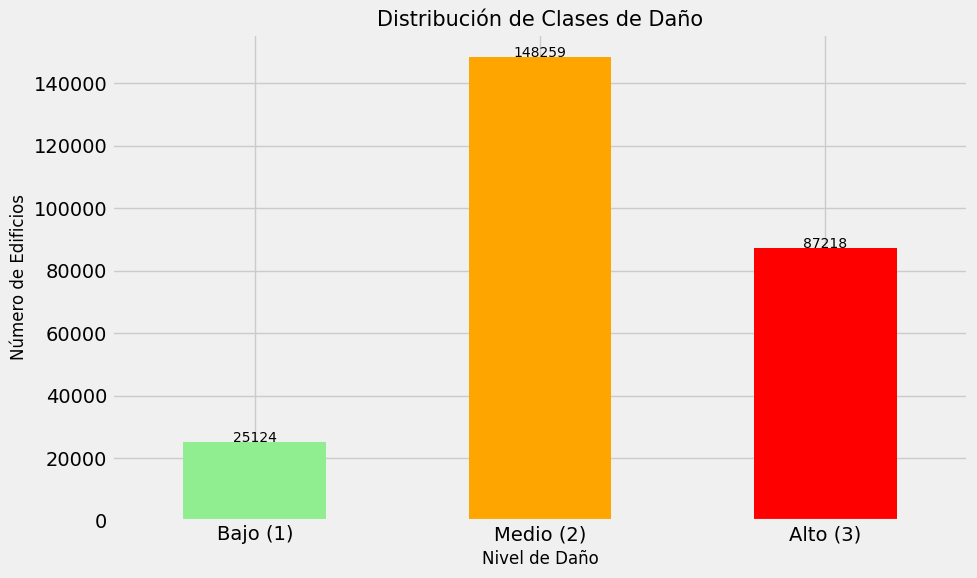
# Statistical summary of important numerical features

train\_data[numerical\_features].describe()

#### **2.3.2 Resultado**

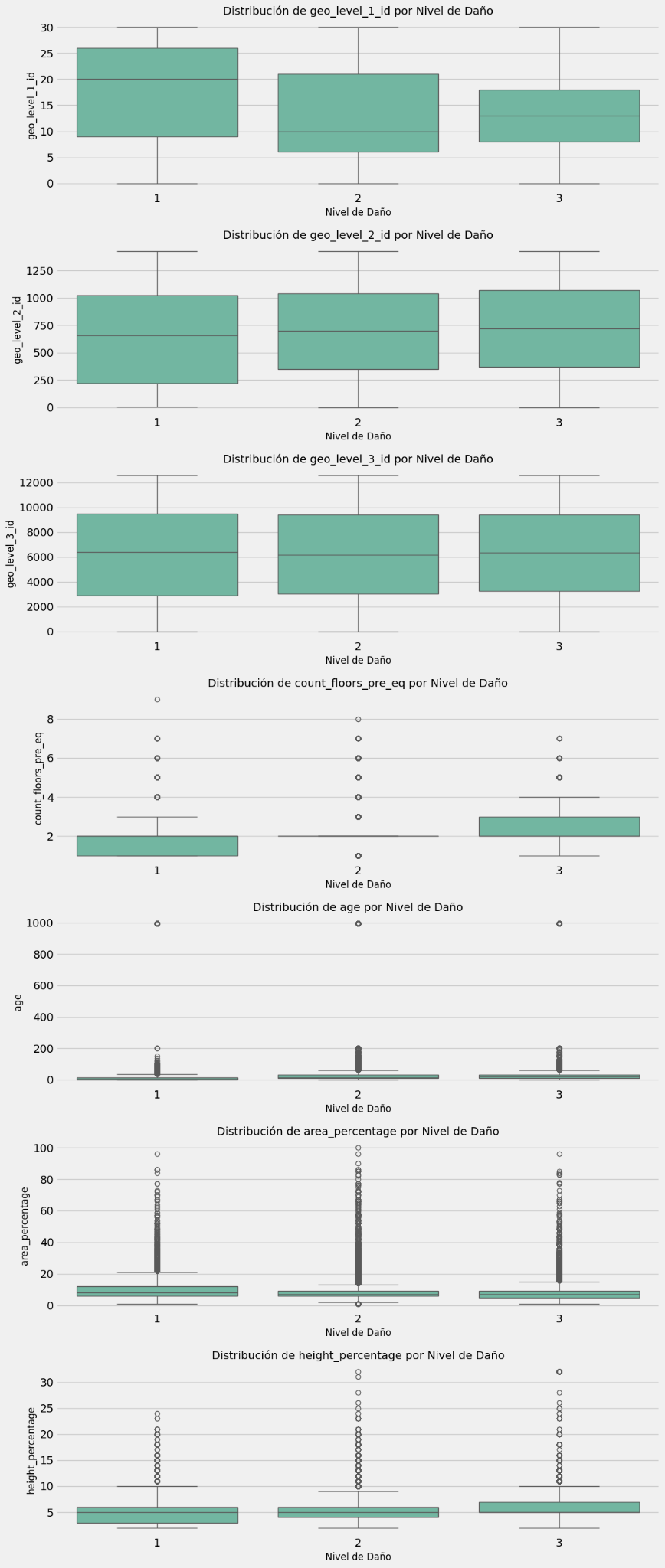
##### **2.3.2.1 Distribución de clases de daño**

El resultado indica la cantidad de casos de destrucción de cada nivel.



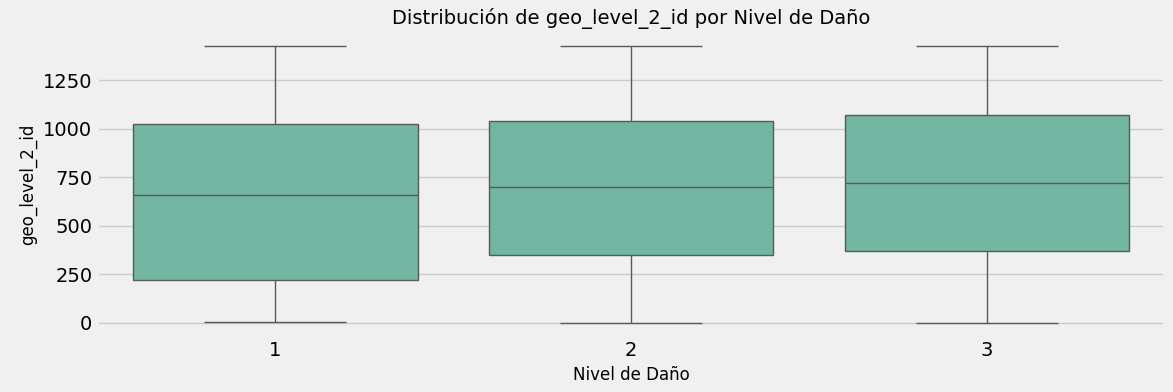
##### **2.3.2.2 Nivel de daño 1**

Este gráfico muestra la distribución geográfica comparada con el nivel de daño 1.



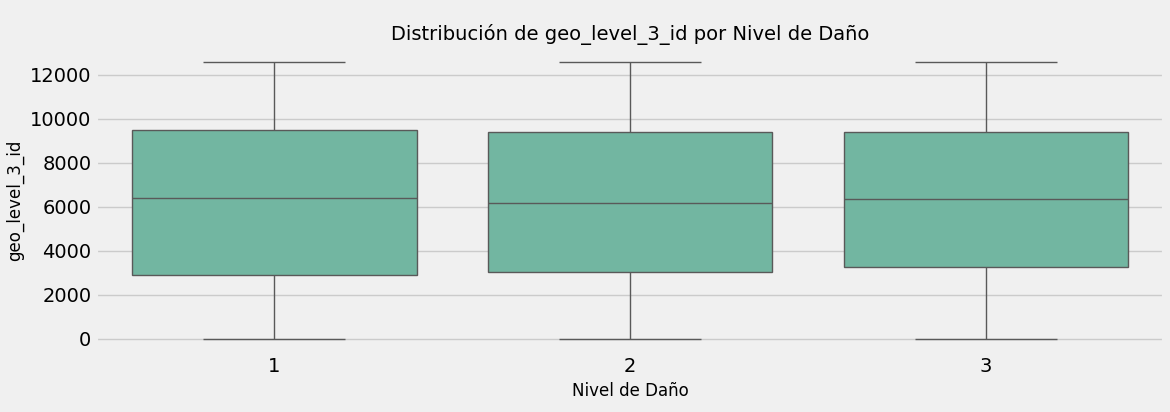
##### **2.3.2.3 Nivel de daño 2**

Este gráfico muestra la distribución geográfica comparada con el nivel de daño 2.



##### **2.3.2.4 Nivel de daño 3**

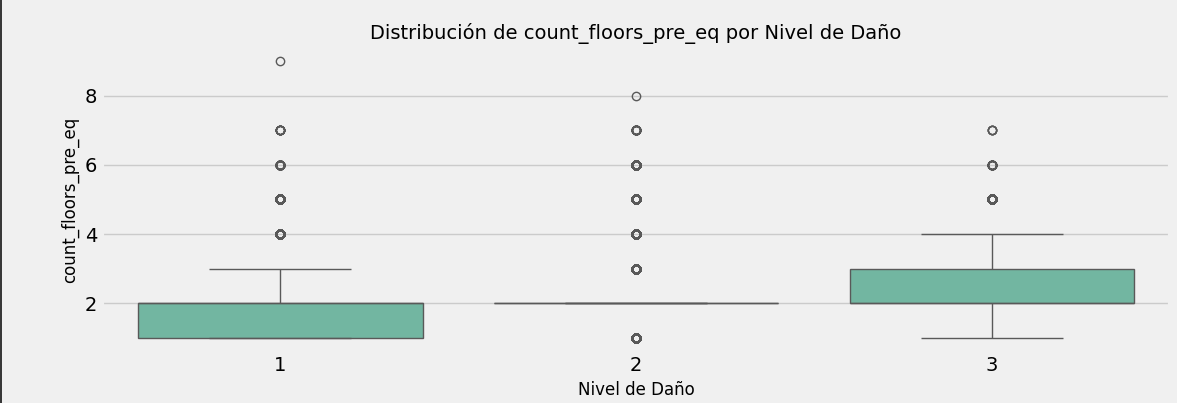
Este gráfico muestra la distribución geográfica comparada con el nivel de daño 3.



##### 

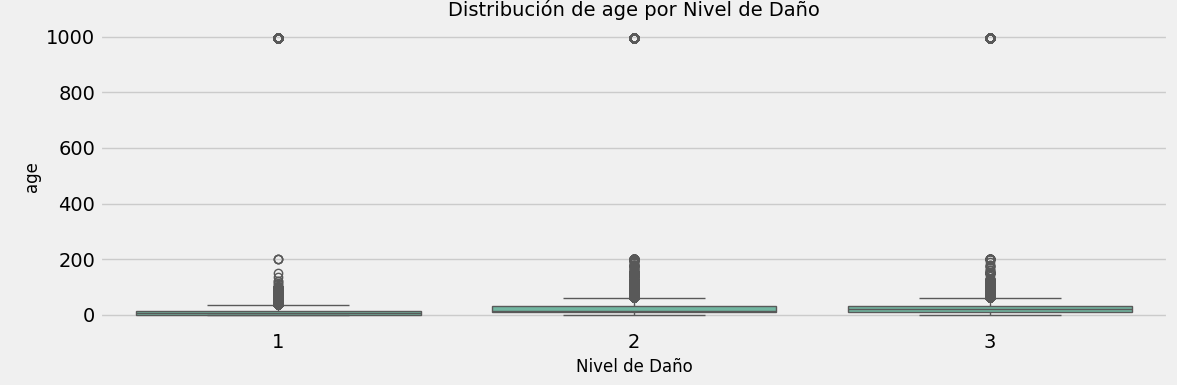
##### **2.3.2.5 Count floors**

Este gráfico muestra la cantidad de pisos del edificio comparados con el nivel de daño.



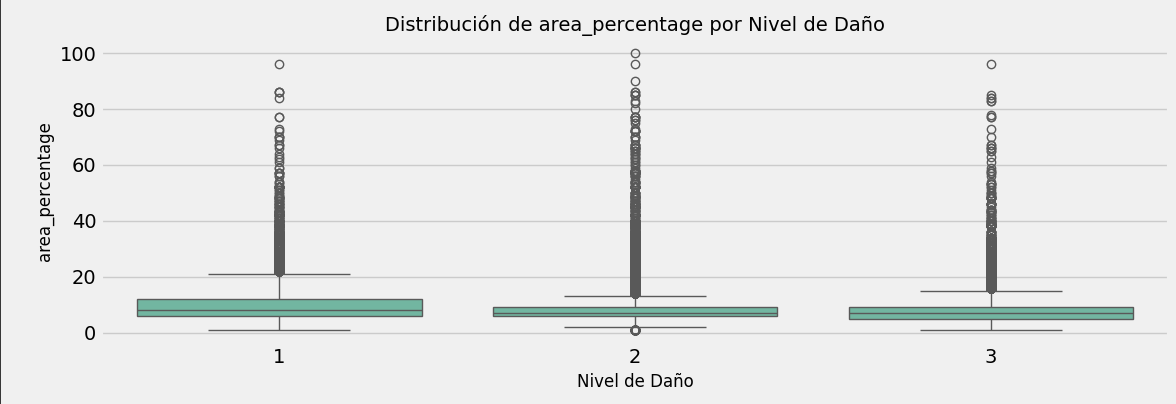
##### **2.3.2.6 Edad**

Este gráfico muestra la cantidad de años del edificio comparados con el nivel de daño.



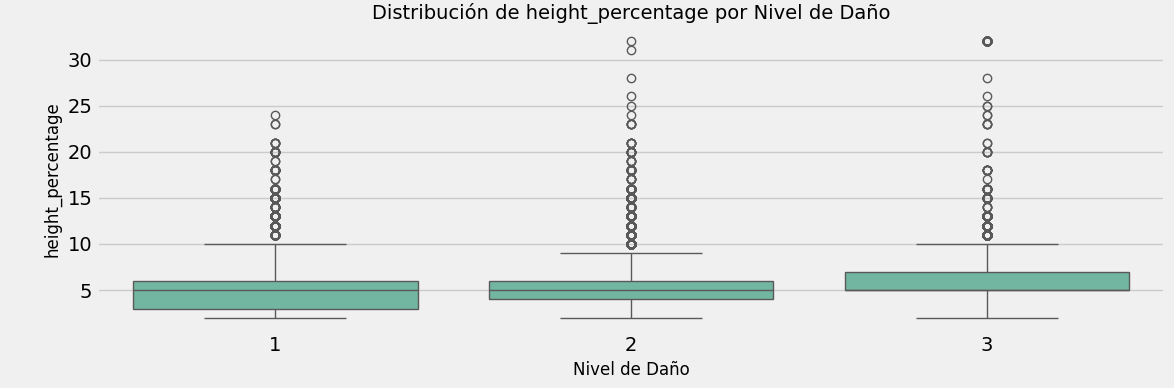
##### **2.3.2.7 Porcentaje por área**

Este gráfico muestra la cantidad de distribución del area comparado con el nivel de daño.



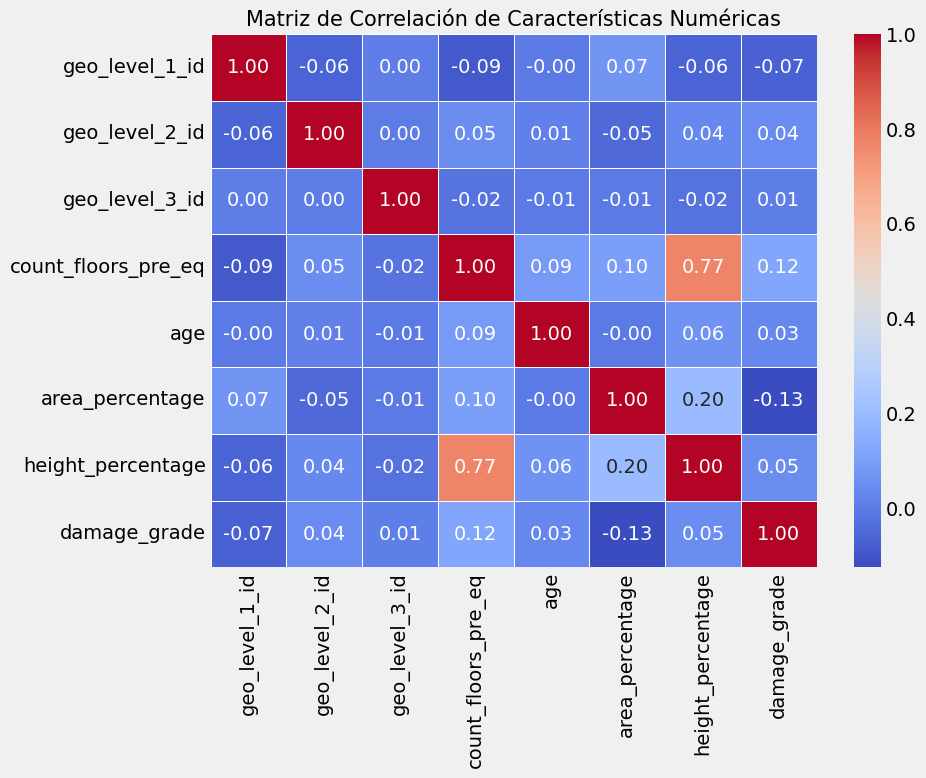
##### **2.3.2.8 Porcentaje por altura**

Este gráfico muestra la altura del edificio comparado con el nivel de daño.



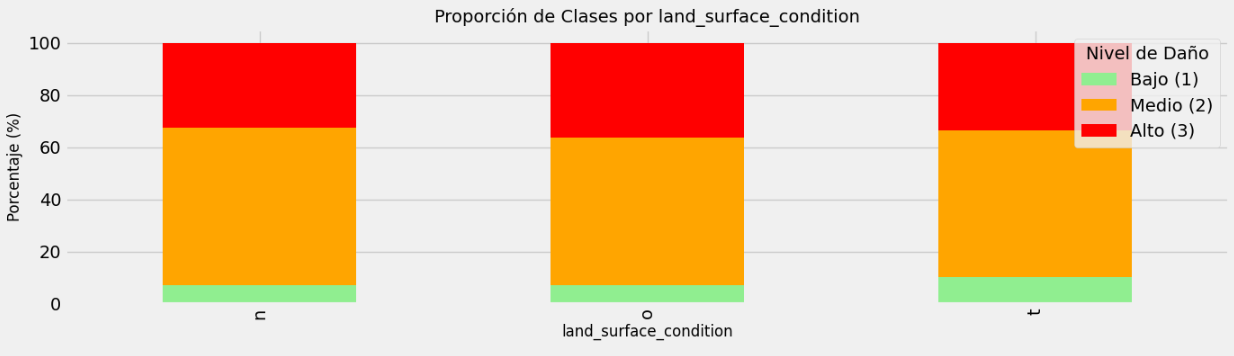
##### **2.3.2.9 Matriz de correlación**

La matriz muestra la correlación entre los datos más relevantes.



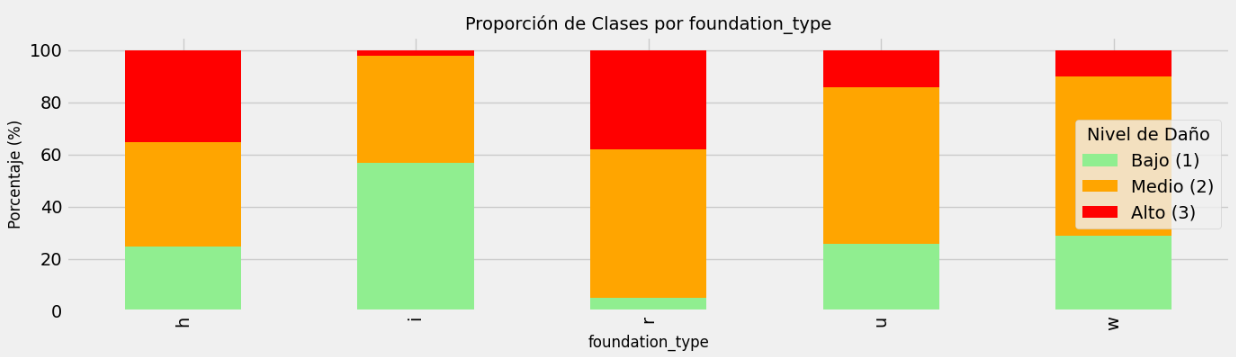
##### **2.3.2.10 Proporción de clases por condición del suelo**

Muestra el porcentaje de clases por condición del suelo



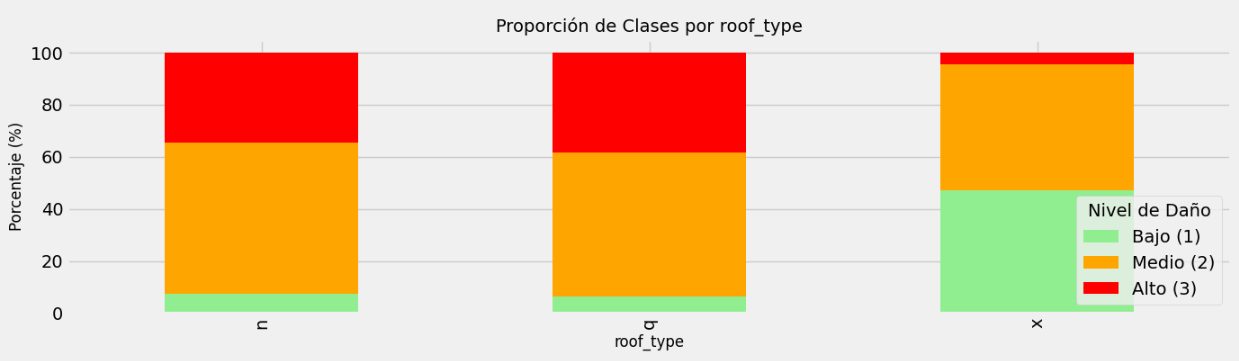
##### **2.3.2.11 Proporción de clases por tipo de cimentación**

Muestra el porcentaje de clases por tipo de cimentación



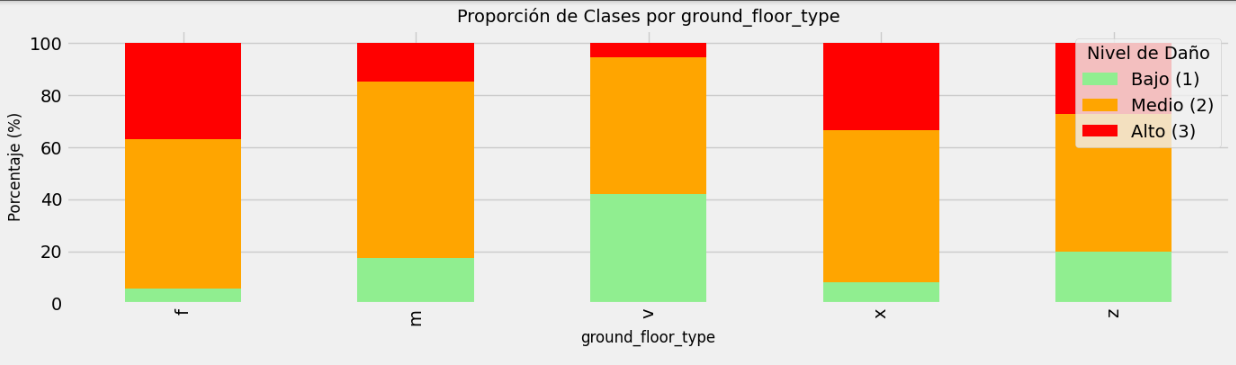
##### **2.3.2.12 Proporción de clases por tipo de tejado**

Muestra el porcentaje de clases por tipo de tejado



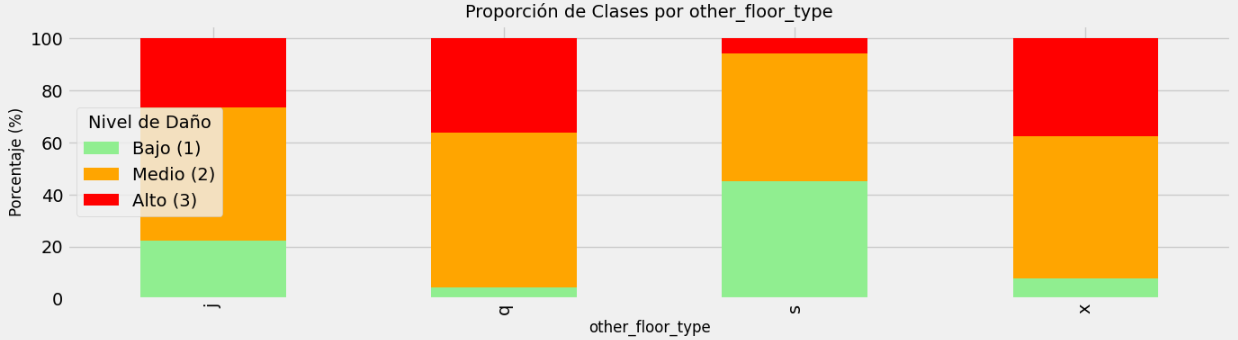
##### **2.3.2.13 Proporción de clases por tipo de suelo**

Muestra el porcentaje de clases por tipo de planta del edificio



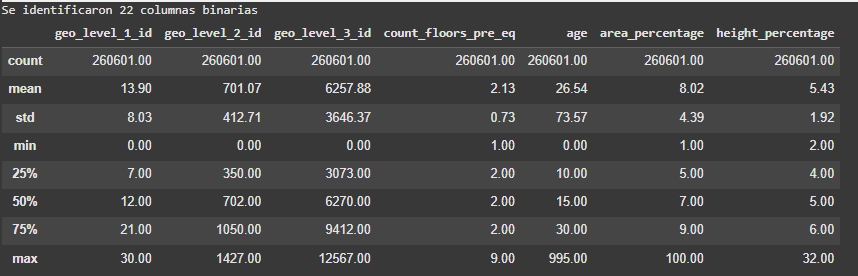
##### **2.3.2.14 Proporción de clases por otro tipo de suelo**

Muestra el porcentaje de clases por tipo de suelo



##### **2.3.2.15 Tabla de datos**

Muestra los datos anteriores en formato tabla.



### **2.4 Selección de características**

#### **2.4.1 Sin dendrogramas**

Este código realiza una selección de características utilizando varios métodos: F-test (ANOVA), información mutua, importancia de características de Random Forest, y análisis de componentes principales (PCA), para identificar las variables más relevantes para predecir el daño de los edificios. Además, combina las características seleccionadas de estos métodos para obtener un conjunto final de variables importantes para el modelo.

##### **2.4.1.1 Código**

# 1. Prepare the data for feature selection

X\_encoded = convert\_categorical\_to\_numeric(train\_data, categorical\_cols).drop(['building\_id'], axis=1)

y = train\_data['damage\_grade'] # Target variable

# 2. F-Test (ANOVA) for feature selection

print("\n--- Feature Selection Based on F-Test (ANOVA) ---")

selector\_f = SelectKBest(f\_classif, k=20)

X\_kbest = selector\_f.fit\_transform(X\_encoded, y)

feature\_scores\_f = pd.DataFrame({'Feature': X\_encoded.columns, 'F-Score': selector\_f.scores\_, 'P-Value': selector\_f.pvalues\_})

top\_features\_f = feature\_scores\_f.sort\_values('F-Score', ascending=False).head(20)

print("Top 20 features according to F-Test:")

display(top\_features\_f)

# Plot the top 15 features

plt.figure(figsize=(12, 8))

sns.barplot(x='F-Score', y='Feature', data=top\_features\_f.head(15))

plt.title('Top 15 Features Based on F-Test (ANOVA)', fontsize=15)

plt.tight\_layout()

plt.show()

# 3. Mutual Information for feature selection

print("\n--- Feature Selection Based on Mutual Information ---")

selector\_mi = SelectKBest(mutual\_info\_classif, k=20)

X\_mi = selector\_mi.fit\_transform(X\_encoded, y)

feature\_scores\_mi = pd.DataFrame({'Feature': X\_encoded.columns, 'Mutual Information': selector\_mi.scores\_})

top\_features\_mi = feature\_scores\_mi.sort\_values('Mutual Information', ascending=False).head(20)

print("Top 20 features according to Mutual Information:")

display(top\_features\_mi)

# Plot the top 15 features

plt.figure(figsize=(12, 8))

sns.barplot(x='Mutual Information', y='Feature', data=top\_features\_mi.head(15))

plt.title('Top 15 Features Based on Mutual Information', fontsize=15)

plt.tight\_layout()

plt.show()

# 4. Feature Importance using RandomForest

print("\n--- Feature Selection Based on RandomForest Importance ---")

feature\_selector\_rf = RandomForestClassifier(n\_estimators=100, random\_state=42, n\_jobs=-1)

feature\_selector\_rf.fit(X\_encoded, y)

feature\_importances = pd.DataFrame({'Feature': X\_encoded.columns, 'Importance': feature\_selector\_rf.feature\_importances\_})

top\_features\_rf = feature\_importances.sort\_values('Importance', ascending=False).head(20)

print("Top 20 features according to RandomForest:")

display(top\_features\_rf)

# Plot the top 15 most important features

plt.figure(figsize=(12, 8))

sns.barplot(x='Importance', y='Feature', data=top\_features\_rf.head(15))

plt.title('Top 15 Features Based on Random Forest', fontsize=15)

plt.tight\_layout()

plt.show()

# 5. Automatically select features above the average importance threshold

selector\_model = SelectFromModel(feature\_selector\_rf, threshold='mean')

selected\_features = X\_encoded.columns[selector\_model.fit\_transform(X\_encoded, y).get\_support()]

print(f"\nAutomatically selected features by RandomForest: {len(selected\_features)}")

print(sorted(selected\_features))

# 6. Principal Component Analysis (PCA)

print("\n--- Principal Component Analysis (PCA) ---")

X\_pca = PCA().fit\_transform(StandardScaler().fit\_transform(X\_encoded))

cumulative\_variance\_ratio = np.cumsum(PCA().explained\_variance\_ratio\_)

n\_components\_95 = np.argmax(cumulative\_variance\_ratio >= 0.95) + 1

print(f"Number of components needed to explain 95% of the variance: {n\_components\_95}")

# Plot cumulative explained variance

plt.figure(figsize=(12, 6))

plt.plot(range(1, len(cumulative\_variance\_ratio) + 1), cumulative\_variance\_ratio, marker='o', linestyle='-')

plt.axhline(y=0.95, color='r', linestyle='--')

plt.axvline(x=n\_components\_95, color='g', linestyle='--')

plt.text(n\_components\_95 + 1, 0.85, f'95% with {n\_components\_95} components', fontsize=12)

plt.title('Cumulative Explained Variance vs Number of Components', fontsize=15)

plt.xlabel('Number of Components', fontsize=12)

plt.ylabel('Cumulative Explained Variance', fontsize=12)

plt.grid(True)

plt.tight\_layout()

plt.show()

# 7. Common features across methods

common\_features\_anova\_mi = set(top\_features\_f['Feature']).intersection(top\_features\_mi['Feature'])

common\_features\_anova\_rf = set(top\_features\_f['Feature']).intersection(top\_features\_rf['Feature'])

common\_features\_mi\_rf = set(top\_features\_mi['Feature']).intersection(top\_features\_rf['Feature'])

common\_features\_all = common\_features\_anova\_mi.intersection(top\_features\_rf['Feature'])

print("\n--- Common Features Across Selection Methods ---")

print(f"Common features in ANOVA and MI: {len(common\_features\_anova\_mi)}")

print(f"Common features in ANOVA and RF: {len(common\_features\_anova\_rf)}")

print(f"Common features in MI and RF: {len(common\_features\_mi\_rf)}")

print(f"Common features in all three methods: {len(common\_features\_all)}")

print("Features selected by all three methods:", sorted(common\_features\_all))

# 8. Final feature selection combining different methods

selected\_features\_from\_dendrogram = selected\_features # Features identified earlier from clustering

selected\_features\_from\_statistical = list(common\_features\_anova\_mi.union(common\_features\_anova\_rf, common\_features\_mi\_rf))

final\_selected\_features = list(set(selected\_features\_from\_dendrogram).union(selected\_features\_from\_statistical))

print("\n--- Final Feature Selection ---")

print(f"Total selected features: {len(final\_selected\_features)}")

print("Final list of selected features:", sorted(final\_selected\_features))

##### 

##### 

##### 

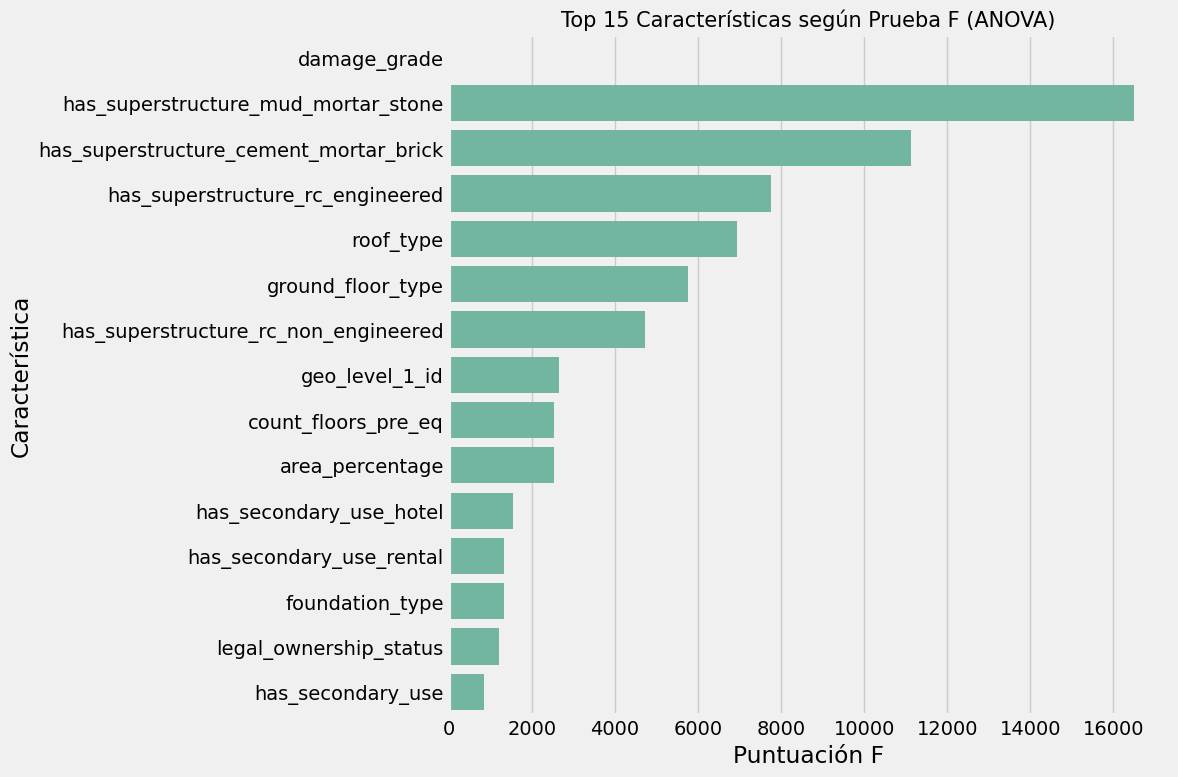
##### **2.4.1.2 Ejecución**

###### **2.4.1.2.1 Tabla F (Anova)**

Análisis de varianza en busca de diferencias significativas:

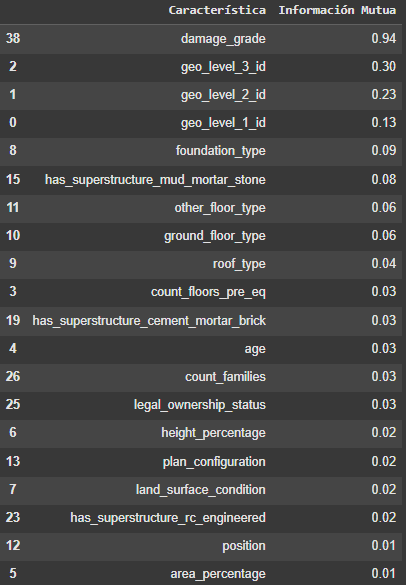


###### **2.4.1.2.2 Tabla F (Anova) visual**

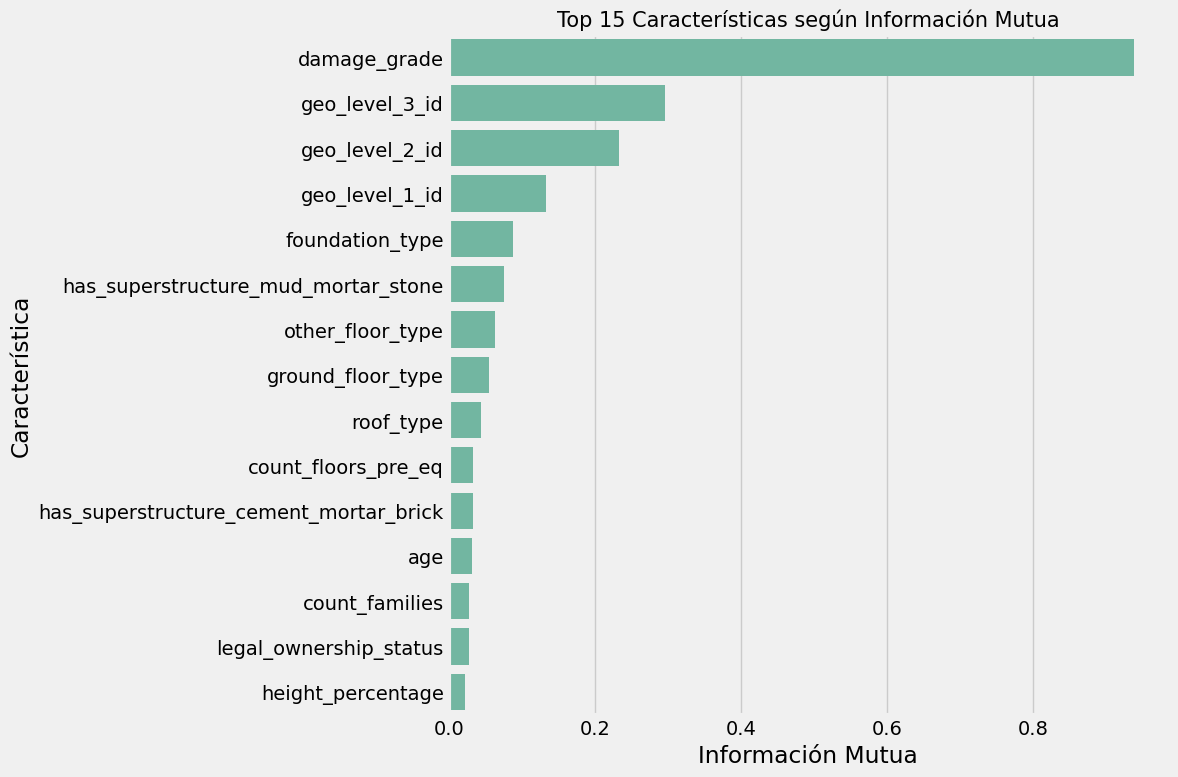


###### **2.4.1.2.3 Tabla información mutua**

Tabla comparativa de la información mutua contrastada a través de un valor numérico.

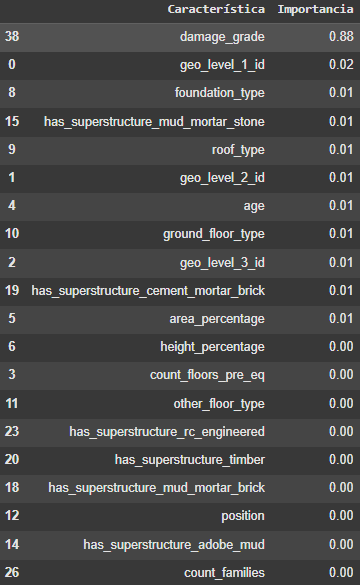


###### **2.4.1.2.4 Tabla información mutua visual**

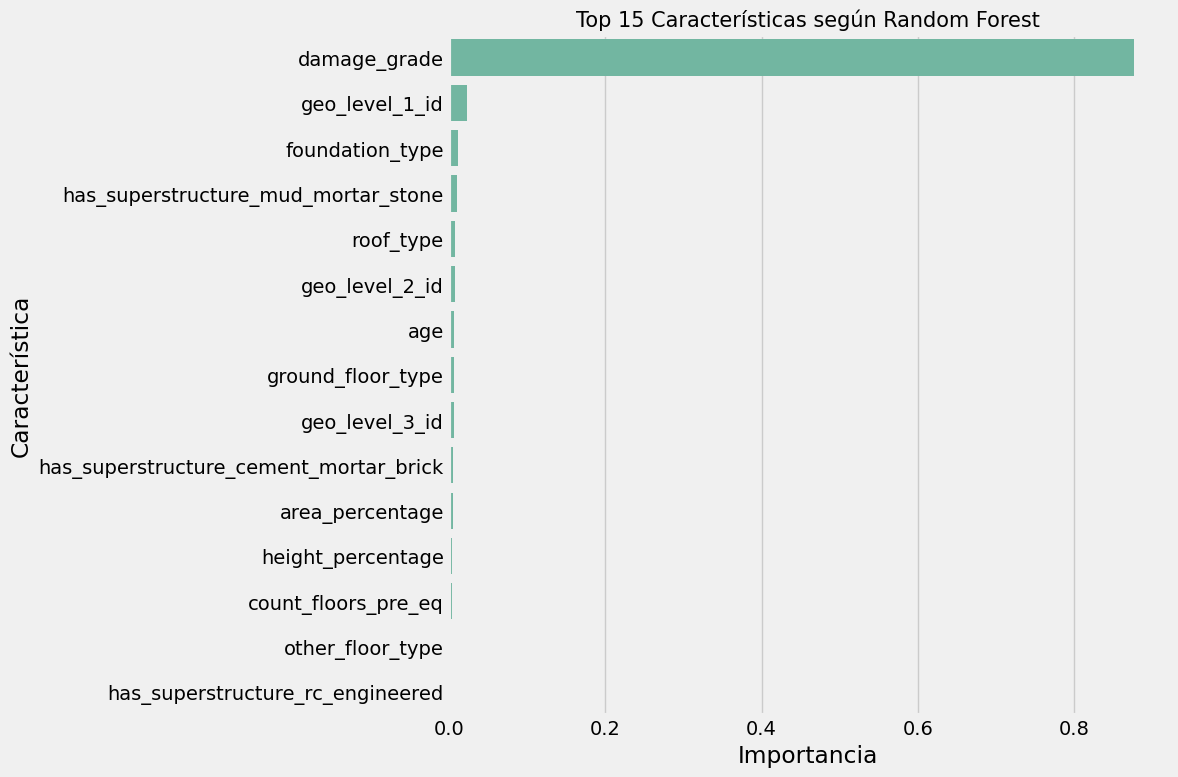


###### **2.4.1.2.5 Tabla RandomForest**

Datos obtenidos del modelo randomForest y su contribución de las variables en el modelo.

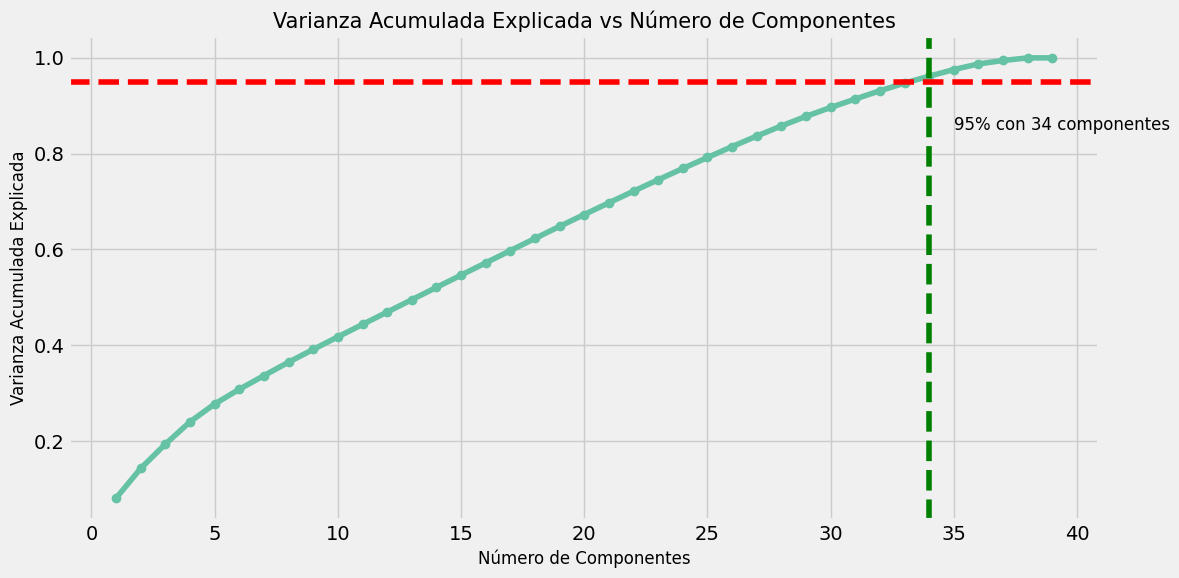


###### **2.4.1.2.6 RandomForest visual**



###### **2.4.1.2.7 Varianza acumulada vs número de componentes**

Esta tabla ayuda a saber cuántos componentes son necesarios para capturar la mayor parte de la información en los datos.



###### 

###### **2.4.1.2.8 Características comunes**

*['area\_percentage', 'count\_floors\_pre\_eq', 'damage\_grade', 'foundation\_type', 'geo\_level\_1\_id', 'ground\_floor\_type', 'has\_superstructure\_cement\_mortar\_brick', 'has\_superstructure\_mud\_mortar\_stone', 'has\_superstructure\_rc\_engineered', 'roof\_type']*

###### **2.4.1.2.9 Características seleccionadas**

*['age', 'area\_percentage', 'count\_families', 'count\_floors\_pre\_eq', 'damage\_grade', 'foundation\_type', 'geo\_level\_1\_id', 'geo\_level\_2\_id', 'geo\_level\_3\_id', 'ground\_floor\_type', 'has\_superstructure\_adobe\_mud', 'has\_superstructure\_cement\_mortar\_brick', 'has\_superstructure\_mud\_mortar\_brick', 'has\_superstructure\_mud\_mortar\_stone', 'has\_superstructure\_rc\_engineered', 'has\_superstructure\_timber', 'height\_percentage', 'legal\_ownership\_status', 'other\_floor\_type', 'position', 'roof\_type']*

#### **2.4.2 Con dendrogramas**

Este código se enfoca en seleccionar las características más importantes para construir un modelo. Primero, escala las variables numéricas y crea un dendrograma para agruparlas según su similitud. Luego, convierte las variables categóricas en números para ver cómo se relacionan con el objetivo (daño de los edificios), visualiza estas relaciones con una matriz de correlación y, finalmente, selecciona las características más relevantes para el modelo, incluyendo algunas variables binarias.

##### **2.4.2.1 Código**

# Select numerical features for analysis

X\_scaled = StandardScaler().fit\_transform(train\_data[numerical\_features])

# Create and visualize the dendrogram for feature clustering

plt.figure(figsize=(14, 10))

dend = hierarchy.dendrogram(

hierarchy.linkage(X\_scaled.T, method='ward'),

labels=numerical\_features,

orientation='right',

leaf\_font\_size=12,

color\_threshold=5

)

plt.title('Dendrogram of Numerical Features', fontsize=16)

plt.xlabel('Distance', fontsize=14)

plt.axvline(x=5, color='red', linestyle='--', alpha=0.7)

plt.tight\_layout()

plt.show()

# Convert categorical features to numerical for correlation analysis

def convert\_categorical\_to\_numeric(df, categorical\_cols):

return df[categorical\_cols].apply(lambda col: col.astype('category').cat.codes)

categorical\_cols = [col for col in train\_data.columns if train\_data[col].dtype == 'object' and col != 'building\_id']

train\_data\_encoded = train\_data.copy()

train\_data\_encoded[categorical\_cols] = convert\_categorical\_to\_numeric(train\_data, categorical\_cols)

# Select a subset of features for correlation matrix

selected\_features = numerical\_features + categorical\_cols[:5] + ['damage\_grade']

# Compute and visualize the correlation matrix

plt.figure(figsize=(14, 12))

correlation = train\_data\_encoded[selected\_features].corr()

sns.heatmap(correlation, mask=np.triu(np.ones\_like(correlation, dtype=bool)),

annot=True, fmt='.2f', cmap='coolwarm', linewidths=0.5, vmin=-1, vmax=1)

plt.title('Correlation Matrix Between Features and Target Variable', fontsize=16)

plt.tight\_layout()

plt.show()

# Select relevant features based on dendrogram and correlation analysis

selected\_features = [

'count\_floors\_pre\_eq', 'age', 'area\_percentage', 'height\_percentage',

'land\_surface\_condition', 'foundation\_type', 'roof\_type',

'ground\_floor\_type', 'other\_floor\_type'

]

# Include top 10 binary features

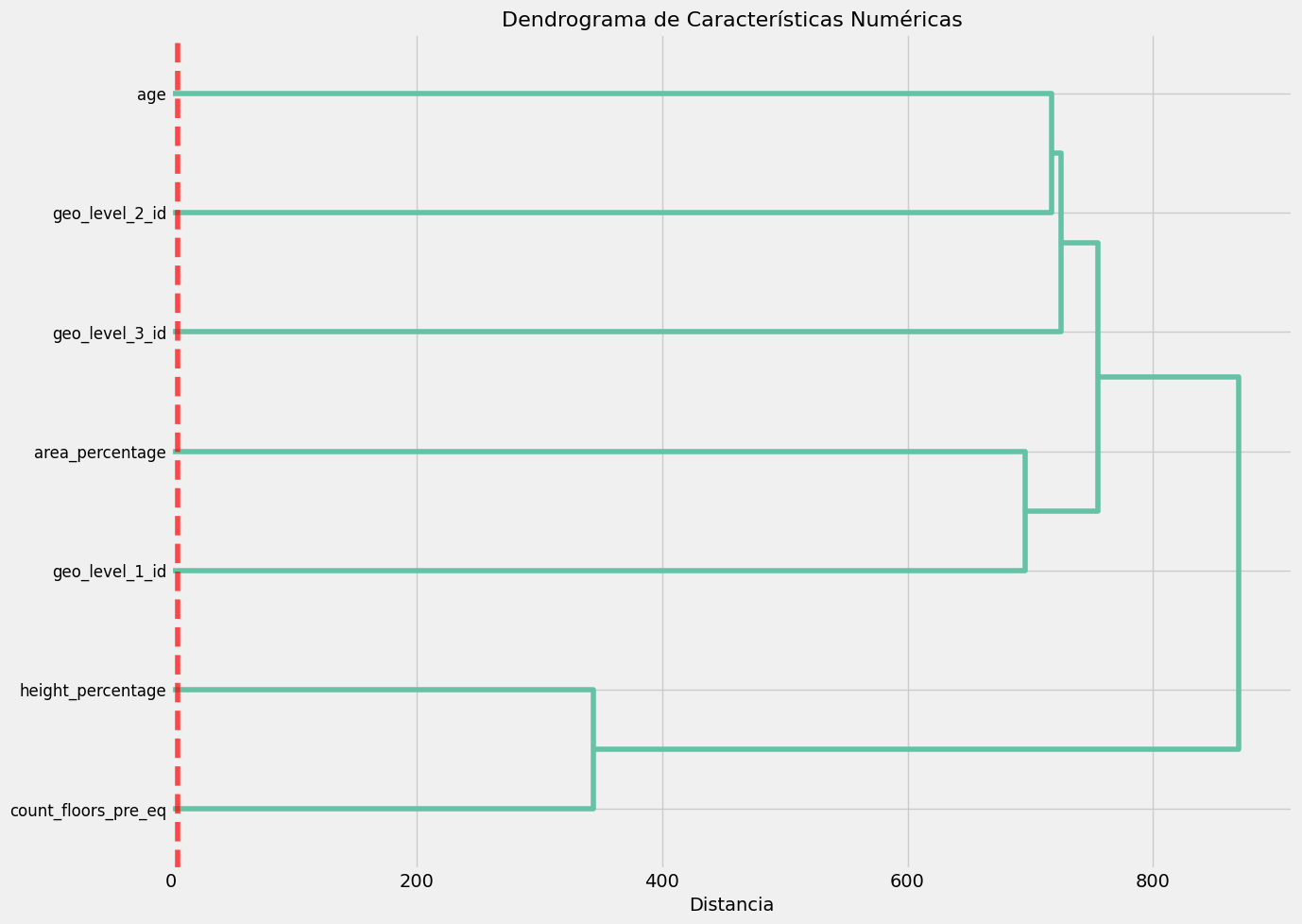
selected\_features += binary\_columns[:10]

print("Selected features for modeling:", selected\_features)

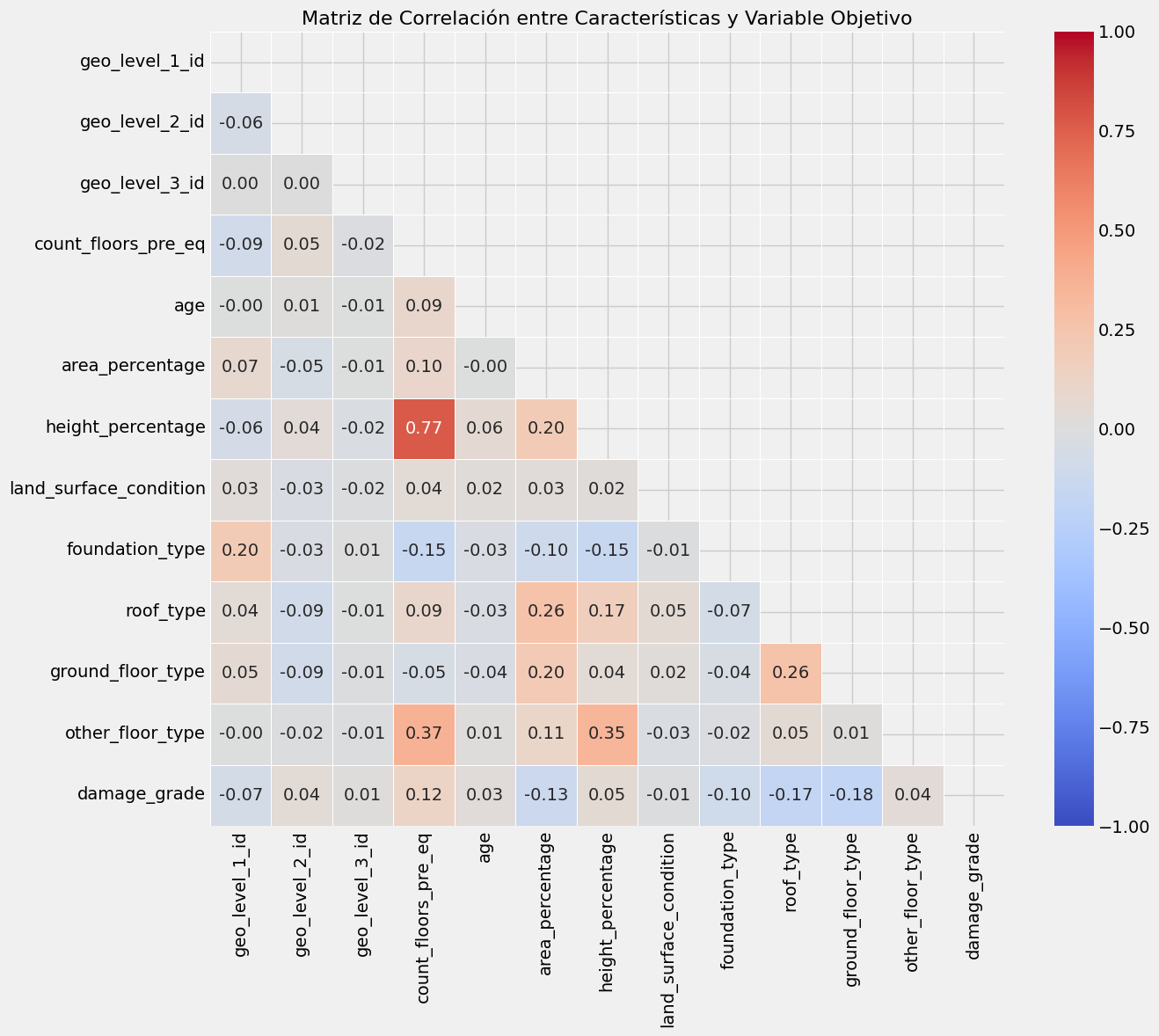
##### **2.4.2.2 Ejecución**

###### **2.4.2.2.1 Dendograma de características numéricas**

Este dendrograma es una representación gráfica del agrupamiento jerárquico de las características numéricas, mostrando cómo se agrupan o se dividen en función de su similitud.



###### **2.4.2.2.2 Matriz de correlación**



###### **2.4.2.2.3 Características seleccionadas**

###### Características seleccionadas para modelado: ['count\_floors\_pre\_eq', 'age', 'area\_percentage', 'height\_percentage', 'land\_surface\_condition', 'foundation\_type', 'roof\_type', 'ground\_floor\_type', 'other\_floor\_type', 'has\_superstructure\_adobe\_mud', 'has\_superstructure\_mud\_mortar\_stone', 'has\_superstructure\_stone\_flag', 'has\_superstructure\_cement\_mortar\_stone', 'has\_superstructure\_mud\_mortar\_brick', 'has\_superstructure\_cement\_mortar\_brick', 'has\_superstructure\_timber', 'has\_superstructure\_bamboo', 'has\_superstructure\_rc\_non\_engineered', 'has\_superstructure\_rc\_engineered']

### **2.5 Preprocesamiento de datos y selección de muestra**

Este apartado se encarga de preparar y preprocesar los datos para entrenar un modelo. Primero, elimina las columnas irrelevantes, luego define las columnas categóricas y numéricas. Utiliza una estrategia de muestreo avanzada para asegurarse de que la muestra sea representativa, tomando en cuenta la distribución geográfica, características estructurales y niveles de daño. Después, divide los datos en conjuntos de entrenamiento y prueba, aplica un preprocesamiento que escala las características numéricas y codifica las categóricas, y finalmente guarda el preprocesador para su uso posterior.

#### **2.5.1 Comando**

# Preprocessing the data

X = train\_data.drop(['building\_id', 'damage\_grade'], axis=1) # Drop irrelevant columns

y = train\_data['damage\_grade'] # Target variable

# Identify categorical and numerical columns

categorical\_cols = X.select\_dtypes(include='object').columns

numerical\_cols = X.select\_dtypes(exclude='object').columns

# Explanation of the sampling strategy

print("Sampling strategy:\n- Stratified sampling with geographic diversity and structural characteristics.")

# Advanced sampling function to ensure diverse and representative samples

def advanced\_sampling(df, y, sample\_size):

geo\_groups = df.groupby(['geo\_level\_1\_id', 'geo\_level\_2\_id'])

sampled\_indices = []

for name, group in geo\_groups:

group\_size = len(group)

group\_sample\_size = max(1, int(group\_size / len(df) \* sample\_size))

for damage\_level in [1, 2, 3]:

damage\_indices = group[y == damage\_level].index

if len(damage\_indices) > 0:

damage\_sample\_size = max(1, int(group\_sample\_size \* (sum(y[group.index] == damage\_level) / group\_size)))

sorted\_indices = df.loc[damage\_indices].sort\_values(by=['age', 'count\_floors\_pre\_eq', 'area\_percentage']).index[:damage\_sample\_size]

sampled\_indices.extend(sorted\_indices)

if len(sampled\_indices) < sample\_size:

remaining = sample\_size - len(sampled\_indices)

additional\_indices = df.sort\_values(by=['foundation\_type', 'roof\_type', 'height\_percentage']).index[:remaining]

sampled\_indices.extend(additional\_indices)

return df.loc[sampled\_indices], y.loc[sampled\_indices]

# Calculate the sample size (2% of the total dataset)

sample\_size = int(0.02 \* len(train\_data))

# Apply the advanced sampling method

X\_sampled, y\_sampled = advanced\_sampling(X, y, sample\_size)

# Check class distribution in the sampled data

plt.figure(figsize=(10, 6))

sns.countplot(x=y\_sampled, palette=['lightgreen', 'orange', 'red'])

plt.title('Class Distribution in the Selected Sample', fontsize=15)

plt.xticks([0, 1, 2], ['Low (1)', 'Medium (2)', 'High (3)'])

plt.tight\_layout()

plt.show()

# Split data into training and testing sets (80% train, 20% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_sampled, y\_sampled, test\_size=0.2, random\_state=42, stratify=y\_sampled)

# Display sizes of training and testing sets

print(f"Training set size: {X\_train.shape[0]} samples")

print(f"Test set size: {X\_test.shape[0]} samples")

# Define preprocessor for scaling and encoding

preprocessor = ColumnTransformer([

('num', StandardScaler(), numerical\_cols), # Scale numerical features

('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_cols) # One-hot encode categorical features

])

# Apply preprocessing to the training and test data

X\_train\_processed = preprocessor.fit\_transform(X\_train)

X\_test\_processed = preprocessor.transform(X\_test)

# Save preprocessor for future use

with open('preprocessor.pkl', 'wb') as file:

pickle.dump(preprocessor, file)

#### **2.5.2 Resultado**

##### 



### **2.6 Lazy Predict**

La instalación de esta librería se indica antes de los imports al inicio del porgrama.

Este código utiliza LazyPredict para comparar rápidamente múltiples modelos de clasificación sin necesidad de configuraciones complicadas. Ajusta varios modelos a los datos de entrenamiento y prueba, muestra los resultados de todos los modelos en términos de precisión y F1-Score, y visualiza los mejores 15 modelos según estos dos métricos, utilizando gráficos de barras para facilitar la comparación.

#### **2.6.1 Código**

# Run LazyPredict to quickly compare multiple models

clf = LazyClassifier(verbose=0, ignore\_warnings=True, custom\_metric=None)

# Fit models to the training and testing data

models, predictions = clf.fit(X\_train\_processed, X\_test\_processed, y\_train, y\_test)

# Display the results of all models

print("Model Comparison using LazyPredict:")

display(models)

# Visualizing the top 15 models by accuracy

plt.figure(figsize=(12, 8))

# Sort models by accuracy and select the top 15

models\_accuracy = models.sort\_values(by='Accuracy', ascending=False)[:15]

# Create a barplot for accuracy

sns.barplot(x=models\_accuracy.index, y=models\_accuracy['Accuracy'], palette='viridis')

plt.title('Top 15 Models by Accuracy', fontsize=15)

plt.xticks(rotation=90, fontsize=10) # Rotate labels for better readability

plt.ylabel('Accuracy', fontsize=12)

plt.tight\_layout()

plt.show()

# Visualizing the top 15 models by F1-Score (our main evaluation metric)

plt.figure(figsize=(12, 8))

# Sort models by F1-Score and select the top 15

models\_f1 = models.sort\_values(by='F1 Score', ascending=False)[:15]

# Create a barplot for F1-Score

sns.barplot(x=models\_f1.index, y=models\_f1['F1 Score'], palette='plasma')

plt.title('Top 15 Models by F1-Score', fontsize=15)

plt.xticks(rotation=90, fontsize=10) # Rotate labels for better readability

plt.ylabel('F1-Score', fontsize=12)

plt.tight\_layout()

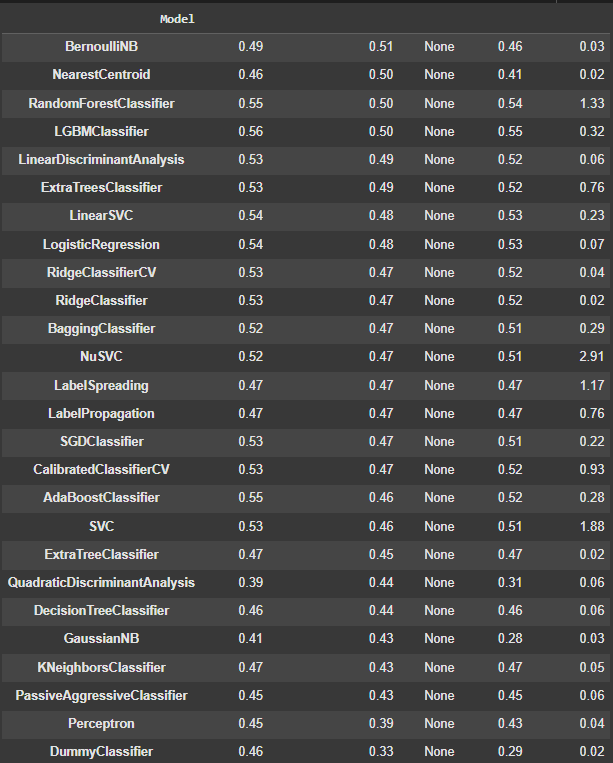
plt.show()

#### 

#### **2.6.2 Resultado**

##### **2.6.2.1 Tabla comparativa**

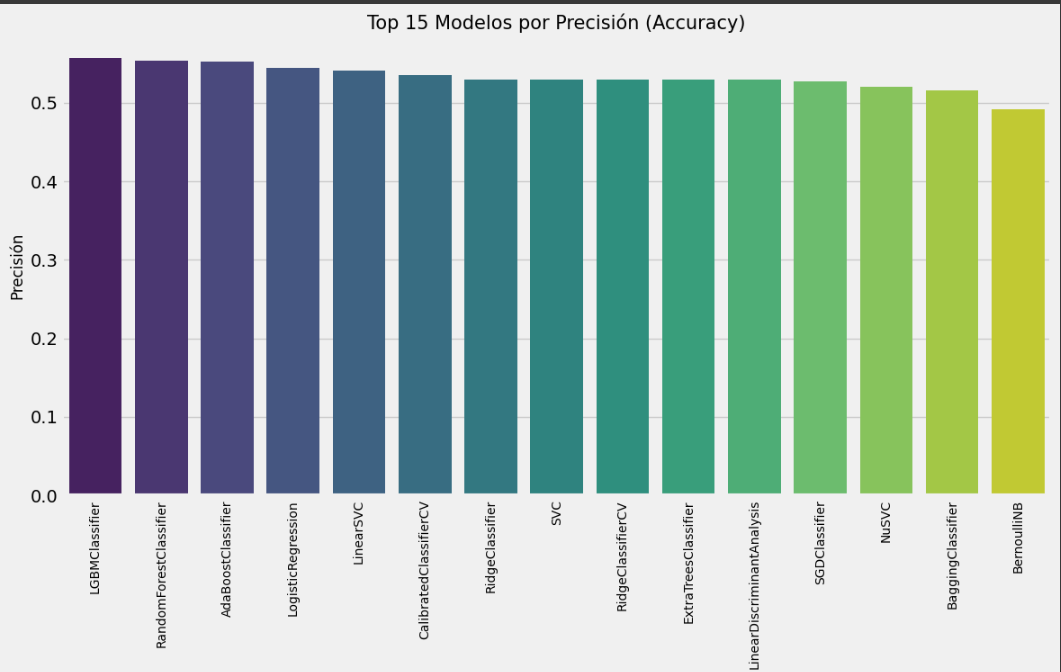
Tabla comparativa entre los modelos y sus precisiones:



##### **2.6.2.2 Comparativa visual**

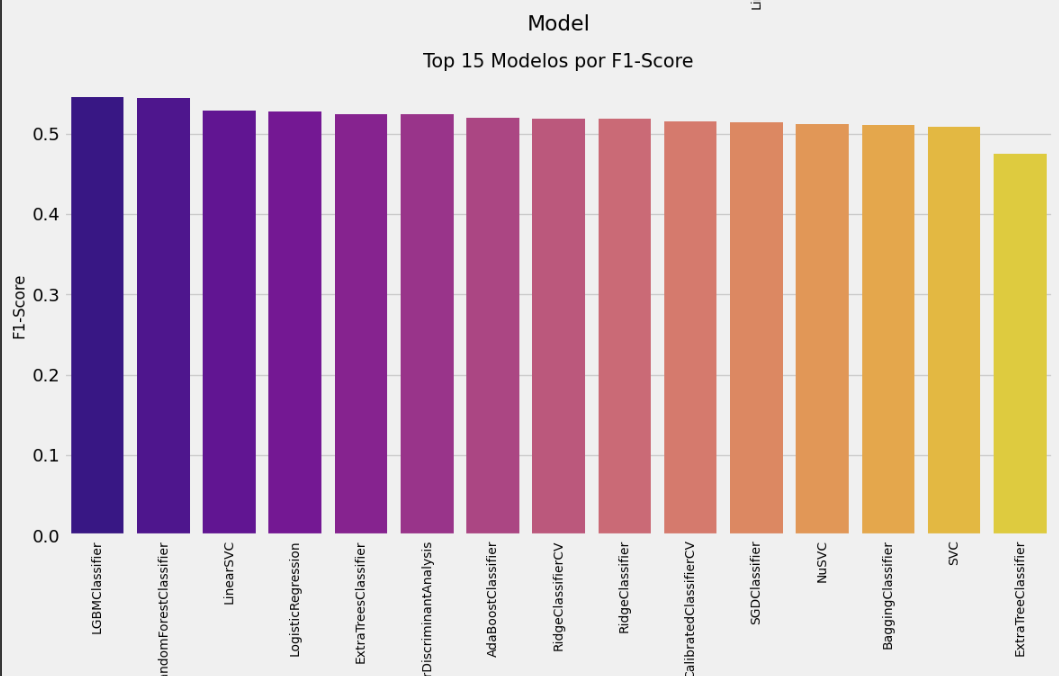
###### **2.6.2.2.1 Comparativa visual (Precisión)**

Comparativa en función a la precisión



###### **2.6.2.2.2 Comparativa visual (F1)**

Comparativa en función al F1-score (mide el balance entre la exactitud y la capacidad de detectar todas las instancias relevantes, es decir, la capacidad de predicción).



### **2.7 Modelos**

#### **2.7.1 Modelos de árbol**

##### **2.7.1.1 Randomforest**

###### **2.7.1.1.1 Código**

# Inicializamos el modelo de RandomForestClassifier

rf\_model = RandomForestClassifier(random\_state=42, n\_jobs=-1)

# Definimos los parámetros para la búsqueda de hiperparámetros (ajuste)

param\_dist\_rf = {

'n\_estimators': [100, 200, 300], # Número de árboles en el bosque

'max\_depth': [None, 10, 20, 30], # Profundidad máxima de los árboles

'min\_samples\_split': [2, 5, 10], # Número mínimo de muestras para dividir un nodo

'min\_samples\_leaf': [1, 2, 4], # Número mínimo de muestras en una hoja

'max\_features': ['sqrt', 'log2'] # Selección de características para la división de cada árbol

}

# Generar combinaciones aleatorias de parámetros

param\_list = list(ParameterSampler(param\_dist\_rf, n\_iter=20, random\_state=42))

# Inicializamos variables para almacenar el mejor resultado

best\_score = 0

best\_params = None

results = []

# Configuramos la validación cruzada con 3 pliegues

cv = StratifiedKFold(n\_splits=3, shuffle=True, random\_state=42)

# Bucle con barra de progreso para optimizar RandomForest

print("Iniciando optimización para RandomForestClassifier con visualización de progreso...")

for params in tqdm(param\_list, desc="Optimizando RandomForest"):

model = RandomForestClassifier(random\_state=42, n\_jobs=-1, \*\*params)

scores = []

# Realizamos validación cruzada manual

for train\_idx, val\_idx in cv.split(X\_train\_processed, y\_train):

# Dividimos los datos en entrenamiento y validación según los índices

if isinstance(X\_train\_processed, np.ndarray):

X\_fold\_train, X\_fold\_val = X\_train\_processed[train\_idx], X\_train\_processed[val\_idx]

else:

X\_fold\_train = X\_train\_processed[train\_idx]

X\_fold\_val = X\_train\_processed[val\_idx]

y\_fold\_train = y\_train.iloc[train\_idx]

y\_fold\_val = y\_train.iloc[val\_idx]

# Entrenamos y evaluamos el modelo

model.fit(X\_fold\_train, y\_fold\_train)

y\_pred = model.predict(X\_fold\_val)

score = f1\_score(y\_fold\_val, y\_pred, average='micro') # Usamos F1-score como métrica

scores.append(score)

# Calculamos el F1-score promedio para este conjunto de parámetros

mean\_score = np.mean(scores)

results.append((params, mean\_score))

# Si encontramos un modelo mejor, actualizamos los resultados

if mean\_score > best\_score:

best\_score = mean\_score

best\_params = params

print(f"\nNuevo mejor F1-score: {best\_score:.4f} con parámetros:")

for key, value in params.items():

print(f" {key}: {value}")

# Creamos el modelo final con los mejores parámetros encontrados

best\_rf = RandomForestClassifier(random\_state=42, n\_jobs=-1, \*\*best\_params)

best\_rf.fit(X\_train\_processed, y\_train)

# Información de entrenamiento

print("\nEntrenamiento completo.")

print(f"Mejores parámetros para RandomForestClassifier: {best\_params}")

print(f"Mejor F1-score en validación cruzada: {best\_score:.4f}")

# Evaluamos el modelo en el conjunto de prueba

y\_pred\_rf = best\_rf.predict(X\_test\_processed)

# Calculamos el F1-score en el conjunto de prueba

rf\_f1 = f1\_score(y\_test, y\_pred\_rf, average='micro')

print(f"F1-score (micro) en conjunto de prueba: {rf\_f1:.4f}")

# Visualizamos la matriz de confusión para evaluar el rendimiento del modelo

plt.figure(figsize=(10, 8))

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_rf)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Oranges',

xticklabels=['Bajo (1)', 'Medio (2)', 'Alto (3)'],

yticklabels=['Bajo (1)', 'Medio (2)', 'Alto (3)'])

plt.title('Matriz de Confusión - Random Forest', fontsize=15)

plt.ylabel('Clase Real', fontsize=12)

plt.xlabel('Clase Predicha', fontsize=12)

plt.tight\_layout()

plt.show()

# Mostrar el informe detallado de clasificación

print("Informe de clasificación - Random Forest:")

print(classification\_report(y\_test, y\_pred\_rf))

# Visualización de las características más importantes según el modelo

if hasattr(best\_rf, 'feature\_importances\_'):

# Obtener las importancias de las características

importances = best\_rf.feature\_importances\_

indices = np.argsort(importances)[-20:] # Top 20 características más importantes

# Graficar las 20 características más importantes

plt.figure(figsize=(12, 8))

plt.barh(range(len(indices)), importances[indices])

plt.yticks(range(len(indices)), [f'Feature {i}' for i in indices])

plt.title('Top 20 Características Importantes - Random Forest', fontsize=15)

plt.xlabel('Importancia', fontsize=12)

plt.tight\_layout()

plt.show()

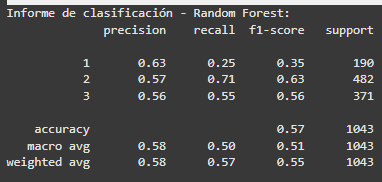
# Guardar el modelo entrenado para su uso posterior

with open('random\_forest\_model.pkl', 'wb') as file:

pickle.dump(best\_rf, file)

###### **2.7.1.1.2 Resultado**

Tabla de precisión y capacidad predictiva.



##### **2.7.1.2 BeggingClassifier**

###### **2.7.1.2.1 Código**

# First, we’ll set up our model using a DecisionTree as the base estimator inside a BaggingClassifier

from sklearn.tree import DecisionTreeClassifier

base\_estimator = DecisionTreeClassifier(random\_state=42)

bagging\_model = BaggingClassifier(estimator=base\_estimator, random\_state=42, n\_jobs=-1)

# Now, we define the parameters that we'll test to optimize the model.

param\_dist\_bagging = {

'n\_estimators': [10, 50, 100], # Number of base estimators (trees) in the bagging ensemble

'max\_samples': [0.5, 0.7, 1.0], # Fraction of samples to train each base estimator on

'max\_features': [0.5, 0.7, 1.0], # Fraction of features to use for each base estimator

'bootstrap': [True, False], # Whether or not to sample with replacement

'estimator\_\_max\_depth': [None, 10, 20], # Maximum depth of each tree (helps prevent overfitting)

'estimator\_\_min\_samples\_split': [2, 5, 10], # Minimum samples needed to split an internal node

'estimator\_\_min\_samples\_leaf': [1, 2, 4] # Minimum samples needed to be at a leaf node

}

# We generate random combinations of the above parameters to try out

param\_list = list(ParameterSampler(param\_dist\_bagging, n\_iter=20, random\_state=42))

# We will store the results of each parameter combination here

best\_score = 0

best\_params = None

results = []

# Set up cross-validation with 3 splits

cv = StratifiedKFold(n\_splits=3, shuffle=True, random\_state=42)

# We’ll now run a loop to optimize the BaggingClassifier model

print("Starting optimization for BaggingClassifier... (you’ll see the progress here!)")

for params in tqdm(param\_list, desc="Optimizing BaggingClassifier"):

# We separate parameters for the base estimator and the BaggingClassifier itself

estimator\_params = {}

bagging\_params = {}

for key, value in params.items():

if key.startswith('estimator\_\_'):

# Extract the parameter name without the 'estimator\_\_' prefix

param\_name = key.replace('estimator\_\_', '')

estimator\_params[param\_name] = value

else:

bagging\_params[key] = value

# Now, create the base estimator (decision tree) with the extracted parameters

base\_est = DecisionTreeClassifier(random\_state=42, \*\*estimator\_params)

# Create the BaggingClassifier with the base estimator and other parameters

model = BaggingClassifier(estimator=base\_est, random\_state=42, n\_jobs=-1, \*\*bagging\_params)

scores = []

# Perform manual cross-validation

for train\_idx, val\_idx in cv.split(X\_train\_processed, y\_train):

# Extract the data for this fold

X\_fold\_train, X\_fold\_val = X\_train\_processed[train\_idx], X\_train\_processed[val\_idx]

y\_fold\_train, y\_fold\_val = y\_train.iloc[train\_idx], y\_train.iloc[val\_idx]

# Train the model on the fold's training data and make predictions

model.fit(X\_fold\_train, y\_fold\_train)

y\_pred = model.predict(X\_fold\_val)

# Calculate the F1-score for the fold

score = f1\_score(y\_fold\_val, y\_pred, average='micro')

scores.append(score)

# Calculate the average F1-score across all folds

mean\_score = np.mean(scores)

results.append((params, mean\_score))

# If this model has the best score so far, we save it

if mean\_score > best\_score:

best\_score = mean\_score

best\_params = params

print(f"\nNew best F1-score: {best\_score:.4f} with these parameters:")

for key, value in params.items():

print(f" {key}: {value}")

# After testing all the parameter combinations, we'll create the final model using the best parameters

# Separate the best parameters for the base estimator and BaggingClassifier

estimator\_params = {}

bagging\_params = {}

for key, value in best\_params.items():

if key.startswith('estimator\_\_'):

param\_name = key.replace('estimator\_\_', '')

estimator\_params[param\_name] = value

else:

bagging\_params[key] = value

# Create the final base estimator with the best parameters

best\_base\_estimator = DecisionTreeClassifier(random\_state=42, \*\*estimator\_params)

# Create the final BaggingClassifier model

best\_bagging = BaggingClassifier(

estimator=best\_base\_estimator,

random\_state=42,

n\_jobs=-1,

\*\*bagging\_params

)

# Train the final model on the full training set

best\_bagging.fit(X\_train\_processed, y\_train)

# Output the details about the best model and its parameters

print("\nTraining complete.")

print("Best parameters for BaggingClassifier:")

print("Base estimator parameters:")

for key, value in estimator\_params.items():

print(f" {key}: {value}")

print("Bagging parameters:")

for key, value in bagging\_params.items():

print(f" {key}: {value}")

print(f"Best F1-score in cross-validation: {best\_score:.4f}")

# Evaluate the trained model on the test set

y\_pred\_bagging = best\_bagging.predict(X\_test\_processed)

# Calculate and print the F1-score on the test set

bagging\_f1 = f1\_score(y\_test, y\_pred\_bagging, average='micro')

print(f"F1-score (micro) on test set: {bagging\_f1:.4f}")

# Visualize the confusion matrix to see how well the model performed on each class

plt.figure(figsize=(10, 8))

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_bagging)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Greens',

xticklabels=['Low (1)', 'Medium (2)', 'High (3)'],

yticklabels=['Low (1)', 'Medium (2)', 'High (3)'])

plt.title('Confusion Matrix - BaggingClassifier', fontsize=15)

plt.ylabel('True Class', fontsize=12)

plt.xlabel('Predicted Class', fontsize=12)

plt.tight\_layout()

plt.show()

# Print a detailed classification report for further analysis

print("Classification report - BaggingClassifier:")

print(classification\_report(y\_test, y\_pred\_bagging))

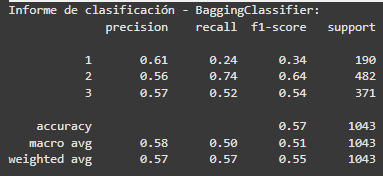
# Save the trained model to a file for future use

with open('bagging\_model.pkl', 'wb') as file:

pickle.dump(best\_bagging, file)

###### **2.7.1.2.2 Resultado**

Tabla de precisión y capacidad predictiva.



##### **2.7.1.3 LGBMClassifier**

###### **2.7.1.3.1 Código**

# Setting up the initial model with a focus on high precision

lgbm\_model = LGBMClassifier(random\_state=42, n\_jobs=-1)

# Define the parameter grid with a focus on general accuracy

param\_dist\_lgbm = {

'n\_estimators': [300, 500, 700, 1000], # More trees for stability

'learning\_rate': [0.01, 0.05, 0.1], # Varying learning rates

'max\_depth': [7, 9, 11], # Moderate depths

'num\_leaves': [31, 63, 127], # Different leaf configurations

'min\_child\_samples': [20, 50, 100], # Higher values to prevent overfitting

'subsample': [0.8, 0.9, 1.0], # Complete sampling to avoid bias

'colsample\_bytree': [0.8, 0.9, 1.0], # Feature sampling options

'min\_split\_gain': [0.0, 0.01], # Control split gains

'reg\_alpha': [0.0, 0.1, 1.0], # Stronger L1 regularization

'reg\_lambda': [0.0, 0.1, 1.0], # Stronger L2 regularization

'boosting': ['gbdt', 'dart'], # Different boosting algorithms to try

'verbose': [-1] # Suppress verbosity

}

# Create random combinations of these parameters for optimization

param\_list = list(ParameterSampler(param\_dist\_lgbm, n\_iter=30, random\_state=42))

# Variables to track the best performance

best\_accuracy = 0

best\_params = None

results = []

# Setting up cross-validation

cv = StratifiedKFold(n\_splits=3, shuffle=True, random\_state=42)

# Starting the optimization loop

print("Starting optimization for maximum accuracy...")

for params in tqdm(param\_list, desc="Optimizing LGBMClassifier for accuracy"):

model = LGBMClassifier(random\_state=42, n\_jobs=-1, \*\*params)

accuracies = []

f1\_scores = []

# Perform manual cross-validation

for train\_idx, val\_idx in cv.split(X\_train\_processed, y\_train):

# Get training and validation data for this fold

X\_fold\_train, X\_fold\_val = X\_train\_processed[train\_idx], X\_train\_processed[val\_idx]

y\_fold\_train, y\_fold\_val = y\_train.iloc[train\_idx], y\_train.iloc[val\_idx]

# Train the model and evaluate performance

model.fit(X\_fold\_train, y\_fold\_train)

y\_pred = model.predict(X\_fold\_val)

# Calculate accuracy and F1-score for this fold

acc = accuracy\_score(y\_fold\_val, y\_pred)

f1 = f1\_score(y\_fold\_val, y\_pred, average='micro')

accuracies.append(acc)

f1\_scores.append(f1)

# Calculate average metrics across all folds

mean\_accuracy = np.mean(accuracies)

mean\_f1 = np.mean(f1\_scores)

results.append((params, mean\_accuracy, mean\_f1))

# Update the best model if this one is better

if mean\_accuracy > best\_accuracy:

best\_accuracy = mean\_accuracy

best\_params = params

print(f"\nNew accuracy record: {best\_accuracy:.4f} with parameters:")

for key, value in params.items():

if key != 'verbose': # Skip parameters not relevant to output

print(f" {key}: {value}")

print(f"Associated F1-score: {mean\_f1:.4f}")

# Create the best model using the optimal parameters

best\_lgbm = LGBMClassifier(random\_state=42, n\_jobs=-1, \*\*best\_params)

# Train the final model on the entire training dataset

print("\nTraining the final model with the best parameters...")

best\_lgbm.fit(X\_train\_processed, y\_train)

print("\nTraining complete.")

print(f"Best parameters for maximum accuracy: {best\_params}")

print(f"Best accuracy in cross-validation: {best\_accuracy:.4f}")

# Evaluate the final model on the test set

y\_pred\_lgbm = best\_lgbm.predict(X\_test\_processed)

# Print detailed evaluation metrics

accuracy = accuracy\_score(y\_test, y\_pred\_lgbm)

lgbm\_f1 = f1\_score(y\_test, y\_pred\_lgbm, average='micro')

lgbm\_f1\_per\_class = f1\_score(y\_test, y\_pred\_lgbm, average=None)

print(f"\nTest set results:")

print(f"Accuracy: {accuracy:.4f}")

print(f"F1-score (micro): {lgbm\_f1:.4f}")

print(f"F1-score per class: Class 1: {lgbm\_f1\_per\_class[0]:.4f}, Class 2: {lgbm\_f1\_per\_class[1]:.4f}, Class 3: {lgbm\_f1\_per\_class[2]:.4f}")

# Plot confusion matrix for better understanding of misclassifications

plt.figure(figsize=(10, 8))

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_lgbm)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues',

xticklabels=['Low (1)', 'Medium (2)', 'High (3)'],

yticklabels=['Low (1)', 'Medium (2)', 'High (3)'])

plt.title('Confusion Matrix - LGBMClassifier Maximum Accuracy', fontsize=15)

plt.ylabel('True Class', fontsize=12)

plt.xlabel('Predicted Class', fontsize=12)

plt.tight\_layout()

plt.show()

# Detailed classification report

print("\nClassification report - Optimized LGBMClassifier for Accuracy:")

print(classification\_report(y\_test, y\_pred\_lgbm))

# Learning curve for further analysis (optional)

if 'n\_estimators' in best\_params:

n\_estimators = best\_params['n\_estimators']

learning\_rates = [0.01, 0.05, 0.1, 0.2]

plt.figure(figsize=(12, 8))

for lr in learning\_rates:

eval\_set = [(X\_test\_processed, y\_test)]

model = LGBMClassifier(

n\_estimators=n\_estimators,

learning\_rate=lr,

random\_state=42,

n\_jobs=-1,

verbose=-1

)

model.fit(X\_train\_processed, y\_train,

eval\_set=eval\_set,

eval\_metric='multi\_logloss') # Suppress verbosity during fit()

results = model.evals\_result\_['valid\_0']['multi\_logloss']

plt.plot(range(1, len(results) + 1), results, label=f'learning\_rate={lr}')

plt.xlabel('Number of Trees')

plt.ylabel('Log Loss')

plt.title('Effect of Learning Rate on Model Performance')

plt.legend()

plt.grid(True)

plt.show()

# Feature importance analysis to understand model behavior

plt.figure(figsize=(12, 8))

if hasattr(best\_lgbm, 'feature\_importances\_'):

importances = best\_lgbm.feature\_importances\_

indices = np.argsort(importances)[-20:] # Top 20 features

plt.barh(range(len(indices)), importances[indices])

plt.yticks(range(len(indices)), [f'Feature {i}' for i in indices])

plt.title('Top 20 Important Features - High Precision Model', fontsize=15)

plt.xlabel('Importance', fontsize=12)

plt.tight\_layout()

plt.show()

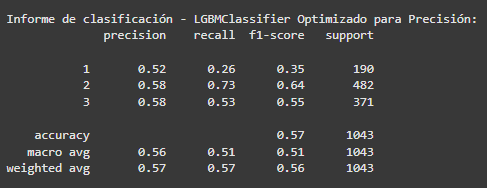
# Save the final high-precision model for future use

with open('lgbm\_model\_high\_precision.pkl', 'wb') as file:

pickle.dump(best\_lgbm, file)

###### **2.7.1.3.2 Resultado**

Tabla de precisión y capacidad predictiva.



#### **2.7.4 SVG**

##### **2.7.4.1 Código**

# Let's start by setting up the SVC (Support Vector Classifier) model with the option to output probabilities

svm\_model = SVC(probability=True, random\_state=42)

# We'll define the hyperparameters we want to test for the model

param\_dist\_svm = {

'C': [0.1, 1, 10], # This controls how strictly we separate the classes (regularization)

'kernel': ['linear', 'rbf'], # The type of decision boundary we want (linear or more flexible 'rbf')

'gamma': ['scale', 'auto', 0.1] # Controls how much influence each training point has on the decision boundary

}

# If we have more than 5000 samples, we'll use a smaller subset to speed up training

if X\_train\_processed.shape[0] > 5000:

from sklearn.model\_selection import train\_test\_split

# Take a random sample of 5000 samples to train the model (just for quicker experimentation)

X\_train\_svm, \_, y\_train\_svm, \_ = train\_test\_split(

X\_train\_processed, y\_train,

train\_size=5000, # Limit to 5000 samples

random\_state=42,

stratify=y\_train # Make sure the classes are proportionally represented in the sample

)

print(f"Using a subset of {X\_train\_svm.shape[0]} samples to train the SVM")

else:

# If there aren't many samples, just use all of them

X\_train\_svm = X\_train\_processed

y\_train\_svm = y\_train

# Generate random combinations of the hyperparameters to explore

param\_list = list(ParameterSampler(param\_dist\_svm, n\_iter=10, random\_state=42))

# Variables to keep track of the best model we've found

best\_score = 0

best\_params = None

results = []

# We'll use cross-validation to test the model's performance

cv = StratifiedKFold(n\_splits=3, shuffle=True, random\_state=42)

# This loop will try different combinations of hyperparameters and evaluate them

print("Starting optimization for SVC with progress visualization...")

for params in tqdm(param\_list, desc="Optimizing SVC"):

model = SVC(probability=True, random\_state=42, \*\*params)

scores = []

# Train and evaluate the model on different folds of the data

for train\_idx, val\_idx in cv.split(X\_train\_svm, y\_train\_svm):

# Split the data into training and validation sets for this fold

X\_fold\_train, X\_fold\_val = X\_train\_svm[train\_idx], X\_train\_svm[val\_idx]

y\_fold\_train, y\_fold\_val = y\_train\_svm.iloc[train\_idx], y\_train\_svm.iloc[val\_idx]

# Train the model on the training fold and make predictions on the validation fold

model.fit(X\_fold\_train, y\_fold\_train)

y\_pred = model.predict(X\_fold\_val)

# Calculate the F1-score for this fold

score = f1\_score(y\_fold\_val, y\_pred, average='micro')

scores.append(score)

# Calculate the average F1-score across all folds

mean\_score = np.mean(scores)

results.append((params, mean\_score))

# If we found a better model, keep track of it

if mean\_score > best\_score:

best\_score = mean\_score

best\_params = params

print(f"\nNew best F1-score: {best\_score:.4f} with parameters:")

for key, value in params.items():

print(f" {key}: {value}")

# Now that we've found the best parameters, let's train the model with the full training set

best\_svm = SVC(probability=True, random\_state=42, \*\*best\_params)

print("\nTraining the final SVC model with the full dataset...")

best\_svm.fit(X\_train\_processed, y\_train)

# Output some details about the model training

print("\nTraining complete.")

print(f"Best parameters for SVC: {best\_params}")

print(f"Best F1-score in cross-validation: {best\_score:.4f}")

# Evaluate the trained model on the test set to see how it performs on unseen data

y\_pred\_svm = best\_svm.predict(X\_test\_processed)

# Calculate the F1-score for the test set predictions

svm\_f1 = f1\_score(y\_test, y\_pred\_svm, average='micro')

print(f"F1-score (micro) on the test set: {svm\_f1:.4f}")

# Visualize how well the model predicted each class using a confusion matrix

plt.figure(figsize=(10, 8))

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_svm)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Purples',

xticklabels=['Low (1)', 'Medium (2)', 'High (3)'],

yticklabels=['Low (1)', 'Medium (2)', 'High (3)'])

plt.title('Confusion Matrix - SVM', fontsize=15)

plt.ylabel('True Class', fontsize=12)

plt.xlabel('Predicted Class', fontsize=12)

plt.tight\_layout()

plt.show()

# Print a detailed classification report (Precision, Recall, F1-score for each class)

print("Classification report - SVM:")

print(classification\_report(y\_test, y\_pred\_svm))

# Save the trained model for later use

with open('svm\_model.pkl', 'wb') as file:

pickle.dump(best\_svm, file)

##### **2.7.4.2 Resultado**

Tabla de precisión y capacidad predictiva.

##### 

#### **2.7.5 Comparación de los modelos**

##### **2.7.5.1 Código**

# Definir F1-score de LGBMClassifier con RandomizedSearchCV

lgbm\_randomized\_f1 = 0.7198

# Recopilar métricas de los modelos seleccionados

model\_names = ['LGBMClassifier (GridSearch)', 'LGBMClassifier (RandomizedSearch)', 'RandomForest', 'SVM']

f1\_scores\_test = [lgbm\_f1, lgbm\_randomized\_f1, rf\_f1, svm\_f1]

# Crear DataFrame para comparación visual

comparison\_df = pd.DataFrame({

'Modelo': model\_names,

'F1-Score (Test)': f1\_scores\_test,

})

# Mostrar tabla de comparación

print("Comparación de Modelos por F1-Score:")

display(comparison\_df.sort\_values(by='F1-Score (Test)', ascending=False))

# Visualización de comparación de F1-Scores

plt.figure(figsize=(12, 6))

sns.barplot(x='Modelo', y='F1-Score (Test)', data=comparison\_df.sort\_values(by='F1-Score (Test)', ascending=False), palette='viridis')

plt.title('Comparación de Modelos por F1-Score', fontsize=15)

plt.ylabel('F1-Score', fontsize=12)

plt.xticks(rotation=15, ha='right')

plt.ylim(min(f1\_scores\_test) - 0.05, 1.0) # Ajustar el límite inferior para mejor visualización

plt.grid(axis='y', linestyle='--', alpha=0.7)

# Agregar etiquetas de valor sobre las barras

for i, model in enumerate(comparison\_df.sort\_values(by='F1-Score (Test)', ascending=False)['Modelo']):

idx = model\_names.index(model)

plt.text(i, f1\_scores\_test[idx] + 0.01, f'{f1\_scores\_test[idx]:.4f}', ha='center', fontsize=9)

plt.tight\_layout()

plt.show()

# Determinar el mejor modelo basado en el conjunto de test

best\_model\_idx = f1\_scores\_test.index(max(f1\_scores\_test))

best\_model\_name = model\_names[best\_model\_idx]

print(f"El mejor modelo es: {best\_model\_name} con F1-Score de {max(f1\_scores\_test):.4f}")

# Comparación específica entre implementaciones de LGBMClassifier (GridSearch vs RandomizedSearch)

print("\nComparación entre implementaciones de LGBMClassifier:")

lgbm\_comparison = comparison\_df[comparison\_df['Modelo'].str.contains('LGBMClassifier')]

display(lgbm\_comparison)

# Visualización comparativa de F1-Score por clase entre GridSearch y RandomizedSearch

lgbm\_grid\_f1\_classes = [0.5013, 0.7758, 0.6845]

lgbm\_random\_f1\_classes = [0.49, 0.78, 0.69]

# Crear DataFrame con comparación por clase

class\_comparison = pd.DataFrame({

'Clase': ['Bajo (1)', 'Medio (2)', 'Alto (3)'] \* 2,

'Modelo': ['GridSearch'] \* 3 + ['RandomizedSearch'] \* 3,

'F1\_Score': lgbm\_grid\_f1\_classes + lgbm\_random\_f1\_classes

})

# Visualización de F1-Score por clase

plt.figure(figsize=(12, 7))

sns.barplot(x='Clase', y='F1\_Score', hue='Modelo', data=class\_comparison, palette=['#2C7FB8', '#7FBC41'])

plt.title('Comparación de F1-Score por Clase: GridSearch vs RandomizedSearch', fontsize=15)

plt.ylabel('F1-Score', fontsize=12)

plt.xlabel('Nivel de Daño', fontsize=12)

plt.ylim(0.4, 0.8) # Ajuste de límite superior para centrar la visualización

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.legend(title='Enfoque de Optimización')

# Agregar etiquetas de valor sobre las barras

for i, row in enumerate(class\_comparison.itertuples()):

plt.text(i % 3 - 0.2 + (i // 3) \* 0.4, row.F1\_Score + 0.01, f'{row.F1\_Score:.3f}',

ha='center', fontsize=9, fontweight='bold')

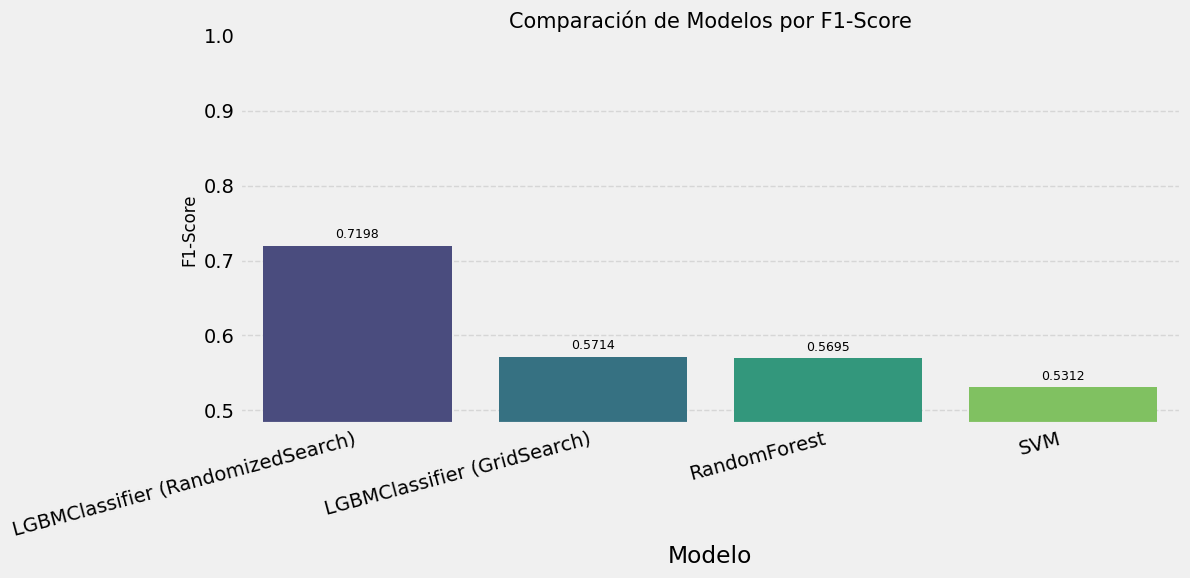
plt.tight\_layout()

plt.show()

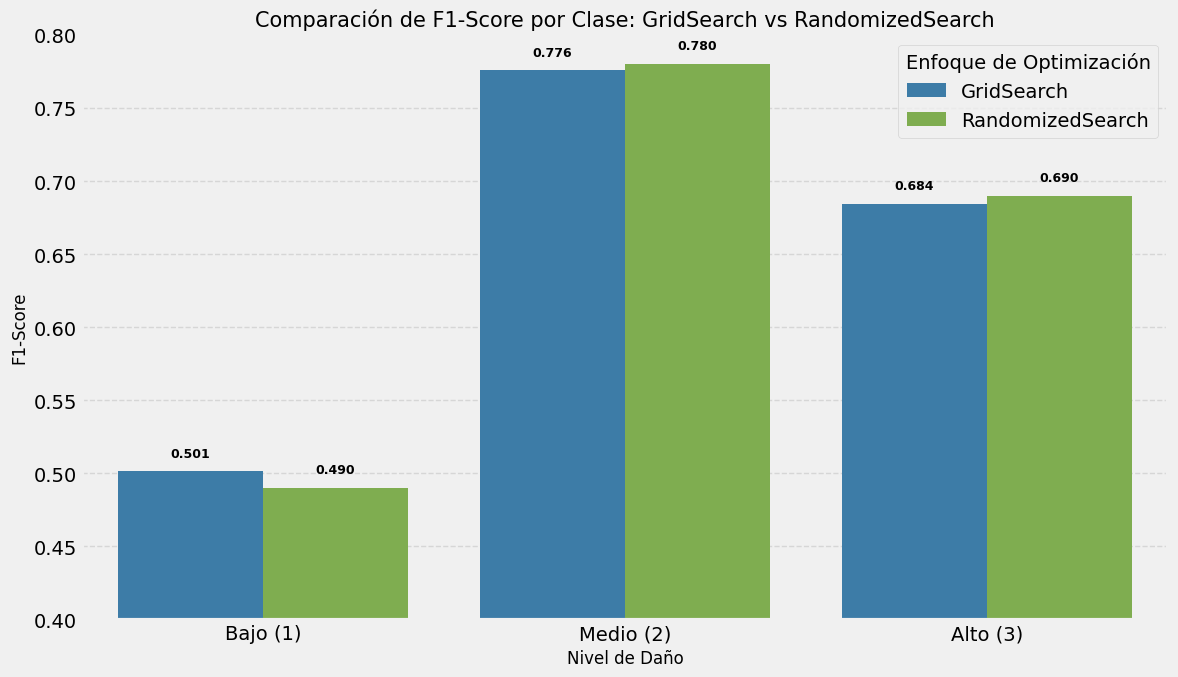
##### **2.7.5.2 Resultado**

###### **2.7.5.2.1 Modelos F1**

Resultados de la comparación entre los modelos F1 juntos a los modelos árbol y svm.



###### **2.7.5.2.2 Comparación Gridsearch vs randomizedSearch**

Esta tabla atiende a si mejora si añado gridsearch o randomizeSearch

#### **2.7.6 Aplicación de RandomizedSearchCV para LGBMClassifier**

##### **2.7.6.1 Código**

# Ensure the best model name is defined

if 'best\_model\_name' not in globals():

raise ValueError("The 'best\_model\_name' variable is not defined.")

print(f"Starting final optimization for the best model: {best\_model\_name}")

# Define parameters based on the best model selected

if best\_model\_name == 'LGBMClassifier':

model\_class = LGBMClassifier

final\_param\_dist = {

'n\_estimators': [200, 300, 500, 700],

'learning\_rate': [0.01, 0.03, 0.05, 0.07],

'max\_depth': [7, 9, 11, 15],

'num\_leaves': [31, 63, 127],

'min\_child\_samples': [10, 20, 30],

'subsample': [0.7, 0.8, 0.9],

'colsample\_bytree': [0.7, 0.8, 0.9],

'reg\_alpha': [0, 0.1, 0.5],

'reg\_lambda': [0, 0.1, 0.5]

}

base\_params = {'random\_state': 42, 'n\_jobs': -1}

elif best\_model\_name == 'BaggingClassifier':

model\_class = BaggingClassifier

base\_est\_params = {

'max\_depth': [10, 20, 30, None],

'min\_samples\_split': [2, 3, 5],

'min\_samples\_leaf': [1, 2, 4]

}

final\_param\_dist = {

'n\_estimators': [50, 100, 200, 300],

'max\_samples': [0.5, 0.7, 0.8, 1.0],

'max\_features': [0.5, 0.7, 0.8, 1.0],

'bootstrap': [True, False]

}

# Add base estimator parameters to final param distribution

for param, values in base\_est\_params.items():

final\_param\_dist[f'base\_estimator\_\_{param}'] = values

base\_params = {'base\_estimator': DecisionTreeClassifier(random\_state=42), 'random\_state': 42, 'n\_jobs': -1}

elif best\_model\_name == 'RandomForest':

model\_class = RandomForestClassifier

final\_param\_dist = {

'n\_estimators': [200, 300, 400, 500],

'max\_depth': [15, 20, 30, None],

'min\_samples\_split': [2, 3, 5, 7],

'min\_samples\_leaf': [1, 2, 3, 4],

'max\_features': ['sqrt', 'log2'],

'bootstrap': [True, False],

'class\_weight': [None, 'balanced', 'balanced\_subsample']

}

base\_params = {'random\_state': 42, 'n\_jobs': -1}

else: # For SVM

model\_class = SVC

final\_param\_dist = {

'C': [0.1, 0.5, 1, 5, 10],

'kernel': ['linear', 'rbf', 'poly'],

'gamma': ['scale', 'auto', 0.01, 0.1, 1],

'class\_weight': [None, 'balanced']

}

base\_params = {'probability': True, 'random\_state': 42}

# Split 20% of the data for training

X\_train\_final, X\_unused, y\_train\_final, y\_unused = train\_test\_split(

X\_train\_processed, y\_train, test\_size=0.8, stratify=y\_train, random\_state=42

)

# 3-fold cross-validation within the 20% training set

cv = StratifiedKFold(n\_splits=3, shuffle=True, random\_state=42)

print(f"Using only 20% ({len(X\_train\_final)} samples) for training and validating on 1/3 of this set.")

# Generate parameter combinations for hyperparameter tuning

param\_list = list(ParameterSampler(final\_param\_dist, n\_iter=30, random\_state=42))

best\_score\_final = 0

best\_params\_final = None

print(f"Starting final optimization for {best\_model\_name} with {len(param\_list)} combinations...")

# Perform the search across the parameters

for params in tqdm(param\_list, desc=f"Optimizing {best\_model\_name}"):

model = model\_class(\*\*base\_params, \*\*params)

scores = []

# Cross-validation loop

for train\_idx, val\_idx in cv.split(X\_train\_final, y\_train\_final):

X\_fold\_train = X\_train\_final.iloc[train\_idx] if hasattr(X\_train\_final, 'iloc') else X\_train\_final[train\_idx]

X\_fold\_val = X\_train\_final.iloc[val\_idx] if hasattr(X\_train\_final, 'iloc') else X\_train\_final[val\_idx]

y\_fold\_train = y\_train\_final.iloc[train\_idx] if hasattr(y\_train\_final, 'iloc') else y\_train\_final[train\_idx]

y\_fold\_val = y\_train\_final.iloc[val\_idx] if hasattr(y\_train\_final, 'iloc') else y\_train\_final[val\_idx]

# Train model and evaluate

model.fit(X\_fold\_train, y\_fold\_train)

y\_pred = model.predict(X\_fold\_val)

scores.append(f1\_score(y\_fold\_val, y\_pred, average='micro'))

mean\_score = np.mean(scores)

if mean\_score > best\_score\_final:

best\_score\_final = mean\_score

best\_params\_final = params

print(f"\nNew best F1-score: {best\_score\_final:.4f} with parameters: {params}")

# Train final model with the 20% of data

final\_model = model\_class(\*\*base\_params, \*\*best\_params\_final)

final\_model.fit(X\_train\_final, y\_train\_final)

print(f"Best F1-score from cross-validation: {best\_score\_final:.4f}")

y\_pred\_final = final\_model.predict(X\_test\_processed)

final\_f1 = f1\_score(y\_test, y\_pred\_final, average='micro')

print(f"Final F1-score on test set: {final\_f1:.4f}")

# Plot confusion matrix

plt.figure(figsize=(10, 8))

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_final)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='YlGnBu')

plt.title(f'Final Confusion Matrix - {best\_model\_name}', fontsize=15)

plt.ylabel('True Class', fontsize=12)

plt.xlabel('Predicted Class', fontsize=12)

plt.tight\_layout()

plt.show()

# Print detailed classification report

print(classification\_report(y\_test, y\_pred\_final))

# Save the trained model

if not os.path.exists("models"):

os.makedirs("models")

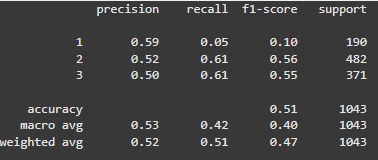
model\_path = os.path.join("models", "final\_optimized\_model.pkl")

with open(model\_path, 'wb') as file:

pickle.dump(final\_model, file)

print(f"Model saved at {model\_path}")

##### **2.7.6.2 Resultado**



### **2.8 Predicción y csv**

Esta parte carga un modelo entrenado y preprocesa los datos de prueba para generar predicciones. Luego, crea un archivo CSV con los resultados y, si está en Colab, permite descargar el archivo. Finalmente, visualiza la distribución de las predicciones en un gráfico y muestra un mensaje indicando que el proceso ha finalizado.

#### **2.8.1 Código**

# Check if we are in Google Colab to enable file download

try:

from google.colab import files

is\_colab = True # If in Google Colab, set flag to True

except ImportError:

is\_colab = False # If not in Colab, set flag to False

# Print current date and time, and user information

print(f"Current date and time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")

print(f"Current user: Saultr21")

print("\n=== GENERATING PREDICTIONS FOR SUBMISSION ===\n")

# Load the final optimized model (or use the one already in memory)

try:

with open('final\_optimized\_model.pkl', 'rb') as file:

final\_model = pickle.load(file) # Load model from file

print("Final model loaded successfully")

except:

print("Using the final model already in memory")

# Load test data for predictions

test\_values\_url = "https://raw.githubusercontent.com/AdrianYArmas/IaBigData/refs/heads/main/SNS/3%20%20-%20Algoritmos%20y%20herramientas%20para%20el%20aprendizaje%20supervisado%20/3.7%20%20Predicci%C3%B3n%20de%20Riesgo%20de%20derrumbamiento\_Terremotos/dataset/test\_values.csv"

test\_values = pd.read\_csv(test\_values\_url) # Read test data from URL

print(f"Test data loaded: {test\_values.shape} records")

# Save the building IDs for submission

test\_building\_ids = test\_values['building\_id'].values

# Preprocess the test data before making predictions

print("Preprocessing test data...")

X\_test\_submission = preprocessor.transform(test\_values) # Assuming preprocessor is already defined

print(f"Test data preprocessed successfully")

# Generate predictions (with progress bar for large files)

print("Generating predictions...")

test\_predictions = final\_model.predict(X\_test\_submission) # Use model to predict

print(f"Predictions generated for {len(test\_predictions)} buildings")

# Create a DataFrame for submission

submission\_df = pd.DataFrame({

'building\_id': test\_building\_ids, # Use building IDs from the test data

'damage\_grade': test\_predictions # Predicted damage grades

})

# Verify the submission file is saved correctly

submission\_file = 'submission.csv'

submission\_df.to\_csv(submission\_file, index=False) # Save DataFrame to CSV

print(f"Submission file generated: {submission\_file}")

# Check if the file exists after saving

if os.path.exists(submission\_file):

print(f"The file '{submission\_file}' has been saved successfully.")

else:

print(f"There was an issue saving the file '{submission\_file}'.")

# If running in Google Colab, allow the user to download the file

if is\_colab:

files.download(submission\_file) # Enable download in Colab

# Show the first few rows of the submission file

print("\nFirst rows of the submission file:")

display(submission\_df.head(10))

# Plot the distribution of predicted damage grades

plt.figure(figsize=(10, 6))

sns.countplot(x=submission\_df['damage\_grade'], palette=['lightgreen', 'orange', 'red'])

plt.title('Distribution of Predicted Damage Grades', fontsize=15)

plt.xlabel('Damage Level', fontsize=12)

plt.ylabel('Number of Buildings', fontsize=12)

plt.xticks([0, 1, 2], ['Low (1)', 'Medium (2)', 'High (3)'])

# Add values above the bars for counts and percentages

counts = submission\_df['damage\_grade'].value\_counts().sort\_index()

for i, count in enumerate(counts):

plt.text(i, count + 100, f"{count} ({count/len(submission\_df)\*100:.1f}%)",

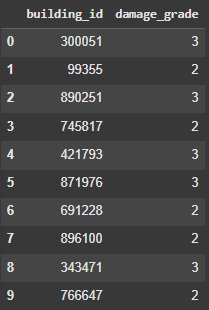
ha='center', fontsize=10)

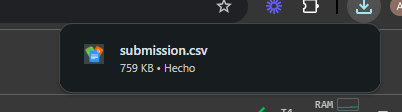
plt.tight\_layout()

plt.show()

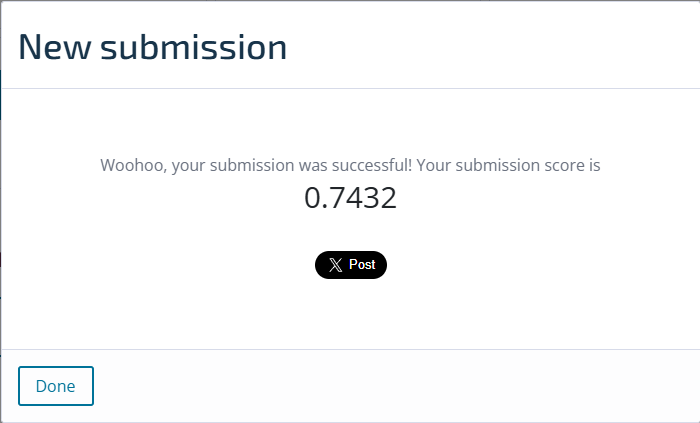
#### **2.8.2 Resultado**

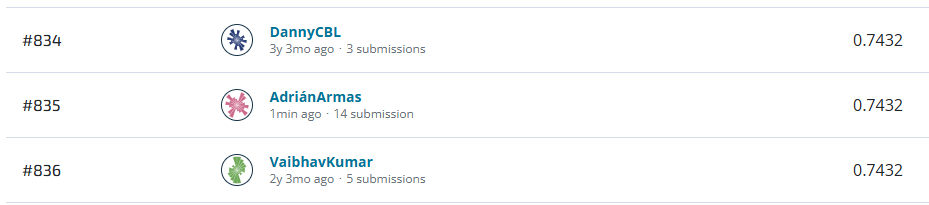
##### **2.8.1.1 Resultado del archivo**





## **3. Resultado**

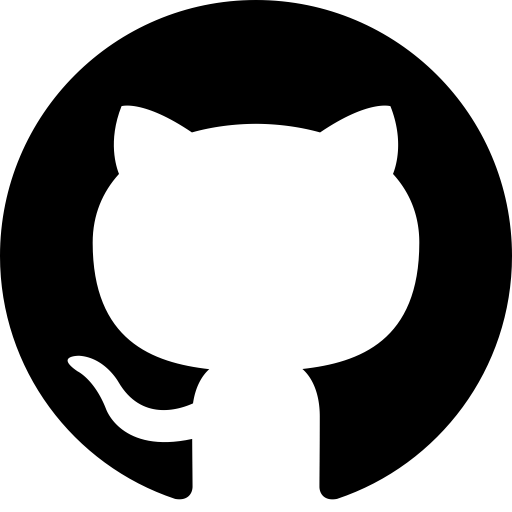




## **4. Problemas encontrados**

* La capacidad de cómputo de las herramientas.
* La necesidad de exportar e importar modelos para hacer más eficiente el programa.

## **5. Github y Colab**

[](https://github.com/AdrianYArmas/IaBigData/tree/main/SNS/3%20%20-%20Algoritmos%20y%20herramientas%20para%20el%20aprendizaje%20supervisado%20/3.7%20%20Predicci%C3%B3n%20de%20Riesgo%20de%20derrumbamiento_Terremotos) [](https://colab.research.google.com/drive/12YodvJJ0lKyidFGc0z4ru6zd0lRa84Ef?usp=sharing)