Clasificación de terremotos

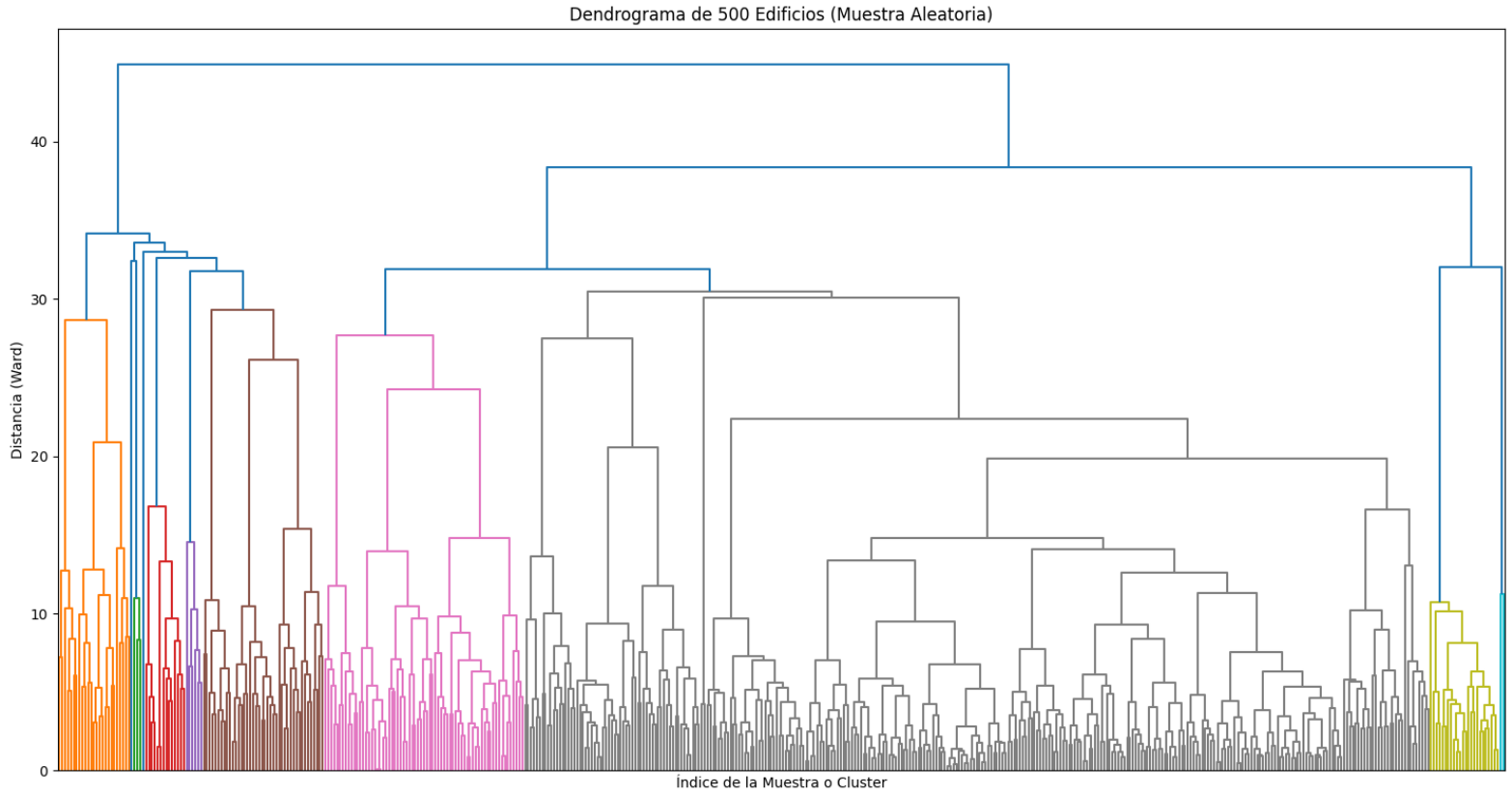
Link a github: Github

Este código realiza clustering jerárquico sobre una muestra aleatoria de datos de edificios. Primero, selecciona 500 muestras aleatorias (o menos si no hay suficientes datos) y las extrae junto con sus etiquetas. Luego, escala las características usando StandardScaler para mejorar la agrupación.

Después, aplica clustering jerárquico con el método Ward, que minimiza la varianza dentro de los clusters. Finalmente, genera un dendrograma para visualizar la estructura de agrupación, permitiendo identificar posibles agrupaciones de edificios según la distancia de corte.

```
# Seleccionar una muestra aleatoria de los datos de entrenamiento procesados
n_samples = 500
if n_samples > X_train_processed.shape[0]:
    n_samples = X_train_processed.shape[0]

sample_indices = np.random.choice(X_train_processed.index, n_samples, replace=False)
X_train_sample = X_train_processed.loc[sample_indices]
y_train_sample_labels = y_train.loc[sample_indices]

# Escalar los datos de la muestra para clustering jerárquico
scaler = StandardScaler()
X_train_sample_scaled = scaler.fit_transform(X_train_sample)

# Realizar clustering jerárquico usando el método 'ward'
Z_buildings = linkage(X_train_sample_scaled, method='ward')

# Crear y visualizar el dendrograma
plt.figure(figsize=(15, 8))
plt.title(f"Dendrograma de {n_samples} Edificios (Muestra Aleatoria)")
plt.xlabel("Índice de la Muestra o Cluster")
plt.ylabel("Distancia (Ward)")

dendrogram(
    Z_buildings,
    leaf_rotation=90.,
    leaf_font_size=8.,
    no_labels=True
)

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

Aquí el resultado:



Luego, entrenamos los modelos con lazyClassifier:

```
clf = LazyClassifier(verbose=1, ignore_warnings=True, custom_metric=None)

#Vamos a usar una porción de los datasets para evitar problemas de memoria
models, predictions = clf.fit(X_train[:len(X_train)//2], X_test[:len(X_test)//2], y_train[:len(y_train)//2], y_test[:len(y_test)//2])
models
```

Aquí el resultado:

Model	Accuracy	Balanced Accuracy	ROC AUC	F1 Score	Time Taken
LGBMClassifier	0.69	0.58	None	0.68	0.26
BaggingClassifier	0.65	0.56	None	0.65	0.70
NearestCentroid	0.46	0.55	None	0.44	0.05
RandomForestClassifier	0.67	0.55	None	0.66	1.43
ExtraTreesClassifier	0.65	0.55	None	0.64	1.39
BernoulliN	0.50	0.54	None	0.50	0.05

BernoulliNB	0.60	0.54	None	0.60	0.13
DecisionTreeClassifier	0.65	0.54	None	0.63	0.71
AdaBoostClassifier	0.37	0.50	None	0.22	0.05
GaussianNB	0.58	0.50	None	0.55	0.10
LinearDiscriminantAnalysis	0.57	0.50	None	0.57	0.05
ExtraTreeClassifier	0.55	0.49	None	0.55	5.41
LabelSpreading	0.55	0.49	None	0.55	4.06
LabelPropagation	0.36	0.49	None	0.22	0.10
QuadraticDiscriminantAnalysis	0.58	0.49	None	0.57	0.24
KNeighborsClassifier	0.60	0.47	None	0.56	0.20
LogisticRegression	0.61	0.47	None	0.57	7.16
SVC	0.60	0.45	None	0.55	2.78
CalibratedClassifierCV	0.60	0.45	None	0.55	0.71
LinearSVC	0.59	0.44	None	0.55	0.36
SGDClassifier	0.60	0.44	None	0.54	0.08
RidgeClassifierCV	0.60	0.43	None	0.54	0.05
RidgeClassifier	0.55	0.41	None	0.47	0.10
PassiveAggressiveClassifier	0.47	0.39	None	0.45	0.09
Perceptron	0.58	0.33	None	0.42	0.04
DummyClassifier					

Luego, se elimina la columna building_id de las características (X) y se extrae de Y para mantenerla disponible si es necesaria más adelante. Además, se redefine Y para que sólo contenga la variable objetivo damage_grade.

División en Entrenamiento y Prueba: Se utiliza train_test_split para repartir los datos en conjuntos de entrenamiento y prueba en una proporción de 80/20, asegurando que la distribución de las clases se mantenga (gracias a stratify=Y).
Preprocesamiento con Label Encoding: Se identifican las columnas categóricas (de tipo object) y se aplica Label Encoding. Cada columna categórica se transforma utilizando un encoder ajustado únicamente con los datos de entrenamiento. Para el conjunto de prueba se aseguran etiquetas

coherentes, asignando un valor especial ('<unknown>') a aquellas categorías no vistas durante el entrenamiento.

Verificación y Consola: Se imprimen las dimensiones de los conjuntos procesados y se comprueba que no existan valores nulos (NaNs) tras la codificación, garantizando la integridad de los datos para el modelado posterior.

Aquí el código:

```
# Eliminar 'building_id' de las características y de Y
X = X.drop('building_id', axis=1)
building_ids_y = Y['building_id'] # Guardar por si acaso, aunque no se usa para entrenar
Y = Y['damage_grade'] # Usar directamente la serie de etiquetas

# Dividir en entrenamiento y prueba ANTES del preprocesamiento detallado
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=42, stratify=Y)
# Usar stratify para clasificación

# --- Preprocesamiento (Label Encoding) ---
categorical_cols = X_train.select_dtypes(include=['object']).columns
label_encoders = {}

# Crear copias para evitar SettingWithCopyWarning
X_train_processed = X_train.copy()
X_test_processed = X_test.copy()

print("Aplicando Label Encoding...")
for column in categorical_cols:
    print(f" - Columna: {column}")
    le = LabelEncoder()
    # Ajustar SOLO con datos de entrenamiento
    X_train_processed[column] = le.fit_transform(X_train_processed[column])
    # Transformar entrenamiento y prueba con el mismo encoder ajustado
    # Manejar valores desconocidos en el test set asignando un valor especial (ej: -1 o len(classes))
     X_test_processed[column] = X_test_processed[column].map(lambda s: '<unknown>' if s not in
le.classes_ else s)
    le.classes_ = np.append(le.classes_, '<unknown>')
    X_test_processed[column] = le.transform(X_test_processed[column])

    label_encoders[column] = le

print("\nPreprocesamiento completado.")
print("Formas de los datos procesados:")
print("X_train_processed:", X_train_processed.shape)
print("X_test_processed:", X_test_processed.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)

# Verificar si hay NaNs (puede ocurrir si el LabelEncoder falla en algún caso)
print("\nNaNs en X_train_processed:", X_train_processed.isnull().sum().sum())
print("NaNs en X_test_processed:", X_test_processed.isnull().sum().sum())
```

Aquí el resultado:

```
Aplicando Label Encoding...
 - Columna: land_surface_condition
 - Columna: foundation_type
 - Columna: roof_type
 - Columna: ground_floor_type
 - Columna: other_floor_type
 - Columna: position
 - Columna: plan_configuration
 - Columna: legal_ownership_status

Preprocesamiento completado.
Formas de los datos procesados:
X_train_processed: (23164, 38)
X_test_processed: (5791, 38)
y_train: (23164,)
y_test: (5791,)

NaNs en X_train_processed: 0
NaNs en X_test_processed: 0
```

Después, definimos y configuramos un modelo base y los espacios de búsqueda de hiperparámetros para optimizarlo (Usaremos RandomForesClassifier):

```
# Modelo base
rf_clf = RandomForestClassifier(random_state=42, n_jobs=-1) # n_jobs=-1 usa todos los cores

# Espacio de búsqueda para GridSearchCV (combinaciones específicas)
param_grid = {
    'n_estimators': [100, 200],          # Número de árboles
    'max_depth': [None, 10, 20],         # Profundidad máxima
    'min_samples_split': [2, 5],         # Mínimo de muestras para dividir un nodo
    'min_samples_leaf': [1, 3],          # Mínimo de muestras en un nodo hoja
    # 'max_features': ['sqrt', 'log2']   # Número de features a considerar (opcional)
    # 'class_weight': [None, 'balanced'] # Ponderación de clases (opcional, útil si hay desbalance)
}

# Espacio de búsqueda para RandomizedSearchCV (distribuciones o rangos)
param_dist = {
    'n_estimators': randint(50, 300),     # Enteros aleatorios entre 50 y 299
    'max_depth': [None] + list(randint(5, 30).rvs(5)), # None o 5 profundidades aleatorias
    'min_samples_split': randint(2, 11),  # Enteros aleatorios entre 2 y 10
    'min_samples_leaf': randint(1, 6),    # Enteros aleatorios entre 1 y 5
    # 'max_features': ['sqrt', 'log2', None],
    # 'class_weight': [None, 'balanced']
}

# Número de iteraciones para RandomizedSearch (más iteraciones = más búsqueda, pero más tiempo)
n_iter_search = 20
```

Ejecutamos el GridSearch:

```
# --- Ejecutar GridSearchCV ---
print("\n--- Iniciando GridSearchCV ---")
# Usaremos 'accuracy' como métrica, podrías usar 'f1_weighted' si las clases están desbalanceadas
# cv=3 para validación cruzada con 3 folds (puedes aumentarlo si tienes tiempo/recursos)
grid_search = GridSearchCV(rf_clf, param_grid, cv=3, scoring='accuracy', verbose=2, n_jobs=-1) # verbose alto para ver progreso

# Entrenar con los datos procesados
grid_search.fit(X_train_processed, y_train)

print("\n--- Resultados GridSearchCV ---")
print("Mejores parámetros encontrados:")
print(grid_search.best_params_)
print("\nMejor puntuación (accuracy) en validación cruzada:")
print(grid_search.best_score_)

# Evaluar el mejor modelo encontrado por GridSearch en el conjunto de prueba
best_grid_model = grid_search.best_estimator_
y_pred_grid = best_grid_model.predict(X_test_processed)
print("\nReporte de clasificación en Test (GridSearchCV):")
print(classification_report(y_test, y_pred_grid))
print("Accuracy en Test (GridSearchCV):", accuracy_score(y_test, y_pred_grid))
```

Aquí el resultado:

```
--- Iniciando GridSearchCV ---
Fitting 3 folds for each of 24 candidates, totalling 72 fits

--- Resultados GridSearchCV ---
Mejores parámetros encontrados:
{'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 200}

Mejor puntuación (accuracy) en validación cruzada:
0.6820066371290306

Reporte de clasificación en Test (GridSearchCV):
              precision    recall  f1-score   support

           1       0.59      0.34      0.43       559
           2       0.69      0.84      0.75      3285
           3       0.71      0.54      0.61      1947

    accuracy                           0.69      5791
   macro avg       0.66      0.57      0.60      5791
weighted avg       0.69      0.69      0.68      5791

Accuracy en Test (GridSearchCV): 0.6883094456915904
```

Y el ramdomizedSearch:

```
print("\n--- Iniciando RandomizedSearchCV ---")
random_search = RandomizedSearchCV(rf_clf, param_distributions=param_dist,
```

```
                    n_iter=n_iter_search, cv=3, scoring='accuracy',
                    verbose=2, random_state=42, n_jobs=-1)

# Entrenar con los datos procesados
random_search.fit(X_train_processed, y_train)

print("\n--- Resultados RandomizedSearchCV ---")
print("Mejores parámetros encontrados:")
print(random_search.best_params_)
print("\nMejor puntuación (accuracy) en validación cruzada:")
print(random_search.best_score_)

# Evaluar el mejor modelo encontrado por RandomSearch en el conjunto de prueba
best_random_model = random_search.best_estimator_
y_pred_random = best_random_model.predict(X_test_processed)
print("\nReporte de clasificación en Test (RandomizedSearchCV):")
print(classification_report(y_test, y_pred_random))
print("Accuracy en Test (RandomizedSearchCV):", accuracy_score(y_test, y_pred_random))
```

```
--- Iniciando RandomizedSearchCV ---
Fitting 3 folds for each of 20 candidates, totalling 60 fits

--- Resultados RandomizedSearchCV ---
Mejores parámetros encontrados:
{'max_depth': np.int64(28), 'min_samples_leaf': 2, 'min_samples_split': 3, 'n_estimators': 251}

Mejor puntuación (accuracy) en validación cruzada:
0.682913037400798

Reporte de clasificación en Test (RandomizedSearchCV):
              precision    recall  f1-score   support

           1       0.60      0.34      0.43       559
           2       0.68      0.85      0.76      3285
           3       0.73      0.51      0.60      1947

    accuracy                           0.69      5791
   macro avg       0.67      0.57      0.60      5791
weighted avg       0.69      0.69      0.67      5791

Accuracy en Test (RandomizedSearchCV): 0.688654809186669
```