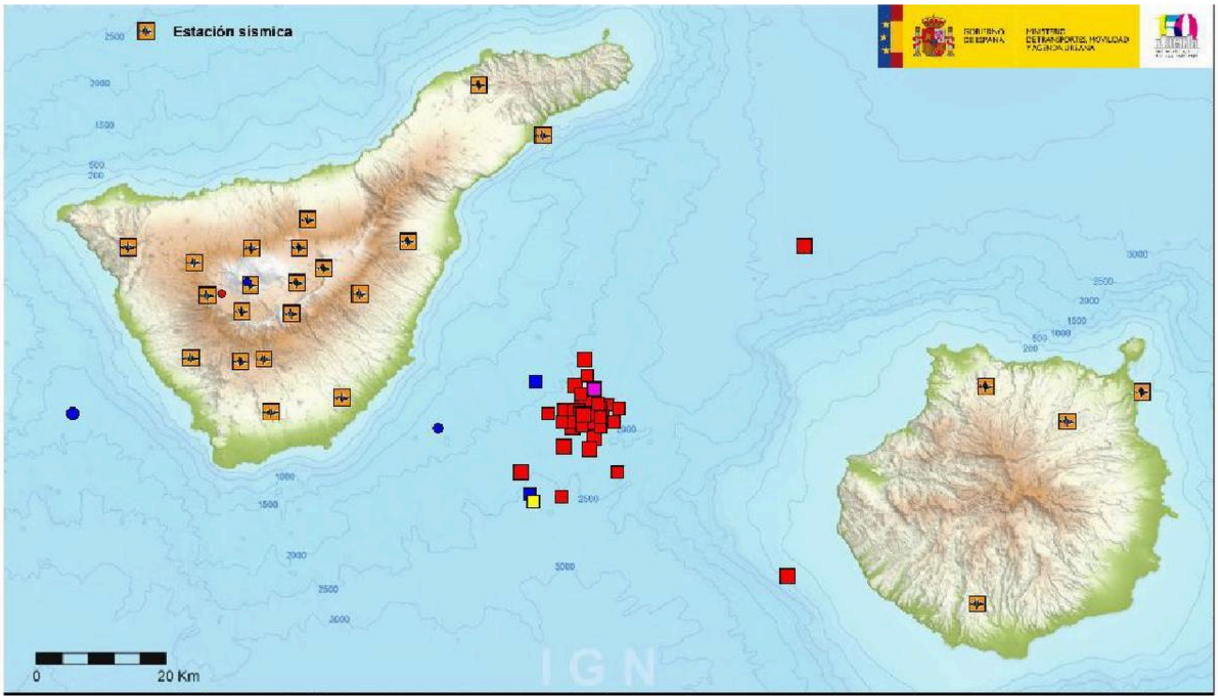
Predicción de Riesgo de derrumbamiento
Terremotos



Adrián Yared Armas de la Nuez

Contenido

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
```

```
Requirement already satisfied: lazypredict in /usr/local/lib/python3.11/dist-packages (0.2.13)
Requirement already satisfied: click in /usr/local/lib/python3.11/dist-packages (from lazypredict) (8.1.8)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (from lazypredict) (
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from lazypredict) (2.2.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from lazypredict) (4.67.1)
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-packages (from lazypredict) (1.4.2)
Requirement already satisfied: lightgbm in /usr/local/lib/python3.11/dist-packages (from lazypredict) (4.5.
Requirement already satisfied: xgboost in /usr/local/lib/python3.11/dist-packages (from lazypredict) (2.1.4
Requirement already satisfied: numpy>=1.17.0 in /usr/local/lib/python3.11/dist-packages (from lightgbm->laz
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from lightgbm->lazypredict
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pand
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->lazypr
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->lazy
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.11/dist-packages (from xgboost->l
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2
```

2.2 Dataset

2.2.1 Descarga

```
# Download dataset from github
train_values_url =
"https://raw.githubusercontent.com/AdrianYArmas/IaBigData/refs/heads/ma
in/SNS/3%20%20-%20Algoritmos%20y%20herramientas%20para%20el%20aprendiza
je%20supervisado%20/3.7%20%20Predicci%C3%B3n%20de%20Riesgo%20de%20derru
mbamiento_Terremotos/dataset/train_values.csv"
train_labels_url =
"https://raw.githubusercontent.com/AdrianYArmas/IaBigData/refs/heads/ma
in/SNS/3%20%20-%20Algoritmos%20y%20herramientas%20para%20el%20aprendiza
je%20supervisado%20/3.7%20%20Predicci%C3%B3n%20de%20Riesgo%20de%20derru
mbamiento_Terremotos/dataset/train_labels.csv"
test_values_url  =
"https://raw.githubusercontent.com/AdrianYArmas/IaBigData/refs/heads/ma
in/SNS/3%20%20-%20Algoritmos%20y%20herramientas%20para%20el%20aprendiza
je%20supervisado%20/3.7%20%20Predicci%C3%B3n%20de%20Riesgo%20de%20derru
mbamiento_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

```
Dimensiones del conjunto de datos de entrenamiento (features): (260601, 39)
Dimensiones del conjunto de datos de entrenamiento (labels): (260601, 2)
Dimensiones del conjunto de datos de prueba: (86868, 39)
```

	building_id	geo_level_1_id	geo_level_2_id	geo_level_3_id	count_floors_pre_eq	age	ar
0	802906	6	487	12198	2	30	
1	28830	8	900	2812	2	10	
2	94947	21	363	8973	2	10	
3	590882	22	418	10694	2	10	
4	201944	11	131	1488	3	30	

5 rows × 39 columns

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()
```

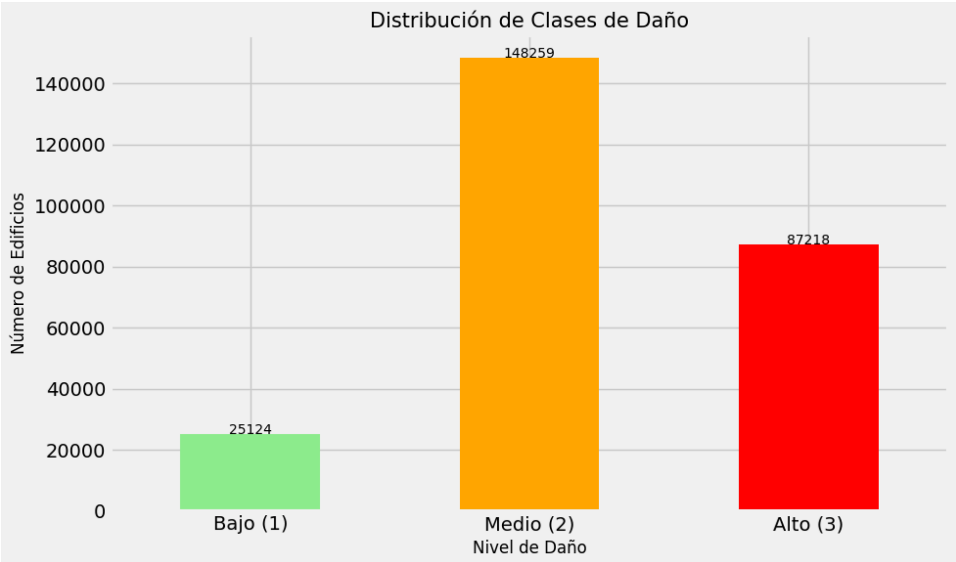
```
damage_counts.plot(kind='bar', color=['lightgreen', 'orange', 'red'])
plt.title('Damage Class Distribution', fontsize=15)
plt.xlabel('Damage Level', fontsize=12)
plt.ylabel('Number of Buildings', fontsize=12)
plt.xticks([0, 1, 2], ['Low (1)', 'Medium (2)', 'High (3)'])

# 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

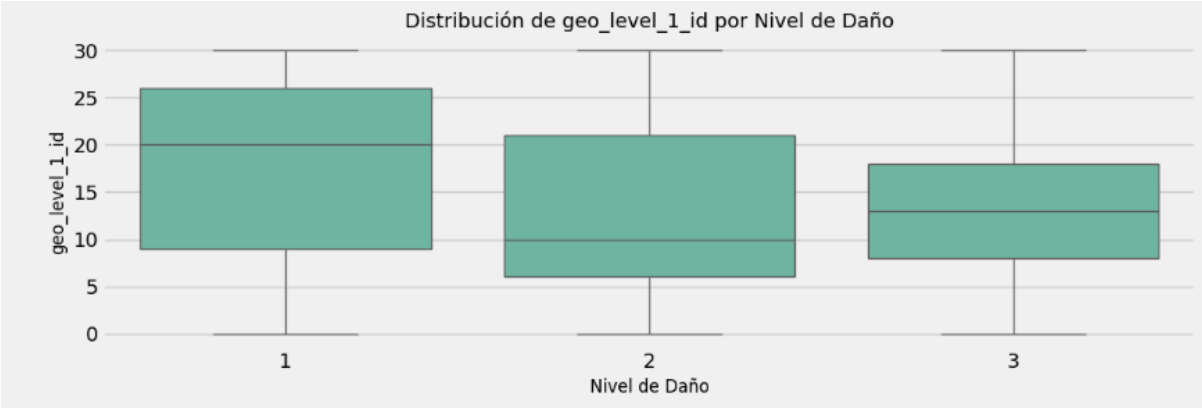
2.3.2.1 Distribución de clases de daño

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



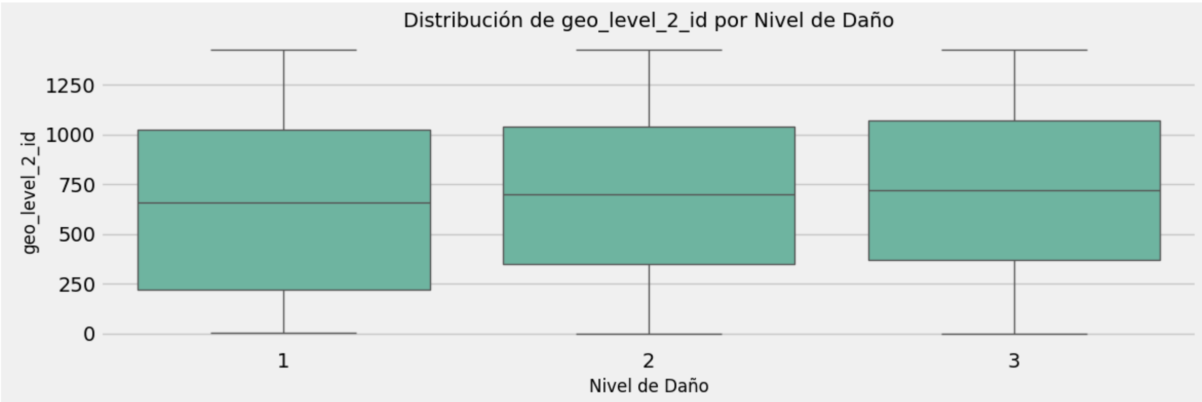
2.3.2.2 Nivel de daño 1

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



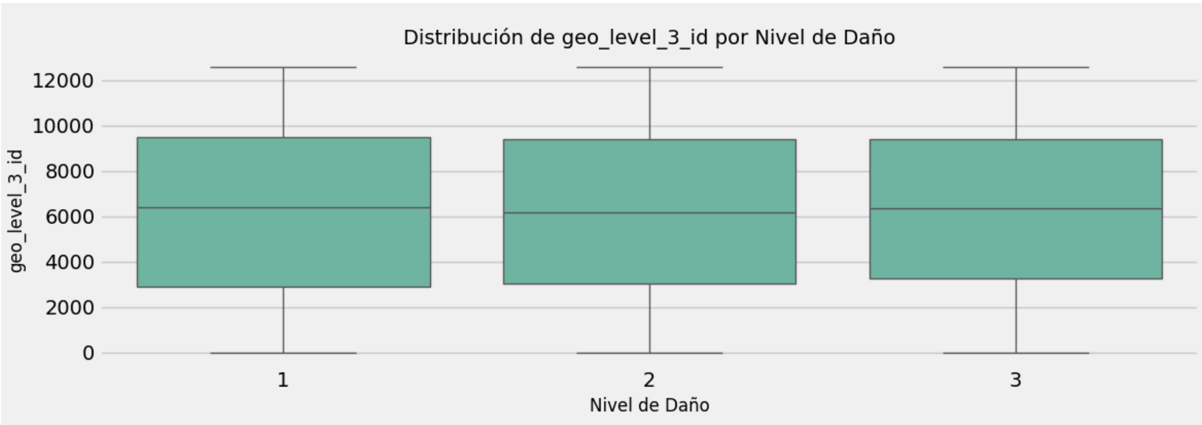
2.3.2.3 Nivel de daño 2

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



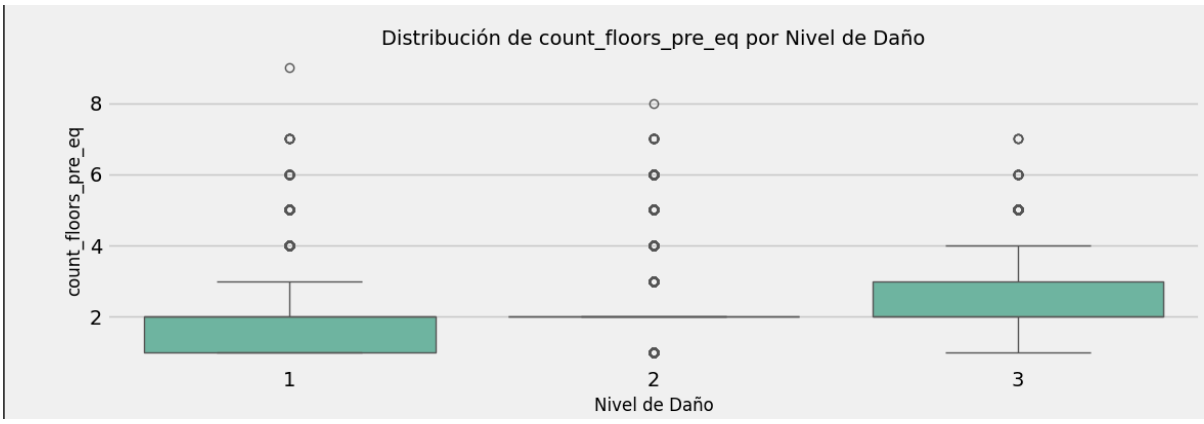
2.3.2.4 Nivel de daño 3

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



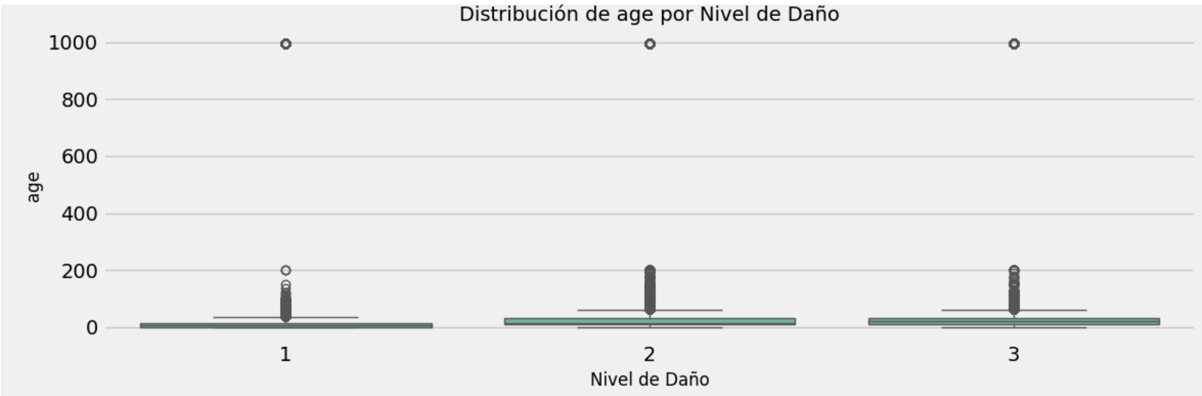
2.3.2.5 Count floors

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



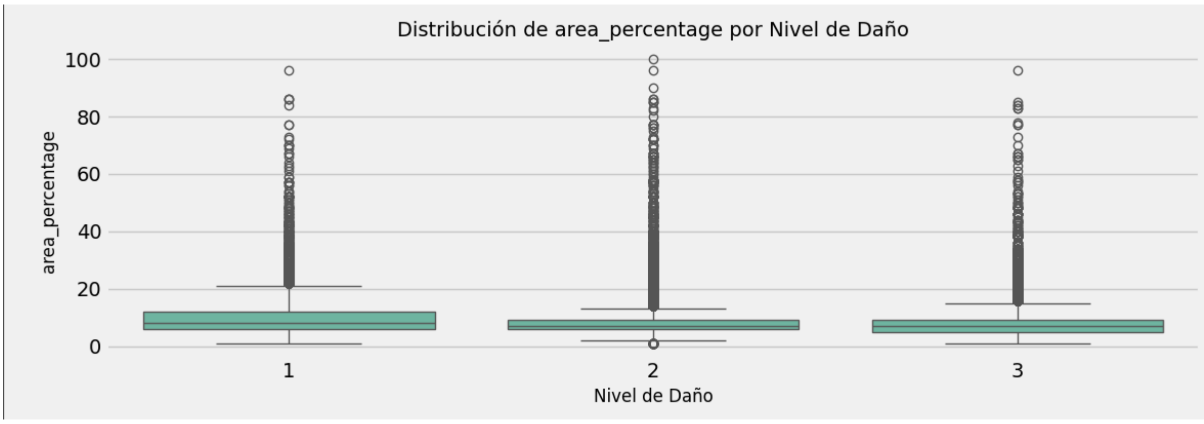
2.3.2.6 Edad

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



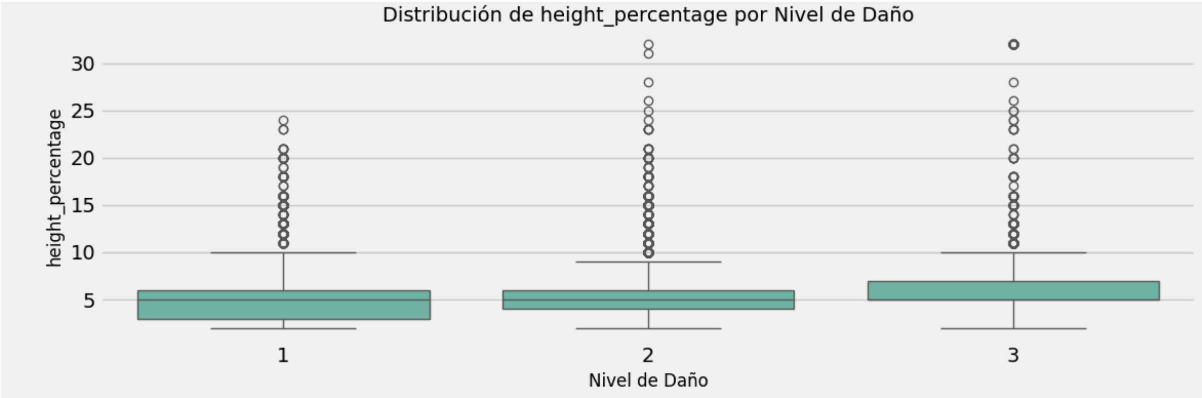
2.3.2.7 Porcentaje por área

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



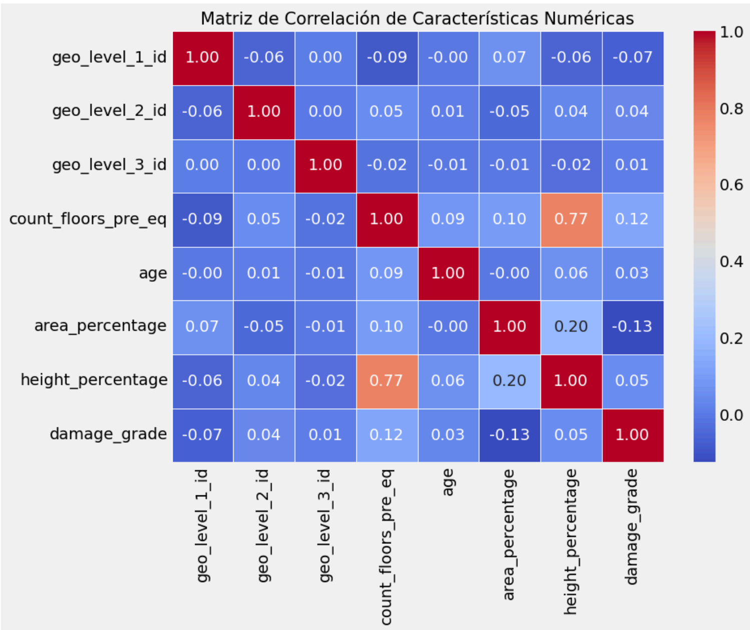
2.3.2.8 Porcentaje por altura

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



2.3.2.9 Matriz de correlación

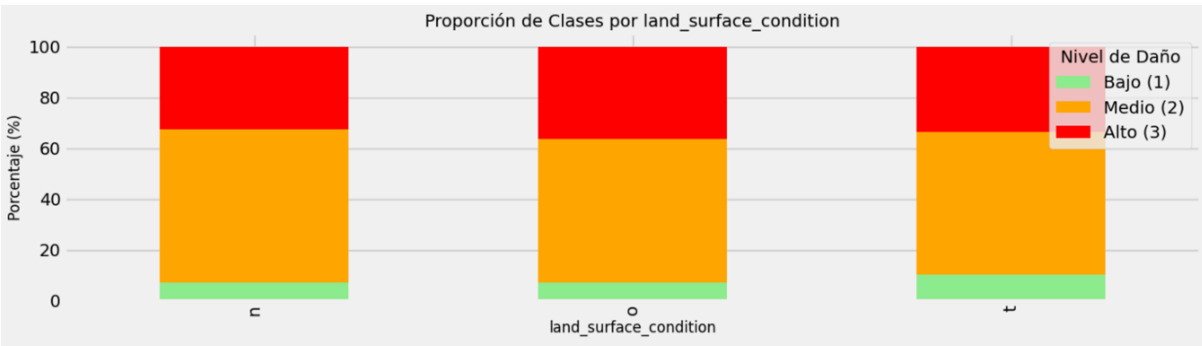
La matriz muestra la correlación entre los datos más relevantes.



	geo_level_1_id	geo_level_2_id	geo_level_3_id	count_floors_pre_eq	age	area_percentage	height_percentage	damage_grade
geo_level_1_id	1.00	-0.06	0.00	-0.09	-0.00	0.07	-0.06	-0.07
geo_level_2_id	-0.06	1.00	0.00	0.05	0.01	-0.05	0.04	0.04
geo_level_3_id	0.00	0.00	1.00	-0.02	-0.01	-0.01	-0.02	0.01
count_floors_pre_eq	-0.09	0.05	-0.02	1.00	0.09	0.10	0.77	0.12
age	-0.00	0.01	-0.01	0.09	1.00	-0.00	0.06	0.03
area_percentage	0.07	-0.05	-0.01	0.10	-0.00	1.00	0.20	-0.13
height_percentage	-0.06	0.04	-0.02	0.77	0.06	0.20	1.00	0.05
damage_grade	-0.07	0.04	0.01	0.12	0.03	-0.13	0.05	1.00

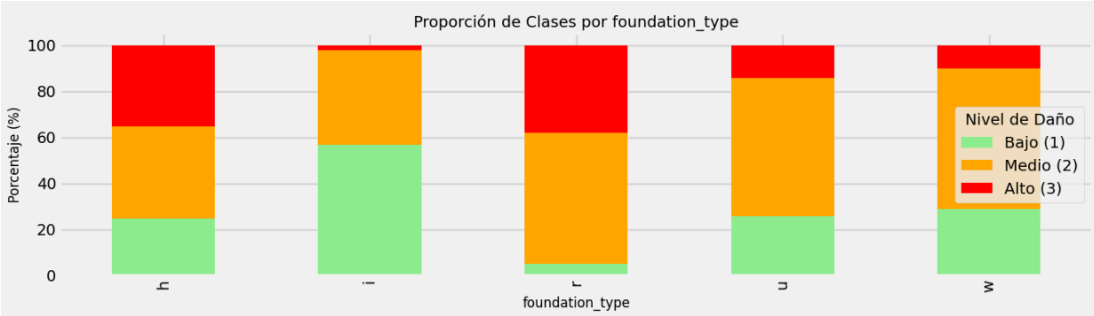
2.3.2.10 Proporción de clases por condición del suelo

Muestra el porcentaje de clases por condición del suelo



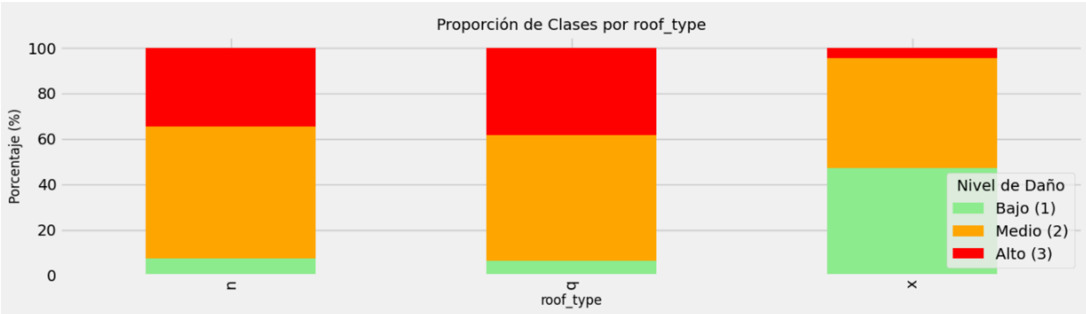
2.3.2.11 Proporción de clases por tipo de cimentación

Muestra el porcentaje de clases por tipo de cimentación



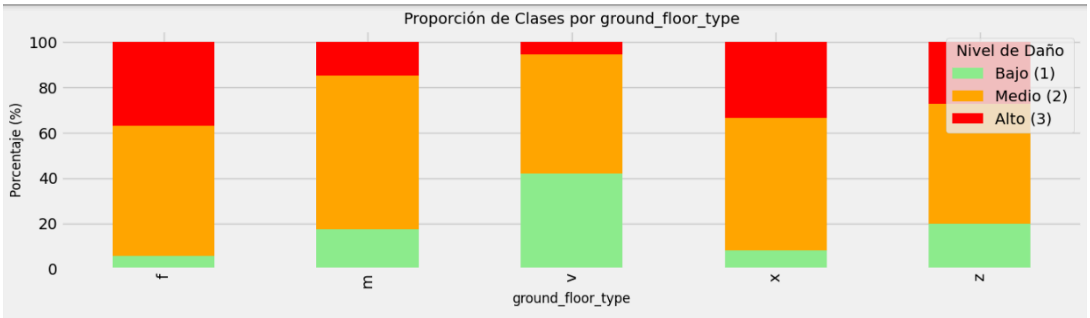
2.3.2.12 Proporción de clases por tipo de tejado

Muestra el porcentaje de clases por tipo de tejado



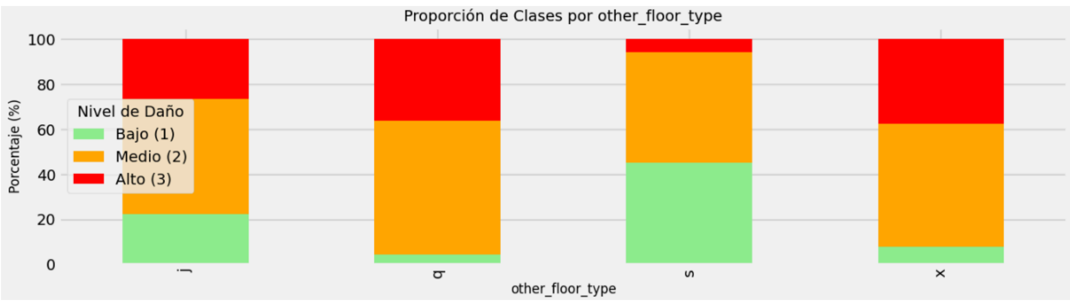
2.3.2.13 Proporción de clases por tipo de suelo

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.

Se identificaron 22 columnas binarias

	geo_level_1_id	geo_level_2_id	geo_level_3_id	count_floors_pre_eq	age	area_percentage	height_percentage
count	260601.00	260601.00	260601.00	260601.00	260601.00	260601.00	260601.00
mean	13.90	701.07	6257.88	2.13	26.54	8.02	5.43
std	8.03	412.71	3646.37	0.73	73.57	4.39	1.92
min	0.00	0.00	0.00	1.00	0.00	1.00	2.00
25%	7.00	350.00	3073.00	2.00	10.00	5.00	4.00
50%	12.00	702.00	6270.00	2.00	15.00	7.00	5.00
75%	21.00	1050.00	9412.00	2.00	30.00	9.00	6.00
max	30.00	1427.00	12567.00	9.00	995.00	100.00	32.00

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_fro
m_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))
```

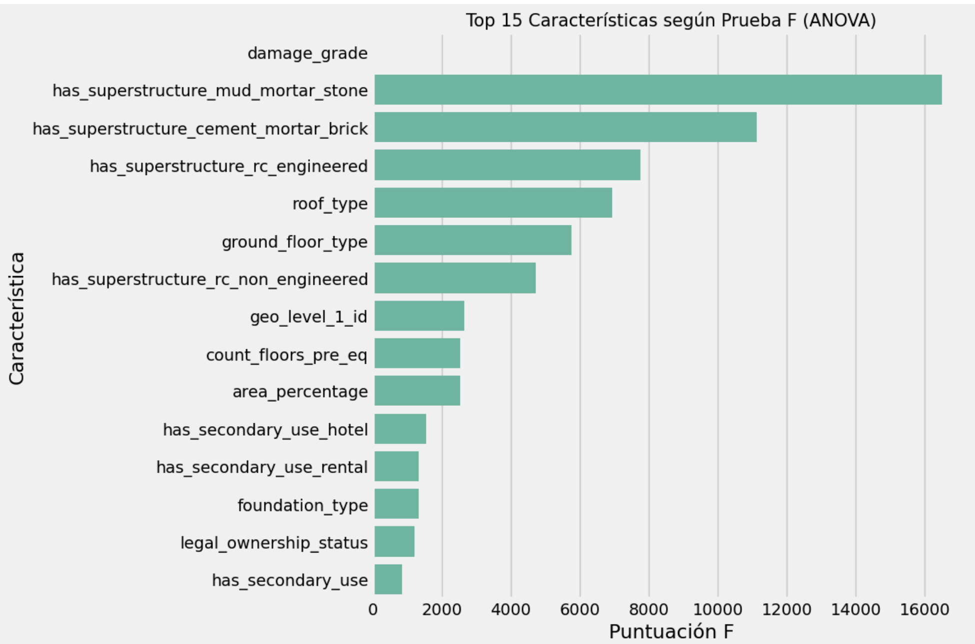
2.4.1.2 Ejecución

2.4.1.2.1 Tabla F (Anova)

Análisis de varianza en busca de diferencias significativas:

	Característica	Puntuación F	P-valor
38	damage_grade	inf	0.00
15	has_superstructure_mud_mortar_stone	16490.39	0.00
19	has_superstructure_cement_mortar_brick	11120.19	0.00
23	has_superstructure_rc_engineered	7757.59	0.00
9	roof_type	6944.70	0.00
10	ground_floor_type	5750.58	0.00
22	has_superstructure_rc_non_engineered	4721.92	0.00
0	geo_level_1_id	2657.79	0.00
3	count_floors_pre_eq	2544.84	0.00
5	area_percentage	2529.05	0.00
29	has_secondary_use_hotel	1537.67	0.00
30	has_secondary_use_rental	1342.10	0.00
8	foundation_type	1341.40	0.00
25	legal_ownership_status	1212.06	0.00
27	has_secondary_use	841.80	0.00
14	has_superstructure_adobe_mud	739.41	0.00
20	has_superstructure_timber	659.20	0.00
16	has_superstructure_stone_flag	576.44	0.00
21	has_superstructure_bamboo	538.55	0.00
18	has_superstructure_mud_mortar_brick	531.78	0.00

2.4.1.2.2 Tabla F (Anova) visual

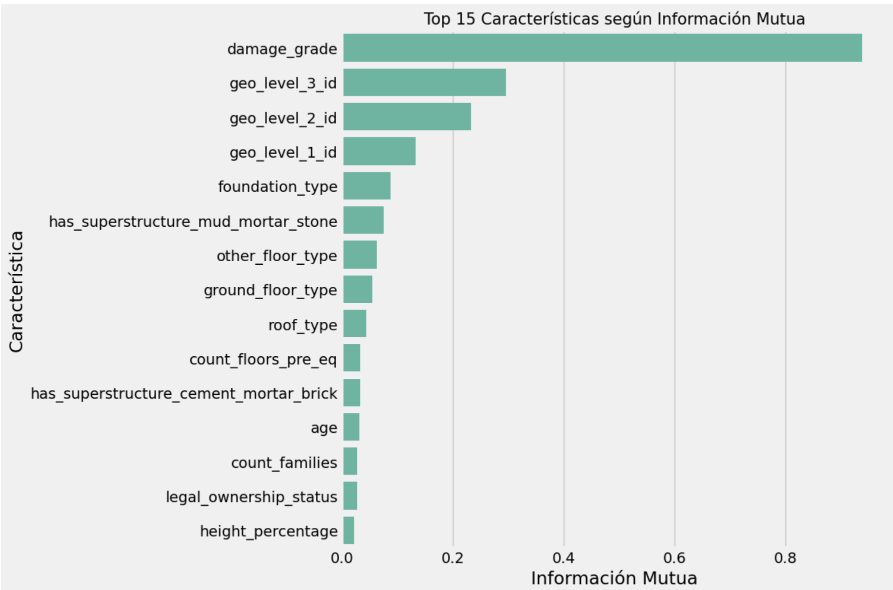


2.4.1.2.3 Tabla información mutua

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

	Característica	Información Mutua
38	damage_grade	0.94
2	geo_level_3_id	0.30
1	geo_level_2_id	0.23
0	geo_level_1_id	0.13
8	foundation_type	0.09
15	has_superstructure_mud_mortar_stone	0.08
11	other_floor_type	0.06
10	ground_floor_type	0.06
9	roof_type	0.04
3	count_floors_pre_eq	0.03
19	has_superstructure_cement_mortar_brick	0.03
4	age	0.03
26	count_families	0.03
25	legal_ownership_status	0.03
6	height_percentage	0.02
13	plan_configuration	0.02
7	land_surface_condition	0.02
23	has_superstructure_rc_engineered	0.02
12	position	0.01
5	area_percentage	0.01

2.4.1.2.4 Tabla información mutua visual

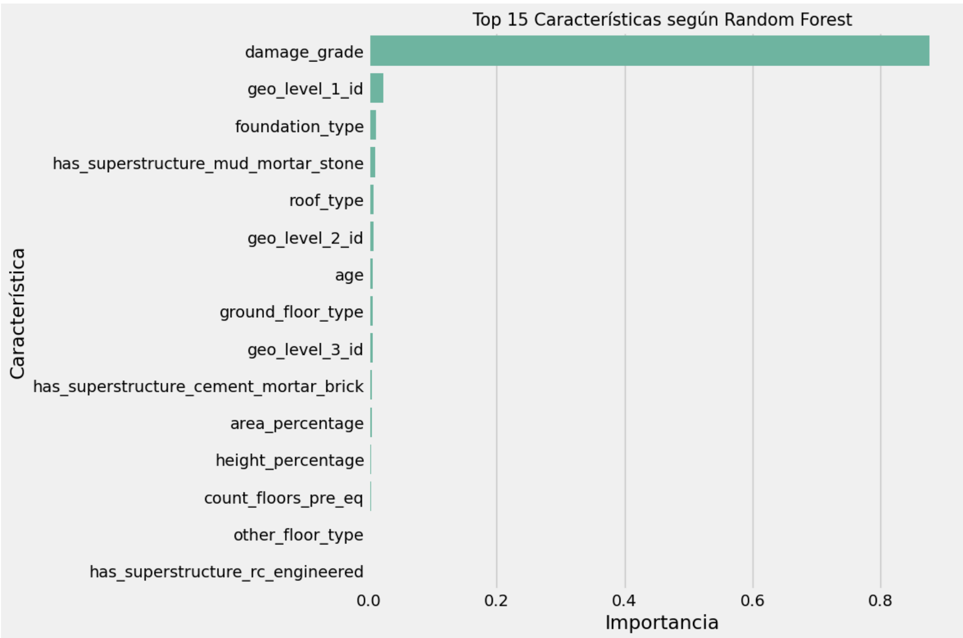


2.4.1.2.5 Tabla RandomForest

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

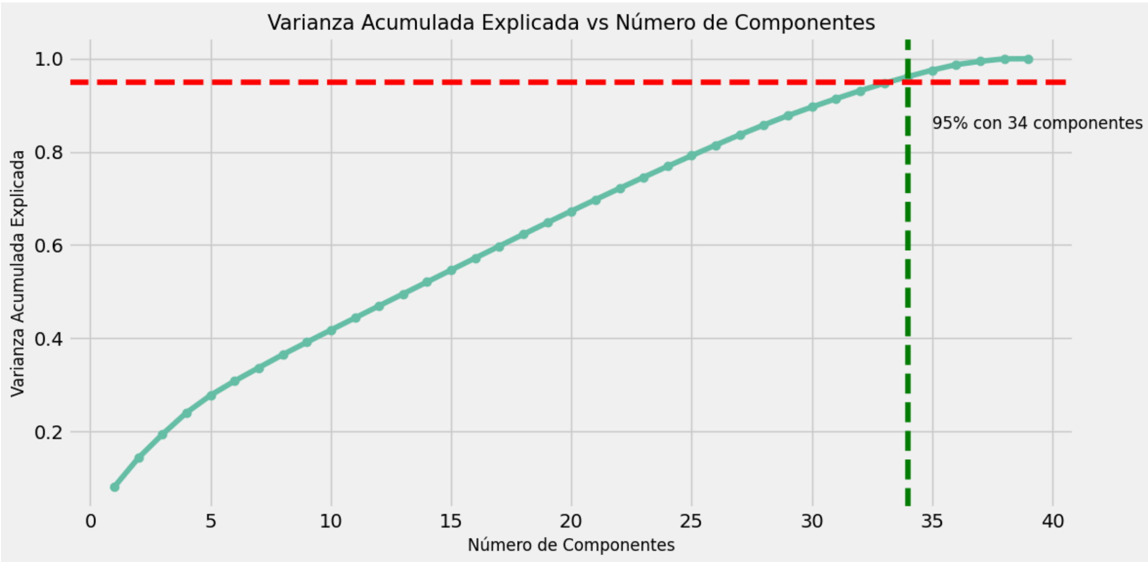
	Característica	Importancia
38	damage_grade	0.88
0	geo_level_1_id	0.02
8	foundation_type	0.01
15	has_superstructure_mud_mortar_stone	0.01
9	roof_type	0.01
1	geo_level_2_id	0.01
4	age	0.01
10	ground_floor_type	0.01
2	geo_level_3_id	0.01
19	has_superstructure_cement_mortar_brick	0.01
5	area_percentage	0.01
6	height_percentage	0.00
3	count_floors_pre_eq	0.00
11	other_floor_type	0.00
23	has_superstructure_rc_engineered	0.00
20	has_superstructure_timber	0.00
18	has_superstructure_mud_mortar_brick	0.00
12	position	0.00
14	has_superstructure_adobe_mud	0.00
26	count_families	0.00

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])
```

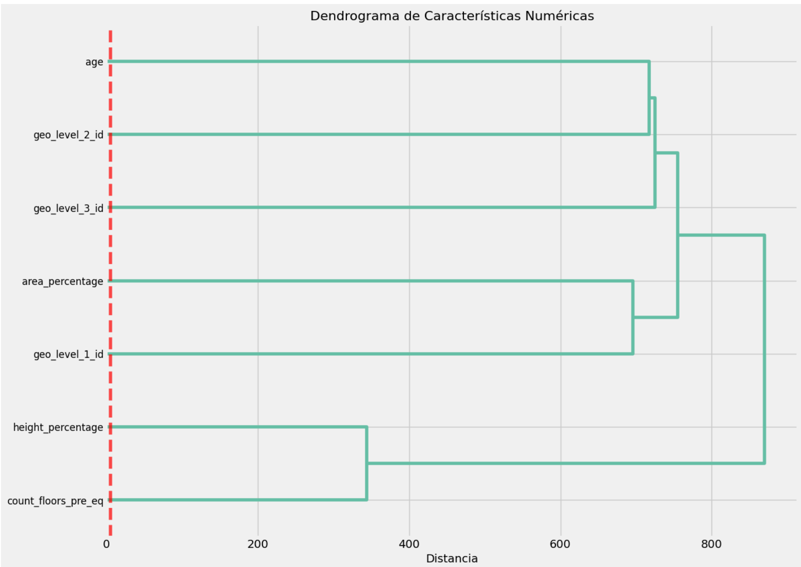
```
# Create and visualize the dendrogram for feature clustering
plt.figure(figsize=(14, 10))
dend = hierarchy.dendrogram(
    hierarchy.linkage(X_scaled.T, method='ward'),
    labels=numerical_features,
    orientation='right',
    leaf_font_size=12,
    color_threshold=5
)
plt.title('Dendrogram of 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)
```

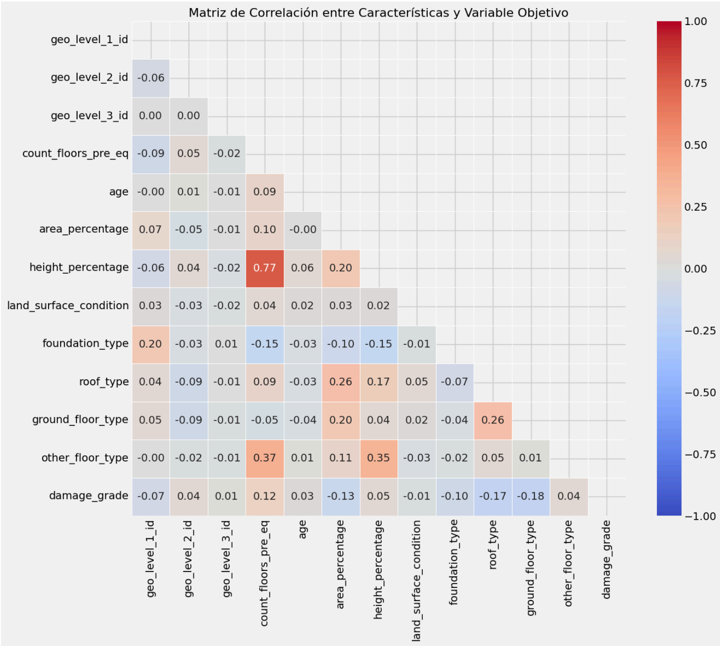
2.4.2.2 Ejecución

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.

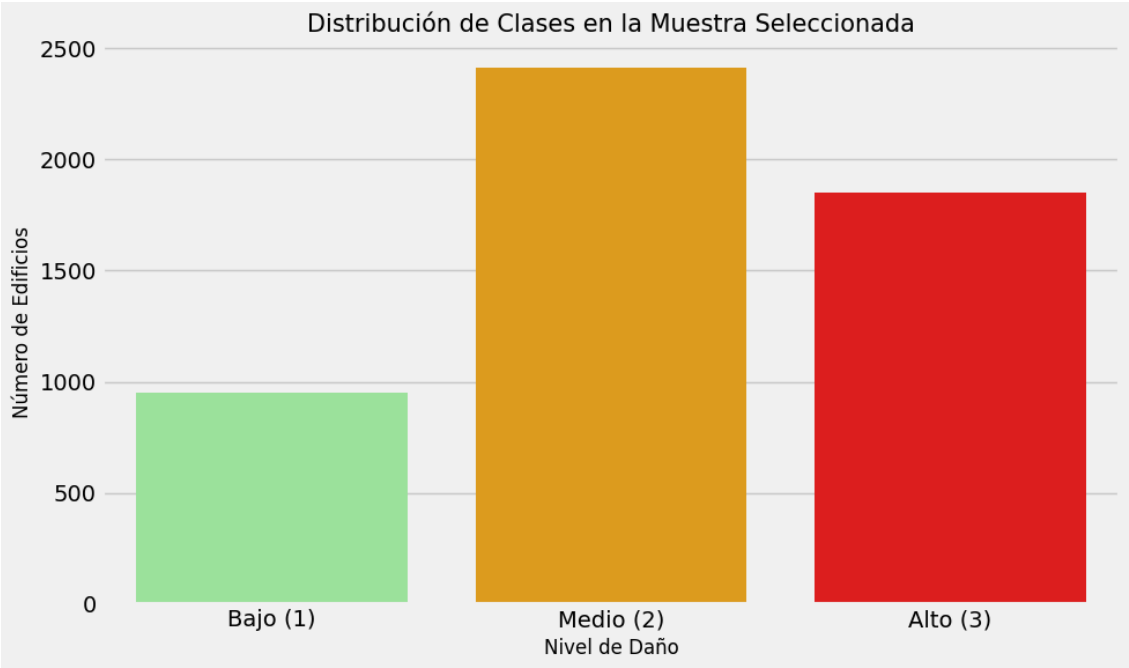
2.5.1 Comando

```
# 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




```
Tamaño conjunto entrenamiento: 4169 muestras
Tamaño conjunto prueba: 1043 muestras
```

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

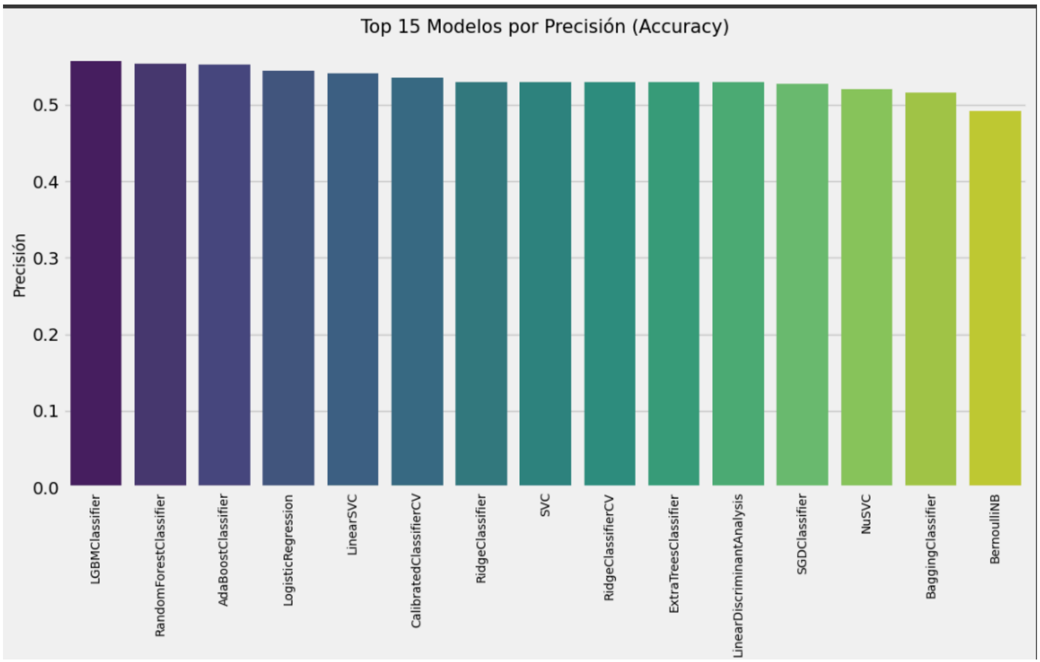
Tabla comparativa entre los modelos y sus precisiones:

Model					
BernoulliNB	0.49	0.51	None	0.46	0.03
NearestCentroid	0.46	0.50	None	0.41	0.02
RandomForestClassifier	0.55	0.50	None	0.54	1.33
LGBMClassifier	0.56	0.50	None	0.55	0.32
LinearDiscriminantAnalysis	0.53	0.49	None	0.52	0.06
ExtraTreesClassifier	0.53	0.49	None	0.52	0.76
LinearSVC	0.54	0.48	None	0.53	0.23
LogisticRegression	0.54	0.48	None	0.53	0.07
RidgeClassifierCV	0.53	0.47	None	0.52	0.04
RidgeClassifier	0.53	0.47	None	0.52	0.02
BaggingClassifier	0.52	0.47	None	0.51	0.29
NuSVC	0.52	0.47	None	0.51	2.91
LabelSpreading	0.47	0.47	None	0.47	1.17
LabelPropagation	0.47	0.47	None	0.47	0.76
SGDClassifier	0.53	0.47	None	0.51	0.22
CalibratedClassifierCV	0.53	0.47	None	0.52	0.93
AdaBoostClassifier	0.55	0.46	None	0.52	0.28
SVC	0.53	0.46	None	0.51	1.88
ExtraTreeClassifier	0.47	0.45	None	0.47	0.02
QuadraticDiscriminantAnalysis	0.39	0.44	None	0.31	0.06
DecisionTreeClassifier	0.46	0.44	None	0.46	0.06
GaussianNB	0.41	0.43	None	0.28	0.03
KNeighborsClassifier	0.47	0.43	None	0.47	0.05
PassiveAggressiveClassifier	0.45	0.43	None	0.45	0.06
Perceptron	0.45	0.39	None	0.43	0.04
DummyClassifier	0.46	0.33	None	0.29	0.02

2.6.2.2 Comparativa visual

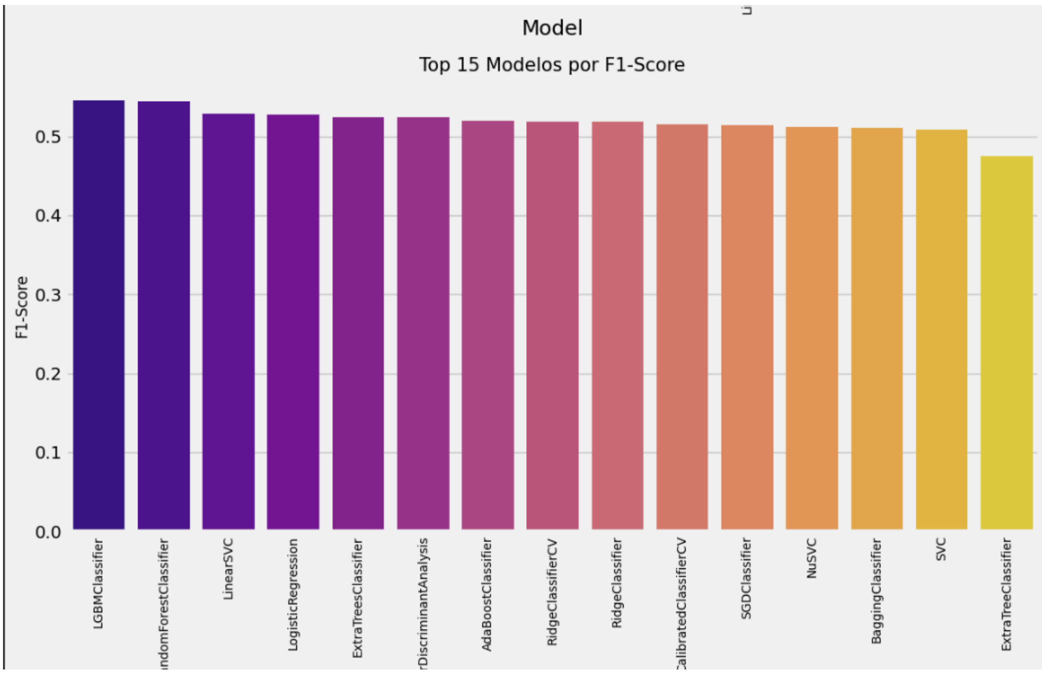
2.6.2.2.1 Comparativa visual (Precisión)

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.

```
Informe de clasificación - Random Forest:
              precision    recall  f1-score   support

           1       0.63      0.25      0.35       190
           2       0.57      0.71      0.63       482
           3       0.56      0.55      0.56       371

    accuracy                           0.57      1043
   macro avg       0.58      0.50      0.51      1043
weighted avg       0.58      0.57      0.55      1043
```

2.7.1.2 BaggingClassifier

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.

Informe de clasificación - BaggingClassifier:

	precision	recall	f1-score	support
1	0.61	0.24	0.34	190
2	0.56	0.74	0.64	482
3	0.57	0.52	0.54	371
accuracy			0.57	1043
macro avg	0.58	0.50	0.51	1043
weighted avg	0.57	0.57	0.55	1043

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.

```
Informe de clasificación - LGBMClassifier Optimizado para Precisión:
              precision    recall  f1-score   support

           1       0.52      0.26      0.35       190
           2       0.58      0.73      0.64       482
           3       0.58      0.53      0.55       371

    accuracy                           0.57      1043
   macro avg       0.56      0.51      0.51      1043
weighted avg       0.57      0.57      0.56      1043
```

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.

Informe de clasificación - SVM:

	precision	recall	f1-score	support
1	0.51	0.26	0.35	190
2	0.55	0.65	0.59	482
3	0.51	0.51	0.51	371
accuracy			0.53	1043
macro avg	0.52	0.48	0.48	1043
weighted avg	0.53	0.53	0.52	1043

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)
```

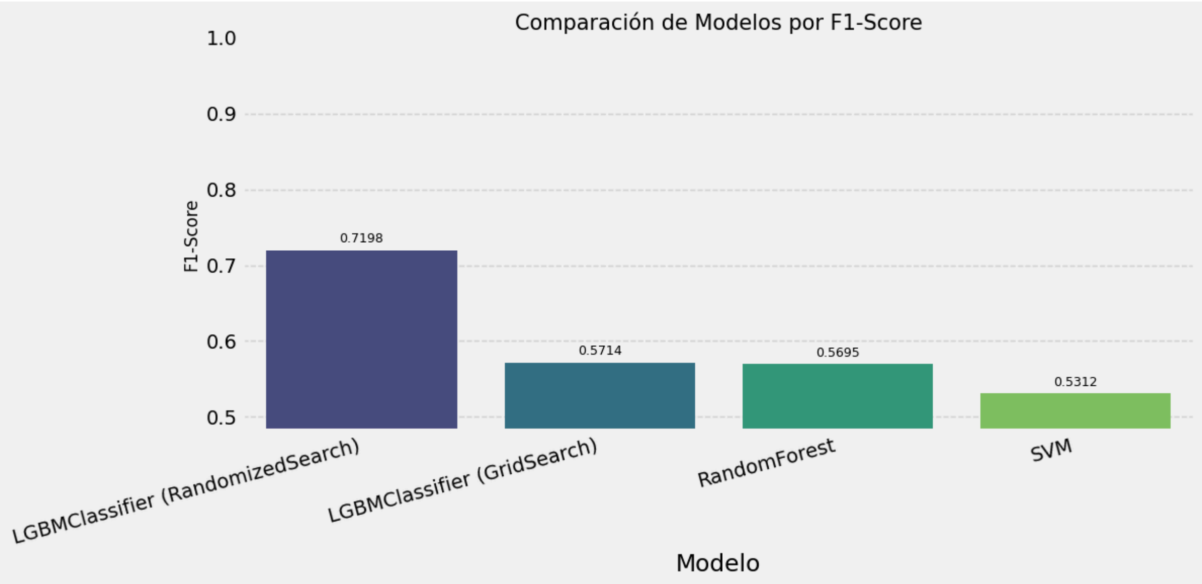
```
plt.tight_layout()
plt.show()

# 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')
```

```python
# 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

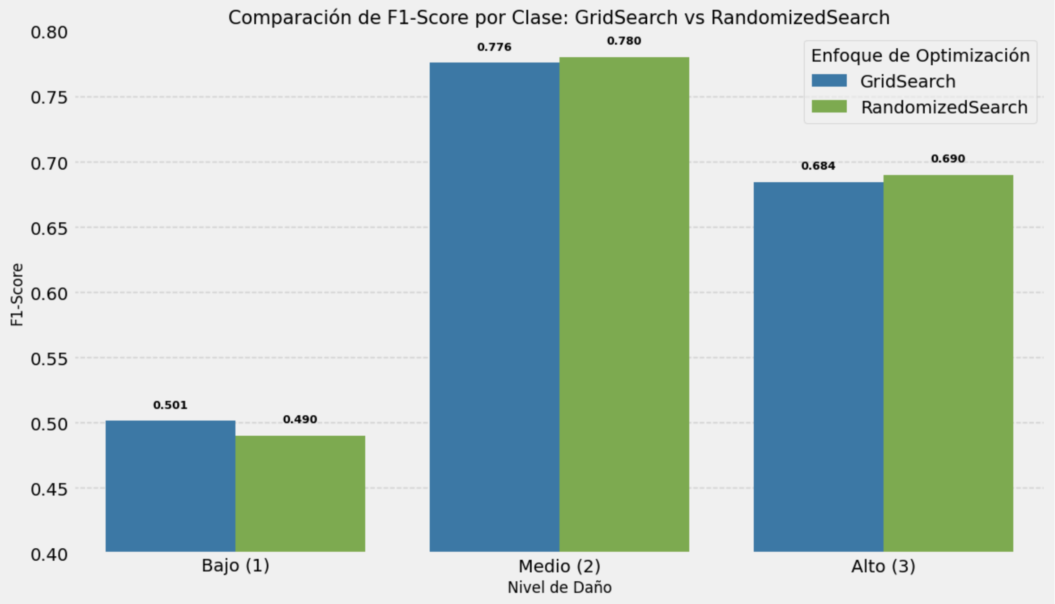
2.7.5.2.1 Modelos F1

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

```
              precision    recall  f1-score   support

           1       0.59      0.05      0.10       190
           2       0.52      0.61      0.56       482
           3       0.50      0.61      0.55       371

    accuracy                           0.51      1043
   macro avg       0.53      0.42      0.40      1043
weighted avg       0.52      0.51      0.47      1043
```

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:
```

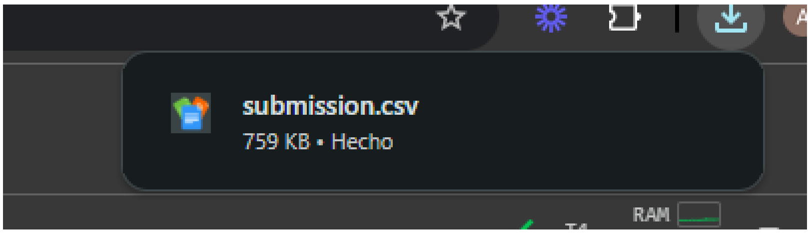
```
    print("Using the final model already in memory")

# 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()
```

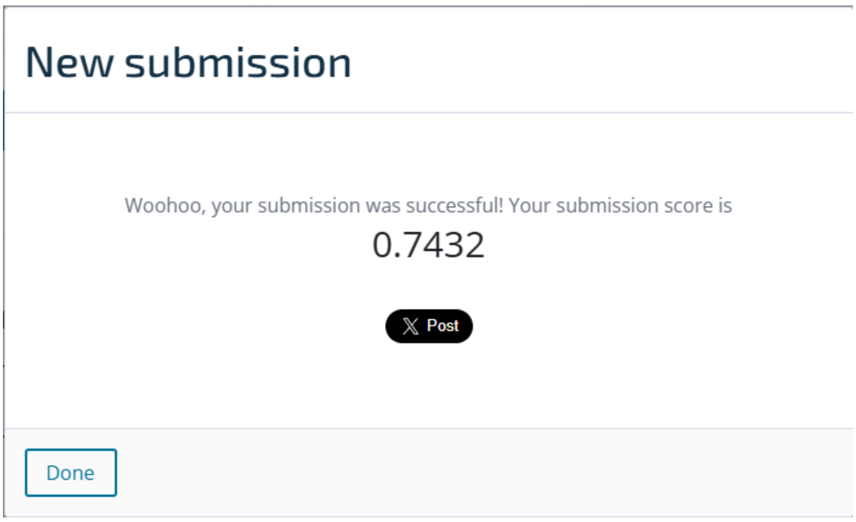
2.8.2 Resultado

2.8.1.1 Resultado del archivo

	building_id	damage_grade
0	300051	3
1	99355	2
2	890251	3
3	745817	2
4	421793	3
5	871976	3
6	691228	2
7	896100	2
8	343471	3
9	766647	2



3. Resultado



#834	DannyCBL 3y 3mo ago · 3 submissions	0.7432
#835	AdriánArmas 1min ago · 14 submission	0.7432
#836	VaibhavKumar 2y 3mo ago · 5 submissions	0.7432

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