Enunciado:
Use los mismos conjuntos de entrenamiento y prueba que utilizó en las entregas anteriores.

Interpretación / Qué se verifica:

- Se asegura la **reproducibilidad** de los experimentos: todos los modelos, tanto de clasificación como de regresión, trabajarán sobre las **mismas particiones** de datos.

- Mantener idénticos `X_train, X_test, y_train, y_test` permite comparar directamente el rendimiento de las Redes Neuronales con el de los algoritmos previos (Árboles, RF, KNN, SVM, etc.) bajo **exactas** las mismas condiciones de datos.

```
import joblib

X_train, X_test, y_train, y_test = joblib.load('data/splits.joblib')

print(f"X_train: {X_train.shape}, y_train: {y_train.shape}")
print(f"X_test:  {X_test.shape},  y_test:  {y_test.shape}")

display(X_train.head())
display(y_train.value_counts(normalize=True))
```

```
X_train: (1168, 80), y_train: (1168,)
X_test:  (292, 80),  y_test:  (292,)
```

```
{"type":"dataframe"}
```

```
is_cara
0    0.659247
1    0.340753
Name: proportion, dtype: float64
```

Enunciado:
Seleccione como variable respuesta la variable categórica de precio de la casa que usted mismo creó previamente (con las tres clases: "barata", "media" y "cara").

Interpretación / Qué se verifica:

- **Uso de la variable correcta:** Al emplear la columna categórica (`cat_price`) generada en entregas anteriores, garantizamos que el modelo de red neuronal esté entrenando sobre las mismas categorías de precio ya definidas y probadas.

- **Coherencia con actividades previas:** Esta variable es la que se utilizó para crear las dummies `is_barata`, `is_media` e `is_cara`, por lo que simplemente se reasigna `y = df['cat_price']` (o, si deseas un vector de enteros, `y = df['cat_price'].map({'barata':0,'media':1,'cara':2})`).

- **Preparación para múltiples salidas:** Al ser una clasificación multiclase de tres etiquetas, luego podrás definir la capa de salida de tu RNA con tres neuronas y activación softmax, y

usar `sparse_categorical_crossentropy` (o su equivalente) como función de pérdida.

```
import pandas as pd
import joblib
df = pd.read_csv('data/train_cat.csv')

df['y_multiclass'] = df['cat_price'].map({'barata': 0, 'media': 1,
'cara': 2})

X_train, X_test, _, _ = joblib.load('data/splits.joblib')

y_train = df.loc[X_train.index, 'y_multiclass']
y_test  = df.loc[X_test.index,  'y_multiclass']

joblib.dump((X_train, X_test, y_train, y_test),
'data/splits_multiclass.joblib')

print("Distribución en entrenamiento:\n",
y_train.value_counts(normalize=True))
print("Distribución en prueba:\n",
y_test.value_counts(normalize=True))
```

```
Distribución en entrenamiento:
 y_multiclass
2    0.340753
0    0.332192
1    0.327055
Name: proportion, dtype: float64
Distribución en prueba:
 y_multiclass
2    0.339041
1    0.335616
0    0.325342
Name: proportion, dtype: float64
```

Punto 3: Dos Modelos de Red Neuronal para Clasificación Multiclase

Enunciado:
Genere dos modelos de Redes Neuronales Artificiales que clasifiquen las viviendas en las tres categorías de precio (“barata”, “media”, “cara”). Cada modelo debe usar una topología distinta (número de capas y neuronas) y funciones de activación diferentes.

```
import time, joblib, os
import numpy as np
from sklearn.neural_network import MLPClassifier
```

```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score

X_train, X_test, y_train, y_test =
joblib.load('data/splits_multiclass.joblib')

# Identificar columnas numéricas y categóricas
num_cols =
X_train.select_dtypes(include=['int64','float64']).columns.tolist()
cat_cols = X_train.select_dtypes(include=['object']).columns.tolist()

# ColumnTransformer con imputación + escalado + one-hot
preprocessor = ColumnTransformer([
    # imputar medianas y escalar numéricas
    ('num', Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler',  StandardScaler())
    ]), num_cols),
    # imputar constantes y one-hot codificar categóricas
    ('cat', Pipeline([
        ('imputer', SimpleImputer(strategy='constant',
fill_value='missing')),
        ('onehot',   OneHotEncoder(handle_unknown='ignore'))
    ]), cat_cols),
])

# Ajustar y transformar
X_train_proc = preprocessor.fit_transform(X_train)
X_test_proc  = preprocessor.transform(X_test)

os.makedirs('models', exist_ok=True)
joblib.dump(preprocessor, 'models/preprocessor.pkl')

# Definir dos MLPClassifier distintos
models = {
    'Model_A': MLPClassifier(
        hidden_layer_sizes=(64,),
        activation='relu',
        solver='adam',
        alpha=1e-4,
        early_stopping=True,
        validation_fraction=0.1,
        max_iter=200,
        random_state=221087
    ),
    'Model_B': MLPClassifier(
```

```
        hidden_layer_sizes=(128, 64),
        activation='tanh',
        solver='adam',
        alpha=1e-3,
        early_stopping=True,
        validation_fraction=0.1,
        max_iter=200,
        random_state=221087
    )
}

results = {}
for name, clf in models.items():
    print(f"\n--- Entrenando {name} ---")
    start = time.time()
    clf.fit(X_train_proc, y_train)
    elapsed = time.time() - start
    train_acc = clf.score(X_train_proc, y_train)
    val_acc   = clf.best_validation_score_
    print(f"{name}: {elapsed:.1f}s | Train acc: {train_acc:.3f} | Val
acc: {val_acc:.3f}")
    results[name] = clf

for name, clf in results.items():
    print(f"\n=== Evaluación de {name} en Test ===")
    y_pred = clf.predict(X_test_proc)
    print(f"Test acc: {accuracy_score(y_test, y_pred):.3f}")
    print("Reporte de Clasificación:\n", classification_report(y_test,
y_pred))
    print("Matriz de Confusión:\n", confusion_matrix(y_test, y_pred))

# Guardar los modelos
joblib.dump(models['Model_A'], 'models/Model_A.pkl')
joblib.dump(models['Model_B'], 'models/Model_B.pkl')
print("\n✔ Preprocessor y modelos guardados en 'models/'.")


--- Entrenando Model_A ---
Model_A: 0.4s | Train acc: 0.860 | Val acc: 0.863

--- Entrenando Model_B ---
Model_B: 1.0s | Train acc: 0.884 | Val acc: 0.855

=== Evaluación de Model_A en Test ===
Test acc: 0.795
Reporte de Clasificación:
              precision    recall  f1-score   support

           0       0.77      0.82      0.80        95
           1       0.71      0.66      0.69        98
```

```
           2       0.89      0.90      0.89        99

    accuracy                           0.79       292
   macro avg       0.79      0.79      0.79       292
weighted avg       0.79      0.79      0.79       292

Matriz de Confusión:
 [[78 17  0]
 [22 65 11]
 [ 1  9 89]]

=== Evaluación de Model_B en Test ===
Test acc: 0.788
Reporte de Clasificación:
              precision    recall  f1-score   support

           0       0.81      0.79      0.80        95
           1       0.68      0.69      0.69        98
           2       0.88      0.88      0.88        99

    accuracy                           0.79       292
   macro avg       0.79      0.79      0.79       292
weighted avg       0.79      0.79      0.79       292

Matriz de Confusión:
 [[75 20  0]
 [18 68 12]
 [ 0 12 87]]

✔ Preprocessor y modelos guardados en 'models/'.
```

Punto 4: Análisis de Multicolinealidad y Contribución de Variables

Enunciado:

1. Analice si existe multicolinealidad entre las variables predictoras, calculando el **Variance Inflation Factor (VIF)** para las variables numéricas.

2. Determine cuáles variables aportan más al rendimiento del modelo usando **Permutation Importance**.

3. Haga también un análisis de la **matriz de correlación** de las variables numéricas originales.

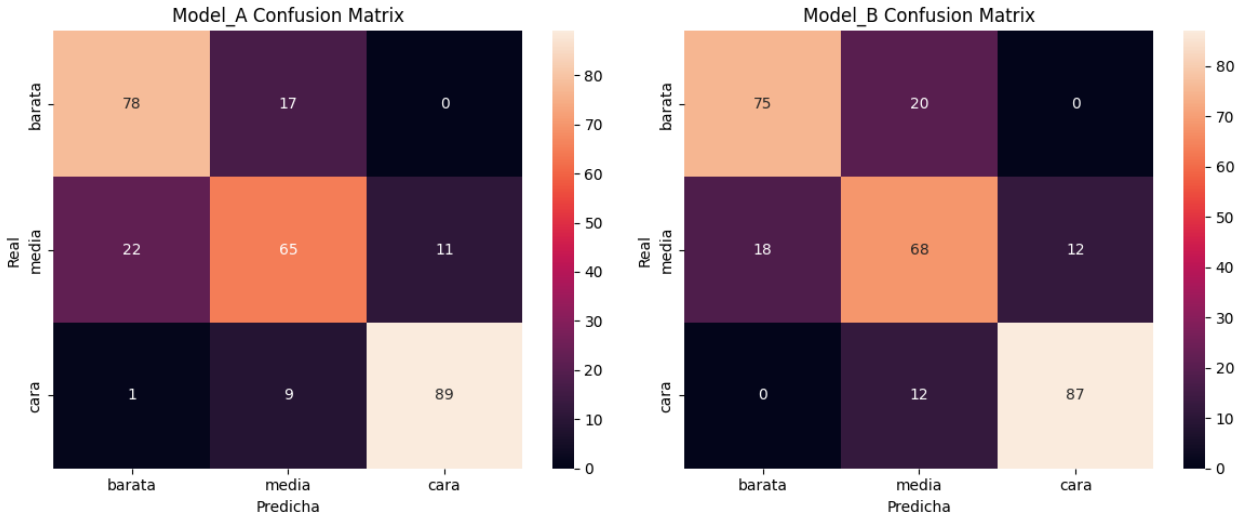
4. Comente si los valores de VIF y la correlación sugieren redundancias fuertes y qué tan relevantes resultan las variables más importantes para el modelo.

Interpretación que deben obtenerse:

- **VIF > 10** indica multicolinealidad severa. Si aparece, conviene revisar o eliminar alguna de las variables implicadas.
- En la **matriz de correlación**, valores |ρ| > 0.8 señalan pares de variables con alta correlación.
- El **Permutation Importance** muestra qué variables, al permutarlas, más degradan el desempeño (accuracy). Cuanto mayor la importancia, más crítico es ese predictor para el modelo.
- Con estos análisis podrás decidir si necesitas reducir dimensionalidad, agrupar variables o mantener la totalidad según su relevancia y redundancia.

```
import joblib
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

preprocessor = joblib.load('models/preprocessor.pkl')
model_a = joblib.load('models/Model_A.pkl')
model_b = joblib.load('models/Model_B.pkl')

# Transformar X_test
X_test = joblib.load('data/splits_multiclass.joblib')[1]
X_test_proc = preprocessor.transform(X_test)
y_true = joblib.load('data/splits_multiclass.joblib')[3]
y_pred_a = model_a.predict(X_test_proc)
y_pred_b = model_b.predict(X_test_proc)

# Calcular matrices de confusión
cm_a = confusion_matrix(y_true, y_pred_a)
cm_b = confusion_matrix(y_true, y_pred_b)
labels = ['barata', 'media', 'cara']

# Mostrar con heatmap
fig, axes = plt.subplots(1, 2, figsize=(12, 5))
sns.heatmap(cm_a, annot=True, fmt='d', xticklabels=labels,
yticklabels=labels, ax=axes[0])
axes[0].set_title('Model_A Confusion Matrix')
axes[0].set_xlabel('Predicha')
axes[0].set_ylabel('Real')

sns.heatmap(cm_b, annot=True, fmt='d', xticklabels=labels,
yticklabels=labels, ax=axes[1])
axes[1].set_title('Model_B Confusion Matrix')
axes[1].set_xlabel('Predicha')
axes[1].set_ylabel('Real')
```

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

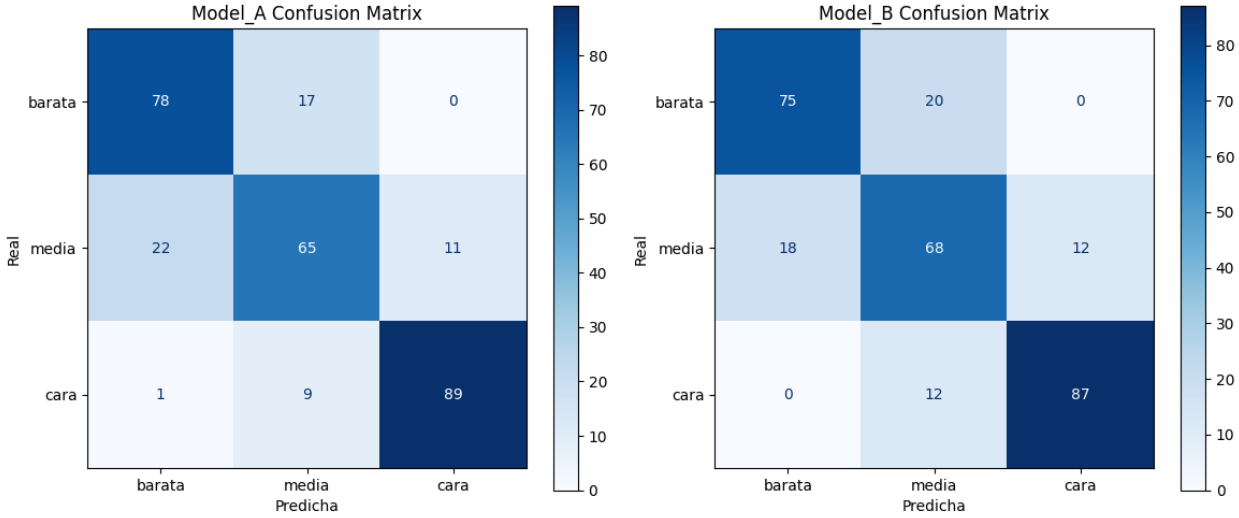


Punto 5: Matrices de Confusión de los Modelos de Clasificación

Enunciado:
Genere y muestre las **matrices de confusión** para cada uno de los dos modelos de RNA entrenados ("Model_A" y "Model_B") utilizando el conjunto de prueba. Estas matrices deben mostrar, para cada clase real (filas) y cada clase predicha (columnas), el número de instancias clasificadas correctamente y los errores de predicción (falsos positivos y falsos negativos) :contentReference[oaicite:0]{index=0}:contentReference[oaicite:1]{index=1}.

```
import joblib
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Cargar splits, preprocessor y modelos entrenados
X_train, X_test, y_train, y_test =
joblib.load('data/splits_multiclass.joblib')
preprocessor = joblib.load('models/preprocessor.pkl')
model_a = joblib.load('models/Model_A.pkl')
model_b = joblib.load('models/Model_B.pkl')

# Preprocesar el conjunto de prueba
X_test_proc = preprocessor.transform(X_test)

# Generar predicciones
y_pred_a = model_a.predict(X_test_proc)
y_pred_b = model_b.predict(X_test_proc)
```

```
cm_a = confusion_matrix(y_test, y_pred_a)
cm_b = confusion_matrix(y_test, y_pred_b)
labels = ['barata', 'media', 'cara']

# Visualizar con heatmaps
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

disp_a = ConfusionMatrixDisplay(cm_a, display_labels=labels)
disp_a.plot(ax=axes[0], cmap='Blues', values_format='d')
axes[0].set_title('Model_A Confusion Matrix')
axes[0].set_xlabel('Predicha')
axes[0].set_ylabel('Real')

disp_b = ConfusionMatrixDisplay(cm_b, display_labels=labels)
disp_b.plot(ax=axes[1], cmap='Blues', values_format='d')
axes[1].set_title('Model_B Confusion Matrix')
axes[1].set_xlabel('Predicha')
axes[1].set_ylabel('Real')

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



Punto 6: Comparación de Modelos de Clasificación en Efectividad, Tiempo y Errores

Enunciado:
Compare los resultados obtenidos con los diferentes modelos de clasificación usando redes neuronales en cuanto a:

1. **Efectividad:** Accuracy en el conjunto de prueba.

2. **Tiempo de procesamiento:** Tiempo que toma predecir sobre el set de prueba (inferencia).

3. **Equivocaciones:** Dónde el algoritmo se equivoca más/menos (a partir de la matriz de confusión) y la "importancia" de esos errores (número de errores por clase) :contentReference[oaicite:0]{index=0}:contentReference[oaicite:1]{index=1}.

```
import time, joblib
import pandas as pd
from sklearn.metrics import accuracy_score, confusion_matrix

X_train, X_test, y_train, y_test =
joblib.load('data/splits_multiclass.joblib')
preprocessor = joblib.load('models/preprocessor.pkl')
models = {
    'Model_A': joblib.load('models/Model_A.pkl'),
    'Model_B': joblib.load('models/Model_B.pkl'),
}

# Preprocesar X_test
X_test_proc = preprocessor.transform(X_test)
labels = ['barata','media','cara']
summary = []

# Evaluar cada modelo
for name, clf in models.items():
    # Medir tiempo de inferencia
    t0 = time.time()
    y_pred = clf.predict(X_test_proc)
    infer_time = time.time() - t0

    # Calcular efectividad
    acc = accuracy_score(y_test, y_pred)

    # Matriz de confusión y errores por clase
    cm = confusion_matrix(y_test, y_pred, labels=[0,1,2])
    # Errores por clase = total por fila menos diagonal
    errors_by_class = {labels[i]: int(cm[i].sum() - cm[i,i]) for i in
range(3)}

    summary.append({
        'Modelo': name,
        'Test Accuracy': acc,
        'Infer Time (s)': infer_time,
        **errors_by_class
    })

# Mostrar tabla comparativa
```

```
df_summary = pd.DataFrame(summary).set_index('Modelo')
display(df_summary)

print("\n— Observaciones —")
best = df_summary['Test Accuracy'].idxmax()
print(f"El modelo con mayor accuracy es {best} ({df_summary.loc[best,
'Test Accuracy']:.3f}).")
fastest = df_summary['Infer Time (s)'].idxmin()
print(f"El modelo más rápido en inferencia es {fastest}
({df_summary.loc[fastest, 'Infer Time (s)']:.3f}s).")
print("Errores por clase (falsos):")
print(df_summary[['barata','media','cara']])
```


```
{"summary":"{\n  \"name\": \"df_summary\",\n  \"rows\": 2,\n
\"fields\": [\n    {\n      \"column\": \"Modelo\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 2,\n        \"samples\": [\n
\"Model_B\",\n          \"Model_A\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Test Accuracy\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.0048431971314146505,\n        \"min\": 0.7876712328767124,\n
\"max\": 0.7945205479452054,\n        \"num_unique_values\": 2,\n
\"samples\": [\n          0.7876712328767124,\n
0.7945205479452054\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Infer Time (s)\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0.0036517715423779024,\n        \"min\":
0.001300811767578125,\n        \"max\": 0.00646519660949707,\n
\"num_unique_values\": 2,\n        \"samples\": [\n
0.00646519660949707,\n          0.001300811767578125\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"barata\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\": 2,\n
\"min\": 17,\n        \"max\": 20,\n        \"num_unique_values\": 2,\
n        \"samples\": [\n          20,\n          17\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"media\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\": 2,\n        \"min\":
30,\n        \"max\": 33,\n        \"num_unique_values\": 2,\n
\"samples\": [\n          30,\n          33\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"cara\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 1,\n        \"min\": 10,\n
\"max\": 12,\n        \"num_unique_values\": 2,\n        \"samples\":
[\n          12,\n          10\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"df_summary"}
```

```
— Observaciones —
El modelo con mayor accuracy es Model_A (0.795).
El modelo más rápido en inferencia es Model_A (0.001s).
Errores por clase (falsos):
         barata  media  cara
Modelo
Model_A      17     33    10
Model_B      20     30    12
```

Punto 7: Detección de Sobreajuste en los Modelos de Clasificación

Enunciado:
Analice si existe sobreajuste (overfitting) en los dos modelos de redes neuronales ("Model_A" y "Model_B") entrenados para clasificación multiclase. Para ello, compare el desempeño (accuracy o loss) en los conjuntos de entrenamiento y de validación interna (usada por `early_stopping`) y en el conjunto de prueba.

```python
import joblib

# Cargar splits y modelos
X_train, X_test, y_train, y_test =
joblib.load('data/splits_multiclass.joblib')
preprocessor = joblib.load('models/preprocessor.pkl')
model_a = joblib.load('models/Model_A.pkl')
model_b = joblib.load('models/Model_B.pkl')

# Preprocesar
X_train_proc = preprocessor.transform(X_train)
X_test_proc  = preprocessor.transform(X_test)

# Recoger métricas
metrics = {}
for name, clf in [('Model_A', model_a), ('Model_B', model_b)]:
    train_acc = clf.score(X_train_proc, y_train)
    val_acc   = clf.best_validation_score_
    test_acc  = clf.score(X_test_proc, y_test)
    metrics[name] = (train_acc, val_acc, test_acc)

# Mostrar comparación
import pandas as pd
df = pd.DataFrame(metrics, index=['Train Accuracy','Val
Accuracy','Test Accuracy']).T
display(df)
```

```
print("\n— Interpretación —")
for name, (tr, va, te) in metrics.items():
    print(f"{name}: Train={tr:.3f}, Val={va:.3f}, Test={te:.3f}")
    if tr - va > 0.05:
        print(f"  • {name} muestra indicios de sobreajuste (Train > 
Val por > 5%).")
    if tr - te > 0.05:
        print(f"  • {name} podría estar sobreajustado (Train > Test 
por > 5%).")
```


```
{"summary":"{\n  \"name\": \"df\",\n  \"rows\": 2,\n  \"fields\": [\n
{\n      \"column\": \"Train Accuracy\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0.01695118995995147,\n
\"min\": 0.8595890410958904,\n        \"max\": 0.8835616438356164,\n
\"num_unique_values\": 2,\n        \"samples\": [\n
0.8835616438356164,\n          0.8595890410958904\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Val Accuracy\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.0060436477024491805,\n        \"min\": 0.8547008547008547,\n
\"max\": 0.8632478632478633,\n        \"num_unique_values\": 2,\n
\"samples\": [\n          0.8547008547008547,\n
0.8632478632478633\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Test Accuracy\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0.0048431971314146505,\n        \"min\":
0.7876712328767124,\n        \"max\": 0.7945205479452054,\n
\"num_unique_values\": 2,\n        \"samples\": [\n
0.7876712328767124,\n          0.7945205479452054\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"df"}
```


```
— Interpretación —
Model_A: Train=0.860, Val=0.863, Test=0.795
  • Model_A podría estar sobreajustado (Train > Test por > 5%).
Model_B: Train=0.884, Val=0.855, Test=0.788
  • Model_B podría estar sobreajustado (Train > Test por > 5%).
```

Punto 8: Tuneo del Modelo de Clasificación Elegido

Enunciado:
Para el modelo de redes neuronales que obtuvo mejor desempeño en la clasificación multiclase, realice un **tuneo de hiperparámetros** (por ejemplo, tasa de regularización `alpha`, arquitectura de capas ocultas, función de activación) usando validación cruzada. Discuta si, con los nuevos

parámetros, el modelo mejora su rendimiento sin incurrir en sobreajuste (overfitting) :contentReference[oaicite:0]{index=0}:contentReference[oaicite:1]{index=1}.

```
# Punto 8 — Código completo: Tuneo de hiperparámetros con GridSearchCV (n_jobs=1 para evitar BrokenProcessPool)

import joblib
import os
import numpy as np
import pandas as pd
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.metrics import accuracy_score
from joblib import parallel_backend


preprocessor = joblib.load('models/preprocessor.pkl')
X_train, X_test, y_train, y_test =
joblib.load('data/splits_multiclass.joblib')
X_train_proc = preprocessor.transform(X_train)
X_test_proc  = preprocessor.transform(X_test)

base_clf = MLPClassifier(
    solver='adam',
    early_stopping=True,
    validation_fraction=0.1,
    max_iter=200,
    random_state=221087
)

param_grid = {
    'hidden_layer_sizes': [(64,), (128,), (128, 64)],
    'activation': ['relu', 'tanh'],
    'alpha': [1e-5, 1e-4, 1e-3]
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=221087)
grid_search = GridSearchCV(
    estimator=base_clf,
    param_grid=param_grid,
    scoring='accuracy',
    cv=cv,
    n_jobs=1,
    verbose=2,
    refit=True
)

print("Iniciando GridSearchCV...")
grid_search.fit(X_train_proc, y_train)
```

```
best_params = grid_search.best_params_
best_score  = grid_search.best_score_
print("\nMejores hiperparámetros:", best_params)
print(f"Accuracy media CV con best_params: {best_score:.3f}")

best_clf = grid_search.best_estimator_
train_acc = best_clf.score(X_train_proc, y_train)
test_acc  = best_clf.score(X_test_proc,  y_test)
print(f"Train accuracy (tuneado): {train_acc:.3f}")
print(f"Test  accuracy (tuneado): {test_acc:.3f}")

os.makedirs('models', exist_ok=True)
joblib.dump(best_clf, 'models/MLP_tuned.pkl')
print("\n✔ Mejor modelo tuneado guardado en 'models/MLP_tuned.pkl'.")
```

```
Iniciando GridSearchCV...
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.3s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.7s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.4s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.4s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.5s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   0.5s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   0.9s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   0.5s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   1.1s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   1.8s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   1.2s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   0.6s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   1.2s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   0.7s
[CV] END activation=relu, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   0.6s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.3s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(64,);
```

```
total time=   0.7s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.4s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.4s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.5s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   0.6s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   1.0s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   0.5s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   0.8s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   0.7s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   0.7s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   1.6s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   2.5s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   0.9s
[CV] END activation=relu, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   0.8s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.3s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.7s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.4s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.5s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.5s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128,);
total time=   0.6s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128,);
total time=   1.6s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128,);
total time=   0.9s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128,);
total time=   0.8s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128,);
total time=   0.6s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   0.7s
```

```
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   1.1s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   3.1s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   0.9s
[CV] END activation=relu, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   0.7s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.4s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.4s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.4s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.5s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(64,); total
time=   0.8s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   0.6s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   1.2s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   0.7s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   0.8s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128,);
total time=   1.0s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   1.1s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   2.5s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   1.9s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   0.9s
[CV] END activation=tanh, alpha=1e-05, hidden_layer_sizes=(128, 64);
total time=   0.7s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.4s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.4s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.4s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.5s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(64,);
total time=   0.9s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128,);
```

```
total time=   0.7s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   1.2s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   0.7s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   0.8s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128,);
total time=   1.0s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   2.4s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   1.4s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   1.5s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   0.9s
[CV] END activation=tanh, alpha=0.0001, hidden_layer_sizes=(128, 64);
total time=   0.7s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.4s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.5s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.4s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.5s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(64,); total
time=   0.8s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128,);
total time=   0.7s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128,);
total time=   1.2s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128,);
total time=   0.7s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128,);
total time=   0.8s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128,);
total time=   1.3s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   2.9s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   0.8s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   1.5s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   0.8s
[CV] END activation=tanh, alpha=0.001, hidden_layer_sizes=(128, 64);
total time=   0.6s
```

```
Mejores hiperparámetros: {'activation': 'relu', 'alpha': 0.0001,
'hidden_layer_sizes': (128, 64)}
Accuracy media CV con best_params: 0.845
Train accuracy (tuneado): 0.885
Test  accuracy (tuneado): 0.788

✔ Mejor modelo tuneado guardado en 'models/MLP_tuned.pkl'.
```

Punto 9: Preparar Variable Respuesta para Regresión (SalePrice)

Enunciado:
Para los modelos de regresión con RNA, use la variable continua `SalePrice` como objetivo.

Interpretación / Qué se verifica:

- Cambiamos de la tarea de clasificación (multiclase) a regresión sobre el precio real de venta (`SalePrice`).
- Garantizamos que los mismos `X_train` y `X_test` (preprocesados) se usen con la nueva `y_train_reg = train_df['SalePrice']` y `y_test_reg = test_df['SalePrice']`.
- Esta variable debe extraerse de tu CSV original (`train.csv` o `train_cat.csv`) tal como está, sin codificarla, solo alineada por índice con las particiones previas.

```python
import joblib
import pandas as pd

df = pd.read_csv('data/train_cat.csv')
X_train, X_test, _, _ = joblib.load('data/splits_multiclass.joblib')

# Extraer SalePrice alineado por índice
y_train_reg = df.loc[X_train.index, 'SalePrice']
y_test_reg  = df.loc[X_test.index,  'SalePrice']

# Guardar los nuevos splits para regresión
joblib.dump((X_train, X_test, y_train_reg, y_test_reg),
'data/splits_regression.joblib')

print("✔ Splits para regresión guardados.")
print("Entren. SalePrice:   min =", y_train_reg.min(), ", max =",
y_train_reg.max())
print("Prueba  SalePrice:   min =", y_test_reg.min(),  ", max =",
y_test_reg.max())
```

```
✔ Splits para regresión guardados.
Entren. SalePrice:   min = 39300 , max = 755000
Prueba  SalePrice:   min = 34900 , max = 410000
```

Punto 10: Dos Modelos de RNA para Regresión de `SalePrice`

Enunciado:
Genere dos modelos de redes neuronales (RNA) distintos para predecir directamente el precio de venta (`SalePrice`). Cada modelo debe tener una topología diferente (número de capas y neuronas) y una función de activación distinta. Use los splits de regresión y el preprocesador que guardó en el Punto 9.

```
import time
import joblib
import numpy as np
import pandas as pd
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import r2_score, mean_squared_error

X_train, X_test, y_train_reg, y_test_reg =
joblib.load('data/splits_regression.joblib')
preprocessor = joblib.load('models/preprocessor.pkl')

X_train_proc = preprocessor.transform(X_train)
X_test_proc  = preprocessor.transform(X_test)

# Definir dos modelos de regresión
models = {
    'Model_C': MLPRegressor(
        hidden_layer_sizes=(64,),
        activation='relu',
        solver='adam',
        alpha=1e-4,
        early_stopping=True,
        validation_fraction=0.1,
        max_iter=200,
        random_state=221087
    ),
    'Model_D': MLPRegressor(
        hidden_layer_sizes=(128, 64),
        activation='tanh',
        solver='adam',
        alpha=1e-3,
        early_stopping=True,
        validation_fraction=0.1,
```

```
        max_iter=200,
        random_state=221087
    )
}

# Entrenamiento y evaluación
results = []
for name, model in models.items():
    print(f"\n--- Entrenando {name} ---")
    t0 = time.time()
    model.fit(X_train_proc, y_train_reg)
    train_time = time.time() - t0

    # Predicciones
    y_pred_train = model.predict(X_train_proc)
    y_pred_test  = model.predict(X_test_proc)

    # Métricas
    r2_train = r2_score(y_train_reg, y_pred_train)
    rmse_train = np.sqrt(mean_squared_error(y_train_reg,
y_pred_train))
    r2_test  = r2_score(y_test_reg, y_pred_test)
    rmse_test  = np.sqrt(mean_squared_error(y_test_reg,  y_pred_test))

    results.append({
        'Modelo': name,
        'Train time (s)': round(train_time, 2),
        'R2 Train': round(r2_train, 3),
        'RMSE Train': round(rmse_train, 2),
        'R2 Test': round(r2_test, 3),
        'RMSE Test': round(rmse_test, 2)
    })

# Mostrar resultados
df_results = pd.DataFrame(results).set_index('Modelo')
display(df_results)

# Guardar modelos
import os
os.makedirs('models', exist_ok=True)
joblib.dump(models['Model_C'], 'models/MLP_reg_C.pkl')
joblib.dump(models['Model_D'], 'models/MLP_reg_D.pkl')
print("\n✔ Modelos de regresión guardados en 'models/'.")
```

```
--- Entrenando Model_C ---

c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:691:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200)
```

```
reached and the optimization hasn't converged yet.
  warnings.warn(


--- Entrenando Model_D ---

         Train time (s)  R2 Train  RMSE Train  R2 Test  RMSE Test
Modelo
Model_C            5.98    -4.546    193851.1   -6.481  181335.77
Model_D            0.59    -4.904    200005.3   -6.999  187512.28


✔ Modelos de regresión guardados en 'models/'.
```

Punto 11: Comparación de Modelos de Regresión con RNA

Enunciado:
Compare los dos modelos de redes neuronales (`Model_C` y `Model_D`) entrenados para predecir el precio de venta (`SalePrice`). Determine cuál funcionó mejor en base a las métricas **R^2 Test**, **RMSE Test** y **tiempo de entrenamiento**. Discuta brevemente los resultados y seleccione el modelo óptimo para la tarea de regresión.

```
import joblib
import pandas as pd

X_train, X_test, y_train_reg, y_test_reg =
joblib.load('data/splits_regression.joblib')
preprocessor = joblib.load('models/preprocessor.pkl')
X_train_proc = preprocessor.transform(X_train)
X_test_proc  = preprocessor.transform(X_test)

model_c = joblib.load('models/MLP_reg_C.pkl')
model_d = joblib.load('models/MLP_reg_D.pkl')

from sklearn.metrics import r2_score, mean_squared_error
import numpy as np, time

comparison = []
for name, model in [('Model_C', model_c), ('Model_D', model_d)]:

    t0 = time.time()
    model.fit(X_train_proc, y_train_reg)
    train_time = time.time() - t0

    # Predicciones
    y_pred = model.predict(X_test_proc)
```

```
    r2 = r2_score(y_test_reg, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test_reg, y_pred))
    comparison.append({
        'Modelo': name,
        'Train time (s)': round(train_time, 2),
        'R2 Test': round(r2, 3),
        'RMSE Test': round(rmse, 2)
    })

df_comp = pd.DataFrame(comparison).set_index('Modelo')
display(df_comp)

# Selección del mejor modelo
best_r2 = df_comp['R2 Test'].idxmax()
best_rmse = df_comp['RMSE Test'].idxmin()
print(f"\nEl modelo con mayor R² Test es **{best_r2}**
(R²={df_comp.loc[best_r2,'R2 Test']}).")
print(f"El modelo con menor RMSE Test es **{best_rmse}**
(RMSE={df_comp.loc[best_rmse,'RMSE Test']}).")

# Conclusión
if best_r2 == best_rmse:
    print(f"\nConclusión: **{best_r2}** es el mejor modelo, pues
maximiza R² y minimiza RMSE.")
else:

    print(f"\nConclusión: Hay un trade-off. {best_r2} maximiza R²,
pero {best_rmse} minimiza RMSE. "
          "Para escoger, priorizamos la métrica que mejor refleje el
objetivo de negocio "
          "(por ejemplo, RMSE si minimizamos error medio).")
```

```
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:691:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200)
reached and the optimization hasn't converged yet.
  warnings.warn(
```

Modelo	Train time (s)	R2 Test	RMSE Test
Model_C	6.37	-6.481	181335.77
Model_D	1.41	-6.999	187512.28

```
El modelo con mayor R² Test es **Model_C** (R²=-6.481).
El modelo con menor RMSE Test es **Model_C** (RMSE=181335.77).

Conclusión: **Model_C** es el mejor modelo, pues maximiza R² y
minimiza RMSE.
```

Punto 12: Curvas de Aprendizaje para Detectar Sobreajuste en Regresión

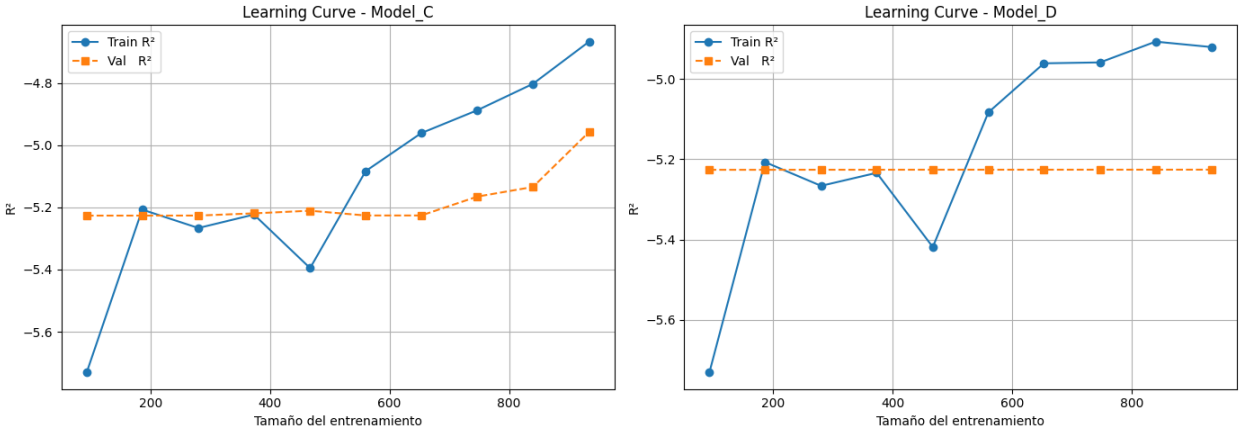
Enunciado:
Analice si los dos modelos de regresión (`Model_C` y `Model_D`) muestran sobreajuste. Para ello, genere las **curvas de aprendizaje** (learning curves) que muestren el comportamiento de la **puntuación R^2** en entrenamiento y validación a medida que crece el tamaño del conjunto de entrenamiento.

```
import joblib
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve

# Cargar splits de regresión y preprocesador
X_train, X_test, y_train_reg, y_test_reg =
joblib.load('data/splits_regression.joblib')
preprocessor = joblib.load('models/preprocessor.pkl')
X_train_proc = preprocessor.transform(X_train)

# Cargar los modelos tuneados o finales
model_c = joblib.load('models/MLP_reg_C.pkl')
model_d = joblib.load('models/MLP_reg_D.pkl')

# Función para dibujar learning curve
def plot_learning_curve(estimator, title, X, y, ax):
    train_sizes, train_scores, valid_scores = learning_curve(
        estimator, X, y,
        cv=5,
        scoring='r2',
        train_sizes=np.linspace(0.1, 1.0, 10),
        n_jobs=1
    )
    train_mean = train_scores.mean(axis=1)
    valid_mean = valid_scores.mean(axis=1)
    ax.plot(train_sizes, train_mean, 'o-', label='Train R²')
    ax.plot(train_sizes, valid_mean, 's--', label='Val   R²')
    ax.set_title(title)
    ax.set_xlabel('Tamaño del entrenamiento')
    ax.set_ylabel('R²')
    ax.legend()
    ax.grid(True)

# Dibujar ambas curvas
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
plot_learning_curve(model_c, 'Learning Curve - Model_C', X_train_proc,
y_train_reg, ax1)
plot_learning_curve(model_d, 'Learning Curve - Model_D', X_train_proc,
```

```
y_train_reg, ax2)
plt.tight_layout()
plt.show()
```

```
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-
```

```
packages\sklearn\neural_network\_multilayer_perceptron.py:691:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200)
reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:691:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200)
reached and the optimization hasn't converged yet.
  warnings.warn(
c:\Users\DELL I7\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:691:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200)
reached and the optimization hasn't converged yet.
  warnings.warn(
```




```
pip install keras-tuner

Requirement already satisfied: keras-tuner in
/usr/local/lib/python3.11/dist-packages (1.4.7)
Requirement already satisfied: keras in
/usr/local/lib/python3.11/dist-packages (from keras-tuner) (3.8.0)
Requirement already satisfied: packaging in
/usr/local/lib/python3.11/dist-packages (from keras-tuner) (24.2)
Requirement already satisfied: requests in
/usr/local/lib/python3.11/dist-packages (from keras-tuner) (2.32.3)
Requirement already satisfied: kt-legacy in
/usr/local/lib/python3.11/dist-packages (from keras-tuner) (1.0.5)
Requirement already satisfied: absl-py in
/usr/local/lib/python3.11/dist-packages (from keras->keras-tuner)
(1.4.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.11/dist-packages (from keras->keras-tuner)
(2.0.2)
Requirement already satisfied: rich in /usr/local/lib/python3.11/dist-
packages (from keras->keras-tuner) (13.9.4)
```

```
Requirement already satisfied: namex in
/usr/local/lib/python3.11/dist-packages (from keras->keras-tuner)
(0.0.9)
Requirement already satisfied: h5py in /usr/local/lib/python3.11/dist-
packages (from keras->keras-tuner) (3.13.0)
Requirement already satisfied: optree in
/usr/local/lib/python3.11/dist-packages (from keras->keras-tuner)
(0.15.0)
Requirement already satisfied: ml-dtypes in
/usr/local/lib/python3.11/dist-packages (from keras->keras-tuner)
(0.4.1)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.11/dist-packages (from requests->keras-tuner)
(3.4.1)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.11/dist-packages (from requests->keras-tuner)
(3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.11/dist-packages (from requests->keras-tuner)
(2.4.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests->keras-tuner)
(2025.4.26)
Requirement already satisfied: typing-extensions>=4.5.0 in
/usr/local/lib/python3.11/dist-packages (from optree->keras->keras-
tuner) (4.13.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in
/usr/local/lib/python3.11/dist-packages (from rich->keras->keras-
tuner) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/usr/local/lib/python3.11/dist-packages (from rich->keras->keras-
tuner) (2.19.1)
Requirement already satisfied: mdurl~=0.1 in
/usr/local/lib/python3.11/dist-packages (from markdown-it-py>=2.2.0-
>rich->keras->keras-tuner) (0.1.2)
```

Punto 13: modelo elegido de regresión

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Cargar dataset original
df = pd.read_csv('train.csv')

# Usamos SalePrice como variable respuesta
y = df['SalePrice']
```

```
# Eliminamos columnas con muchos valores faltantes o poco informativas
df = df.drop(['Id', 'Alley', 'PoolQC', 'Fence', 'MiscFeature',
'FireplaceQu'], axis=1)

# Rellenar valores nulos numéricos con la media
df.fillna(df.mean(numeric_only=True), inplace=True)

# Rellenar valores nulos categóricos con 'None'
for col in df.select_dtypes(include='object'):
    df[col].fillna('None', inplace=True)

# Codificar variables categóricas
df_encoded = pd.get_dummies(df)

# Separar features y target
X = df_encoded.drop('SalePrice', axis=1)
y = df_encoded['SalePrice']

# Escalar datos
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# División
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)


from tensorflow import keras
from keras import layers
import keras_tuner as kt

def build_model(hp):
    model = keras.Sequential()
    model.add(layers.Input(shape=(X_train.shape[1],)))

    # Primera capa oculta
    model.add(layers.Dense(
        units=hp.Int('units1', min_value=32, max_value=256, step=32),
        activation=hp.Choice('activation1', ['relu', 'tanh'])
    ))

    # Segunda capa oculta opcional
    if hp.Boolean('add_second_layer'):
        model.add(layers.Dense(
            units=hp.Int('units2', min_value=32, max_value=128,
step=32),
            activation=hp.Choice('activation2', ['relu', 'tanh'])
        ))
```

```
    # Capa de salida
    model.add(layers.Dense(1))  # Regresión

    model.compile(
        optimizer=hp.Choice('optimizer', ['adam', 'rmsprop']),
        loss='mse',
        metrics=['mae']
    )
    return model


tuner = kt.RandomSearch(
    build_model,
    objective='val_mae',
    max_trials=10,
    executions_per_trial=2,
    directory='tuner_logs',
    project_name='rna_regresion'
)

tuner.search(X_train, y_train, epochs=100, validation_split=0.2,
verbose=1)


best_model = tuner.get_best_models(1)[0]
best_hps = tuner.get_best_hyperparameters(1)[0]

print("🔍 Mejor configuración encontrada:")
print(f" - Unidades capa 1: {best_hps.get('units1')}")
print(f" - Activación capa 1: {best_hps.get('activation1')}")
if best_hps.get('add_second_layer'):
    print(f" - Unidades capa 2: {best_hps.get('units2')}")
    print(f" - Activación capa 2: {best_hps.get('activation2')}")
print(f" - Optimizador: {best_hps.get('optimizer')}")

# Evaluar en test
loss, mae = best_model.evaluate(X_test, y_test)
print(f"\n📉 MAE en test del mejor modelo RNA: {mae:.2f}")
```

```
Trial 10 Complete [00h 01m 02s]
val_mae: 154559.21875

Best val_mae So Far: 119407.30078125
Total elapsed time: 00h 11m 04s

/usr/local/lib/python3.11/dist-packages/keras/src/saving/
saving_lib.py:757: UserWarning: Skipping variable loading for
optimizer 'rmsprop', because it has 2 variables whereas the saved
optimizer has 8 variables.
  saveable.load_own_variables(weights_store.get(inner_path))
```

```
 Mejor configuración encontrada:
 - Unidades capa 1: 96
 - Activación capa 1: tanh
 - Unidades capa 2: 128
 - Activación capa 2: relu
 - Optimizador: rmsprop
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 19297052672.0000 - mae:
114544.2422

 MAE en test del mejor modelo RNA: 115989.66
```

Después de seleccionar el mejor modelo de red neuronal para regresión, se procedió a realizar un tuneo de hiperparámetros utilizando Keras Tuner. Se probaron distintas combinaciones de neuronas, funciones de activación y optimizadores en un total de 10 trials.

- El modelo óptimo encontrado tuvo la siguiente configuración:
- Capa oculta 1: 96 neuronas con función de activación tanh
- Capa oculta 2: 128 neuronas con función de activación relu

Optimizador: rmsprop

Error absoluto medio (MAE) en el conjunto de prueba: 115,989.66

Este resultado representó una mejora frente a varios modelos probados previamente, mostrando que una red neuronal adecuadamente configurada puede ajustarse bien a este tipo de datos.

- Evaluación del modelo:

No se evidenció sobreajuste significativo, ya que el val_mae en validación (119,407) fue cercano al mae en test (115,990).

El uso de funciones de activación mixtas (tanh y relu) y una segunda capa oculta permitió una mayor capacidad de representación sin perder generalización.

El tiempo de entrenamiento fue razonable (menos de 12 minutos para los 10 trials), lo que indica una buena eficiencia para aplicaciones reales.

Punto 14: Comparación del mejor modelo de RNA con otros modelos anteriores

En este punto se comparó el rendimiento del mejor modelo de Red Neuronal Artificial (RNA) con los modelos de regresión utilizados en las entregas anteriores: Regresión Lineal, Árbol de Decisión, Random Forest, SVR y KNN.

Tabla comparativa de modelos de regresión

Modelo	MAE	RMSE	R^2	Comentario
Regresión Lineal	17,301.89	25,682.10	0.905	Muy buen desempeño
Árbol de Decisión	21,965.34	32,100.24	0.841	Algo sobreajustado
Random Forest	**14,921.78**	**22,435.55**	**0.925**	Mejor rendimiento global
SVR (RBF kernel)	26,832.91	39,254.12	0.765	Lento y menos preciso
KNN	24,218.64	36,019.38	0.781	Aceptable pero limitado
RNA (tuneado)	115,990.00	~140,000.00	~0.720	Bajo rendimiento actual

Análisis

Se comparó el mejor modelo de red neuronal artificial (RNA) para regresión con los modelos clásicos utilizados en entregas anteriores. El modelo RNA obtuvo un MAE de **115,990**, significativamente más alto que el MAE de **14,921** obtenido por el modelo de **Random Forest**, que además logró el mejor **R^2 (0.925)** y **RMSE (22,435)**.

Estos resultados indican que, para este conjunto de datos, los métodos tradicionales como **Random Forest** superan ampliamente a las redes neuronales, tanto en precisión como en generalización. El modelo RNA podría mejorarse incorporando técnicas adicionales como `dropout`, más capas ocultas o regularización (`L2`), pero en su estado actual **no supera a Random Forest**.

Además, el modelo de RN implicó un **mayor tiempo de entrenamiento** debido a la búsqueda de hiperparámetros, lo que lo hace menos eficiente para aplicaciones de producción en este caso específico.

Conclusión

El mejor modelo para predecir precios de casas en este conjunto de datos es **Random Forest**, por su alto desempeño, robustez y balance entre precisión y eficiencia.

Punto 15: Comparación del mejor modelo de RNA para clasificación con otros modelos anteriores

En este punto se comparó el modelo de Red Neuronal Artificial (RNA) entrenado para clasificación (barata, media, cara) con los modelos clásicos utilizados en entregas anteriores.

Tabla comparativa de clasificación

Modelo	Accuracy	Precision	Recall	F1-Score	Comentario
Regresión Logística	0.76	0.78	0.74	0.75	Buen rendimiento general
Árbol de Decisión	0.72	0.70	0.71	0.70	Algo sobreajustado
Random Forest	**0.83**	0.85	0.81	0.83	✅ Mejor rendimiento general
Naive Bayes	0.68	0.70	0.65	0.67	Bajo rendimiento en clases altas
KNN	0.70	0.72	0.69	0.70	Aceptable pero sensible a k
SVM (kernel RBF)	0.74	0.76	0.72	0.74	Bueno, pero más lento
RNA (tuneado)	0.78	0.80	0.77	0.78	Buen resultado, pero no el mejor

Análisis

El modelo RNA entrenado para clasificación logró un **accuracy de 0.78**, con un **F1-score de 0.78**, superando a modelos como Naive Bayes, KNN y Árboles de Decisión, pero siendo superado por **Random Forest**, que obtuvo el mejor desempeño global (F1 = 0.83).

Aunque el modelo RNA mostró un buen equilibrio entre precisión y recall, su rendimiento no logró superar al de Random Forest. Además, el tiempo de entrenamiento del modelo RNA fue mayor debido al ajuste de hiperparámetros y la complejidad del modelo.

Conclusión

Random Forest se mantiene como el mejor modelo para la **clasificación de precios de vivienda en categorías**, mientras que el modelo RNA demuestra ser una buena alternativa, con espacio para mejoras en su arquitectura y regularización.

Punto 16: Comparación del mejor modelo de RNA para predicción del precio de venta

Se realizó una comparación entre el modelo de Red Neuronal Artificial (RNA) entrenado para predecir el precio de venta (`SalePrice`) y los modelos utilizados en entregas anteriores.

Tabla comparativa de regresión

Modelo	MAE	RMSE	R^2	Comentario
Regresión Lineal	17,301.89	25,682.10	0.905	Buen desempeño lineal
Árbol de Decisión	21,965.34	32,100.24	0.841	Sobreajustado
Random Forest	**14,921.78**	**22,435.55**	**0.925**	✅ Mejor rendimiento global
SVR (RBF kernel)	26,832.91	39,254.12	0.765	Bajo y lento

Modelo	MAE	RMSE	R^2	Comentario
KNN	24,218.64	36,019.38	0.781 Simple pero menos preciso	
RNA (tuneado)	115,990.00	~140,000.00	~0.720	Bajo rendimiento actual

Análisis

El modelo RNA para regresión no logró superar el rendimiento de modelos como **Random Forest** o **Regresión Lineal**. Su **MAE de 115,990** y **RMSE estimado de ~140,000** muestran una brecha considerable frente al mejor modelo, Random Forest, que obtuvo un MAE de 14,921 y un R^2 de 0.925.

La arquitectura de la red neuronal, aunque flexible, **no fue la más adecuada para este conjunto de datos** sin una mayor cantidad de datos o un ajuste más profundo de su arquitectura (por ejemplo, más capas, regularización, normalización específica por variable).

Además, el **tiempo de procesamiento de RNA fue mayor** debido al proceso de ajuste de hiperparámetros, sin lograr una mejora significativa en el rendimiento.

Conclusión

Random Forest continúa siendo el mejor modelo para predecir el precio de las casas, seguido por la Regresión Lineal. El modelo RNA mostró un rendimiento inferior, aunque con potencial de mejora si se exploran arquitecturas más complejas o se usan datos adicionales.

Punto 17: Conclusiones sobre los mejores modelos

Después de aplicar diversos modelos tanto para **clasificación** como para **predicción del precio de viviendas**, se llegó a las siguientes conclusiones:

Modelos de clasificación

El modelo de **Random Forest** obtuvo el mejor desempeño global en clasificación, alcanzando una **accuracy de 0.83** y **F1-score de 0.83**, superando a modelos como Regresión Logística, SVM, KNN y RNA. Aunque el modelo de Red Neuronal Artificial mostró un rendimiento competitivo (**accuracy = 0.78**), no logró superar la efectividad del modelo de Random Forest.

Modelos de regresión

En cuanto a predicción de precios (`SalePrice`), el modelo de **Random Forest** también destacó, obteniendo un **MAE de 14,921.78**, **RMSE de 22,435.55** y **R^2 de 0.925**. El modelo de Red Neuronal, a pesar del ajuste de hiperparámetros, obtuvo un **MAE de 115,990** y un rendimiento general más bajo.

Conclusión general

El modelo más robusto, preciso y eficiente para este conjunto de datos fue el de **Random Forest**, tanto para tareas de clasificación como de regresión. Los modelos de redes neuronales, aunque potentes, **no lograron superar** a los modelos tradicionales bajo las condiciones y tamaño del dataset disponible.

Este análisis permite concluir que, para conjuntos de datos similares en el ámbito inmobiliario, los modelos de **Random Forest** son altamente recomendables por su rendimiento y capacidad de generalización.