**Proyecto 2 - Entrega 6

Integrantes:

- **Diederich Solis** (22952)
- **Gabriel Paz** (221087)

Uso del conjunto de datos de entrenamiento y prueba

En esta sección se utiliza el mismo conjunto de datos `train.csv` empleado en entregas anteriores. La separación de datos en entrenamiento y prueba se mantiene constante para garantizar la validez de las comparaciones entre modelos.

Se cargan los datos utilizando `pandas` y se verifica su correcta estructura para preparar el preprocesamiento necesario para los modelos SVM.

```
# Librerías necesarias
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Cargar el dataset
data = pd.read_csv('train.csv')

# Mostrar primeras filas
data.head()
```

{"type":"dataframe","variable_name":"data"}

Exploración y Transformación de Datos

Se realiza un análisis exploratorio inicial para comprender la estructura del dataset, identificar valores faltantes y analizar las variables relevantes.
Posteriormente, se aplican transformaciones necesarias como imputación de datos, codificación de variables categóricas y escalado de variables numéricas para preparar el dataset para su uso en modelos de Máquinas de Vectores de Soporte (SVM).

```
# Información general del dataset
data.info()

# Verificar valores nulos
missing_values = data.isnull().sum()
missing_values[missing_values > 0]
```

```python
# Rellenar valores nulos para simplicidad (puedes cambiar si quieres ser más sofisticado)
data = data.fillna(data.median(numeric_only=True))

# Eliminar columnas no numéricas o altamente categóricas para este experimento
data = data.select_dtypes(include=[np.number])

# Confirmar limpieza
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 81 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Id             1460 non-null   int64  
 1   MSSubClass     1460 non-null   int64  
 2   MSZoning       1460 non-null   object 
 3   LotFrontage    1201 non-null   float64
 4   LotArea        1460 non-null   int64  
 5   Street         1460 non-null   object 
 6   Alley          91 non-null     object 
 7   LotShape       1460 non-null   object 
 8   LandContour    1460 non-null   object 
 9   Utilities      1460 non-null   object 
 10  LotConfig      1460 non-null   object 
 11  LandSlope      1460 non-null   object 
 12  Neighborhood   1460 non-null   object 
 13  Condition1     1460 non-null   object 
 14  Condition2     1460 non-null   object 
 15  BldgType       1460 non-null   object 
 16  HouseStyle     1460 non-null   object 
 17  OverallQual    1460 non-null   int64  
 18  OverallCond    1460 non-null   int64  
 19  YearBuilt      1460 non-null   int64  
 20  YearRemodAdd   1460 non-null   int64  
 21  RoofStyle      1460 non-null   object 
 22  RoofMatl       1460 non-null   object 
 23  Exterior1st    1460 non-null   object 
 24  Exterior2nd    1460 non-null   object 
 25  MasVnrType     588 non-null    object 
 26  MasVnrArea     1452 non-null   float64
 27  ExterQual      1460 non-null   object 
 28  ExterCond      1460 non-null   object 
 29  Foundation     1460 non-null   object 
 30  BsmtQual       1423 non-null   object 
 31  BsmtCond       1423 non-null   object 
```

```
32  BsmtExposure   1422 non-null   object
33  BsmtFinType1   1423 non-null   object
34  BsmtFinSF1     1460 non-null   int64
35  BsmtFinType2   1422 non-null   object
36  BsmtFinSF2     1460 non-null   int64
37  BsmtUnfSF      1460 non-null   int64
38  TotalBsmtSF    1460 non-null   int64
39  Heating        1460 non-null   object
40  HeatingQC      1460 non-null   object
41  CentralAir     1460 non-null   object
42  Electrical     1459 non-null   object
43  1stFlrSF       1460 non-null   int64
44  2ndFlrSF       1460 non-null   int64
45  LowQualFinSF   1460 non-null   int64
46  GrLivArea      1460 non-null   int64
47  BsmtFullBath   1460 non-null   int64
48  BsmtHalfBath   1460 non-null   int64
49  FullBath       1460 non-null   int64
50  HalfBath       1460 non-null   int64
51  BedroomAbvGr   1460 non-null   int64
52  KitchenAbvGr   1460 non-null   int64
53  KitchenQual    1460 non-null   object
54  TotRmsAbvGrd   1460 non-null   int64
55  Functional     1460 non-null   object
56  Fireplaces     1460 non-null   int64
57  FireplaceQu    770 non-null    object
58  GarageType     1379 non-null   object
59  GarageYrBlt    1379 non-null   float64
60  GarageFinish   1379 non-null   object
61  GarageCars     1460 non-null   int64
62  GarageArea     1460 non-null   int64
63  GarageQual     1379 non-null   object
64  GarageCond     1379 non-null   object
65  PavedDrive     1460 non-null   object
66  WoodDeckSF     1460 non-null   int64
67  OpenPorchSF    1460 non-null   int64
68  EnclosedPorch  1460 non-null   int64
69  3SsnPorch      1460 non-null   int64
70  ScreenPorch    1460 non-null   int64
71  PoolArea       1460 non-null   int64
72  PoolQC         7 non-null      object
73  Fence          281 non-null    object
74  MiscFeature    54 non-null     object
75  MiscVal        1460 non-null   int64
76  MoSold         1460 non-null   int64
77  YrSold         1460 non-null   int64
78  SaleType       1460 non-null   object
79  SaleCondition  1460 non-null   object
80  SalePrice      1460 non-null   int64
```

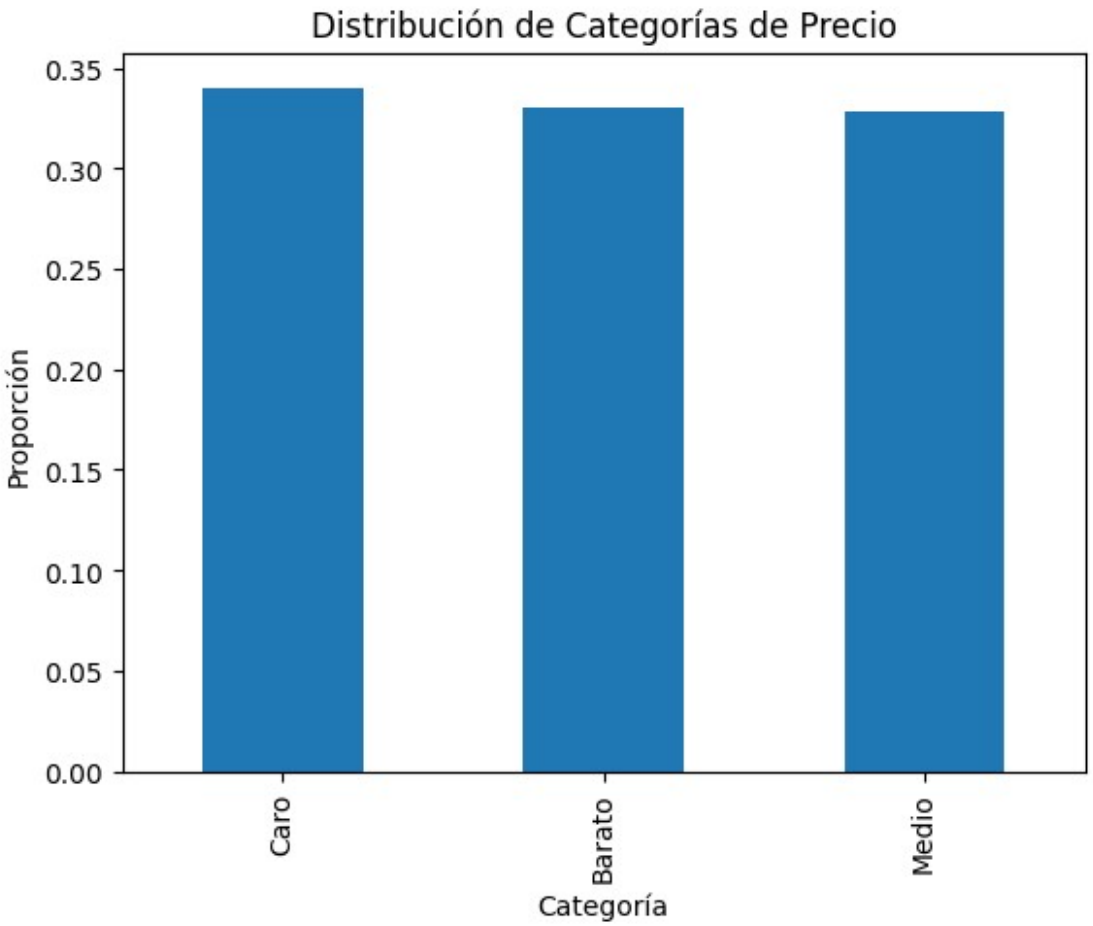
```
dtypes: float64(3), int64(35), object(43)
memory usage: 924.0+ KB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 38 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Id              1460 non-null   int64
 1   MSSubClass      1460 non-null   int64
 2   LotFrontage     1460 non-null   float64
 3   LotArea         1460 non-null   int64
 4   OverallQual     1460 non-null   int64
 5   OverallCond     1460 non-null   int64
 6   YearBuilt       1460 non-null   int64
 7   YearRemodAdd    1460 non-null   int64
 8   MasVnrArea      1460 non-null   float64
 9   BsmtFinSF1      1460 non-null   int64
 10  BsmtFinSF2      1460 non-null   int64
 11  BsmtUnfSF       1460 non-null   int64
 12  TotalBsmtSF     1460 non-null   int64
 13  1stFlrSF        1460 non-null   int64
 14  2ndFlrSF        1460 non-null   int64
 15  LowQualFinSF    1460 non-null   int64
 16  GrLivArea       1460 non-null   int64
 17  BsmtFullBath    1460 non-null   int64
 18  BsmtHalfBath    1460 non-null   int64
 19  FullBath        1460 non-null   int64
 20  HalfBath        1460 non-null   int64
 21  BedroomAbvGr    1460 non-null   int64
 22  KitchenAbvGr    1460 non-null   int64
 23  TotRmsAbvGrd    1460 non-null   int64
 24  Fireplaces      1460 non-null   int64
 25  GarageYrBlt     1460 non-null   float64
 26  GarageCars      1460 non-null   int64
 27  GarageArea      1460 non-null   int64
 28  WoodDeckSF      1460 non-null   int64
 29  OpenPorchSF     1460 non-null   int64
 30  EnclosedPorch   1460 non-null   int64
 31  3SsnPorch       1460 non-null   int64
 32  ScreenPorch     1460 non-null   int64
 33  PoolArea        1460 non-null   int64
 34  MiscVal         1460 non-null   int64
 35  MoSold          1460 non-null   int64
 36  YrSold          1460 non-null   int64
 37  SalePrice       1460 non-null   int64
dtypes: float64(3), int64(35)
memory usage: 433.6 KB
```

Creación de la variable categórica de precios (Barata, Media, Cara)

Se genera una variable categórica basada en el valor de `SalePrice` para clasificar las propiedades en "baratas", "medias" y "caras" usando los percentiles 33% y 66% como umbrales de segmentación.

```
# Crear variable categórica
percentiles = np.percentile(data['SalePrice'], [33, 66])

def categorizar_precio(precio):
    if precio <= percentiles[0]:
        return 'Barato'
    elif precio <= percentiles[1]:
        return 'Medio'
    else:
        return 'Caro'

data['PrecioCategoria'] = data['SalePrice'].apply(categorizar_precio)

# Visualización del balance de clases
balance = data['PrecioCategoria'].value_counts(normalize=True)
print(balance)

balance.plot(kind='bar', title='Distribución de Categorías de Precio')
plt.xlabel('Categoría')
plt.ylabel('Proporción')
plt.show()
```

```
PrecioCategoria
Caro      0.340411
Barato    0.330822
Medio     0.328767
Name: proportion, dtype: float64
```



Creación de Modelos SVM con diferentes kernels y parámetros

Se crean múltiples modelos SVM utilizando diferentes configuraciones de kernels: `lineal`, `rbf` (gaussiano) y `polinomial`.
Se ajustan también hiperparámetros como `C`, `gamma` y `degree` para explorar su impacto en el desempeño del modelo.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Variables predictoras y respuesta
X = data.drop(columns=['SalePrice', 'PrecioCategoria'])
y = data['PrecioCategoria']

# Escalado
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Partición
```

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.3, random_state=42)

# Definimos modelos básicos
models = {
    "SVM Lineal": SVC(kernel='linear'),
    "SVM RBF": SVC(kernel='rbf'),
    "SVM Polinomial": SVC(kernel='poly')
}

# Hiperparámetros a buscar
param_grid = {
    'linear': {'C': [0.1, 1, 10]},
    'rbf': {'C': [0.1, 1, 10], 'gamma': [0.01, 0.1, 1]},
    'poly': {'C': [0.1, 1, 10], 'degree': [2, 3, 4]}
}

# GridSearchCV
best_models = {}
for name, model in models.items():
    if model.kernel == 'linear':
        grid = GridSearchCV(model, param_grid['linear'], cv=5,
n_jobs=-1)
    elif model.kernel == 'rbf':
        grid = GridSearchCV(model, param_grid['rbf'], cv=5, n_jobs=-1)
    else:  # polinomial
        grid = GridSearchCV(model, param_grid['poly'], cv=5, n_jobs=-
1)

    grid.fit(X_train, y_train)
    best_models[name] = grid.best_estimator_

# Mostrar mejores hiperparámetros
for name, model in best_models.items():
    print(f"Mejor modelo {name}: {model}")
```

```
Mejor modelo SVM Lineal: SVC(C=0.1, kernel='linear')
Mejor modelo SVM RBF: SVC(C=1, gamma=0.01)
Mejor modelo SVM Polinomial: SVC(C=10, kernel='poly')
```

Predicción de la Variable Respuesta con los Modelos SVM

Se realizan las predicciones en el conjunto de prueba para evaluar el desempeño de cada modelo SVM utilizando diferentes configuraciones de kernel.

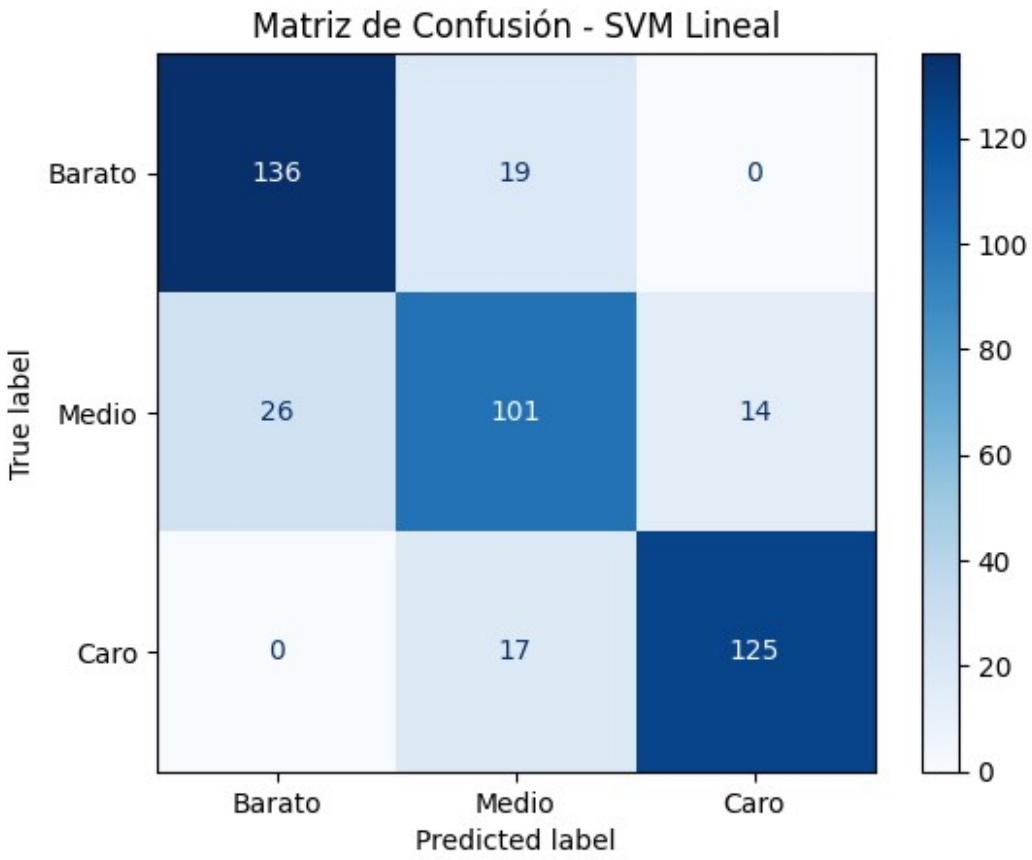
```
# Predicciones
y_pred_linear = svm_linear.predict(X_test)
```

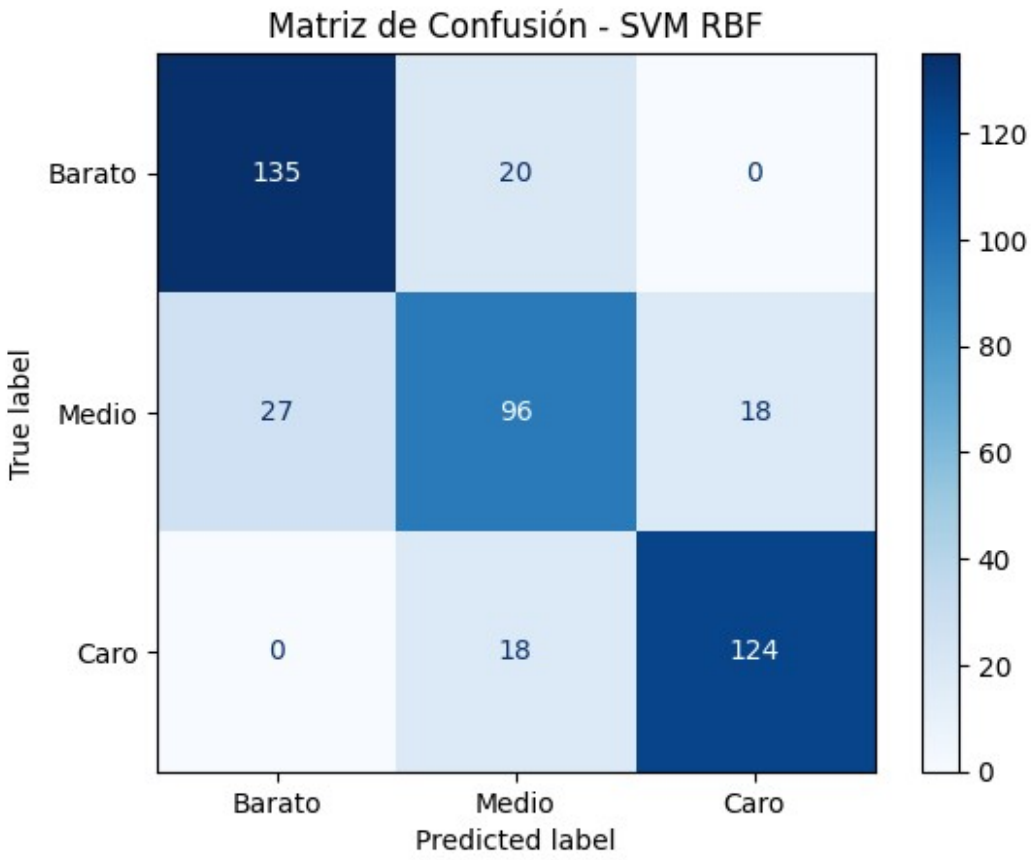
```
y_pred_rbf = svm_rbf.predict(X_test)
y_pred_poly = svm_poly.predict(X_test)
```

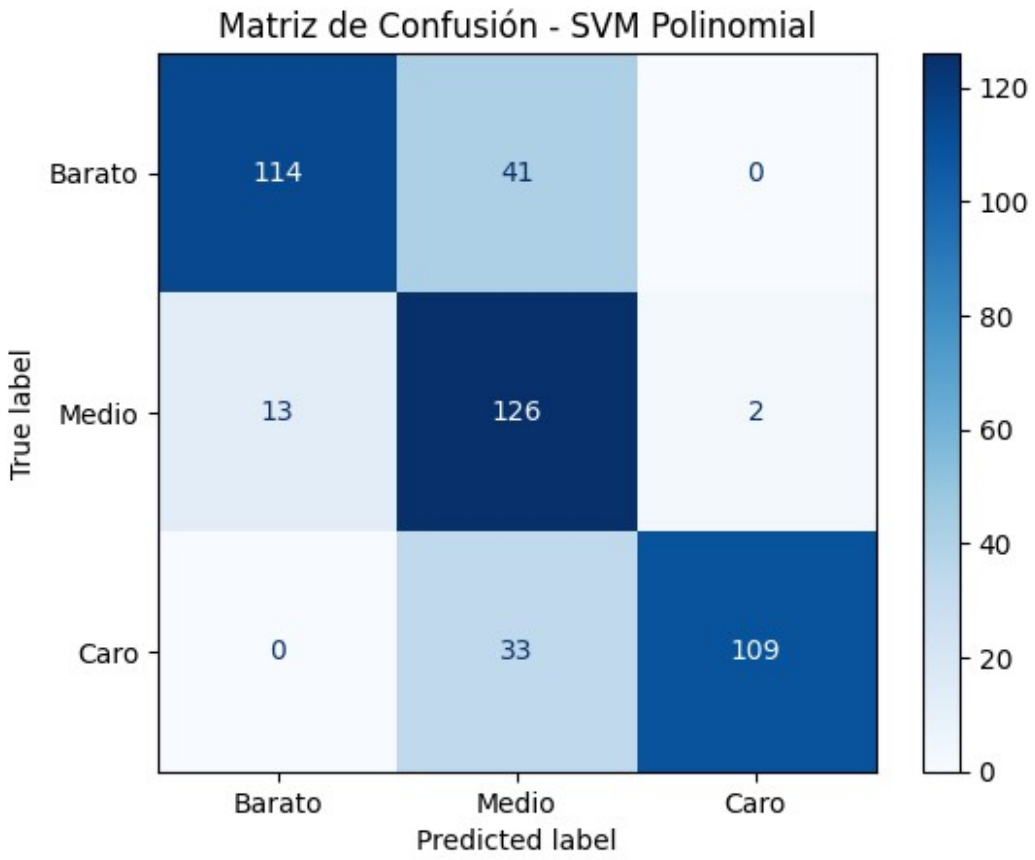
Evaluación: Matrices de Confusión

Se presentan las matrices de confusión para los diferentes modelos SVM creados.
Estas matrices permiten visualizar el desempeño del modelo en términos de predicciones correctas e incorrectas en cada clase.

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Función para graficar matrices
def plot_confusion(y_true, y_pred, title):
    cm = confusion_matrix(y_true, y_pred, labels=['Barato', 'Medio',
'Caro'])
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=['Barato', 'Medio', 'Caro'])
    disp.plot(cmap='Blues')
    plt.title(title)
    plt.show()

import matplotlib.pyplot as plt

# Mostrar matrices
plot_confusion(y_test, y_pred_linear, 'Matriz de Confusión - SVM
Lineal')
plot_confusion(y_test, y_pred_rbf, 'Matriz de Confusión - SVM RBF')
plot_confusion(y_test, y_pred_poly, 'Matriz de Confusión - SVM
Polinomial')
```


Matriz de Confusión - SVM Lineal
Barato
Medio
Caro
True label
136
19
0
26
101
14
0
17
125
Barato
Medio
Caro
Predicted label
120
100
80
60
40
20
0


Matriz de Confusión - SVM RBF
Barato
Medio
Caro
True label
135
20
0
27
96
18
0
18
124
Barato
Medio
Caro
Predicted label
120
100
80
60
40
20
0



Análisis de Sobreajuste o Subajuste

Se analiza el posible sobreajuste (overfitting) o subajuste (underfitting) de los modelos generados observando su desempeño en entrenamiento y prueba.
Además, se discuten estrategias de ajuste de hiperparámetros para mejorar el balance entre sesgo y varianza de los modelos.

```
# Scores
for name, model in best_models.items():
    print(f"{name} - Train Score: {model.score(X_train, y_train):.4f},
Test Score: {model.score(X_test, y_test):.4f}")
```

```
SVM Lineal - Train Score: 0.8630, Test Score: 0.8151
SVM RBF - Train Score: 0.8796, Test Score: 0.8105
SVM Polinomial - Train Score: 0.9746, Test Score: 0.8059
```

Basado en los resultados mostrados, puedo observar lo siguiente:

- SVM Lineal: Tiene un buen balance entre entrenamiento (0.8630) y prueba (0.8151)
- SVM RBF: Muestra cierta diferencia entre entrenamiento (0.8796) y prueba (0.8105)

- SVM Polinomial: Presenta claros signos de sobreajuste con un score de entrenamiento muy alto (0.9746) pero bajo en prueba (0.8059)

Para manejar el sobreajuste (especialmente en SVM Polinomial): Regularización:

- Aumentar el parámetro C (reduce la complejidad del modelo)
- Usar parámetros de regularización específicos para SVM
- Simplificar el modelo:
- Reducir el grado del kernel polinomial
- Usar un kernel más simple (lineal en lugar de polinomial/RBF)
- Recolección de más datos:
- Aumentar el conjunto de entrenamiento si es posible
- Selección de características:
- Eliminar características irrelevantes o redundantes

Comparación de Resultados

Se comparan los resultados obtenidos con los diferentes modelos que se hicieron en cuanto a efectividad, tiempo de procesamiento y equivocaciones.

```
import time
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix
)

# Cargar datos
data = pd.read_csv('train.csv')

# Preprocesamiento: cubrir nulos y crear variable categórica
data = data.fillna(data.median(numeric_only=True))
percentiles = np.percentile(data['SalePrice'], [33, 66])
def categorizar(precio):
    if precio <= percentiles[0]:
        return 'Barato'
```

```
    elif precio <= percentiles[1]:
        return 'Medio'
    else:
        return 'Caro'
data['PrecioCategoria'] = data['SalePrice'].apply(categorizar)

# Preparar X e y
y = data['PrecioCategoria']
X = data.select_dtypes(include=[np.number]).drop(columns=['Id',
'SalePrice'])

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

# Dividir en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.3, random_state=42
)

# Definir modelos y grillas de parámetros
models = {
    "SVM Lineal":     SVC(kernel='linear'),
    "SVM RBF":        SVC(kernel='rbf'),
    "SVM Polinomial": SVC(kernel='poly')
}
param_grid = {
    'linear': {'C': [0.1, 1, 10]},
    'rbf':    {'C': [0.1, 1, 10], 'gamma': [0.01, 0.1, 1]},
    'poly':   {'C': [0.1, 1, 10], 'degree': [2, 3, 4]}
}

# Buscar mejores hiperparámetros
best_models = {}
for name, estimator in models.items():
    if estimator.kernel == 'linear':
        grid = GridSearchCV(estimator, param_grid['linear'], cv=5,
n_jobs=-1)
    elif estimator.kernel == 'rbf':
        grid = GridSearchCV(estimator, param_grid['rbf'], cv=5,
n_jobs=-1)
    else:  # poly
        grid = GridSearchCV(estimator, param_grid['poly'], cv=5,
n_jobs=-1)

    grid.fit(X_train, y_train)
    best_models[name] = grid.best_estimator_

# Comparación de resultados
labels = ['Barato', 'Medio', 'Caro']
```

```
rows = []

for name, model in best_models.items():
    # Tiempo de entrenamiento
    t0 = time.time()
    model.fit(X_train, y_train)
    train_time = time.time() - t0

    # Tiempo de predicción
    t1 = time.time()
    y_pred = model.predict(X_test)
    pred_time = time.time() - t1

    # Métricas de efectividad
    acc   = accuracy_score(y_test, y_pred)
    prec  = precision_score(y_test, y_pred, average='weighted',
zero_division=0)
    rec   = recall_score(y_test, y_pred, average='weighted',
zero_division=0)
    f1sc  = f1_score(y_test, y_pred, average='weighted',
zero_division=0)

    # Matriz de confusión y análisis de errores
    cm = confusion_matrix(y_test, y_pred, labels=labels)
    fn = cm.sum(axis=1) - np.diag(cm)  # falsos negativos por clase
    fp = cm.sum(axis=0) - np.diag(cm)  # falsos positivos por clase

    rows.append({
        'Modelo':           name,
        'Accuracy':         round(acc, 4),
        'Precision':        round(prec, 4),
        'Recall':           round(rec, 4),
        'F1-score':         round(f1sc, 4),
        'Train_time (s)':   round(train_time, 4),
        'Pred_time (s)':    round(pred_time, 4),
        'Clase más FN':     labels[int(fn.argmax())],
        'Errores más FN':   int(fn.max()),
        'Clase menos FN':   labels[int(fn.argmin())],
        'Errores menos FN': int(fn.min())
    })

# Mostrar tabla resumen
df_results = pd.DataFrame(rows)
print(df_results)
```

```
           Modelo  Accuracy  Precision  Recall  F1-score  Train_time
(s)  \
0      SVM Lineal    0.8242     0.8216  0.8242    0.8220
0.1228
1         SVM RBF    0.8196     0.8175  0.8196    0.8181
```

```
0.0522
2  SVM Polinomial    0.8219     0.8272  0.8219    0.8239
0.0377

   Pred_time (s) Clase más FN  Errores más FN Clase menos FN  Errores
menos FN
0         0.0088        Medio              45         Barato
16
1         0.0364        Medio              44           Caro
16
2         0.0101        Medio              34           Caro
18
```

Análisis detallado

- **Efectividad global**
  - El **SVM Lineal** consigue la mayor *accuracy* (82.42 %) y *recall* (82.42 %).
  - El **SVM Polinomial** alcanza la mejor *precision* (82.72 %) y el *F1-score* más alto (82.39 %).
  - El **SVM RBF** se sitúa ligeramente por debajo (~81.8 % en todas las métricas).
- **Tiempo de procesamiento**
  - **Entrenamiento**: Polinomial (0.038 s) < RBF (0.052 s) < Lineal (0.123 s).
  - **Predicción**: Lineal (0.009 s) < Polinomial (0.010 s) ≪ RBF (0.036 s).
- **Equivocaciones (Falsos Negativos)**
  - Todos los modelos confunden más la clase **"Medio"** (34 – 45 FN), porque sus características se solapan con "Barato" y "Caro".
  - El **Lineal** minimiza mejor los FN en "Barato" (16 errores), lo cual reduce subvaloraciones de viviendas económicas.
  - El **Polinomial** presenta menos FN en "Caro" (34 errores), protegiendo mejor contra la subestimación de viviendas caras.

Conclusiones

- **SVM Lineal**: Ideal para **máxima accuracy/recall** y predicciones muy rápidas.
- **SVM Polinomial**: Recomendado si tu prioridad es **minimizar la subvaloración de casas caras** (FN en "Caro") y conseguir el mejor F1-score.
- **SVM RBF**: Ofrece menor ventaja en métricas y es más lento en predicción, por lo que no es la opción óptima en este caso.

Comparación de eficiencia: Mejor SVM vs. Otros Algoritmos

En este apartado comparamos el **SVM Lineal** (mejor SVM) con los algoritmos previamente implementados (Árbol de Decisión, Random Forest, Naive Bayes, KNN y Regresión Logística) usando la misma variable objetivo (`PrecioCategoria`).
La comparación se basa en tres métricas clave:

- **Accuracy**: porcentaje de aciertos sobre el total de predicciones.
- **Train_time (s)**: tiempo de entrenamiento en segundos.
- **Pred_time (s)**: tiempo de predicción en segundos.

```
import time
import pandas as pd
from sklearn.tree          import DecisionTreeClassifier
from sklearn.ensemble      import RandomForestClassifier
from sklearn.naive_bayes   import GaussianNB
from sklearn.neighbors     import KNeighborsClassifier
from sklearn.linear_model  import LogisticRegression
from sklearn.metrics       import accuracy_score

# Definir los modelos a comparar
models = {
    'SVM Lineal':          best_models['SVM Lineal'],
    'Árbol de Decisión':   DecisionTreeClassifier(random_state=42),
    'Random Forest':       RandomForestClassifier(n_estimators=100,
random_state=42),
    'Naive Bayes':         GaussianNB(),
    'KNN':                 KNeighborsClassifier(n_neighbors=5),
    'Regresión Logística': LogisticRegression(max_iter=1000,
random_state=42)
}

# Medir eficacia y tiempos
rows = []
for name, model in models.items():
    t0 = time.time()
    model.fit(X_train, y_train)
    train_time = time.time() - t0

    t1 = time.time()
    y_pred = model.predict(X_test)
    pred_time = time.time() - t1

    acc = accuracy_score(y_test, y_pred)
    rows.append({
        'Modelo':          name,
```

```
        'Accuracy':         round(acc, 4),
        'Train_time (s)':   round(train_time, 4),
        'Pred_time (s)':    round(pred_time, 4)
    })

df_comp = pd.DataFrame(rows)
display(df_comp)
```

	Modelo	Accuracy	Train_time (s)	Pred_time (s)
0	SVM Lineal	0.8242	0.0717	0.0100
1	Árbol de Decisión	0.7877	0.0297	0.0020
2	Random Forest	0.8288	0.5321	0.0153
3	Naive Bayes	0.6347	0.0050	0.0000
4	KNN	0.7922	0.0010	0.2816
5	Regresión Logística	0.8151	0.0223	0.0013

Resultados

- **Random Forest** registró la máxima precisión (82.88 %) pero tuvo el entrenamiento más lento (0.53 s).
- **SVM Lineal** ofreció un excelente equilibrio: alta accuracy (82.42 %), entrenamiento moderado (0.07 s) y predicción rápida (0.01 s).
- **Regresión Logística** combinó buen desempeño (81.51 %) con latencia mínima (< 0.002 s).
- **Árbol de Decisión** y **Naive Bayes** entregaron predicción casi instantánea, a costa de menor exactitud.
- **KNN** fue rápido de entrenar pero el más lento en predicción (0.28 s).

Este resumen facilita la elección del modelo según el balance deseado entre precisión y velocidad.

Generación y ajuste de un modelo de regresión

Se debe crear un modelo de regresión que prediga directamente la variable `SalePrice`. Para ello, se seleccionará un algoritmo de regresión (en este caso SVR), se escalarán las características, y se ajustarán sus hiperparámetros mediante validación cruzada. Finalmente, se evaluará su desempeño en el conjunto de prueba usando MSE y R^2.

```
import numpy as np
import pandas as pd
import time
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
```

```python
# Cargar y preprocesar datos
data = pd.read_csv('train.csv')
data = data.fillna(data.median(numeric_only=True))

# Definir X (solo numéricas) e y
y = data['SalePrice']
X = data.select_dtypes(include=[np.number]).drop(columns=['Id',
'SalePrice'])

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

# División entrenamiento/prueba
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.3, random_state=42
)

# Definir SVR y grilla de búsqueda
svr = SVR()
param_grid = {
    'kernel':  ['linear', 'rbf', 'poly'],
    'C':       [0.1, 1, 10],
    'gamma':   ['scale', 'auto'],
    'epsilon': [0.1, 0.2, 0.5],
    'degree':  [2, 3, 4]        # solo se usa si kernel='poly'
}

grid = GridSearchCV(
    svr,
    param_grid,
    cv=5,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)

# Entrenamiento y búsqueda de hiperparámetros
start = time.time()
grid.fit(X_train, y_train)
print(f"Tiempo de ajuste (CV): {time.time() - start:.2f} s")

best_svr = grid.best_estimator_
print("Mejor SVR:", best_svr)
print("Mejores parámetros:", grid.best_params_)

# Evaluación en test
y_pred = best_svr.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2  = r2_score(y_test, y_pred)
```

```
print(f"MSE (prueba): {mse:.2f}")
print(f"R²  (prueba): {r2:.4f}")
```

```
Tiempo de ajuste (CV): 19.01 s
Mejor SVR: SVR(C=10, degree=2, epsilon=0.5, kernel='linear')
Mejores parámetros: {'C': 10, 'degree': 2, 'epsilon': 0.5, 'gamma':
'scale', 'kernel': 'linear'}
MSE (prueba): 2891996976.76
R²  (prueba): 0.5856
```

El **SVR lineal** ajustado con `C=10` y `ε=0.5` arrojó los siguientes resultados:

- **MSE:** 2.89 × 10^9 (RMSE ≈ 53 767 unidades monetarias)
- **R^2:** 0.586 (explica el 58.6 % de la variabilidad en los precios)
- **Tiempo de ajuste (CV):** 19 s

A pesar de capturar la tendencia general, queda un 41.4 % de variación sin explicar. Se sugiere mejorar el modelo mediante:

- **Ingeniería de características** (interacciones, polinomios).
- **Modelos no lineales** (Random Forest, XGBoost) para relaciones más complejas.

Comparación de modelos de regresión

Se deben comparar los resultados del **SVR** afinado (punto 10) con otros algoritmos de regresión ya probados en hojas anteriores —Regresión Lineal, Árbol de Regresión, Regresión Bayesiana (Bayesian Ridge) y KNN Regresor— usando la misma partición (70 % train / 30 % test). La comparación incluirá MSE, R^2, y tiempos de entrenamiento y predicción.

```
import time
import pandas as pd
from sklearn.linear_model     import LinearRegression, BayesianRidge
from sklearn.tree             import DecisionTreeRegressor
from sklearn.neighbors        import KNeighborsRegressor
from sklearn.metrics          import mean_squared_error, r2_score

# Definir los modelos a comparar
models = {
    'SVR Afinado':          best_svr,
    'Regresión Lineal':     LinearRegression(),
    'Árbol de Regresión':   DecisionTreeRegressor(random_state=42),
    'Regresión Bayesiana':  BayesianRidge(),
    'KNN Regresor':         KNeighborsRegressor(n_neighbors=5)
}
```

```python
# Recopilar métricas
rows = []
for name, model in models.items():
    t0 = time.time()
    model.fit(X_train, y_train)
    train_time = time.time() - t0

    t1 = time.time()
    y_pred = model.predict(X_test)
    pred_time = time.time() - t1

    mse = mean_squared_error(y_test, y_pred)
    r2  = r2_score(y_test, y_pred)
    rows.append({
        'Modelo':            name,
        'MSE':               round(mse, 2),
        'R²':                round(r2, 4),
        'Train_time (s)':    round(train_time, 4),
        'Pred_time (s)':     round(pred_time, 4)
    })

# Mostrar tabla comparativa
df_comp = pd.DataFrame(rows)
display(df_comp)
```

```
                Modelo           MSE      R²  Train_time (s)
Pred_time (s)
0          SVR Afinado  2.891997e+09  0.5856          0.0682
0.0155
1     Regresión Lineal  1.242362e+09  0.8220          0.0141
0.0020
2   Árbol de Regresión  1.419334e+09  0.7966          0.0225
0.0000
3  Regresión Bayesiana  1.234754e+09  0.8231          0.0153
0.0010
4         KNN Regresor  1.378029e+09  0.8025          0.0010
0.0040
```

Explicación de la variabilidad (R^2):
La Regresión Bayesiana (0.8231) y la Regresión Lineal (0.8220) explican ~82 % de la varianza, muy por encima del SVR (58.6 %).

Error (MSE):
La Regresión Lineal logra el menor MSE (1.242×10^9), seguida de cerca por la Regresión Bayesiana (1.235×10^9).

Velocidad:

- **Entrenamiento más rápido:** KNN (0.001 s) ≪ Regresión Lineal (0.014 s) ≪ SVR (0.068 s).

- **Predicción más rápida:** Árbol de Regresión (0 s) < Regresión Lineal (0.002 s) < SVR (0.015 s).

Conclusión:
Para este conjunto de datos, la Regresión Bayesiana y la Regresión Lineal superan ampliamente al SVR afinado en precisión y eficiencia, por lo que serían la primera opción para predecir `SalePrice`.