

Carregando o dataset

```
# Importa bibliotecas necessárias (pandas, numpy, matplotlib, seaborn).  
  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
# Lê o arquivo "cardio_train.csv" usando separador ";".  
# Exibe as primeiras linhas (head) do dataframe.  
  
data_frame = pd.read_csv("base/cardio_train.csv", sep = ";")  
data_frame.head()  
  
id    age   gender  height  weight  ap_hi  ap_lo cholesterol  gluc  
smoke \
0     0  18393      2     168     62.0    110     80            1      1  
0  
1     1  20228      1     156     85.0    140     90            3      1  
0  
2     2  18857      1     165     64.0    130     70            3      1  
0  
3     3  17623      2     169     82.0    150    100            1      1  
0  
4     4  17474      1     156     56.0    100     60            1      1  
0  
  
alco  active  cardio  
0     0        1        0  
1     0        1        1  
2     0        0        1  
3     0        1        1  
4     0        0        0  
  
# Mostra informações dos tipos de dados e contagem de não-nulos (info).  
  
data_frame.info()  
  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 70000 entries, 0 to 69999  
Data columns (total 13 columns):  
 #   Column      Non-Null Count Dtype  
---  ---  
 0   id          70000 non-null  int64  
 1   age          70000 non-null  int64  
 2   gender       70000 non-null  int64
```

```
3    height      70000 non-null  int64
4    weight      70000 non-null  float64
5    ap_hi       70000 non-null  int64
6    ap_lo       70000 non-null  int64
7    cholesterol 70000 non-null  int64
8    gluc        70000 non-null  int64
9    smoke       70000 non-null  int64
10   alco        70000 non-null  int64
11   active      70000 non-null  int64
12   cardio      70000 non-null  int64
dtypes: float64(1), int64(12)
memory usage: 6.9 MB
```

Mostra as dimensões do dataframe.

```
print(data_frame.shape)
(70000, 13)
```

Mostra as colunas do dataframe.

```
print(data_frame.columns)
Index(['id', 'age', 'gender', 'height', 'weight', 'ap_hi', 'ap_lo',
       'cholesterol', 'gluc', 'smoke', 'alco', 'active', 'cardio'],
      dtype='object')
```

Pré-processamento

Conta valores nulos por coluna.

```
data_frame.isnull().sum()
id          0
age         0
gender      0
height      0
weight      0
ap_hi       0
ap_lo       0
cholesterol 0
gluc        0
smoke       0
alco        0
active      0
cardio      0
dtype: int64
```

Verificando se existem duplicados

```
# Remove a coluna "id" (identificador único) e conta registros duplicados.

data_frame.drop("id", axis=1).duplicated().sum()

np.int64(24)
```

Verificando a tipagem dos campos

As funções abaixo foram criadas para informar as medidas de colesterol LDL ou Glicose para converter nas categorias utilizadas no modelo (1- normal, 2 - acima do normal, 3 - alto)

```
#Funcoes para converter colesterol LDL e Glicose em classificacao normal, acima do normal ou alto

# referencia - https://cardiologistaabc.com.br/index.php/our-
destination/colesterol/
#def ldl_to_category(ldl):
#    if ldl < 100:
#        return 1
#    elif ldl < 160:
#        return 2
#    else:
#        return 3

#referencia - https://diabetes.org/about-diabetes/diagnosis
#def glucose_to_category(glucose_mg_dl):
#    if glucose_mg_dl < 100:
#        return 1 # normal
#    elif glucose_mg_dl < 126:
#        return 2 # acima do normal (pré-diabetes)
#    else:
#        return 3 # alto (diabetes)

# Removendo os registros duplicados.

df_sem_duplicadas = data_frame.drop("id", axis=1).drop_duplicates()
df_sem_duplicadas.shape

(69976, 12)

# Filtrar apenas registros válidos para pressão arterial

# ap_hi -> Maior ou igual a 30 e menor ou igual a 280
# ap_lo -> Maior ou igual a 20 e menor ou igual a 200
df_pradonizado = df_sem_duplicadas[
    (df_sem_duplicadas["ap_hi"].between(30, 280)) &
    (df_sem_duplicadas["ap_lo"].between(20, 200))
].copy()

df_pradonizado.shape
```

```
(68771, 12)

# Garantindo que o peso não é uma variável escalável (float)
df_pradonizado["weight"] = df_pradonizado["weight"].astype(int)
```

Exploração dos dados

```
# Garantindo que os campos booleanos serão lidos corretamente

df_pradonizado["smoke"] = df_pradonizado["smoke"].astype(bool)
df_pradonizado["alco"] = df_pradonizado["alco"].astype(bool)
df_pradonizado["active"] = df_pradonizado["active"].astype(bool)
df_pradonizado["cardio"] = df_pradonizado["cardio"].astype(bool)

# Gráfico de correlação entre cada coluna e a coluna "cardio"

features = df_pradonizado.drop(columns=["cardio"])

correlacoes = features.corrwith(df_pradonizado["cardio"])
correlacoes = correlacoes.sort_values()

plt.figure(figsize=(10, 6))
bars = plt.barh(correlacoes.index, correlacoes.values)

plt.xlabel("Correlação com cardio")
plt.title("Correlação das variáveis com problema cardíaco")

plt.axvline(0, linestyle="--")

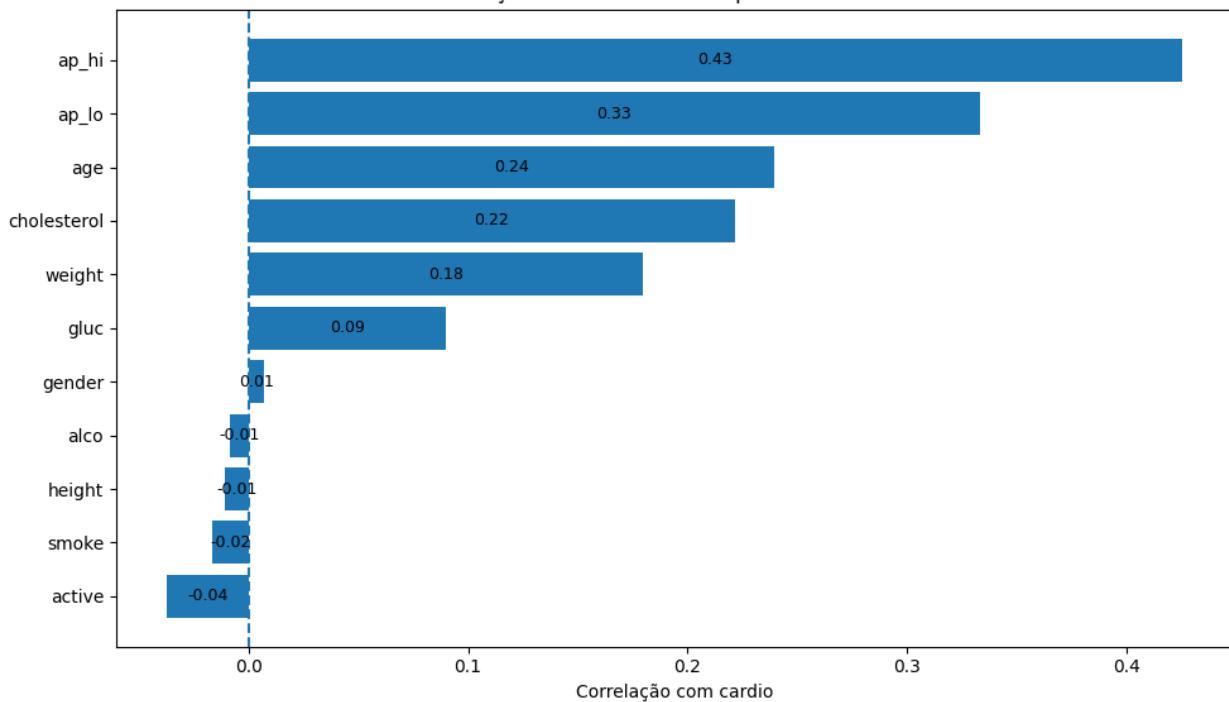
# === RÓTULOS DENTRO DAS BARRAS ===

for bar in bars:
    valor = bar.get_width()
    y = bar.get_y() + bar.get_height() / 2

    plt.text(
        valor / 2,
        y,
        f"{valor:.2f}",
        va="center",
        ha="center",
        fontsize=9,
        color="black"
    )

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

Correlação das variáveis com problema cardíaco



```
# Coeficientes de correlação entre colunas separadamente
```

```
plt.rcParams["figure.figsize"] = (18,8)
ax = sns.heatmap(df_pradonizado.corr(), annot = True)
```



```
# =====
# 1. Preparação dos dados
# =====
```

```

df = df_pradonizado.copy()

# Garantir que são binárias (0/1)
cols_binarias = ["smoke", "alco", "active", "cardio"]
df[cols_binarias] = df[cols_binarias].astype(int)

# Criar rótulo do perfil comportamental
df["perfil"] = (
    "Smoke=" + df["smoke"].astype(str) +
    " | Alco=" + df["alco"].astype(str) +
    " | Active=" + df["active"].astype(str)
)

# =====
# 2. Cálculo do risco por perfil
# =====

risco_por_perfil = (
    df
    .groupby("perfil")["cardio"]
    .mean()
    .sort_values()
)
risco_percentual = risco_por_perfil * 100

# =====
# 3. Gráfico
# =====

plt.figure(figsize=(12, 6))

bars = plt.barh(
    risco_percentual.index,
    risco_percentual.values
)

plt.xlabel("Percentual com problema cardíaco (%)")
plt.title("Risco de problema cardíaco por combinação de comportamentos")

# Rótulos de valor nas barras
for bar in bars:
    valor = bar.get_width()
    y = bar.get_y() + bar.get_height() / 2

    plt.text(
        valor + 0.5,
        y,

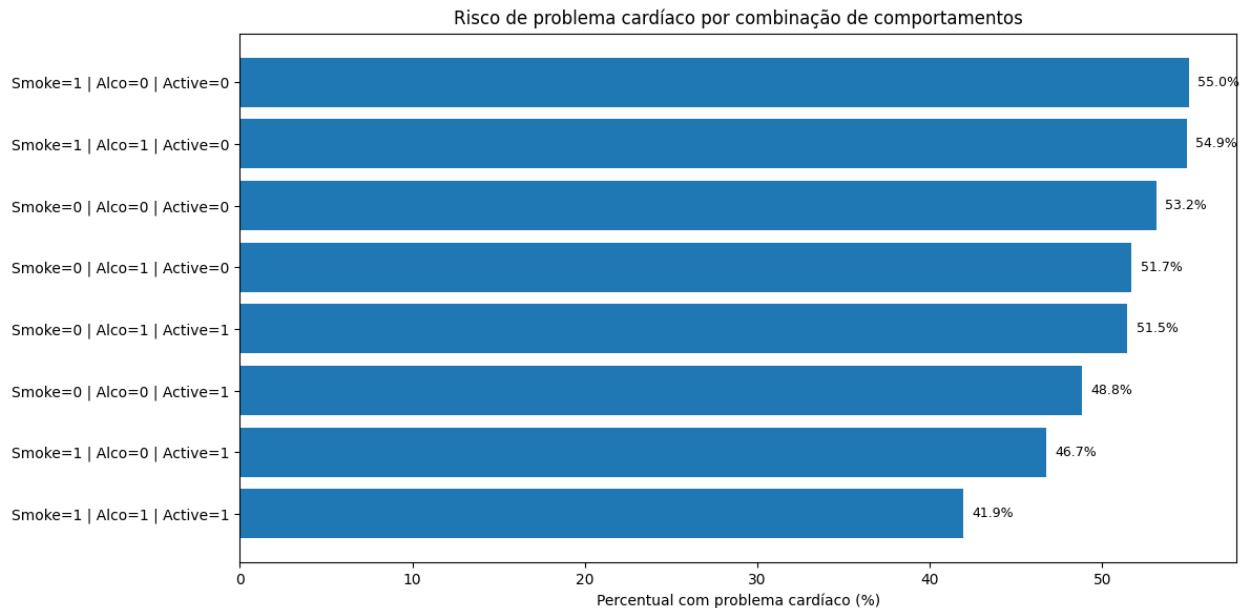
```

```

        f" {valor:.1f}%",
        va="center",
        fontsize=9
    )

plt.tight_layout()
plt.show()

```



Criando data_frame apenas de resultados onde o paciente apresentou problemas cardiovasculares

```

data_frame_cardio_true = df_pradonizado[df_pradonizado["cardio"] == True].copy()

df = data_frame_cardio_true.copy()
df["age"] = (df["age"] // 365).astype(int)
df["age_range_5"] = (df["age"] // 5) * 5

percentual_por_faixa = (
    df["age_range_5"]
    .value_counts(normalize=True)
    .sort_index() * 100
)

maior_percentual = percentual_por_faixa.max()
faixa_pico = percentual_por_faixa.idxmax()
legenda = f"Pico: {maior_percentual:.2f}% ({faixa_pico}-{faixa_pico+4} anos)"

fig, axes = plt.subplots(1, 2, figsize=(18, 6))

```

```

# === Gráfico de linhas ===
axes[0].plot(percentual_por_faixa.index, percentual_por_faixa.values)

axes[0].set_xlabel("Faixa etária (anos)")
axes[0].set_ylabel("Percentual (%)")

axes[0].axhline(
    y=maior_percentual,
    linestyle="--",
    label = legenda)

axes[0].axvline(x=faixa_pico, linestyle="--")

# === Gráfico de barras horizontais ===
bars = axes[1].barh(percentual_por_faixa.index,
percentual_por_faixa.values)

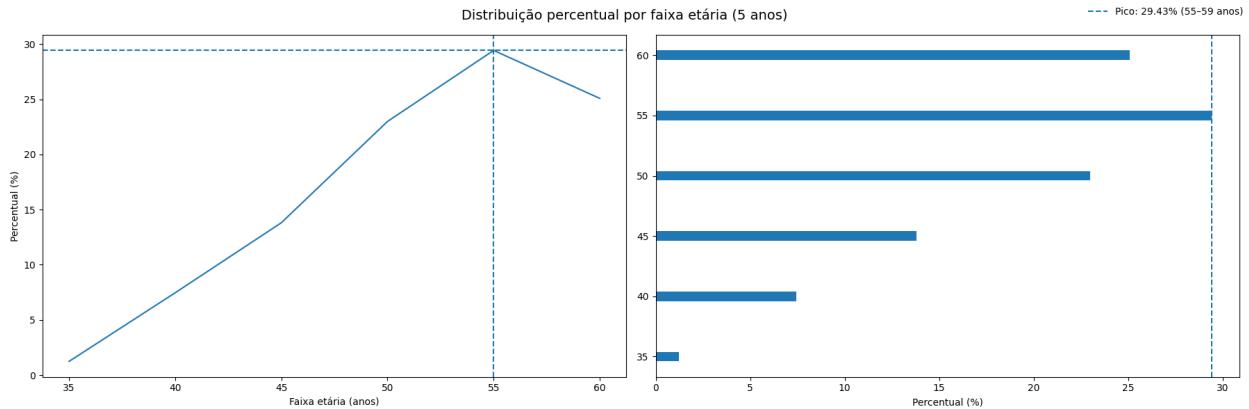
axes[1].set_xlabel("Percentual (%)")
axes[1].axvline(x=maior_percentual,
    linestyle="--")

# === Legenda ===
handles, labels = axes[0].get_legend_handles_labels()

fig.legend(
    handles,
    labels,
    frameon=False
)

fig.suptitle("Distribuição percentual por faixa etária (5 anos)",
fontsize=14)
plt.tight_layout()
plt.show()

```



Treinamento

Importando as bibliotecas utilizadas na criação dos modelos

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score, RocCurveDisplay, roc_curve, accuracy_score,
recall_score
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB, BernoulliNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
import shap
import random
```

Separando em treino e teste

```
x = df_pradonizado.drop(["cardio"], axis = 1)
y = df_pradonizado["cardio"] # Target

x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, stratify=y, random_state=42 )

# Padronizando os valores

scaler = StandardScaler()
x_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)
```

Teste Modelo 1: KNeighborsClassifier

Tentando encontrar o melhor K

```
#Tentando encontrar o melhor K

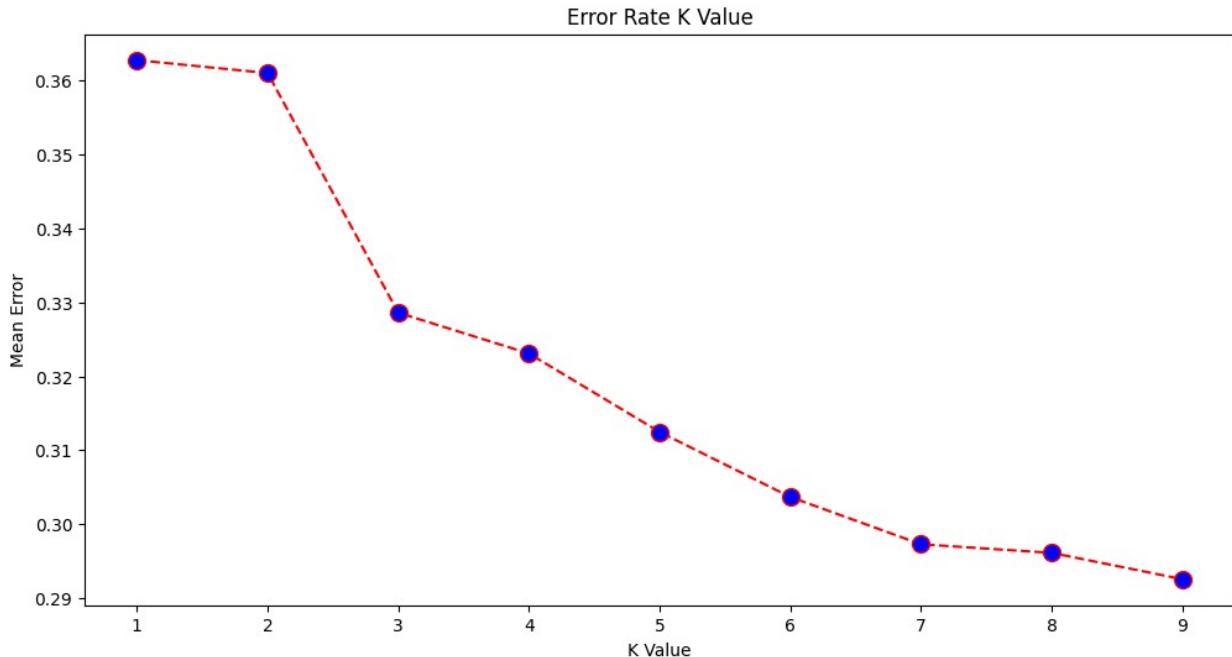
error = []

for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(x_scaled, y_train)
    pred_i = knn.predict(x_test_scaled)
    error.append(np.mean(pred_i != y_test))
```

```

plt.figure(figsize=(12,6))
plt.plot(range(1,10), error, color='red', linestyle='dashed',
marker='o', markerfacecolor='blue', markersize=10)
plt.title("Error Rate K Value")
plt.xlabel("K Value")
plt.ylabel("Mean Error")
Text(0, 0.5, 'Mean Error')

```



K com menos erros : 7 e 9

```
best_knn = 7
```

Criando o Pipeline para executar o StandardScaler e KNN

```

pipeline_knn = Pipeline([
    ("scaler", StandardScaler()),
    ("knn", KNeighborsClassifier(
        n_neighbors=best_knn,
        weights="distance"
    ))
])
pipeline_knn.fit(x_train, y_train)

Pipeline(steps=[('scaler', StandardScaler()),
    ('knn',
        KNeighborsClassifier(n_neighbors=7,
weights='distance'))])

```

```

y_pred = pipeline_knn.predict(x_test)
y_proba = pipeline_knn.predict_proba(x_test)[:, 1]

print(accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

0.6934932751726645
[[4883 2063]
 [2153 4656]]
      precision    recall   f1-score   support
  False       0.69      0.70      0.70      6946
   True       0.69      0.68      0.69      6809

   accuracy          0.69      0.69      0.69      13755
   macro avg       0.69      0.69      0.69      13755
weighted avg       0.69      0.69      0.69      13755

```

69% dos casos positivos seriam identificados corretamente os 31% dos doentes positivos seriam dados como negativo. Isto é ruim clinicamente, temos que aumentar o recall

```

# Aumentando a probabilidade de ser positivo
# isso vai aumentar o caso de FALSO POSITIVOS, mas clinicamente é mais
# aceitável do que FALSO NEGATIVO
y_pred_custom = (y_proba >= 0.30).astype(int)

print(accuracy_score(y_test, y_pred_custom))
print(confusion_matrix(y_test, y_pred_custom))
print(classification_report(y_test, y_pred_custom))

0.6601235914213014
[[3452 3494]
 [1181 5628]]
      precision    recall   f1-score   support
  False       0.75      0.50      0.60      6946
   True       0.62      0.83      0.71      6809

   accuracy          0.66      0.66      0.66      13755
   macro avg       0.68      0.66      0.65      13755
weighted avg       0.68      0.66      0.65      13755

```

Teste Modelo 2 : Classificador Naive Bayes Gauseano

```

x = df_pradonizado.drop(["cardio"], axis = 1)
y = df_pradonizado["cardio"]
print(x)

```

```

      age gender height weight ap_hi ap_lo cholesterol gluc
smoke \
0    18393     2    168     62    110     80             1     1
False
1    20228     1    156     85    140     90             3     1
False
2    18857     1    165     64    130     70             3     1
False
3    17623     2    169     82    150    100             1     1
False
4    17474     1    156     56    100     60             1     1
False
...
...
69995  19240     2    168     76    120     80             1     1
True
69996  22601     1    158    126    140     90             2     2
False
69997  19066     2    183    105    180     90             3     1
False
69998  22431     1    163     72    135     80             1     2
False
69999  20540     1    170     72    120     80             2     1
False

      alco active
0    False   True
1    False   True
2    False  False
3    False   True
4    False  False
...
69995  False   True
69996  False   True
69997  True  False
69998  False  False
69999  False   True

[68771 rows x 11 columns]

X_train, X_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=42)

modelo_naive = GaussianNB()
modelo_naive.fit(X_train, y_train)
y_pred = modelo_naive.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(accuracy_score(y_test, y_pred))

```

```

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

0.7144311159578335
[[5651 1284]
 [2644 4176]]
      precision    recall   f1-score   support
  False       0.68     0.81     0.74     6935
   True       0.76     0.61     0.68     6820
   accuracy          0.71          0.71        13755
    macro avg       0.72     0.71     0.71        13755
 weighted avg       0.72     0.71     0.71        13755

```

Teste Modelo 3 : Classificador Naive Bayes Bernoulli

```

x = df_pradonizado.drop(["cardio"], axis = 1)
y = df_pradonizado["cardio"]
print(x)

      age gender height weight ap_hi ap_lo cholesterol gluc
smoke \
0     18393     2     168     62    110     80             1     1
False
1     20228     1     156     85    140     90             3     1
False
2     18857     1     165     64    130     70             3     1
False
3     17623     2     169     82    150    100             1     1
False
4     17474     1     156     56    100     60             1     1
False
...
...
69995  19240     2     168     76    120     80             1     1
True
69996  22601     1     158    126    140     90             2     2
False
69997  19066     2     183    105    180     90             3     1
False
69998  22431     1     163     72    135     80             1     2
False
69999  20540     1     170     72    120     80             2     1
False

      alco active
0    False   True
1    False   True

```

```

2      False  False
3      False  True
4      False  False
...
69995  False  True
69996  False  True
69997  True   False
69998  False  False
69999  False  True

[68771 rows x 11 columns]

X_train, X_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=42)

modelo_naive = BernoulliNB()
modelo_naive.fit(X_train, y_train)
y_pred = modelo_naive.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
accuracy

0.5211922937113777

print(accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

0.5211922937113777
[[5686 1249]
 [5337 1483]]
          precision    recall  f1-score   support
      False       0.52      0.82      0.63     6935
      True       0.54      0.22      0.31     6820

      accuracy                           0.52    13755
     macro avg       0.53      0.52      0.47    13755
  weighted avg       0.53      0.52      0.47    13755

```

Teste Modelo 4 : DecisionTreeClassifier

```

x = df_pradonizado.drop(["cardio"], axis = 1)
y = df_pradonizado["cardio"] # Target

x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, stratify=y, random_state=42)

# Padronizando os valores
scaler = StandardScaler()

```

```

x_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

##### DecisionTreeClassifier

clf = DecisionTreeClassifier(max_depth=5, random_state=42)
clf.fit(x_train, y_train)

y_pred = clf.predict(x_test)

print("Acurácia:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))

Acurácia: 0.7242457288258815
      precision    recall   f1-score   support
      False        0.69      0.82      0.75      6946
      True         0.78      0.62      0.69      6809
      accuracy           0.72      0.72      0.72      13755
      macro avg       0.73      0.72      0.72      13755
      weighted avg     0.73      0.72      0.72      13755

```

O modelo acerta cerca de 72% dos casos. É um desempenho razoável com uma árvore rasa (profundidade 5), indicando que já há padrões capturados sem exagerar na complexidade.

Teste Modelo 5 : RandomForestClassifier

```

x = df_pradonizado.drop(["cardio"], axis = 1)
y = df_pradonizado["cardio"] # Target

x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, stratify=y, random_state=42)

# Padronizando os valores
scaler = StandardScaler()
x_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

##### RandomForestClassifier

# Criando o modelo Random Forest
rf_clf = RandomForestClassifier(
    n_estimators=100,          # número de árvores
    max_depth=None,           # profundidade máxima (None = cresce até o
fim)                         # fim
    random_state=42,
    n_jobs=-1                 # usa todos os núcleos do processador
)

# Treinando o modelo

```

```

rf_clf.fit(x_train, y_train)

# Fazendo previsões
y_pred_rf = rf_clf.predict(x_test)

# Avaliando o modelo
print("Acurácia RandomForest:", accuracy_score(y_test, y_pred_rf))
print("\nMatriz de Confusão:\n", confusion_matrix(y_test, y_pred_rf))
print("\nRelatório de Classificação:\n", classification_report(y_test,
y_pred_rf))

Acurácia RandomForest: 0.7092693565976008

Matriz de Confusão:
[[5040 1906]
 [2093 4716]]

Relatório de Classificação:
      precision    recall   f1-score   support
  False        0.71     0.73     0.72     6946
  True        0.71     0.69     0.70     6809

  accuracy          0.71      --      0.71     13755
  macro avg       0.71     0.71     0.71     13755
weighted avg     0.71     0.71     0.71     13755

```

O modelo acerta cerca de 71% dos casos.

Teste Modelo 6 : GradientBoostingClassifier

```

x = df_pradonizado.drop(["cardio"], axis = 1)
y = df_pradonizado["cardio"] # Target

x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, stratify=y, random_state=42)

# Padronizando os valores
scaler = StandardScaler()
x_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Criando o modelo Gradient Boosting
gb_clf = GradientBoostingClassifier(
    n_estimators=200,      # número de árvores (quanto maior, mais
robusto, mas mais lento)
    learning_rate=0.1,     # taxa de aprendizado (menor = mais estável,
maior = mais rápido)
    max_depth=3,           # profundidade máxima das árvores
    random_state=42

```

```

)

# Treinando o modelo
gb_clf.fit(x_train, y_train)

# Fazendo previsões
y_pred_gb = gb_clf.predict(x_test)

# Avaliando o modelo
print("Acurácia GradientBoosting:", accuracy_score(y_test, y_pred_gb))
print("\nMatriz de Confusão:\n", confusion_matrix(y_test, y_pred_gb))
print("\nRelatório de Classificação:\n", classification_report(y_test,
y_pred_gb))

Acurácia GradientBoosting: 0.7301344965467103

Matriz de Confusão:
[[5384 1562]
 [2150 4659]]

Relatório de Classificação:
      precision    recall   f1-score   support
  False        0.71     0.78     0.74     6946
  True        0.75     0.68     0.72     6809

  accuracy          0.73
  macro avg        0.73     0.73     0.73     13755
weighted avg       0.73     0.73     0.73     13755

```

Desempenho global de 73%, superior ao RandomForest (71%) e próximo da árvore simples (72%), indicando melhor capacidade de generalização com ajustes moderados.

Teste Modelo 7: Regressão Logística

```
model_logistic = LogisticRegression(max_iter=1000)
```

Validando Funcionamento Inicial do Modelo

```

model_logistic.fit(x_scaled, y_train)
y_prob = model_logistic.predict_proba(x_test_scaled)[:, 1]
y_pred = (y_prob >= 0.5).astype(int)

print(classification_report(y_test, y_pred))
print("Acurácia:", accuracy_score(y_test, y_pred))
print("ROC-AUC:", roc_auc_score(y_test, y_prob))
print("Recall Score:", recall_score(y_test, y_pred))
print("Matriz de Confusão: \n", confusion_matrix(y_test, y_pred))

```

	precision	recall	f1-score	support
False	0.70	0.78	0.74	6946
True	0.75	0.67	0.70	6809
accuracy			0.72	13755
macro avg	0.72	0.72	0.72	13755
weighted avg	0.72	0.72	0.72	13755

Acurácia: 0.7224282079243911

ROC-AUC: 0.78765120366893

Recall Score: 0.6670583051843149

Matriz de Confusão:

```
[[5395 1551]
 [2267 4542]]
```

Identificando melhor conjunto de parâmetros para o modelo

```
pipe_logistic_regression = Pipeline([
    ("scaler", StandardScaler()),
    ("model", LogisticRegression(max_iter=1000))
])

# Cada Solver tem um range de penalty's disponíveis. Por conta disso,
# estão sendo testados três conjuntos de solvers/penalty's diferentes
param_grid = [
    {
        "model_solver": ["lbfgs", "newton-cg", "sag"],
        "model_penalty": ["l2"],
        "model_C": [0.1, 1, 10],
        "model_class_weight": [None, "balanced"]
    },
    {
        "model_solver": ["liblinear"],
        "model_penalty": ["l1", "l2"],
        "model_C": [0.1, 1, 10],
        "model_class_weight": [None, "balanced"]
    },
    {
        "model_solver": ["saga"],
        "model_penalty": ["l1", "l2", "elasticnet"],
        "model_l1_ratio": [0.5],
        "model_C": [0.1, 1, 10],
        "model_class_weight": [None, "balanced"]
    }
]

grid = GridSearchCV(
    estimator=pipe_logistic_regression,
```

```

param_grid=param_grid,
scoring="recall", # priorizando recall
cv=5,
n_jobs=-1,
verbose=2
)

grid.fit(X_train, y_train)
best_model = grid.best_estimator_

print("Melhor Modelo", best_model)

Fitting 5 folds for each of 48 candidates, totalling 240 fits
Melhor Modelo Pipeline(steps=[('scaler', StandardScaler()),
    ('model',
        LogisticRegression(C=1, class_weight='balanced',
max_iter=1000,
                           penalty='l2', solver='newton-
cg'))])

c:\Users\User\Documents\pos\tech challenge-cardio\treinamento-cardio\
env\Lib\site-packages\sklearn\linear_model\_logistic.py:1135:
FutureWarning: 'penalty' was deprecated in version 1.8 and will be
removed in 1.10. To avoid this warning, leave 'penalty' set to its
default value and use 'l1_ratio' or 'C' instead. Use l1_ratio=0
instead of penalty='l2', l1_ratio=1 instead of penalty='l1', and
C=np.inf instead of penalty=None.
warnings.warn(

```

Encontrando o Threshold ideal

```

fpr, tpr, thresholds = roc_curve(y_test, y_prob)
best_idx = np.argmax(tpr - fpr)
# Melhor threshold em geral
best_threshold = thresholds[best_idx]
# Melhor threshold para garantir um recall acima de 80%
best_recall_threshold = thresholds[tpr >= 0.80][0]

print("Threshold ideal:", best_threshold)
print("Threshold ideal para garantir um recall acima de 80%:",
best_recall_threshold)

```

```

Threshold ideal: 0.48445910721289315
Threshold ideal para garantir um recall acima de 80%:
0.3996763656875961

```

Criando novo pipeline com parâmetros e threshold ideais

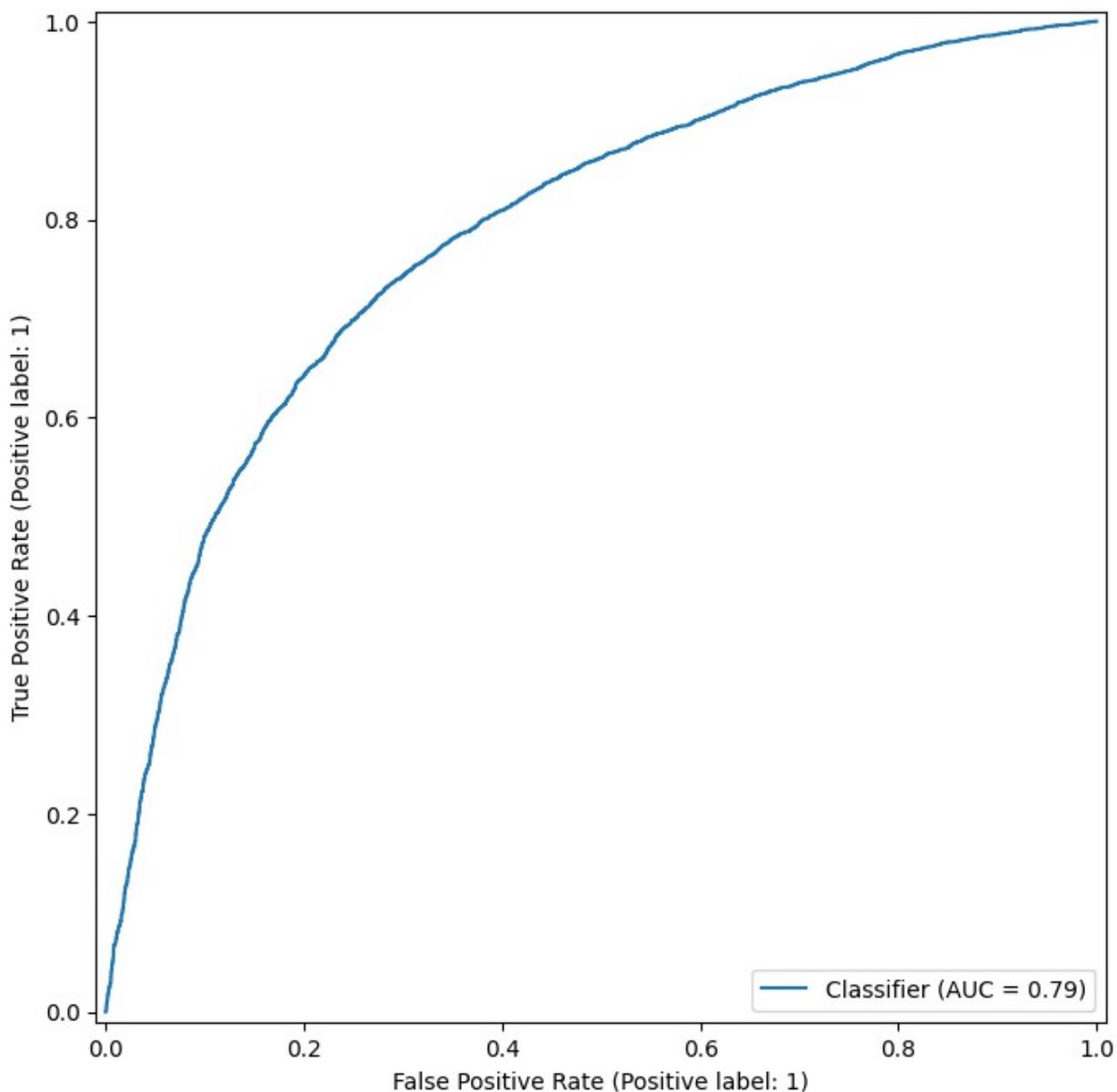
```
pipe_logistic_regression_best = Pipeline([
    ("scaler", StandardScaler()),
    ('model', LogisticRegression(C=1, class_weight='balanced',
max_iter=1000,solver='newton-cg'))
])

pipe_logistic_regression_best.fit(x_train, y_train)

Pipeline(steps=[('scaler', StandardScaler()),
               ('model',
                LogisticRegression(C=1, class_weight='balanced',
max_iter=1000,
                           solver='newton-cg'))])

y_prob = pipe_logistic_regression_best.predict_proba(x_test)[:, 1]

RocCurveDisplay.from_predictions(
    y_test, y_prob / 100
)
plt.show()
```



A curva ROC-AUC indica que, ao comparar aleatoriamente um paciente doente com um paciente saudável, o modelo atribui uma probabilidade maior ao paciente doente em aproximadamente 79% das vezes.

Testando os valores de threshold identificados anteriormente

```
y_pred = (y_prob >= best_threshold).astype(int) # utilizando o melhor threshold, o valor do recall está abaixo do aceitável

print(classification_report(y_test, y_pred))
print("Acurácia:", accuracy_score(y_test, y_pred))
```

```
print("Recall:", recall_score(y_test, y_pred))
print("Matriz de Confusão: \n", confusion_matrix(y_test, y_pred))
```

	precision	recall	f1-score	support
False	0.72	0.75	0.73	6946
True	0.73	0.69	0.71	6809
accuracy			0.72	13755
macro avg	0.73	0.72	0.72	13755
weighted avg	0.73	0.72	0.72	13755

Acurácia: 0.7245365321701199

Recall: 0.6949625495667499

Matriz de Confusão:

```
[[5234 1712]
 [2077 4732]]
```

y_pred = (y_prob >= best_recall_threshold).astype(int) # utilizando o melhor threshold para garantir o recall acima de 80%. Dessa forma, os casos falsos negativos irão diminuir, entretanto teremos menos casos falsos positivos

```
print(classification_report(y_test, y_pred))
print("Acurácia:", accuracy_score(y_test, y_pred))
print("Recall Score:", recall_score(y_test, y_pred))
print("Matriz de Confusão: \n", confusion_matrix(y_test, y_pred))
```

	precision	recall	f1-score	support
False	0.76	0.61	0.68	6946
True	0.67	0.81	0.73	6809
accuracy			0.71	13755
macro avg	0.71	0.71	0.70	13755
weighted avg	0.72	0.71	0.70	13755

Acurácia: 0.7057070156306797

Recall Score: 0.8054046115435453

Matriz de Confusão:

```
[[4223 2723]
 [1325 5484]]
```

Apesar de utilizar um valor abaixo do ideal para o threshold, o modelo perdeu ~2% de acurácia. Entretanto, quando pensamos em um contexto médico/hospitalar, é melhor termos um número de falsos positivos (pacientes que não tem problemas cardíacos, porém foram identificados com tal), do que correr o risco de deixarmos algum paciente doente passar sem identificação.

Utilizando o threshold para priorizar o valor de recall, o modelo apresentou uma acurácia da ~71%, mantendo o valor de recall em ~80%

Utilizando Shap para identificar os fatores mais relevantes para a decisão do modelo

```
X_train_scaled =
pipe_logistic_regression_best.named_steps["scaler"].transform(x_train)
X_test_scaled =
pipe_logistic_regression_best.named_steps["scaler"].transform(x_test)

feature_names = x.columns

explainer = shap.LinearExplainer(
    pipe_logistic_regression_best.named_steps["model"],
    X_train_scaled,
    feature_names=feature_names
)

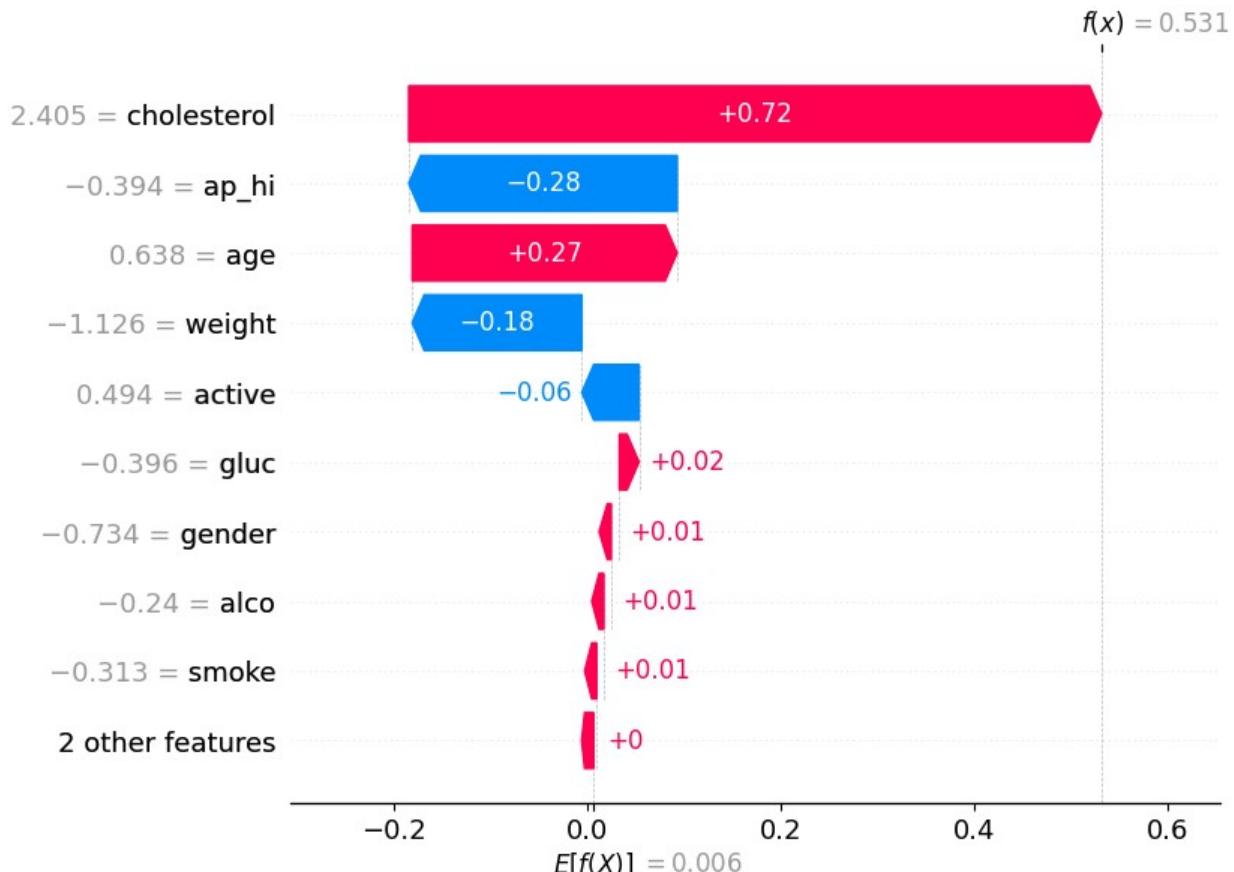
shap_values = explainer(X_test_scaled)

# Selecionando um caso aleatório dentro dos casos separados para teste
i = np.random.randint(0, X_test_scaled.shape[0])

base_log_odds = explainer.expected_value
patient_log_odds = base_log_odds + shap_values[i].values.sum()

prob = 1 / (1 + np.exp(-patient_log_odds))

shap.plots.waterfall(shap_values[i])
print(f"Probabilidade do paciente ter problema cardíaco: {prob}")
```



Probabilidade do paciente ter problema cardíaco: 0.6297940115285232

Salvando o melhor modelo para utilizar na API

```
import joblib
joblib.dump(pipe_logistic_regression_best,
'pipe_logistic_regression.pkl')
['pipe_logistic_regression.pkl']
```