3. Previsão da nota do IMDb

3.0 Imports:

In [157…

```
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import OneHotEncoder
from pathlib import Path
import joblib, sklearn, sys
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.inspection import permutation_importance
import xgboost as xgb
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import root_mean_squared_error



df = pd.read_csv('..\data\processed\df_eda01_plus_5000.csv')
```

```
<>:20: SyntaxWarning: invalid escape sequence '\d'
<>:20: SyntaxWarning: invalid escape sequence '\d'
C:\Users\guima\AppData\Local\Temp\ipykernel_3740\228565998.py:20: SyntaxWarning: invalid escape s
equence '\d'
  df = pd.read_csv('..\data\processed\df_eda01_plus_5000.csv')
```

3.1 Definição do problema

O objetivo é prever a nota do IMDb (**IMDB_Rating**) a partir das demais variáveis disponíveis.

- Como a variável alvo é **contínua (0–10)**, trata-se de um **problema de regressão**.
- Opcionalmente, poderíamos transformar a nota em faixas (ruim/médio/bom/excelente), mas isso implicaria em um **problema de classificação ordinal**, com perda de informação.

Modelos de Regressão considerados

Para esse tipo de tarefa, alguns modelos de regressão supervisionada podem ser aplicados:

- **Regressão Linear**
  Simples e interpretável, útil como baseline. Porém, pode não capturar relações não lineares entre as variáveis.

- **Árvore de Decisão / Random Forest Regressor**
  Captura relações mais complexas, lida bem com variáveis categóricas (via encoding) e não exige normalização. Bom para dados tabulares.

- **Gradient Boosting (XGBoost, LightGBM, HistGradientBoosting)**
  Combina várias árvores de forma sequencial para reduzir o erro residual. Em geral apresenta excelente performance em dados tabulares.
  Observação: **HistGradientBoosting** é a implementação nativa do scikit-learn, otimizada por histogramas (rápida e integrada ao ecossistema sklearn).

- **CatBoost**
  Método de boosting que trata variáveis categóricas de forma mais eficiente (estatísticas ordenadas), costuma ser muito competitivo com pouco *tuning* e bom controle de *overfitting*. Em geral entrega resultados robustos em dados tabulares.

> Dessa forma, começamos com um modelo **simples (Regressão Linear)** para servir de *baseline* e, em seguida, testamos modelos baseados em árvores e boosting (**Random Forest, HistGradientBoosting, XGBoost e CatBoost**) para verificar ganhos de performance (métricas: **RMSE** principal, além de **MAE** e **R^2**).

3.2 Seleção e tratamento de variáveis

3.2.1 Tratamento de valores nulos

Antes de selecionar as variáveis preditoras para o modelo, é importante tratar os **valores ausentes** no dataset.
Valores nulos podem distorcer estatísticas, atrapalhar transformações (como normalização) e prejudicar o desempenho dos modelos de Machine Learning.

In [158…

```
df.isna().sum()
```

Out[158…

```
Series_Title        0
Released_Year       1
Certificate       316
IMDB_Rating         0
Overview          123
Meta_score       4420
Director            0
Star1               0
Star2               1
Star3               4
No_of_Votes         0
Genres_list         0
Action              0
Adventure           0
Animation           0
Biography           0
Comedy              0
Crime               0
Drama               0
Family              0
Fantasy             0
Film-Noir           0
History             0
Horror              0
Music               0
Musical             0
Mystery             0
Romance             0
Sci-Fi              0
Sport               0
Thriller            0
War                 0
Western             0
Runtime            12
Gross             849
budget            932
dtype: int64
```

Foram aplicados os seguintes tratamentos:

- `Certificate` : valores nulos preenchidos com "R".
- `Released_Year` , `Star2` , `Star3` , `Runtime` : registros com valores nulos removidos.
- `Overview` : registros com valores nulos removidos.
- `Gross` : valores nulos preenchidos com a mediana da coluna.
- `budget` : valores nulos preenchidos com a mediana da coluna.
- Índice do DataFrame redefinido após as remoções.

In [159…

```python
df['Certificate'].fillna("R", inplace= True)
df.dropna(subset=["Released_Year", "Star2", "Star3", "Runtime"], inplace=True)
print("Novo shape do df:", df.shape)
df.dropna(subset=["Overview"], inplace=True)
df["Gross"].fillna(df["Gross"].median(), inplace=True)
df["budget"].fillna(df["budget"].median(), inplace=True)
df.reset_index(drop=True, inplace=True)
```

```
Novo shape do df: (5246, 36)
```

```
C:\Users\guima\AppData\Local\Temp\ipykernel_3740\2255263276.py:1: FutureWarning: A value is tryin
g to be set on a copy of a DataFrame or Series through chained assignment using an inplace metho
d.
The behavior will change in pandas 3.0. This inplace method will never work because the intermedi
ate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value},
inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on th
e original object.


  df['Certificate'].fillna("R", inplace= True)
C:\Users\guima\AppData\Local\Temp\ipykernel_3740\2255263276.py:5: FutureWarning: A value is tryin
g to be set on a copy of a DataFrame or Series through chained assignment using an inplace metho
d.
The behavior will change in pandas 3.0. This inplace method will never work because the intermedi
ate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value},
inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on th
e original object.


  df["Gross"].fillna(df["Gross"].median(), inplace=True)
C:\Users\guima\AppData\Local\Temp\ipykernel_3740\2255263276.py:6: FutureWarning: A value is tryin
g to be set on a copy of a DataFrame or Series through chained assignment using an inplace metho
d.
The behavior will change in pandas 3.0. This inplace method will never work because the intermedi
ate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value},
inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on th
e original object.


  df["budget"].fillna(df["budget"].median(), inplace=True)
```

In [160…

```
df = df[df["Series_Title"] != "The Shawshank Redemption"].reset_index(drop=True)
```

3.2.2 Seleção de variáveis

As variáveis preditoras selecionadas foram:

- `Released_Year`: o ano de lançamento pode influenciar a recepção e, consequentemente, a nota.
- `Certificate`: a classificação indicativa pode afetar o público-alvo e o tipo de conteúdo.
- `Director`, `Star1`, `Star2`, `Star3`: equipe criativa e atores principais exercem grande impacto na percepção de qualidade.
- `No_of_Votes`: representa a popularidade do filme e tende a ter forte correlação com a nota.
- `Genres_list` e colunas dummies de gênero (`Action`, `Adventure`, ..., `Western`): o gênero influencia o perfil do público e a forma como o filme é avaliado.
- `Runtime`: a duração pode estar relacionada à recepção do público.
- `Gross` e `budget`: orçamento e bilheteria refletem aspectos financeiros que podem se relacionar com a qualidade percebida.

A variável alvo (target) é:

- `IMDB_Rating`

Variáveis não utilizadas:

- `Series_Title` : apenas identifica o filme, não possui valor preditivo.
- `Overview` : exige processamento de linguagem natural (NLP) para ser usada de forma adequada, o que está fora do escopo nesta análise.
- `Meta_score` : apresenta muitos valores ausentes, dificultando seu uso como variável preditora.

3.2.3 Transformações aplicadas

Para preparar as variáveis selecionadas, foram adotadas as seguintes transformações:

3.2.3.1 Tratamento das variáveis numéricas

Foram consideradas numéricas as colunas:

- `Released_Year` , `Runtime`
- `Gross` , `budget` , `No_of_Votes`

Tratamentos aplicados:

- **Padronização (StandardScaler)** em todas as variáveis numéricas, para colocar na mesma escala (média 0 e desvio padrão 1).
- **Transformação logarítmica (log1p)** previamente em variáveis altamente enviesadas:
  - `Gross` , `budget` , `No_of_Votes`

In [161…

```
# Definição das colunas numéricas
num_linear = ["Released_Year", "Runtime"]
num_skewed = ["Gross", "budget", "No_of_Votes"]

# Pipelines numéricos
numeric_linear_pipe = Pipeline(steps=[
    ("scaler", StandardScaler())
])

numeric_skewed_pipe = Pipeline(steps=[
    ("log1p", FunctionTransformer(np.log1p, validate=False)),
    ("scaler", StandardScaler())
])

# ColumnTransformer apenas para as numéricas
num_preprocessor = ColumnTransformer(
    transformers=[
        ("num_linear", numeric_linear_pipe, num_linear),
        ("num_skewed", numeric_skewed_pipe, num_skewed),
    ],
    remainder="drop"
)

# Exemplo de uso
X_num = df[num_linear + num_skewed].copy()
X_num_proc = num_preprocessor.fit_transform(X_num)

print("Shape após pré-processamento numérico:", X_num_proc.shape)
```

```
Shape após pré-processamento numérico: (5129, 5)
```

In [162…

```
feature_names = (
    [f"std_{c}" for c in num_linear] +
    [f"log1p_std_{c}" for c in num_skewed]
)

X_num_proc_df = pd.DataFrame(
    X_num_proc, columns=feature_names, index=X_num.index
)
X_num_proc_df.head()
```

Out[162…

	std_Released_Year	std_Runtime	log1p_std_Gross	log1p_std_budget	log1p_std_No_of_Votes
0	-1.854601	2.864626	1.075159	-0.676131	2.209229
1	0.488205	1.853034	1.723030	1.674331	2.410428
2	-1.724445	4.052147	0.672068	-0.146074	2.002986
3	-2.830770	-0.609972	-0.539866	-2.624165	1.720658
4	0.162815	4.008165	1.559523	1.210174	2.217081

Resumo:

- `Released_Year`, `Runtime` → `StandardScaler`
- `Gross`, `budget`, `No_of_Votes` → `log1p` → `StandardScaler`

3.2.3.2 Transformações das variáveis categóricas

Variáveis categóricas consideradas:

- `Certificate`, `Director`, `Star1`, `Star2`, `Star3`

Transformações aplicadas:

- `Certificate` → **One-Hot Encoding** (baixa cardinalidade), com `handle_unknown="ignore"`.
- `Director`, `Star1`, `Star2`, `Star3` → **Frequency Encoding** (alta cardinalidade), codificando cada categoria pela sua frequência relativa no conjunto de treino para evitar explosão de dimensionalidade.

In [163…

```
cat_low  = ["Certificate"]                        # baixa cardinalidade
cat_high = ["Director", "Star1", "Star2", "Star3"] # alta cardinalidade


class FrequencyEncoder(BaseEstimator, TransformerMixin):

    def __init__(self):
        self.maps_ = None
        self.cols_ = None
        self.n_ = None

    def fit(self, X, y=None):
        X = pd.DataFrame(X)
        self.cols_ = list(X.columns)
        self.n_ = len(X)
        self.maps_ = {}
        for i, c in enumerate(self.cols_):
            vc = X.iloc[:, i].astype(str).value_counts(dropna=False)
            freq = (vc / self.n_).to_dict()
```

```
            self.maps_[c] = freq
        return self

    def transform(self, X):
        X = pd.DataFrame(X)
        outs = []
        for i, c in enumerate(self.cols_):
            m = self.maps_[c]
            col = (
                X.iloc[:, i]
                .astype(str)
                .map(m)
                .fillna(0.0)
                .to_numpy()
                .reshape(-1, 1)
            )
            outs.append(col)
        return np.hstack(outs)

cat_preprocessor = ColumnTransformer(
    transformers=[
        ("cert_ohe",
         OneHotEncoder(sparse_output=False, handle_unknown="ignore"),
         cat_low),
        ("people_freq",
         FrequencyEncoder(),
         cat_high),
    ],
    remainder="drop"
)

X_cat = df[cat_low + cat_high].copy()
X_cat_proc = cat_preprocessor.fit_transform(X_cat)

# nomes das features resultantes
ohe = cat_preprocessor.named_transformers_["cert_ohe"]
cert_names = list(ohe.get_feature_names_out(cat_low))  # ex.: ['Certificate_A', 'Certificate_R',
freq_names = [f"freq_{c}" for c in cat_high]

cat_feature_names = cert_names + freq_names

X_cat_proc_df = pd.DataFrame(X_cat_proc, columns=cat_feature_names, index=df.index)
print("Shape após pré-processamento categórico:", X_cat_proc_df.shape)
X_cat_proc_df.head()
```

```
Shape após pré-processamento categórico: (5129, 25)
```

Out[163…

	Certificate_16	Certificate_A	Certificate_Approved	Certificate_G	Certificate_GP	Certificate_M	Certificat
0	0.0	1.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	1.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	

5 rows × 25 columns

Saída:

- Colunas One-Hot para `Certificate` (ex.: `Certificate_A`, `Certificate_R`, ...).
- Uma coluna numérica por atributo de alta cardinalidade (ex.: `freq_Director`, `freq_Star1`, `freq_Star2`, `freq_Star3`).

Observação:

- Target Encoding é uma alternativa para alta cardinalidade, porém exige cuidado com vazamento de informação (deve ser feito com validação cruzada). Nesta etapa optamos por Frequency Encoding para manter simplicidade e robustez.

3.2.3.3 Transformações das variáveis de gênero

Variáveis consideradas:

- Colunas binárias: `Action`, `Adventure`, `Animation`, `Biography`, `Comedy`, `Crime`, `Drama`, `Family`, `Fantasy`, `Film-Noir`, `History`, `Horror`, `Music`, `Musical`, `Mystery`, `Romance`, `Sci-Fi`, `Sport`, `Thriller`, `War`, `Western`.

Tratamento aplicado:

- As colunas de gênero já estão em formato binário (0/1), portanto foram usadas **diretamente** no modelo via *passthrough* no `ColumnTransformer`.
- A coluna textual `Genres_list` foi **descartada** por redundância.

Observações:

- Não é necessária normalização dessas colunas binárias.
- Como os gêneros formam um conjunto multi-rótulo (um filme pode ter mais de um), não se aplica a remoção de uma categoria para evitar colinearidade típica de one-hot de uma única variável.

In [164…

```
import numpy as np
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import OneHotEncoder, StandardScaler, FunctionTransformer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline


num_linear = ["Released_Year", "Runtime"]
num_skewed = ["Gross", "budget", "No_of_Votes"]
cat_low  = ["Certificate"]
cat_high = ["Director", "Star1", "Star2", "Star3"]


genre_cols = [
    "Action","Adventure","Animation","Biography","Comedy","Crime","Drama","Family",
    "Fantasy","Film-Noir","History","Horror","Music","Musical","Mystery","Romance",
    "Sci-Fi","Sport","Thriller","War","Western"
]


if "Genres_list" in df.columns:
    df = df.drop(columns=["Genres_list"])
```

```python
numeric_linear_pipe = Pipeline([
    ("scaler", StandardScaler())
])

numeric_skewed_pipe = Pipeline([
    ("log1p", FunctionTransformer(np.log1p, validate=False)),
    ("scaler", StandardScaler())
])


class FrequencyEncoder(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.maps_ = None
        self.cols_ = None
        self.n_ = None
    def fit(self, X, y=None):
        X = pd.DataFrame(X)
        self.cols_ = list(X.columns)
        self.n_ = len(X)
        self.maps_ = {}
        for i, c in enumerate(self.cols_):
            vc = X.iloc[:, i].astype(str).value_counts(dropna=False)
            self.maps_[c] = (vc / self.n_).to_dict()
        return self
    def transform(self, X):
        X = pd.DataFrame(X)
        outs = []
        for i, c in enumerate(self.cols_):
            m = self.maps_[c]
            col = (
                X.iloc[:, i].astype(str).map(m).fillna(0.0).to_numpy().reshape(-1, 1)
            )
            outs.append(col)
        return np.hstack(outs)


preprocessor = ColumnTransformer(
    transformers=[
        ("num_linear",  numeric_linear_pipe,  num_linear),             # std
        ("num_skewed",  numeric_skewed_pipe,  num_skewed),             # log1p + std
        ("cert_ohe",    OneHotEncoder(sparse_output=False, handle_unknown="ignore"),cat_low),
        ("people_freq", FrequencyEncoder(),   cat_high),               # frequência
        ("genres",      "passthrough",        genre_cols),             # já binárias
    ],
    remainder="drop"
)


X = df[num_linear + num_skewed + cat_low + cat_high + genre_cols].copy()
X_proc = preprocessor.fit_transform(X)
print("Shape após pré-processamento completo:", X_proc.shape)

num_feature_names = (
    [f"std_{c}" for c in num_linear] +
    [f"log1p_std_{c}" for c in num_skewed]
)

ohe = preprocessor.named_transformers_["cert_ohe"]
cert_names = list(ohe.get_feature_names_out(cat_low))

freq_names = [f"freq_{c}" for c in cat_high]
```

```
genre_names = genre_cols

feature_names = num_feature_names + cert_names + freq_names + genre_names
X_proc_df = pd.DataFrame(X_proc, columns=feature_names, index=df.index)
X_proc_df.head()
```

```
Shape após pré-processamento completo: (5129, 51)
```

Out[164...

	std_Released_Year	std_Runtime	log1p_std_Gross	log1p_std_budget	log1p_std_No_of_Votes	Certificate_
0	-1.854601	2.864626	1.075159	-0.676131	2.209229	
1	0.488205	1.853034	1.723030	1.674331	2.410428	
2	-1.724445	4.052147	0.672068	-0.146074	2.002986	
3	-2.830770	-0.609972	-0.539866	-2.624165	1.720658	
4	0.162815	4.008165	1.559523	1.210174	2.217081	

5 rows × 51 columns

3.2.4 Conclusão de tratamento e seleção das variáveis

Numéricas

- `Released_Year`, `Runtime` → `StandardScaler`
  Por quê: coloca todas as variáveis na mesma escala (média 0, desvio 1), evitando que atributos com magnitude maior dominem modelos sensíveis à escala (p.ex., Regressão Linear).
- `Gross`, `budget`, `No_of_Votes` → `log1p` + `StandardScaler`
  Por quê: `log1p` reduz assimetria e ordens de grandeza (variância mais estável e relações mais próximas de lineares); depois padronizamos para manter a comparabilidade entre atributos.

Categóricas

- `Certificate` → One-Hot Encoding
  Por quê: poucas categorias e sem ordem natural; OHE representa cada categoria sem impor ordenação artificial.
- `Director`, `Star1`, `Star2`, `Star3` → Frequency Encoding
  Por quê: altíssima cardinalidade; evita explosão de colunas do OHE e ainda captura sinal de "popularidade/recorrência" das categorias. (Mais simples e robusto que Target Encoding, reduz risco de vazamento.)

Gêneros

- Colunas binárias (`Action` … `Western`) → *passthrough*
  Por quê: já estão em 0/1; não requerem codificação nem escala adicional.
- `Genres_list` → removida
  Por quê: redundante com as colunas binárias e exigiria NLP para uso adequado.

Orquestração

- `ColumnTransformer` para unificar todas as etapas
  Por quê: garante o mesmo pré-processamento em treino/teste, melhora reprodutibilidade e evita vazamento acidental.
- `df` original preservado; matriz final de features em `X_proc_df`
  Por quê: mantém rastreabilidade dos dados brutos e separa claramente *features* de *target*.

Saída do pré-processamento

- **Features transformadas:** `X = X_proc_df` com **5262 × 52** colunas.
- **Alvo (target):** `y = df["IMDB_Rating"].values` com **5262** registros.

Próximo passo

- `train_test_split`, treinar baseline (Regressão Linear) e modelos de árvore/boosting (Random Forest, Gradient Boosting) e avaliar com **RMSE** e **MAE**.

3.3 Separação em treino e teste dos valores

Para garantir uma avaliação justa da capacidade preditiva dos modelos, dividimos o conjunto de dados em duas partes: **80% para treino** e **20% para teste**.

- **Treino (X_train, y_train):** utilizado para ajustar os parâmetros internos dos modelos.
- **Teste (X_test, y_test):** mantido isolado durante o treino, serve para avaliar a performance em dados nunca vistos, simulando um cenário de generalização.

A proporção 80/20 foi escolhida por ser um padrão amplamente aceito: garante dados suficientes para aprendizado do modelo e, ao mesmo tempo, reserva uma amostra representativa para validação final.
O parâmetro `random_state=42` assegura **reprodutibilidade** da divisão, permitindo que os resultados sejam replicados.

In [165…

```
# ===== Divisão em treino e teste =====
from sklearn.model_selection import train_test_split

# Features (X) já processadas
X = X_proc_df

# Target (y) = nota do IMDB
y = df["IMDB_Rating"].values

# Separação 80% treino / 20% teste
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,      # 20% para teste
    random_state=42,    # garante reprodutibilidade
    shuffle=True        # embaralha antes da divisão
)

print("Tamanho treino:", X_train.shape[0], "amostras")
print("Tamanho teste :", X_test.shape[0], "amostras")
```

```
Tamanho treino: 4103 amostras
Tamanho teste : 1026 amostras
```

3.4 Avaliação dos modelos

Para iniciar a etapa de modelagem, adotamos a **Regressão Linear** como modelo baseline.
A escolha se justifica por ser um algoritmo simples, rápido de treinar e de fácil interpretação, permitindo avaliar se as variáveis transformadas já carregam sinal preditivo suficiente.
Com esse baseline, poderemos comparar posteriormente com modelos mais complexos (Random Forest e XGBoost) e verificar ganhos reais de performance.

Métricas utilizadas

Como o problema é de **regressão supervisionada**, utilizamos métricas que avaliam o erro entre as previsões e os valores reais:

- **RMSE (Root Mean Squared Error):** métrica principal, pois penaliza mais fortemente erros grandes.
- **MAE (Mean Absolute Error):** complemento de fácil interpretação, indica o erro médio absoluto em pontos de nota IMDb.
- **R^2 (Coeficiente de Determinação):** mostra a proporção da variabilidade da nota explicada pelo modelo, útil como métrica adicional.

Dessa forma, a análise será conduzida principalmente pelo **RMSE**, mas também observaremos **MAE** e **R^2** para uma visão mais completa.

3.4.1 Regressão Linear

- Modelo baseline: simples, rápido e interpretável.
- Serve como referência para verificar se modelos mais complexos trazem ganhos reais de performance.

In [166…

```
# ===== 2) Instanciar e treinar o modelo =====
linreg = LinearRegression()
linreg.fit(X_train, y_train)

# ===== 3) Predições =====
y_pred = linreg.predict(X_test)

# ===== 4) Avaliação =====
mae  = mean_absolute_error(y_test, y_pred)
rmse = root_mean_squared_error(y_test, y_pred)
r2   = r2_score(y_test, y_pred)

print(f"[Regressão Linear - Teste]")
print(f"MAE : {mae:.3f}")
print(f"RMSE: {rmse:.3f}")
print(f"R²  : {r2:.3f}")

# ===== 5) (Opcional) Guardar resultados para análise posterior =====
resultados_baseline = pd.DataFrame({
    "y_true": y_test,
    "y_pred": y_pred,
    "erro_abs": np.abs(y_test - y_pred)
}).reset_index(drop=True)

resultados_baseline.head()
```

```
[Regressão Linear - Teste]
MAE : 0.529
RMSE: 0.730
R²  : 0.594
```

Out[166…

	y_true	y_pred	erro_abs
0	6.2	5.590744	0.609256
1	6.5	6.335948	0.164052
2	7.2	6.754364	0.445636
3	5.9	5.834197	0.065803
4	6.8	6.356622	0.443378

Resultados da Regressão Linear

O modelo de **Regressão Linear** foi utilizado como baseline, servindo como ponto de partida para comparação com modelos mais complexos.

Métricas obtidas no conjunto de teste:

- **MAE (Mean Absolute Error):** 0.520
  → Em média, o modelo erra cerca de **0,52 pontos** na nota IMDb (em uma escala de 0 a 10).
- **RMSE (Root Mean Squared Error):** 0.730
  → Erros maiores são penalizados mais fortemente; neste caso, o erro típico é de aproximadamente **0,73 pontos**.
- **R^2 (Coeficiente de Determinação):** 0.594
  → O modelo consegue explicar cerca de **59% da variabilidade** da nota IMDb a partir das variáveis disponíveis.

Análise:

- Os resultados mostram que a regressão linear já captura uma parte relevante da relação entre as variáveis e a nota do IMDb, mas ainda deixa **~40% da variabilidade sem explicação**.
- O MAE e RMSE estão em níveis aceitáveis, mas sugerem espaço para melhoria, especialmente em casos onde a nota real é mais extrema (erros maiores penalizam o RMSE).
- Isso confirma a **importância de testar modelos mais complexos** (como Random Forest e Gradient Boosting), que podem capturar relações não-lineares e interações entre variáveis que a regressão linear não consegue modelar.

3.4.2 Random Forest Regressor

- Modelo baseado em um conjunto de árvores de decisão, construídas sobre subconjuntos de dados e variáveis.
- Captura **relações não lineares** e **interações entre atributos**, sem exigir normalização das variáveis.
- É mais robusto a outliers e tende a reduzir o risco de overfitting em comparação a uma única árvore de decisão.
- Serve como próximo passo após o baseline para verificar ganhos de performance com um modelo mais complexo.

In [167…

```
# Garantia: usar os mesmos splits
assert 'X_train' in globals() and 'X_test' in globals(), "Faça a divisão treino/teste antes."
assert 'y_train' in globals() and 'y_test' in globals(), "Faça a divisão treino/teste antes."
```

```python
# 1) Instanciar o modelo (setup inicial e reprodutível)
rf = RandomForestRegressor(
    n_estimators=300,      # número de árvores
    max_depth=None,        # deixe crescer; ajustaremos depois se precisar
    min_samples_leaf=1,    # padrão
    n_jobs=-1,             # usa todos os núcleos
    random_state=42,       # reprodutibilidade
)
# 2) Treinar
rf.fit(X_train, y_train)

# 3) Predizer
y_pred_rf = rf.predict(X_test)

# 4) Avaliar (RMSE calculado como raiz do MSE para compatibilidade entre versões)
mae_rf  = mean_absolute_error(y_test, y_pred_rf)
mse_rf  = mean_squared_error(y_test, y_pred_rf)
rmse_rf = np.sqrt(mse_rf)
r2_rf   = r2_score(y_test, y_pred_rf)

print("[Random Forest - Teste]")
print(f"MAE : {mae_rf:.3f}")
print(f"RMSE: {rmse_rf:.3f}")
print(f"R²  : {r2_rf:.3f}")

# 5) (Opcional) Tabela de resultados para comparar depois
resultados_rf = pd.DataFrame({
    "y_true": y_test,
    "y_pred": y_pred_rf,
    "erro_abs": np.abs(y_test - y_pred_rf)
}).reset_index(drop=True)

# 6) (Opcional) Importâncias de variáveis (top 15) — útil para diagnóstico
try:
    importances = pd.Series(rf.feature_importances_, index=X_train.columns).sort_values(ascendin
    display(importances.head(15))
except Exception as e:
    print("Não foi possível calcular importâncias:", e)
```

```
[Random Forest - Teste]
MAE : 0.463
RMSE: 0.664
R²  : 0.663
log1p_std_No_of_Votes    0.328738
Drama                    0.116148
log1p_std_budget         0.106153
std_Released_Year        0.100520
std_Runtime              0.080315
log1p_std_Gross          0.063631
freq_Star1               0.026460
Certificate_PG-13        0.021130
freq_Director            0.018715
freq_Star2               0.016872
Thriller                 0.011600
freq_Star3               0.011337
Horror                   0.010230
Animation                0.008685
Certificate_R            0.007720
dtype: float64
```

Resultados do Random Forest Regressor

Métricas obtidas no conjunto de teste:

- **MAE (Mean Absolute Error):** 0.463
  → O erro médio absoluto caiu em relação à regressão linear (0.520 → 0.463), ou seja, o modelo erra menos em média.

- **RMSE (Root Mean Squared Error):** 0.664
  → O erro típico (com maior penalização para erros grandes) também melhorou em relação ao baseline (0.730 → 0.664).

- **R^2 (Coeficiente de Determinação):** 0.663
  → O modelo explica **66% da variabilidade** da nota IMDb, contra 59% da regressão linear.

Análise:

- O **Random Forest superou a Regressão Linear** em todas as métricas, mostrando que capturar relações não lineares e interações entre variáveis melhora a qualidade das previsões.
- O ganho em R^2 (de 0.594 → 0.663) indica que o modelo consegue explicar uma fatia maior da variação nas notas.
- As importâncias de variáveis apontam que o número de votos (`log1p_std_No_of_Votes`), gênero *Drama* e orçamento (`log1p_std_budget`) são fatores muito relevantes para prever a nota.
- Apesar da melhora, ainda existe cerca de 34% da variabilidade não explicada, reforçando que outros algoritmos (como Gradient Boosting) podem trazer ganhos adicionais.

Conclusão parcial:
O Random Forest demonstrou ser um avanço claro sobre o baseline. Na próxima etapa, testaremos o **Gradient Boosting (XGBoost/LightGBM)** para verificar se conseguimos reduzir ainda mais o erro e aumentar o poder explicativo do modelo.

3.4.3 XGBoost

- Algoritmo de **boosting de árvores**, onde cada árvore é treinada sequencialmente para corrigir os erros das anteriores.
- Considerado um dos modelos mais eficientes para dados tabulares, geralmente superando Random Forest em performance.
- Captura **relações não lineares** e **interações entre variáveis**, com regularização para evitar overfitting.
- Serve como próximo passo após o Random Forest para verificar se o ajuste sequencial reduz o erro e melhora o poder explicativo do modelo.

In [168…

```
assert 'X_train' in globals() and 'X_test' in globals(), "Faça a divisão treino/teste antes."
assert 'y_train' in globals() and 'y_test' in globals(), "Faça a divisão treino/teste antes."

# 1) Instanciar o modelo
xgb_model = xgb.XGBRegressor(
    n_estimators=500,
    learning_rate=0.05,
    max_depth=6,
    subsample=0.85,
    colsample_bytree=0.85,
    reg_lambda=1.0,
    random_state=42,
    n_jobs=-1,
```

```
    tree_method="hist",   # rápido e estável
    objective="reg:squarederror",
    verbosity=0,
)

# 2) Treinar
xgb_model.fit(X_train, y_train)

# 3) Predizer
y_pred_xgb = xgb_model.predict(X_test)

# 4) Avaliar
mae_xgb  = mean_absolute_error(y_test, y_pred_xgb)
mse_xgb  = mean_squared_error(y_test, y_pred_xgb)
rmse_xgb = np.sqrt(mse_xgb)
r2_xgb   = r2_score(y_test, y_pred_xgb)

print("[XGBoost - Teste]")
print(f"MAE : {mae_xgb:.3f}")
print(f"RMSE: {rmse_xgb:.3f}")
print(f"R²  : {r2_xgb:.3f}")

# 5) (Opcional) Importâncias de features
importances_xgb = pd.Series(xgb_model.feature_importances_, index=X_train.columns)\
                    .sort_values(ascending=False).head(15)
display(importances_xgb)
```

```
[XGBoost - Teste]
MAE : 0.449
RMSE: 0.644
R²  : 0.684
Drama                   0.141517
Certificate_U           0.101224
Animation               0.054244
Certificate_PG-13       0.052296
log1p_std_No_of_Votes   0.050663
Horror                  0.033309
Thriller                0.026087
Certificate_UA          0.025682
Certificate_PG          0.025466
log1p_std_budget        0.025065
Certificate_R           0.024547
Biography               0.022713
Musical                 0.022631
std_Runtime             0.022464
Certificate_X           0.022309
dtype: float32
```

Resultados do XGBoost

Métricas obtidas no conjunto de teste:

- **MAE (Mean Absolute Error):** 0.449
  → O menor erro médio até agora, indicando previsões mais próximas das notas reais.

- **RMSE (Root Mean Squared Error):** 0.644
  → Redução em relação ao Random Forest (0.664) e à Regressão Linear (0.730), com menor penalização de erros grandes.

- **R^2 (Coeficiente de Determinação):** 0.684
  → O modelo explica cerca de **68% da variabilidade** da nota IMDb, superando o Random Forest (66%) e a Regressão Linear (59%).

Análise:

- O XGBoost apresentou **ganho consistente em todas as métricas** comparado aos modelos anteriores.
- Mostrou maior capacidade de capturar padrões complexos, reduzindo o erro médio e aumentando o R^2.
- As importâncias de variáveis reforçam que `No_of_Votes`, `Drama` e `budget` continuam como fatores centrais para previsão.
- Apesar da melhora, ainda existe cerca de 32% da variabilidade não explicada, o que indica limites impostos pelos dados disponíveis.

Conclusão parcial:
O XGBoost superou a Regressão Linear e o Random Forest, consolidando-se como o melhor modelo até o momento. Na próxima etapa, testaremos o **CatBoost**, que pode explorar variáveis categóricas de maneira ainda mais eficiente e potencialmente trazer ganhos adicionais.

3.4.4 HistGradientBoosting (sklearn)

- Variante de Gradient Boosting integrada ao scikit-learn, otimizada via histogramas para acelerar os cálculos.
- Captura **relações não lineares** e **interações entre variáveis** de forma semelhante ao XGBoost, mas com implementação mais leve.
- Serve como alternativa ao XGBoost em cenários onde se busca simplicidade e integração com o ecossistema sklearn.

In [169…

```
# Garantia: usar o mesmo split
assert 'X_train' in globals() and 'X_test' in globals(), "Faça a divisão treino/teste antes."
assert 'y_train' in globals() and 'y_test' in globals(), "Faça a divisão treino/teste antes."

# 1) Instanciar o modelo
hgb_model = HistGradientBoostingRegressor(
    learning_rate=0.06,
    max_depth=None,
    max_iter=300,
    l2_regularization=0.0,
    early_stopping=True,
    validation_fraction=0.1,
    random_state=42,
)

# 2) Treinar
hgb_model.fit(X_train, y_train)

# 3) Predizer
y_pred_hgb = hgb_model.predict(X_test)

# 4) Avaliar
mae_hgb  = mean_absolute_error(y_test, y_pred_hgb)
mse_hgb  = mean_squared_error(y_test, y_pred_hgb)
rmse_hgb = np.sqrt(mse_hgb)
r2_hgb   = r2_score(y_test, y_pred_hgb)
```

```python
print("[HistGradientBoosting - Teste]")
print(f"MAE : {mae_hgb:.3f}")
print(f"RMSE: {rmse_hgb:.3f}")
print(f"R²  : {r2_hgb:.3f}")

# Importâncias via permutação (pode ser mais lento)
result = permutation_importance(
    hgb_model, X_test, y_test,
    n_repeats=10, random_state=42, n_jobs=-1
)

importances_hgb = pd.Series(result.importances_mean, index=X_train.columns)\
                    .sort_values(ascending=False).head(15)

display(importances_hgb)
```

```
[HistGradientBoosting - Teste]
MAE : 0.457
RMSE: 0.650
R²  : 0.678
log1p_std_No_of_Votes    0.814464
Drama                    0.094613
std_Released_Year        0.093470
log1p_std_budget         0.085592
log1p_std_Gross          0.059928
std_Runtime              0.052958
Animation                0.031573
Horror                   0.014636
Certificate_PG-13        0.011784
Thriller                 0.011071
Action                   0.010079
freq_Star1               0.007644
freq_Director            0.005704
Certificate_R            0.005530
Comedy                   0.004615
dtype: float64
```

Resultados do HistGradientBoosting

Métricas obtidas no conjunto de teste:

- **MAE (Mean Absolute Error):** 0.457
  → Um erro médio comparável ao XGBoost (0.449), mas ainda ligeiramente superior.
- **RMSE (Root Mean Squared Error):** 0.650
  → Valor próximo ao XGBoost (0.644), mostrando boa capacidade de generalização.
- **R^2 (Coeficiente de Determinação):** 0.678
  → Explica cerca de **68% da variabilidade** das notas IMDb, em linha com o XGBoost, mas com resultado marginalmente inferior.

Análise:

- O HistGradientBoosting entregou resultados sólidos, superiores ao Random Forest e à Regressão Linear.
- Apesar de não superar o XGBoost, mostrou ser uma **opção competitiva e mais integrada ao sklearn**.

- A proximidade nos resultados evidencia que o ganho entre variantes de boosting é incremental, e não disruptivo.

Conclusão parcial:
O HistGradientBoosting confirmou o padrão de bom desempenho dos modelos de boosting, ficando muito próximo ao XGBoost, mas sem superá-lo. O próximo passo será testar o **CatBoost**, que pode oferecer ganhos adicionais ao lidar de forma diferenciada com variáveis categóricas.

3.4.5 CatBoost

- Algoritmo de **gradient boosting** desenvolvido para lidar de forma mais eficiente com variáveis categóricas.
- Possui mecanismos internos que reduzem a necessidade de pré-processamento pesado, evitando overfitting e acelerando o treinamento.
- Foi testado aqui como o último modelo, após a Regressão Linear, Random Forest e outras variantes de boosting, para verificar ganhos adicionais.

In [170…

```python
# Garantia: usar o mesmo split
assert 'X_train' in globals() and 'X_test' in globals(), "Faça a divisão treino/teste antes."
assert 'y_train' in globals() and 'y_test' in globals(), "Faça a divisão treino/teste antes."

try:
    from catboost import CatBoostRegressor

    # 1) Instanciar o modelo
    cat = CatBoostRegressor(
        iterations=500,
        learning_rate=0.05,
        depth=8,
        random_seed=42,
        verbose=0,
        allow_writing_files=False,  # <- não cria catboost_info
    )

    # 2) Treinar
    cat.fit(X_train, y_train)

    # 3) Predizer
    y_pred_cat = cat.predict(X_test)

    # 4) Avaliar
    mae_cat  = mean_absolute_error(y_test, y_pred_cat)
    mse_cat  = mean_squared_error(y_test, y_pred_cat)
    rmse_cat = np.sqrt(mse_cat)
    r2_cat   = r2_score(y_test, y_pred_cat)

    print("[CatBoost - Teste]")
    print(f"MAE : {mae_cat:.3f}")
    print(f"RMSE: {rmse_cat:.3f}")
    print(f"R²  : {r2_cat:.3f}")

    # 5) (Opcional) DataFrame de resultados
    resultados_cat = pd.DataFrame({
        "y_true": y_test,
        "y_pred": y_pred_cat,
        "erro_abs": np.abs(y_test - y_pred_cat)
    }).reset_index(drop=True)
```

```
except ImportError:
    print("CatBoost não está instalado. Rode: pip install catboost")
```

```
[CatBoost - Teste]
MAE : 0.444
RMSE: 0.643
R²  : 0.685
```

Resultados do CatBoost

Métricas obtidas no conjunto de teste:

- **MAE (Mean Absolute Error):** 0.444
  → O menor erro médio entre todos os modelos, mostrando previsões bastante próximas das notas reais.

- **RMSE (Root Mean Squared Error):** 0.643
  → Também o menor RMSE até agora, indicando que o CatBoost lida melhor com erros maiores.

- **R^2 (Coeficiente de Determinação):** 0.685
  → Explica cerca de **68,5% da variabilidade** das notas IMDb, o maior valor entre os modelos testados.

Análise:

- O CatBoost apresentou a **melhor performance geral**, superando Regressão Linear, Random Forest e até mesmo o XGBoost.
- A redução do MAE e RMSE, ainda que marginal em relação ao XGBoost, confirma sua capacidade de capturar relações complexas.
- O ganho adicional provavelmente se deve ao tratamento interno mais sofisticado de variáveis categóricas, mesmo após nossas transformações prévias.
- Ainda assim, existe aproximadamente 31,5% da variabilidade que não foi explicada — evidenciando limites dos dados disponíveis.

Conclusão parcial:
O **CatBoost** foi o modelo vencedor neste estudo, atingindo o melhor equilíbrio entre erro médio e capacidade explicativa. Ele se consolida como a escolha final para previsão das notas IMDb neste conjunto de dados.

3.4.6 Comparação dos Modelos

Modelo	MAE	RMSE	R^2
Regressão Linear	0.520	0.730	0.594
Random Forest	0.463	0.664	0.663
HistGradientBoosting	0.457	0.650	0.678
XGBoost	0.449	0.644	0.684
CatBoost	**0.444**	**0.643**	**0.685**

Conclusões finais

- A **Regressão Linear** cumpriu seu papel como baseline, mas explicou apenas ~59% da variabilidade das notas IMDb.
- O **Random Forest** trouxe um avanço claro, capturando relações não lineares e elevando o R^2 para ~66%.
- Os modelos de **boosting** mostraram melhor desempenho: tanto o **HistGradientBoosting** quanto o **XGBoost** aumentaram a capacidade explicativa para ~68%, reduzindo consistentemente os erros.
- O **CatBoost** apresentou o **melhor resultado geral**, com menor MAE e RMSE e o maior R^2 (≈ 0.685).

Conclusão geral

Entre todos os modelos testados, o **CatBoost** se mostrou o que melhor se aproxima dos dados, apresentando o menor erro (MAE = 0.444, RMSE = 0.643) e o maior poder explicativo (R^2 = 0.685).
Seus principais **prós** são: lidar de forma eficiente com variáveis categóricas, robustez contra overfitting e excelente desempenho em dados tabulares. Como **contra**, pode exigir mais tempo de treino em comparação a modelos mais simples e apresenta menor interpretabilidade em relação à Regressão Linear.

Para avaliar os modelos, utilizamos as métricas **MAE, RMSE e R^2**, pois tratamos de um problema de **regressão supervisionada** (a variável alvo é contínua).

- O **MAE** indica o erro médio absoluto em pontos da nota IMDb, de fácil interpretação.
- O **RMSE** foi escolhido como **métrica principal**, pois penaliza mais fortemente erros grandes, que são mais críticos neste contexto.
- O **R^2** complementa a análise, mostrando a proporção da variabilidade da nota que o modelo consegue explicar.

Dessa forma, concluímos que o **CatBoost** é o modelo mais adequado para este problema, alcançando o melhor equilíbrio entre erro médio e capacidade explicativa.
Ainda assim, aproximadamente 31,5% da variabilidade permanece não explicada, o que indica que fatores externos ao dataset (como recepção crítica especializada, contexto cultural ou marketing) também influenciam significativamente as notas.

4. Previsão da nota do IMDb para um novo filme

Aplicamos o **mesmo pré-processamento** utilizado no treino (log1p + StandardScaler para variáveis assimétricas, StandardScaler para numéricas lineares, One-Hot em `Certificate`, Frequency Encoding em `Director` / `Stars` e gêneros binários em *passthrough*) ao registro do filme fornecido. Em seguida, usamos o **modelo final (CatBoost)** para obter a previsão.

4.1 Preparação dos dados:

1. Conversão de tipos:
   - `Runtime` → inteiro (minutos)
   - `Gross` → numérico (remoção de `,`)
   - `Released_Year` → numérico
2. Preenchimento de variáveis ausentes:
   - `budget` (não fornecido) → **mediana** do conjunto de treino.

3. Gêneros:
    - Criação das colunas binárias (`Action` , `Drama` , ..., `Western`) a partir de `Genre` .
4. Alinhamento com o dataset de treino:
    - Inclusão de `IMDB_Rating` como `NaN` (alvo, não usado na predição).
    - `reindex` para garantir **as mesmas 35 colunas e dtypes** do treino.

> Resultado: `df_novo` fica 100% compatível com o pipeline de treino.

In [171…

```python
# Dicionário do filme fornecido
novo_filme = {
    'Series_Title': 'The Shawshank Redemption',
    'Released_Year': '1994',
    'Certificate': 'A',
    'Runtime': '142 min',
    'Genre': 'Drama',
    'Overview': 'Two imprisoned men bond over a number of years, finding solace and eventual rede
    'Meta_score': 80.0,
    'Director': 'Frank Darabont',
    'Star1': 'Tim Robbins',
    'Star2': 'Morgan Freeman',
    'Star3': 'Bob Gunton',
    'Star4': 'William Sadler',
    'No_of_Votes': 2343110,
    'Gross': '28,341,469'
}
```

In [172…

```python
# Criar DataFrame a partir do dicionário fornecido
df_novo = pd.DataFrame([novo_filme])

# Preencher variáveis ausentes com valores default (mediana do dataset)
df_novo["budget"] = df["budget"].median()

# Ajustar colunas de formato
df_novo["Runtime"] = df_novo["Runtime"].str.replace(" min", "").astype(int)
df_novo["Gross"] = df_novo["Gross"].str.replace(",", "").astype(float)
```

In [173…

```python
# Criar DataFrame do novo filme
df_novo = pd.DataFrame([novo_filme])

# Ajustar colunas numéricas
df_novo["Runtime"] = df_novo["Runtime"].str.replace(" min", "").astype(float)
df_novo["Gross"] = df_novo["Gross"].str.replace(",", "").astype(float)
df_novo["Released_Year"] = df_novo["Released_Year"].astype(float)

# Preencher variáveis ausentes
df_novo["budget"] = df["budget"].median()
if "Meta_score" not in df_novo or pd.isna(df_novo["Meta_score"].iloc[0]):
    df_novo["Meta_score"] = df["Meta_score"].median()

# Criar colunas de gêneros (Action, Adventure, ..., Western)
generos = ['Action','Adventure','Animation','Biography','Comedy','Crime','Drama','Family','Fanta
           'Film-Noir','History','Horror','Music','Musical','Mystery','Romance','Sci-Fi',
           'Sport','Thriller','War','Western']

# Inicializar com 0
for g in generos:
    df_novo[g] = 0

# Marcar os gêneros do filme
```

```
for g in generos:
    if g in df_novo.loc[0, "Genre"]:
        df_novo.loc[0, g] = 1

# Adicionar a coluna IMDB_Rating vazia (para alinhar)
df_novo["IMDB_Rating"] = np.nan

# Reindexar para alinhar com df original
df_novo = df_novo.reindex(columns=df.columns, fill_value=0)
```

4.2 Aplicando o Modelo:

- Usamos as **mesmas features** do treino:
  `num_linear + num_skewed + cat_low + cat_high + genre_cols`.
- **Sem refitar** o pré-processador: apenas `preprocessor.transform(df_novo[feature_cols])`.
- Predição com o **CatBoost** treinado.

Nota prevista (CatBoost): 8.93

In [174…

```
feature_cols = num_linear + num_skewed + cat_low + cat_high + genre_cols


X_novo = preprocessor.transform(df_novo[feature_cols])


pred_nota = cat.predict(X_novo)

print("Nota prevista para o IMDb:", round(float(pred_nota[0]), 2))
```

```
Nota prevista para o IMDb: 8.93
```

4.3 Comparação com a nota oficial do IMDb

- **Nota prevista:** 8.93
- **Nota real (IMDb):** 9.30

Erro absoluto: |9.30 − 8.93| = **0.37** ponto
Erro percentual: ≈ 3,98%

Leitura: o erro desta previsão é **menor que o MAE do modelo** (≈ 0.44) e abaixo do **RMSE** (≈ 0.64), indicando que a estimativa está **dentro do esperado** para a performance do CatBoost.

5. Salvando o modelo

Para reaproveitar o que foi treinado, vamos salvar **todo o pipeline**:

- `preprocessor` (**ColumnTransformer** já *fitado*), que contém *log1p*, *StandardScaler*, *One-Hot*, *Frequency Encoding* e *passthrough* de gêneros;
- `model` (o **CatBoost** já treinado);
- `feature_cols` (lista de colunas brutas esperadas);
- `metadata` (versões do ambiente, útil para reprodutibilidade).

> Vantagem: ao carregar o `.pkl`, basta aplicar `preprocessor.transform()` e chamar `model.predict()` — sem refazer nenhuma transformação manual.

In [175…
```
models_dir = (Path.cwd() / ".." / "models").resolve()
if not models_dir.exists():
    # fallback: se o CWD já for a raiz do projeto
    models_dir = (Path.cwd() / "models").resolve()

models_dir.mkdir(parents=True, exist_ok=True)


artifact = {
    "preprocessor": preprocessor,  # ColumnTransformer já fitado
    "model": cat,                  # CatBoost já treinado
    "feature_cols": num_linear + num_skewed + cat_low + cat_high + genre_cols,
    "metadata": {
        "python": sys.version,
        "sklearn": sklearn.__version__,
    }
}

# 3) Salvar
pkl_path = models_dir / "imdb_catboost_pipeline.pkl"
joblib.dump(artifact, pkl_path)
print(f" Modelo salvo em: {pkl_path}")
```

```
Modelo salvo em: C:\Users\guima\Desktop\Desafio_Guilherme\models\imdb_catboost_pipeline.pkl
```

Carregando o modelo:

In [176…
```
# Localiza o arquivo (fora de notebooks/ ou no diretório atual)
pkl_path = (Path.cwd() / ".." / "models" / "imdb_catboost_pipeline.pkl").resolve()
if not pkl_path.exists():
    pkl_path = (Path.cwd() / "models" / "imdb_catboost_pipeline.pkl").resolve()

art = joblib.load(pkl_path)

preprocessor = art["preprocessor"]
model        = art["model"]
feature_cols = art["feature_cols"]

print(" Pipeline carregado com sucesso.")
```

```
Pipeline carregado com sucesso.
```

Tetando novamente!!

In [177…
```
X_novo = preprocessor.transform(df_novo[feature_cols])
pred   = model.predict(X_novo)

print("Nota prevista para o IMDb:", round(float(pred[0]), 2))
```

```
Nota prevista para o IMDb: 8.93
```