Universidade Federal de Pernambuco

IF1014 - Tópicos avançados em SI 5

Relatório - Variação Paramétrica

Disciplina: IF1014 - Tópicos avançados em SI 5
Docente: Leandro Maciel Almeida
Discentes: Carlos Clark, Ian Gabriel, Igor Rocha, Williams Santos (Grupo 9)

Fase de Variação Paramétrica (CRISP-DM) para o Dataset NOMAO

1. Base de Dados

a. Conjunto

O dataset Nomao foi originalmente utilizado no Nomao Challenge, uma competição de Data Mining organizada pela ALRA (Active Learning in Real-World Applications) em 2012. Seu principal objetivo é auxiliar na identificação de registros duplicados em uma base de dados composta por informações de locais extraídas de diversas fontes.

O desafio central do dataset consiste em determinar se duas entradas distintas representam o mesmo local, mesmo que apresentem diferenças na descrição. Esse problema é particularmente relevante para empresas que dependem de dados geoespaciais, como serviços de mapas, plataformas de delivery, empresas de logística e redes varejistas. A existência de registros duplicados pode comprometer a qualidade dos serviços e a eficiência operacional dessas empresas.

O dataset é composto por dois arquivos principais:

- **nomao.data**: Contém os dados brutos organizados em formato CSV (Comma-Separated Values).
- **nomao.name**: Fornece a descrição detalhada dos atributos presentes no dataset, que incluem informações como nome do local, endereço, cidade, código postal, número de telefone, fax e coordenadas geográficas.

Além disso, foi criado o arquivo nomao.features, que contém apenas os nomes das features presentes no dataset. Esse arquivo foi elaborado para facilitar o processamento e a codificação das informações, simplificando a extração das features relevantes.

b. Características

O dataset Nomao possui um total de 34.466 registros e 120 variáveis. Embora inicialmente todas as colunas tenham sido identificadas como do tipo "object", a análise do arquivo nomao.name revelou que a tipagem real dos atributos é composta por 89 variáveis numéricas e 31 variáveis categóricas.

Os atributos do dataset refletem diferentes aspectos de comparação entre dois registros, como similaridade de nome, endereço e coordenadas geográficas. Algumas variáveis possuem valores contínuos, representando diferenças quantitativas entre os locais, enquanto outras utilizam categorias específicas (n, s, m) para indicar diferentes níveis de correspondência. Além disso, valores ausentes são representados pelo caractere "?", necessitando de um tratamento adequado durante o pré-processamento dos dados.

Para garantir uma análise eficiente e preparar os dados para a modelagem, realizamos a separação entre variáveis numéricas e categóricas. Após essa distinção, constatamos que o dataset é composto por 89 colunas numéricas e 30 colunas categóricas (excluindo a label, que havia sido removida previamente). Essa predominância de variáveis numéricas pode impactar a escolha de técnicas estatísticas e algoritmos de machine learning mais apropriados para a tarefa de classificação.

A estrutura detalhada e a separação dos tipos de variáveis facilitam o tratamento dos dados e permitem uma abordagem mais eficaz na

construção dos modelos preditivos, garantindo maior precisão na detecção de registros duplicados.

2. Preparação dos Dados

a. Divisão do conjunto de teste e treinamento

Para separar o *dataset* Nomao em conjuntos de treino e teste, utilizamos a função `train_test_split` da biblioteca *scikit-learn*, reservando 80% dos dados para treinamento e 20% para teste. Esse procedimento foi realizado de forma estratificada, de modo a manter a proporção original das classes e evitar possíveis vieses de amostragem.

```python
import pandas as pd
from sklearn.model_selection import train_test_split


X = nomao_df
y = label_col              # Coluna-alvo

# Dividindo o dataset em treino (80%) e teste (20%), mantendo a consistência das classes
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    train_size=0.8,
    random_state=42,
    stratify=y
)

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

X_train = X_train.drop(columns=['id'], errors='ignore')
X_test = X_test.drop(columns=['id'], errors='ignore')

train_df = pd.concat([X_train, y_train], axis=1)

test_df = pd.concat([X_test, y_test], axis=1)
```

```
Tamanho do treino: 27572
Tamanho do teste : 6893
```

Em seguida, concatenamos `X_train` e `y_train` para formar o *DataFrame* `train_df`, e fizemos o mesmo com `X_test` e `y_test` para criar o `test_df`. Todas as etapas de pré-processamento (imputação, encoding, normalização, etc.) foram ajustadas exclusivamente em `train_df` e, posteriormente, aplicadas em `test_df`, porém muitas

mudanças que adotamos não possuem natureza estatística, logo a natureza da nossa metodologia previne “Data Leakage”.

b. Remoção da coluna ID

Removemos a coluna de identificação de ambos os datasets, passo necessário para rodar os modelos.

```
X_train = X_train.drop(columns=['id'], errors='ignore')
X_test = X_test.drop(columns=['id'], errors='ignore')
```

c. Seleção de Colunas Numéricas

Identificamos as colunas numéricas utilizando o trecho de código apresentado abaixo, vital para a realização da imputação sobre as colunas numéricas posteriormente.

```
# 2) Selecionar colunas numéricas e preencher valores faltantes com -1
numeric_cols = train_df.select_dtypes(include=[np.number]).columns
train_df[numeric_cols] = train_df[numeric_cols].fillna(-1)
```

d. Imputação

Preenchemos valores ausentes dessas colunas numéricas com -1, de modo a não introduzir estatísticas do dataset (por exemplo, média ou mediana) que poderiam distorcer o modelo. Em seguida, replicamos a mesma operação no conjunto de teste. Nosso embasamento para tal escolha se deu devido a uma hipótese levantada sobre a natureza do problema. O dataset “Nomao” consiste em dados relacionados a localização, logo, utilizar medidas estatísticas pode resultar em distorções na previsão dos modelos, portanto realizamos uma revisão de literatura sobre tal ponto, encontrando uma referência interessante sobre a “Nomao Challenge”, segue o trecho abaixo:

II-A. Análisis de datos y pre-procesamiento

El dataset no presenta datos fuera de rango, ya que todas las variables continuas estan dentro del dominio especificado: entre 0 y 1. Las variables categóricas también respetan el estándar: no hay ninguna que contenga un valor no especificado. Estas variable se convirtieron a variables dummies, con el objetivo de poder aplicar algoritmos que requieran variables numéricas. Las variables categóricas originales son eliminadas del dataset, dejando sólamente las dummies como entrada de los algorítmos.

Todas las variables continuas, excepto las que comienzan con el nombre *clean_name*, tienen datos faltantes. Como el rango de estas variables es de 0 a 1, todas aquellas que tienen datos faltantes se las reemplaza por el valor -1. No hay una justificación teórica de por que se escoge el valor -1, pero los clasificadores respondieron efectivamente a este valor.

II-A. Análise de dados e pré-processamento

O dataset não apresenta dados fora de faixa, pois todas as variáveis contínuas estão dentro do domínio especificado: entre 0 e 1. As variáveis categóricas também respeitam o padrão: não há nenhuma que contenha um valor não especificado. Essas variáveis foram convertidas em variáveis dummy, com o objetivo de aplicar algoritmos que exijam variáveis numéricas. As variáveis categóricas originais são removidas do dataset, mantendo apenas as dummies no momento dos algoritmos.

Todas as variáveis contínuas, exceto aquelas que começam com o nome *clean_name*, apresentam dados faltantes. Como não há valores fora do intervalo de 0 a 1, todos aqueles que possuíam dados faltantes foram substituídos pelo valor -1. Não existe uma justificativa teórica específica para o uso de -1, mas os classificadores respondem de forma eficaz a esse valor.

Por tal motivo, adotamos por hora a mesma metodologia empregada no artigo, vamos primeiramente observar os resultados da aplicação de imputação com o valor -1 em colunas numéricas, posteriormente avaliando a aplicabilidade de inputação estatística em caso de uma performance indesejável do modelo.

e. Verificação

Após o preenchimento, checamos se ainda restavam valores ausentes, confirmando que tanto `train_df` quanto `test_df` estavam livres de `NaN`. (Essa verificação por hora reflete as colunas numéricas apenas).

f.

```
[25]
missing_numeric = train_df[numeric_cols].isnull().sum()
print("Valores faltantes em colunas numéricas (treino):\n", missing_numeric)
print("Total faltante numérico (treino):", missing_numeric.sum())
```

```
Valores faltantes em colunas numéricas (treino):
 clean_name_intersect_min       0
clean_name_intersect_max        0
clean_name_levenshtein_sim      0
clean_name_trigram_sim          0
clean_name_levenshtein_term     0
                               ..
coordinates_lat_diff            0
coordinates_lat_levenshtein     0
coordinates_lat_trigram         0
geocode_coordinates_diff        0
coordinates_diff                0
Length: 89, dtype: int64
Total faltante numérico (treino): 0
```

```
[26]
missing_numeric = test_df[numeric_colunas].isnull().sum()
print("Valores faltantes em colunas numéricas (teste):\n", missing_numeric)
print("Total faltante numérico (teste):", missing_numeric.sum())
```

```
Valores faltantes em colunas numéricas (teste):
 clean_name_intersect_min       0
clean_name_intersect_max        0
clean_name_levenshtein_sim      0
clean_name_trigram_sim          0
clean_name_levenshtein_term     0
                               ..
coordinates_lat_diff            0
coordinates_lat_levenshtein     0
coordinates_lat_trigram         0
geocode_coordinates_diff        0
coordinates_diff                0
Length: 89, dtype: int64
Total faltante numérico (teste): 0
```

g. Seleção de Colunas Categóricas

Usamos `select_dtypes(include=['category'])` para identificar colunas não numéricas, que continham os valores "s", "n" e "m".

```
# Selecionar colunas categóricas corretamente
cat_cols = train_df.select_dtypes(include=['category']).columns
```

h. One-Hot Encoding

Para converter essas colunas em formato adequado aos algoritmos de Machine Learning, aplicamos `pd.get_dummies`, gerando colunas binárias (0/1) para cada categoria distinta. Em seguida, garantimos que as mesmas categorias fossem criadas no conjunto de teste, alinhando a estrutura das colunas de treino e teste.

```
# Selecionar colunas categóricas corretamente (as que não são numéricas de fato)
cat_cols_test = test_df.select_dtypes(include=['category']).columns

# Garantir que há colunas categóricas antes de aplicar One-Hot Encoding
if len(cat_cols_test) > 0:
    # Aplicar One-Hot Encoding
    # 4) Aplicar get_dummies no test_df
    test_df = pd.get_dummies(test_df, columns=cat_cols_test, drop_first=True)
    test_df = test_df.astype('float64')  # Garantir que os tipos são float
    print("One-Hot Encoding aplicado com sucesso!")
    test_df = test_df.reindex(columns=train_df.columns, fill_value=0)  # Garantir que as colunas são as mesmas
else:
    print("Nenhuma coluna categórica encontrada para One-Hot Encoding.")

# Conferir se ainda há valores nulos após a transformação
missing_data_cat = test_df.isnull().sum()
total_missing_cat = missing_data_cat.sum()
print("Valores faltantes após One-Hot Encoding:\n", missing_data_cat[missing_data_cat > 0])
print("Total faltante categórico (teste):", total_missing_cat)
```

```
One-Hot Encoding aplicado com sucesso!
Valores faltantes após One-Hot Encoding:
 Series([], dtype: int64)
Total faltante categórico (teste): 0
```

i. Exportação

Após todo o pré-processamento (remoção de colunas indesejadas, preenchimento de valores faltantes, identificação e codificação de colunas categóricas), gravamos os DataFrames resultantes (`train_df` e `test_df`) no formato PKL. Esse formato binário preserva índices, tipos de dados e a estrutura dos DataFrames de forma mais fiel que o CSV, além de acelerar o carregamento futuro:

Com isso, qualquer etapa de modelagem pode ser retomada diretamente a partir desses arquivos, sem necessidade de repetir o pré-processamento. Esse cuidado assegura reprodutibilidade e economia de tempo em fases posteriores do projeto.

j. Desbalanceamento

Para lidar com o desbalanceamento de classes, foi adotado o SMOTE (Synthetic Minority Over-sampling Technique), que gera amostras sintéticas da classe minoritária apenas no conjunto de treino. Esse procedimento está integrado diretamente em um pipeline junto com o classificador (por exemplo, KNN, Random Forest, etc.). Dessa forma, a cada iteração de validação cruzada (k-fold), o SMOTE é aplicado somente no subconjunto de treino, evitando vazamento de dados para a parte de validação e garantindo que o modelo seja avaliado em dados que mantêm a distribuição real do problema.

Essa estratégia assegura que o treinamento seja feito em dados balanceados (após o oversampling), ao mesmo tempo em que a avaliação reflete fielmente o desempenho em dados não balanceados, preservando a integridade das métricas e evitando resultados inflados artificialmente.

```python
from sklearn.ensemble import RandomForestClassifier

# Definindo o pipeline com SMOTE + RandomForest
pipeline_rf = Pipeline([
    ('smote', SMOTE(random_state=42)),
    ('rf', RandomForestClassifier(random_state=42))
])

# Definindo os hiperparâmetros para o RandomForest utilizando range e prefixando com "rf__"
param_dist_rf = {
    'rf__n_estimators': range(50, 301, 50),            # Valores: 50, 100, 150, 200, 250, 300
    'rf__max_depth': [None] + list(range(5, 26, 5)),      # None, 5, 10, 15, 20, 25
    'rf__min_samples_split': range(2, 11),              # 2 até 10
    'rf__min_samples_leaf': range(1, 5),                # 1 até 4
    'rf__bootstrap': [True, False]                      # Uso de bootstrap: True ou False
}

# Definindo a validação cruzada estratificada
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Configurando o RandomizedSearchCV para o pipeline (SMOTE + RandomForest)
rf_search = RandomizedSearchCV(
    estimator=pipeline_rf,
    param_distributions=param_dist_rf,
    n_iter=50,                  # Número de combinações aleatórias a avaliar
    scoring='f1_macro',         # Métrica de avaliação (pode ser alterada conforme necessário)
    cv=cv,
    random_state=42,
    n_jobs=-1,                  # Utiliza todos os núcleos disponíveis
    verbose=1
)

# Ajusta o pipeline com os dados de treino
rf_search.fit(X_train, y_train)
```

k. Data Leakage

Nossa estratégia se restringe apenas a tratamentos não estatísticos, mesmo assim nós utilizamos todo o pipeline necessário a garantir que tal erro não acontece em nosso dataset, porém comentamos todas as linhas de código com justificativas de que tais passos nem seriam necessários, visto que por exemplo, imputar valores faltantes com -1, não vaza informações.

3. Seleção dos modelos

Os algoritmos utilizados foram os definidos nas instruções da atividade, sendo eles:

a. K-NN

O **K-NN** é um classificador baseado em instâncias que atribui a classe de um novo exemplo com base nas classes dos seus vizinhos mais próximos. É simples, mas pode ser sensível a ruído e à escolha do número de vizinhos (k).

Para a busca de hiperparâmetros, foram testados os seguintes valores:

- **n_neighbors**: Número de vizinhos considerados no processo de classificação ou regressão, variando de 1 a 30.
    - Valores menores podem levar a um modelo mais sensível a ruídos (overfitting).
    - Valores maiores tendem a suavizar a fronteira de decisão, podendo resultar em underfitting.
- **weights**: Tipo de ponderação aplicado aos vizinhos, variando entre *uniform* e *distance*.
    - *uniform*: todos os vizinhos têm o mesmo peso.
    - *distance*: os vizinhos mais próximos recebem maior peso.
- **p**: Define a métrica de distância utilizada, variando entre 1 e 2.
    - *p = 1*: distância Manhattan (L1).
    - *p = 2*: distância Euclidiana (L2).

```
param_dist_knn = {
```

```
    'knn__n_neighbors': list(range(1, 31)),  # 1 a 30
    'knn__weights': ['uniform', 'distance'],
    'knn__p': [1, 2]  # 1=Manhattan, 2=Euclidiana
}
```

b. LVQ (Learning Vector Quantization)

LVQ (Learning Vector Quantization) é uma técnica de aprendizado supervisionado baseada em redes neurais competitivas, onde protótipos representativos são ajustados para melhor distinguir as classes dos dados.

Os parâmetros escolhidos para a busca de parâmetros do LVQ foram os seguintes:

```
param_dist = {
    'n_codebooks': [5, 10, 15, 20, 25],
    'lrate': (0.001, 0.1),  # Intervalo contínuo
    'epochs': [1,2,3]
}
```

c. Árvore de Decisão

A **Árvore de Decisão** é um algoritmo de aprendizado supervisionado que constrói um modelo baseado em regras de decisão derivadas dos dados de treinamento. É amplamente utilizado por sua interpretabilidade e capacidade de capturar padrões não lineares.

Para a busca de hiperparâmetros, foram testados os seguintes valores:

- **criterion**: Função para medir a impureza dos nós, variando entre *gini*, *entropy* e *log_loss*.
    - Cada critério permite diferentes estratégias de divisão.
- **max_depth**: Profundidade máxima da árvore, variando de 1 a 99.

    - Controla a complexidade do modelo, evitando crescimento excessivo da árvore (overfitting).
- **min_samples_split**: Número mínimo de amostras para dividir um nó, variando entre 2 e 10.

    - Impede divisões com poucos dados, auxiliando no controle do sobreajuste.
- **min_samples_leaf**: Número mínimo de amostras em cada folha, variando de 1 a 10.

    - Garante que cada folha resultante tenha amostras suficientes, contribuindo para a estabilidade do modelo.

```
# 1) Definindo o espaço de hiperparâmetros para a Decision Tree

param_dist_dt = {

    'criterion': ['gini', 'entropy', 'log_loss'],

    'max_depth': range(1, 100),

    'min_samples_split': range(2, 11),

    'min_samples_leaf': range(1, 11)

}
```

d. SVM (Support Vector Machine)

O SVM (Support Vector Machine) é um classificador que encontra um hiperplano ótimo para separar as classes, utilizando margens máximas e pode empregar kernels para lidar com dados não linearmente separáveis.

Os valores definidos para a busca de hiperparâmetros do SVM foram os seguintes:

```
# Espaço de busca dos hiperparâmetros do SVM
param_dist_svm = {
    'svm__C': [0.1, 1, 10, 100],
    'svm__kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'svm__degree': [2, 3, 4],
    'svm__gamma': ['scale', 'auto'],
    'svm__class_weight': ['balanced', None]
}
```

e. Random Forest

O Random Forest é um modelo de aprendizado baseado em um conjunto de árvores de decisão, no qual múltiplas árvores são treinadas e suas previsões são combinadas para melhorar a generalização e reduzir o sobreajuste. Esse algoritmo é amplamente utilizado por sua robustez e capacidade de lidar com dados complexos.

Para a busca de hiperparâmetros, foram testados os seguintes valores:

- **n_estimators**: Número de árvores (estimadores) na floresta.
    - Valores testados: 50, 100, 150, 200, 250 e 300.
    - Um número maior de árvores pode melhorar a performance, mas também aumenta o tempo de treinamento.
- **max_depth**: Profundidade máxima de cada árvore.
    - Valores testados: None e os valores 5, 10, 15, 20 e 25.
    - Controla o crescimento das árvores e ajuda a evitar overfitting.
- **min_samples_split**: Número mínimo de amostras necessárias para dividir um nó interno.

    - Valores testados: de 2 a 10.
    - Impede a criação de divisões baseadas em amostras muito pequenas, contribuindo para a robustez do modelo.
- **min_samples_leaf**: Número mínimo de amostras que uma folha deve conter.

    - Valores testados: de 1 a 4.
    - Garante que cada folha tenha dados suficientes, ajudando a reduzir o risco de overfitting.
- **bootstrap**: Indica se as amostras utilizadas para construir cada árvore devem ser obtidas com reposição.

    - Valores testados: `True` e `False`.
    - O uso de bootstrap pode aumentar a diversidade entre as árvores e melhorar a performance do ensemble.

```
param_dist_rf = {
    'rf__n_estimators': range(50, 301, 50),                # Valores: 50, 100, 150, 200, 250, 300
    'rf__max_depth': [None] + list(range(5, 26, 5)),       # None, 5, 10, 15, 20, 25
    'rf__min_samples_split': range(2, 11),                 # 2 até 10
    'rf__min_samples_leaf': range(1, 5),                   # 1 até 4
    'rf__bootstrap': [True, False]                         # Uso de bootstrap: True ou False
}
```

f. Rede Neural MLP (Multilayer Perceptron)

MLP (MultIlayer Perceptron) é um modelo de rede neural profunda com múltiplas camadas totalmente conectadas e ativação não linear, capaz de aprender representações complexas dos dados.

Para a busca de hiperparâmetros, foram testados os seguintes valores:

- **hidden_layer_sizes**: Define a arquitetura das camadas ocultas, ou seja, o número de neurônios em cada camada.

    - Valores testados: (50,), (100,), (50,50), (100,50) e (100,100).
    - Essa variação permite explorar modelos com diferentes profundidades e complexidades.
- **activation**: Função de ativação utilizada nas camadas ocultas.

    relu:

    - A função *Rectified Linear Unit* (ReLU) retorna o próprio valor se ele for positivo e zero caso contrário.
    - É muito popular devido à sua eficiência computacional e à sua capacidade de mitigar o problema do gradiente desaparecido, facilitando o treinamento de redes profundas.
    - Geralmente acelera a convergência durante o treinamento e permite a modelagem de relações não-lineares de forma eficaz.

    tanh:

    - A função hiperbólica tangente (tanh) mapeia os valores de entrada para um intervalo entre -1 e 1.
    - Pode proporcionar uma convergência mais rápida quando os dados estão normalizados, visto que sua saída centrada em zero ajuda na estabilização do treinamento.

        - No entanto, em redes muito profundas, pode sofrer com o problema do gradiente desaparecido, afetando a atualização dos pesos.
    -
- **solver**: Algoritmo de otimização para ajuste dos pesos da rede.
    - Valores testados: *adam* e *sgd*.
    - O *adam* adapta a taxa de aprendizado ao longo do treinamento, enquanto o *sgd* (gradiente descendente estocástico) segue uma abordagem mais simples e direta.
- **alpha**: Parâmetro de regularização (penalidade L2) que ajuda a prevenir o overfitting.
    - Valores testados: 1e-4, 1e-3 e 1e-2.
- **learning_rate_init**: Taxa de aprendizado inicial para o processo de otimização.
    - Valores testados: 0.001, 0.01 e 0.1.
    - Esse parâmetro influencia a velocidade com que o modelo se ajusta aos dados.

```
param_dist_mlp = {

    'mlp__hidden_layer_sizes': [(50,), (100,), (50,50), (100,50),
(100,100)],

    'mlp__activation': ['relu', 'tanh'],

    'mlp__solver': ['adam', 'sgd'],

    'mlp__alpha': [1e-4, 1e-3, 1e-2],

    'mlp__learning_rate_init': [0.001, 0.01, 0.1]

}
```

g. Comitê de Redes Neurais Artificiais

O Comitê de Redes Neurais Artificiais trata-se da combinação de múltiplas redes neurais treinadas independentemente, agregando suas previsões para melhorar a robustez e a generalização.

No pipeline apresentado possui o VotingClassifier que integra três classificadores baseados em MLP com arquiteturas distintas:

- **mlp1**: MLP com duas camadas ocultas de 50 neurônios cada (50,50).
- **mlp2**: MLP com uma única camada oculta de 100 neurônios (100,).
- **mlp3**: MLP com duas camadas ocultas, a primeira com 50 neurônios e a segunda com 30 neurônios (50,30).

O VotingClassifier pode operar em dois modos:

- **Soft Voting**: onde as predições são baseadas na média das probabilidades de cada estimador, permitindo uma ponderação mais suave.
- **Hard Voting**: onde a decisão final é obtida por meio da votação majoritária entre os classificadores.

Além disso, o espaço de busca de hiperparâmetros para o ensemble contempla o ajuste do parâmetro de regularização (*alpha*) de cada MLP, testando os valores 1e-3 e 1e-2, e a escolha entre os modos de votação.

```
pipeline_mlp_ensemble = Pipeline([
    ('smote', SMOTE(random_state=42)),
    ('mlp_ensemble', VotingClassifier(
        estimators=[
```

```
                                                    ('mlp1',
MLPClassifier(hidden_layer_sizes=(50,50), random_state=42)),

                                                    ('mlp2',
MLPClassifier(hidden_layer_sizes=(100,), random_state=42)),

                                                    ('mlp3',
MLPClassifier(hidden_layer_sizes=(50,30), random_state=42))

        ],

              voting='soft'    # 'soft' utiliza médias das
probabilidades; 'hard' utiliza voto majoritário

    ))

])



# Espaço de hiperparâmetros para o VotingClassifier

param_dist_ensemble = {

    'mlp_ensemble__mlp1__alpha': [1e-3, 1e-2],

    'mlp_ensemble__mlp2__alpha': [1e-3, 1e-2],

    'mlp_ensemble__mlp3__alpha': [1e-3, 1e-2],

    'mlp_ensemble__voting': ['hard', 'soft']

}
```

h. Comitê Heterogêneo (Stacking)

O **StackingClassifier** é uma técnica de ensemble que combina diferentes modelos base com um estimador final. Essa abordagem permite que cada modelo aprenda padrões distintos dos dados, enquanto o estimador final (neste caso, uma Regressão Logística) aprende a

integrar as predições dos modelos base para obter uma predição mais robusta.

No pipeline, o balanceamento das classes é realizado pelo SMOTE e os modelos base incluem:

- **KNN**: Classificador baseado em vizinhos próximos.
- **Random Forest**: Modelo ensemble que agrega diversas árvores de decisão.
- **LGBM**: Algoritmo de boosting de gradiente que utiliza LightGBM.

Como estimador final, utiliza-se a **Logistic Regression**, que combina as predições dos modelos base. O grid de hiperparâmetros permite ajustar tanto o estimador final quanto os modelos individuais, abrangendo:

- **final_estimator (Logistic Regression)**:
    - *C*: Valores de 0.01, 0.1, 1, 10 e 100, controlando a força da regularização.
    - *penalty*: Penalidade L2, que é a padrão na Regressão Logística.
- **KNN base**:
    - *n_neighbors*: Variando de 1 a 20, determinando o número de vizinhos considerados na classificação.
- **Random Forest base**:
    - *n_estimators*: Número de árvores testadas com valores 50, 100 e 150.
- **LGBM base**:
    - *num_leaves*: Número de folhas, testando os valores 31, 50 e 70 para ajustar a complexidade do modelo.

```
# Definição do pipeline com StackingClassifier
```

```
pipeline_stack = Pipeline([
    ('smote', SMOTE(random_state=42)),
    ('stack', StackingClassifier(
        estimators=[
            ('knn', KNeighborsClassifier()),
            ('rf', RandomForestClassifier(random_state=42)),
            ('lgbm', LGBMClassifier(random_state=42))
        ],
        final_estimator=LogisticRegression(random_state=42)
    ))
])

param_dist_stack = {
              #    Hiperparâmetros    do    estimador    final
(LogisticRegression)
    'stack__final_estimator__C': [0.01, 0.1, 1, 10, 100],
          'stack__final_estimator__penalty':   ['l2'],      #
LogisticRegression utiliza L2 por padrão

    # Hiperparâmetros do KNN base
    'stack__knn__n_neighbors': list(range(1, 21)),

    # Hiperparâmetros da Random Forest base
    'stack__rf__n_estimators': [50, 100, 150],

     # Hiperparâmetros do LGBM base (por exemplo, número de
folhas)
    'stack__lgbm__num_leaves': [31, 50, 70]
}
```

i. XGBoost

O “XGBoost” é um modelo baseado em boosting, otimizando árvores de decisão de forma sequencial para reduzir erros residuais, sendo altamente eficiente e utilizado em competições de machine learning.

Para este modelo, os valores escolhidos para a busca de hiperparâmetros foram os seguintes:

- **n_estimators**: Número de árvores (estimadores) do modelo.

    - Valores testados: 50, 100, 150, 200, 250 e 300.
    - Um número maior de estimadores pode melhorar a capacidade de aprendizado, mas também aumenta o tempo de treinamento e o risco de overfitting.
- **max_depth**: Profundidade máxima da árvore .

    - Valores testados: 3, 5, 7 e 9.
    - Controla o tamanho das árvores e pode impactar a complexidade do modelo. Profundidades maiores permitem capturar mais padrões, mas aumentam o risco de overfitting.
- **learning_rate**: Taxa de aprendizado do modelo.

    - Valores testados: 0.01, 0.05, 0.1 e 0.2.
    - Define o impacto de cada árvore no modelo final. Taxas menores resultam em aprendizado mais lento, mas podem levar a melhores generalizações quando combinadas com um número maior de estimadores.
- **subsample**: Fração das amostras utilizadas em cada árvore..

    - Valores testados: 0.6, 0.8 e 1.0.
    - Reduzir essa fração pode ajudar a diminuir o overfitting ao tornar o modelo mais robusto, semelhante ao efeito do bootstrap na Random Forest.
- **colsample_bytree**: Fração das colunas (features) utilizadas em cada árvore.

    - Valores testados: 0.6, 0.8 e 1.0.
    - Menores valores aumentam a diversidade entre as árvores, reduzindo a chance de overfitting.

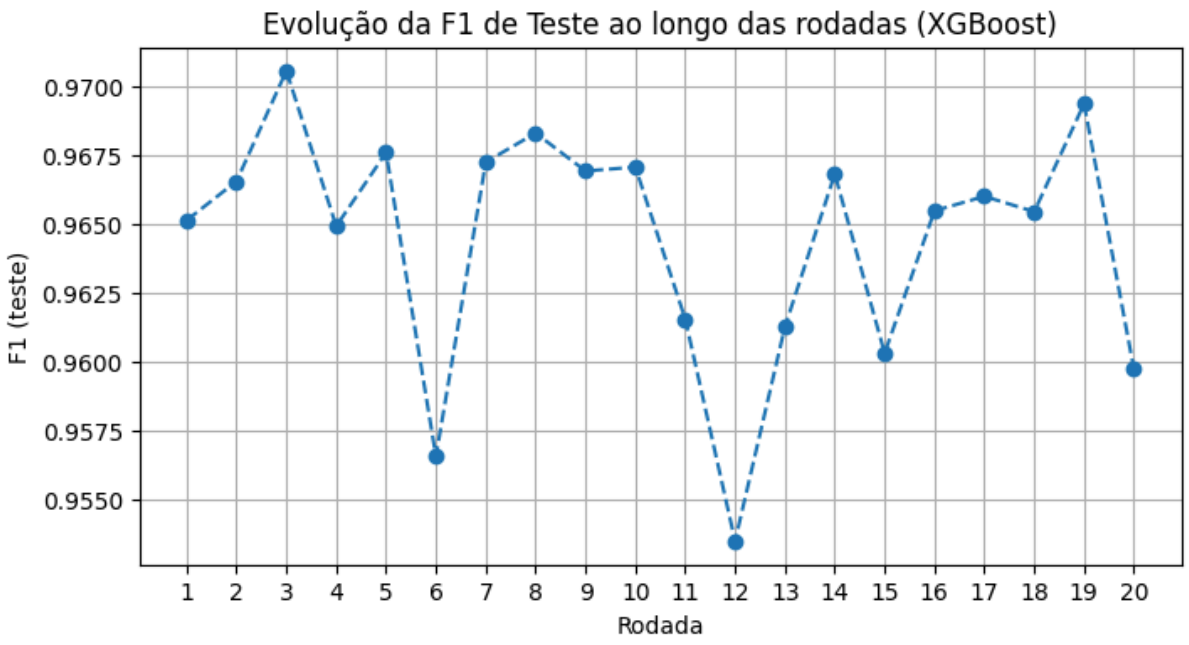
- **gamma**: Parâmetro de regularização que define um limite mínimo de redução na perda para a criação de novos nós..

    - Valores testados: 0, 0.1, 0.2 e 0.3.
    - Valores mais altos tornam o modelo mais conservador, evitando a criação de divisões pouco significativas e reduzindo o risco de overfitting.

```python
# Espaço de busca dos hiperparâmetros para o XGBoost
param_dist_xgb = {
    'xgb__n_estimators': range(50, 301, 50),      # Número de árvores: 50, 100, 150, 200, 250, 300
    'xgb__max_depth': range(3, 11, 2),            # Profundidade máxima: 3, 5, 7, 9
    'xgb__learning_rate': [0.01, 0.05, 0.1, 0.2], # Taxa de aprendizado
    'xgb__subsample': [0.6, 0.8, 1.0],           # Amostragem das instâncias
    'xgb__colsample_bytree': [0.6, 0.8, 1.0],    # Amostragem das colunas
    'xgb__gamma': [0, 0.1, 0.2, 0.3],            # Regularização gamma
}

# Configurando a validação cruzada estratificada (k=5)
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

Depois da variação paramétrica, a melhor configuração de parâmetros obtida foi a seguinte:

```
Melhor rodada: 3
F1 nessa rodada: 0.9705756711414573
Melhores hiperparâmetros: {'xgb__subsample': 0.8, 'xgb__n_estimators': 250, 'xgb__max_depth': 9, 'xgb__learning_rate': 0.2, 'xgb__gamma': 0.1, 'xgb__colsample_bytree': 0.6}
```



j. LightGBM

O “LightGBM” é similar ao “XGBoost”, mas otimizado para lidar com grandes volumes de dados, construindo árvores de forma mais rápida e eficiente, utilizando técnicas como leaf-wise splitting.

Para o LightGBM foram definidos os seguintes valores para a busca de hiperparâmetros:

4. Busca de Hiperparâmetros

a. Método de busca

No notebook, adotamos o **RandomizedSearchCV** da biblioteca *scikit-learn* para realizar a busca de hiperparâmetros. Em vez de testar exaustivamente todas as combinações (como no *GridSearch*), o *RandomizedSearchCV* seleciona aleatoriamente um número predeterminado de combinações de hiperparâmetros (definido por *n_iter*). Para cada combinação:

1. **Validação Cruzada Estratificada**: Dividimos o conjunto de treino em k partições (k-fold), preservando a proporção das classes em cada partição (estratificação).
2. **Treinamento e Avaliação**: Em cada partição, o modelo é treinado nos k-1 blocos e avaliado no bloco restante. A métrica utilizada (por exemplo, *f1_macro*) é calculada em cada fold.
3. **Média e Desvio-Padrão**: A pontuação final de cada combinação de hiperparâmetros é a média (e desvio-padrão) das métricas obtidas ao longo dos k folds.

Repetimos esse processo dentro de um loop, também combinando o parâmetro *n_iter para obter um vasto conjunto de* configurações aleatórias. Ao terminar, o *RandomizedSearchCV* identifica a melhor combinação de hiperparâmetros com base na pontuação média obtida durante a validação cruzada, repetimos para 20 execuções diferentes, registrando os melhores resultados de cada e comparando em um gráfico.

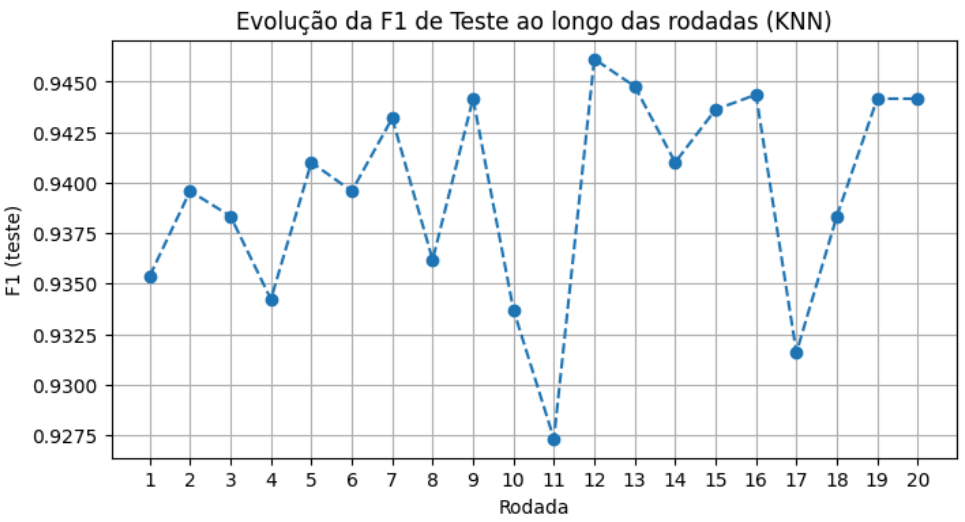
A adoção do *RandomizedSearchCV* permite explorar espaços de hiperparâmetros amplos sem o custo computacional de um *grid search* completo. Cada iteração extrai valores das distribuições definidas para cada hiperparâmetro, possibilitando encontrar boas configurações de forma mais eficiente.

5. Monitoramento e Avaliação

a. Registro de desempenho

i. KNN

- **Sensibilidade ao Número de Vizinhos:**
    - Valores baixos de vizinhos permitiram captar detalhes finos nos dados, porém com risco de overfitting, evidenciado por alta performance no treinamento e desempenho inferior em dados não vistos.
    - Valores mais elevados suavizam a fronteira de decisão, o que pode reduzir a variância, mas em excesso pode levar a underfitting.
- **Impacto da Ponderação (weights) e da Métrica de Distância (p):**
    - A métrica Euclidiana (p = 1) frequentemente apresentou resultados superiores, sugerindo que a estrutura dos dados favorece essa medida de similaridade.
- **Seleção dos Melhores Hiperparâmetros:**
    - A combinação com maior F1, melhor equilibrando a sensibilidade e a generalização foi selecionada, essa sendo a combinação 12 expressa no print abaixo.
    - A análise do histórico de desempenho (média e desvio-padrão) confirmou a robustez do modelo para essa configuração, com discrepâncias mínimas entre os conjuntos de treinamento e teste.



O gráfico mostra a variação do F1 de teste ao longo de 20 rodadas de busca de hiperparâmetros para o KNN. Podemos destacar alguns pontos importantes:

- **Estabilidade Geral**: Apesar de algumas oscilações, a maior parte (apenas um abaixo) dos valores de F1 permanece em um intervalo relativamente alto (acima de 0.93). Isso sugere que, mesmo em rodadas diferentes, o KNN manteve um bom nível de desempenho na maioria das configurações testadas.

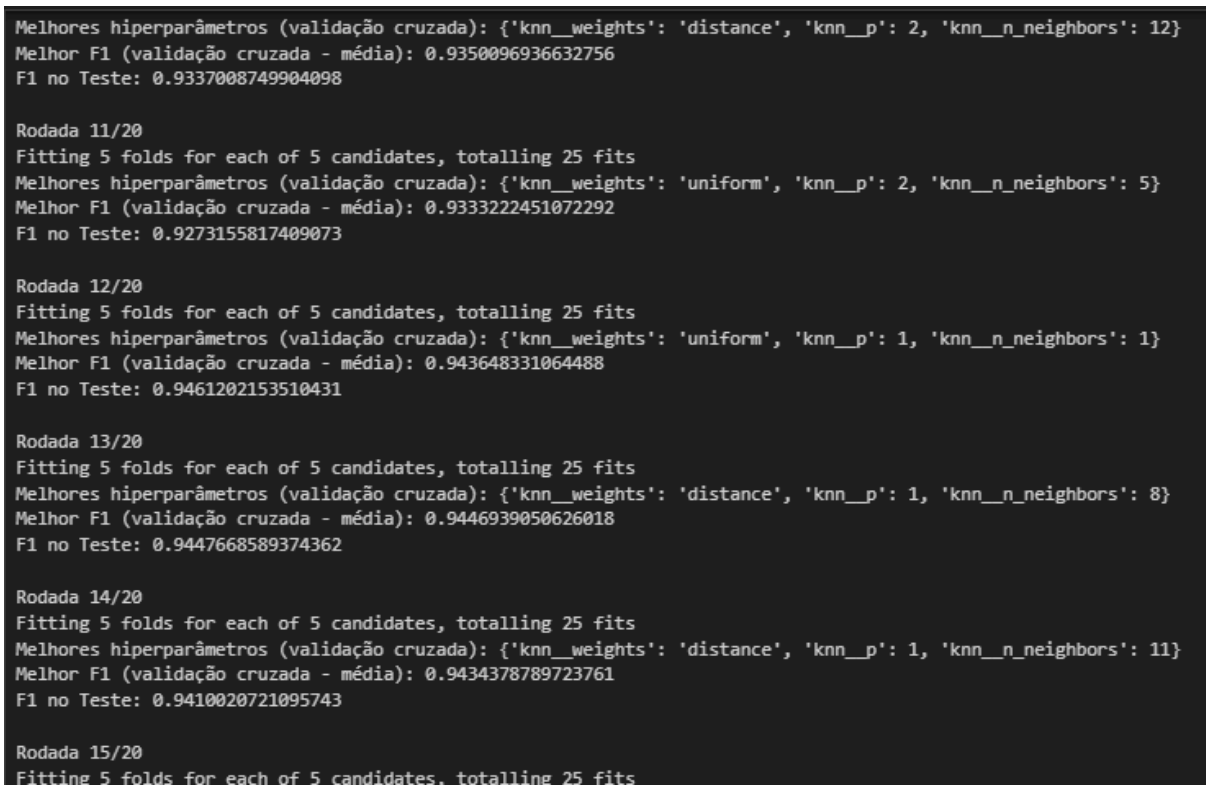
- **Picos de Desempenho**: Observam-se picos próximos ou acima de 0.94 em algumas rodadas (por exemplo, rodadas 16 e 12), indicando que certas combinações de hiperparâmetros se destacaram e potencialmente representam a melhor configuração para o modelo.

- **Queda Pontual**: Há rodadas em que o F1 diminui para valores em torno de 0.927, mostrando que algumas configurações de hiperparâmetros não

foram tão adequadas e podem ter levado a um ajuste menos eficiente nos dados, como na rodada 11.

- **Interpretação das Oscilações**: As flutuações de F1 de uma rodada para outra são comuns em buscas aleatórias, pois cada iteração pode selecionar parâmetros bastante diferentes (por exemplo, variações em *n_neighbors*, *weights* e *p*). Ainda assim, o fato de o modelo se manter em um nível alto na maior parte do tempo reforça a robustez do KNN quando bem ajustado.

- **Validação e Resultados no Conjunto de Teste:**



```
Melhores hiperparâmetros (validação cruzada): {'knn__weights': 'distance', 'knn__p': 2, 'knn__n_neighbors': 12}
Melhor F1 (validação cruzada - média): 0.9350096936632756
F1 no Teste: 0.9337008749904098

Rodada 11/20
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Melhores hiperparâmetros (validação cruzada): {'knn__weights': 'uniform', 'knn__p': 2, 'knn__n_neighbors': 5}
Melhor F1 (validação cruzada - média): 0.9333222451072292
F1 no Teste: 0.9273155817409073

Rodada 12/20
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Melhores hiperparâmetros (validação cruzada): {'knn__weights': 'uniform', 'knn__p': 1, 'knn__n_neighbors': 1}
Melhor F1 (validação cruzada - média): 0.943648331064488
F1 no Teste: 0.9461202153510431

Rodada 13/20
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Melhores hiperparâmetros (validação cruzada): {'knn__weights': 'distance', 'knn__p': 1, 'knn__n_neighbors': 8}
Melhor F1 (validação cruzada - média): 0.9446939050626018
F1 no Teste: 0.9447668589374362

Rodada 14/20
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Melhores hiperparâmetros (validação cruzada): {'knn__weights': 'distance', 'knn__p': 1, 'knn__n_neighbors': 11}
Melhor F1 (validação cruzada - média): 0.9434378789723761
F1 no Teste: 0.9410020721095743

Rodada 15/20
Fitting 5 folds for each of 5 candidates, totalling 25 fits
```

Observe o desempenho na validação e sobre o teste, adotamos como principal métrica o desempenho sobre o

teste, vide o fato de que a rodada 12 foi a de melhor resultado.

```
Melhor rodada: 12
F1 nessa rodada: 0.9461202153510431
Melhores hiperparâmetros: {'knn__weights': 'uniform', 'knn__p': 1, 'knn__n_neighbors': 1}
```

Avaliação do Desempenho com o Melhor Conjunto de Hiperparâmetros

```python
# Célula 6: Treinamento final com os melhores hiperparâmetros

melhores_parametros = best_params_knn[best_run_index]

# Reconstruindo o Pipeline com os parâmetros ideais
best_knn_pipeline = Pipeline([
    ('smote', SMOTE(random_state=42)),
    ('knn', KNeighborsClassifier(
        n_neighbors=melhores_parametros['knn__n_neighbors'],
        weights=melhores_parametros['knn__weights'],
        p=melhores_parametros['knn__p']
    ))
])

# Treina usando TODO o conjunto de treino (X_train, y_train)
best_knn_pipeline.fit(X_train, y_train)
```

[9]

Pipeline

SMOTE

KNeighborsClassifier

Após a busca sistemática de hiperparâmetros e a seleção do melhor conjunto para o KNN, avaliamos o desempenho do modelo no conjunto de teste. Os resultados obtidos são bastante expressivos:

- **Acurácia**: 0.9560
  Indica que o modelo classificou corretamente cerca de 95,6% das instâncias de teste.

- **Precisão**: 0.9463
  Representa a proporção de instâncias realmente positivas dentre

aquelas que o modelo classificou como positivas, evidenciando baixa taxa de falsos positivos.

- **Recall**: 0.9459
  Mostra a proporção de instâncias positivas que foram corretamente identificadas, indicando baixa taxa de falsos negativos.

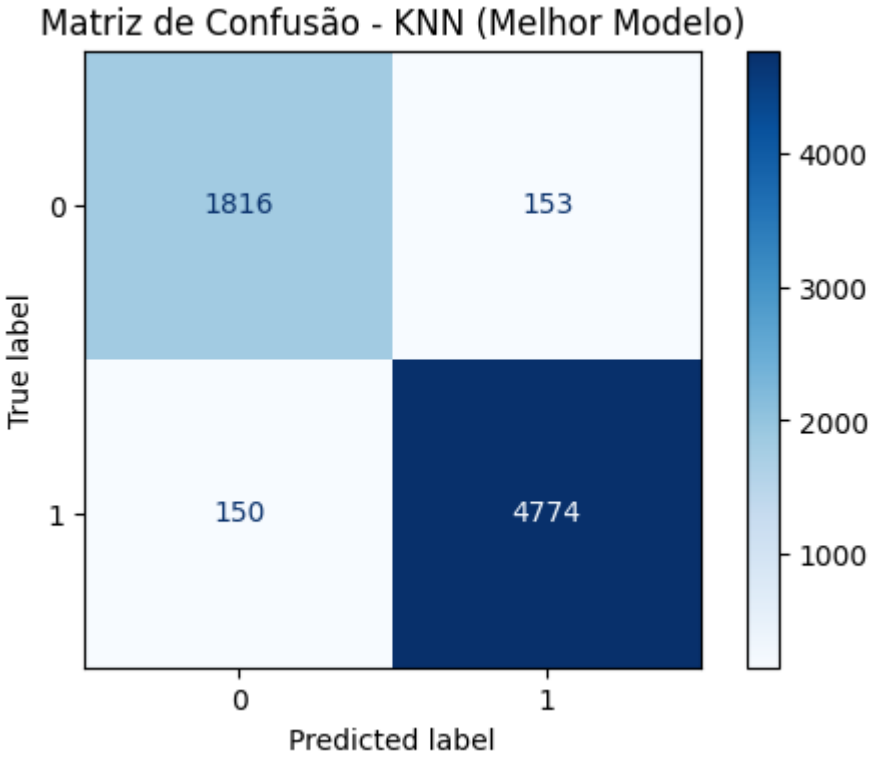
- **F1-score**: 0.9461
  Combina precisão e recall em uma única métrica, confirmando o equilíbrio entre ambos.

- **AUC**: 0.9459
  Reflete a capacidade de discriminação do modelo em diferentes limiares de decisão, apontando para uma separação consistente entre as classes.

```
Desempenho no Conjunto de Teste:
Acurácia  : 0.9560
Precisão  : 0.9463
Recall    : 0.9459
F1-score  : 0.9461
AUC       : 0.9459

Matriz de Confusão:
[[1816  153]
 [ 150 4774]]

Relatório de Classificação:
              precision    recall  f1-score   support

        -1.0       0.92      0.92      0.92      1969
         1.0       0.97      0.97      0.97      4924

    accuracy                           0.96      6893
   macro avg       0.95      0.95      0.95      6893
weighted avg       0.96      0.96      0.96      6893
```

Matriz de Confusão e Curva ROC

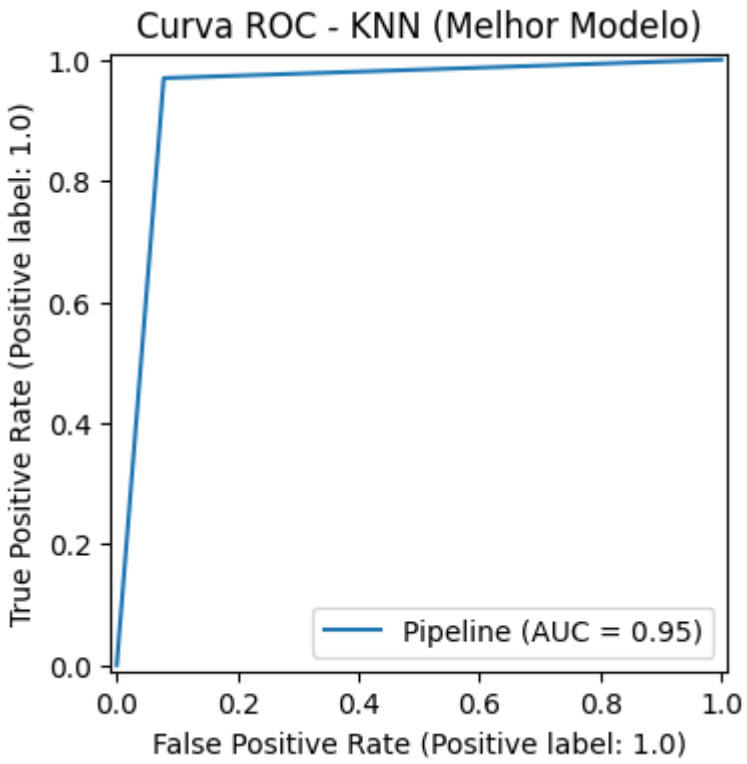
A **Matriz de Confusão** do KNN (Melhor Modelo) demonstra um alto nível de acertos em ambas as classes, com 1816 instâncias da classe 0 (em realidade, -1) classificadas corretamente (verdadeiros negativos) e 4774 instâncias da classe 1 também corretamente classificadas (verdadeiros positivos). Os valores fora da diagonal principal (153 falsos positivos e 150 falsos negativos) representam apenas uma pequena parcela do total, indicando que o modelo mantém boa precisão e recall.



Já a **Curva ROC** (Receiver Operating Characteristic) reforça esse desempenho positivo, exibindo uma curva próxima do canto superior esquerdo do gráfico. O valor de **AUC = 0.95** confirma que o classificador consegue distinguir muito bem as duas classes em diferentes limiares de decisão, mantendo baixas taxas de falsos positivos enquanto detecta corretamente a maioria dos verdadeiros positivos.

Em conjunto, tanto a matriz de confusão quanto a curva ROC evidenciam que o KNN, com a configuração ideal de

hiperparâmetros, apresenta uma capacidade robusta de classificação, equilibrando bem os acertos nas duas classes e oferecendo uma boa separação entre positivos e negativos.



ii. Random Forest

Observando os resultados, podemos destacar:

1. Consistência das Métricas:

   Em quase todas as rodadas, a média do F1 obtida na validação cruzada e o F1 no teste ficam próximos, indicando boa generalização e ausência de overfitting excessivo.

2. Melhores Desempenhos:

   - O F1 no teste chegou a 0.961 na rodada 7.
   - Esse patamar elevado de F1 sugere que, em geral, a Random Forest consegue capturar bem as características do problema, principalmente após a otimização de parâmetros.

3. Hiperparâmetros Destacados:

   - **max_depth**: Frequentemente os valores 25, 20 ou mesmo *None* (sem limite) apareceram.
   - **n_estimators**: Variou em todos os valores com bom desempenho, reforçando que a Random Forest mantém robustez mesmo em faixas distintas de árvores.
   - **min_samples_split e min_samples_leaf**: Valores intermediários (como 4, 5 ou 7 no split e 1 ou 2 nas folhas) aparecem com frequência, sugerindo que restringir um pouco o crescimento das árvores melhora a estabilidade do modelo sem prejudicar a capacidade de aprendizado.
   - **bootstrap**: As rodadas mostram que tanto `True` quanto `False` podem levar a bons resultados, mas há uma ligeira tendência de configurações com `bootstrap=False` alcançarem F1 um pouco mais alto em alguns cenários. Esse fato pode estar relacionado à maior diversidade nas amostras de treinamento quando não se usa substituição, mas

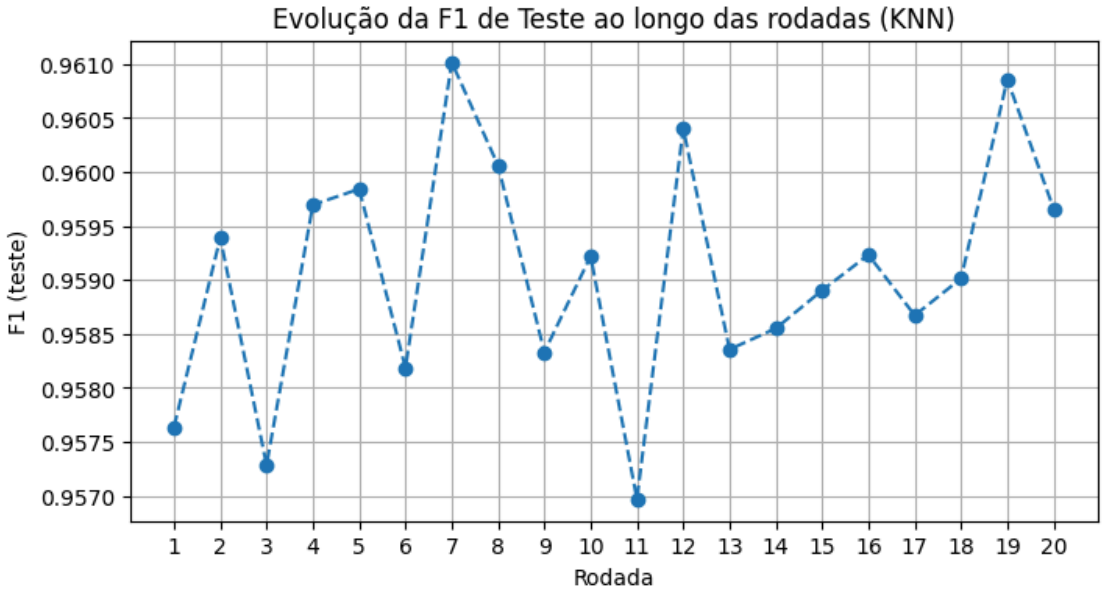
também depende de outras configurações em conjunto.

4. Análise de Variabilidade:

    - Há uma oscilação mínima no F1 de teste, geralmente entre 0.957 e 0.961.
    - Mesmo assim, os valores de F1 permanecem em um nível elevado na maioria das rodadas, reforçando que o Random Forest é um classificador robusto.

Evolução dos Resultados e Identificação do Melhor Modelo

O gráfico mostra a variação do F1 de teste ao longo de 20 rodadas de busca de hiperparâmetros para o modelo, enquanto o trecho de saída confirma a melhor rodada (rodada 7) e os respectivos hiperparâmetros selecionados. Alguns destaques:



1. Flutuações Naturais ao Longo das Rodadas

- As pontuações de F1 apresentaram oscilações esperadas, típicas de uma busca aleatória (RandomizedSearchCV). Cada rodada sorteia combinações diferentes de parâmetros, resultando em variações na capacidade de generalização.

2. Tendência de Desempenho Elevado

    - Ainda que haja altos e baixos, todas as rodadas mantiveram o F1 em um patamar elevado (acima de 0.957), indicando que o modelo tem boa robustez na classificação e que várias configurações de parâmetros podem levar a resultados satisfatórios.

3. Melhor Rodada (Rodada 7)

    - De acordo com o relatório, a rodada 7 obteve a maior pontuação de F1 no teste (em torno de 0.961).
    - Os hiperparâmetros que se mostraram ideais nessa rodada foram:
        - n_estimators: 100
        - min_samples_split: 5
        - min_samples_leaf: 1
        - max_depth: 25
        - bootstrap: False

```
Melhor rodada: 7
F1 nessa rodada: 0.9610156747599901
Melhores hiperparâmetros: {'rf__n_estimators': 100, 'rf__min_samples_split': 5, 'rf__min_samples_leaf': 1, 'rf__max_depth': 25, 'rf__bootstrap': False}
```

Avaliação do Desempenho no Conjunto de Teste (Melhor Modelo)

Os resultados apresentados demonstram um alto nível de acerto e excelente capacidade de discriminação por parte do modelo Random Forest, configurado com os hiperparâmetros selecionados na busca. Eis os principais destaques:

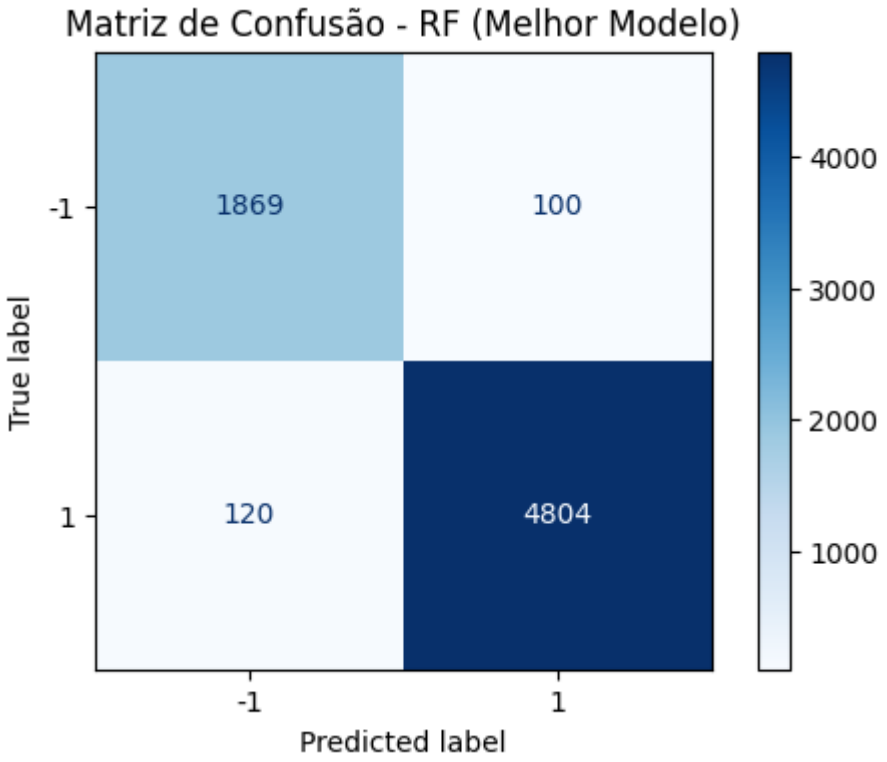
```
Desempenho no Conjunto de Teste:
Acurácia  : 0.9681
Precisão  : 0.9596
Recall    : 0.9624
F1-score  : 0.9610
AUC       : 0.9943

Matriz de Confusão:
[[1869  100]
 [ 120 4804]]

Relatório de Classificação:
              precision    recall  f1-score   support

        -1.0       0.94      0.95      0.94      1969
         1.0       0.98      0.98      0.98      4924

    accuracy                           0.97      6893
   macro avg       0.96      0.96      0.96      6893
weighted avg       0.97      0.97      0.97      6893
```

- Métricas Principais:
    - Acurácia = 0.9681: Cerca de 96,8% das instâncias de teste foram classificadas corretamente.
    - Precisão = 0.9596 e Recall = 0.9624: A alta precisão indica uma baixa taxa de falsos positivos, enquanto o recall elevado mostra que a maioria das instâncias positivas foi corretamente identificada (poucos falsos negativos).
    - F1-score = 0.9610: Combina precisão e recall, confirmando o equilíbrio entre ambos.
    - AUC = 0.9943: Indica uma separação quase perfeita entre as classes ao variar o limiar de decisão.

No geral, essas métricas mostram que a Random Forest ajustada via busca de hiperparâmetros atingiu excelente desempenho, equilibrando muito bem a identificação correta das duas classes e apresentando baixas taxas de erro. O elevado valor de AUC (0.9943) reforça a capacidade do modelo em distinguir os exemplos negativos e positivos ao longo de diferentes limiares de

decisão, tornando-o uma opção confiável para aplicações que exigem alta precisão e recall.

Matriz de Confusão e Curva ROC do Melhor Modelo (Random Forest)

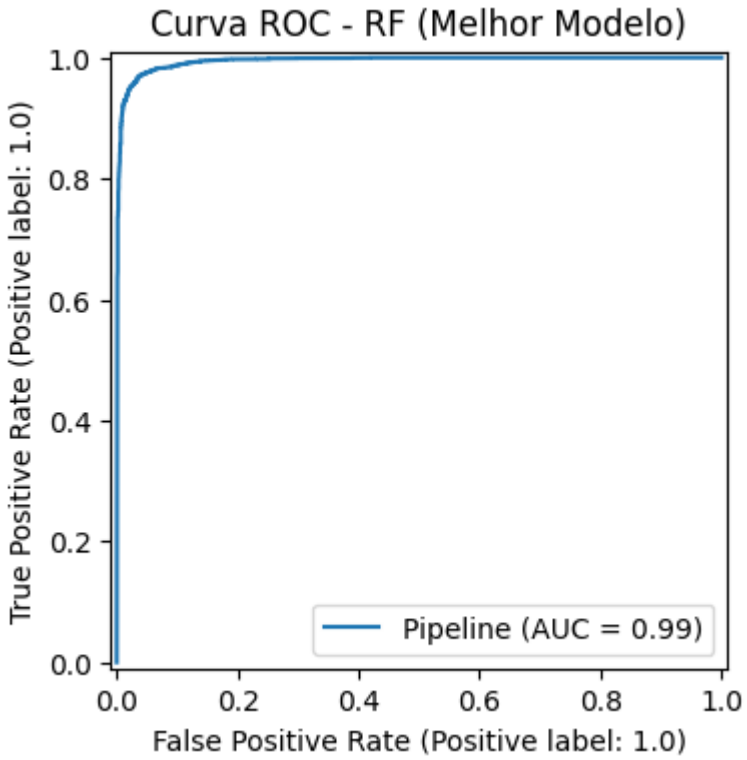


A **Matriz de Confusão** ilustra a distribuição das previsões em relação aos rótulos reais, fornecendo uma visão clara de como o modelo se comporta para cada classe:

- **Classe -1 (lugares diferentes)**:
  - 1869 acertos (verdadeiros negativos).
  - 100 instâncias classificadas incorretamente como positivas (falsos positivos).
- **Classe 1 (lugares iguais)**:
  - 4804 acertos (verdadeiros positivos).
  - 120 instâncias deixaram de ser identificadas como positivas (falsos negativos).

Em termos proporcionais, percebe-se que a maior parte das instâncias foi classificada corretamente, resultando em taxas de

erro baixas tanto para falsos positivos quanto para falsos negativos.



iii. Decision Tree

A seguir, destacam-se alguns pontos observados nos resultados:

1. F1-score na Validação Cruzada vs. F1 no Teste
    - Em geral, o F1 médio na validação cruzada ficou entre 0.938 e 0.940, enquanto o F1 no teste variou aproximadamente de 0.935 a 0.941.
2. Critério de Divisão (criterion)

- Observou-se a seleção de `'log_loss'` em várias rodadas, mas também houve momentos em que `'entropy'` foi escolhido. Não houve um predomínio absoluto de um único critério, indicando que diferentes cenários podem favorecer cada um.
- O critério `'gini'` não apareceu nas melhores configurações dessas rodadas, mas isso não significa necessariamente pior desempenho – apenas que, nas combinações sorteadas, `'entropy'` e `'log_loss'` se destacaram.

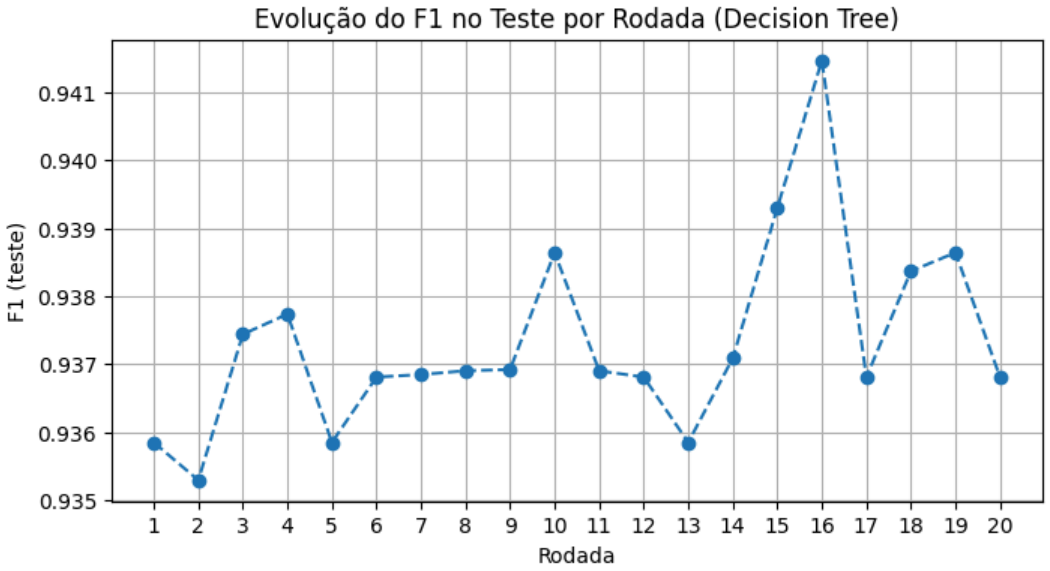
3. Profundidade da Árvore (max_depth)

    - Mesmo com profundidades elevadas, o modelo não apresentou F1 muito distante entre treino e validação.

4. min_samples_split e min_samples_leaf

    - O valor 4 de min_samples_split foi o escolhido na melhor combinação.
    - Para min_samples_leaf, valores menores apareceram em rodadas ótimas.

Variação ao Longo das Rodadas



- Em geral, o F1 permaneceu em um nível relativamente alto, variando em uma faixa estreita (cerca de 0.935 a 0.938) na maioria das rodadas.
- Uma rodada apresenta um pico acima de 0.94, indicando que certas configurações de hiperparâmetros foram particularmente eficazes em capturar a estrutura dos dados.

Melhor Rodada: 16

```
Melhor rodada: 16
F1 nessa rodada: 0.9414704643643181
Melhores hiperparâmetros: {'dt__min_samples_split': 4, 'dt__min_samples_leaf': 3, 'dt__max_depth': 35, 'dt__criterion': 'entropy'}
```

- Na rodada 16, o F1 no teste atingiu cerca de 0.9415, valor mais alto dentre as 20 iterações.
- Os hiperparâmetros que levaram a esse desempenho foram:
    - min_samples_split = 4
    - min_samples_leaf = 3
    - max_depth = 35
    - criterion = 'entropy'
- Essa combinação sugere que uma profundidade intermediária (35) e restrições de divisão moderadas (min_samples_split = 4 e

min_samples_leaf = 3) ajudaram a equilibrar a capacidade de aprendizado da árvore sem gerar sobreajuste.

Interpretação dos Resultados

- A árvore de decisão conseguiu alcançar uma F1 elevada (acima de 0.94) quando os parâmetros foram ajustados de forma a permitir maior profundidade, mas com controle suficiente para não superespecializar.
- O critério `'entropy'` aparece na melhor configuração, mostrando que, para esses dados, medir a impureza dos nós através da entropia pode trazer bons resultados.

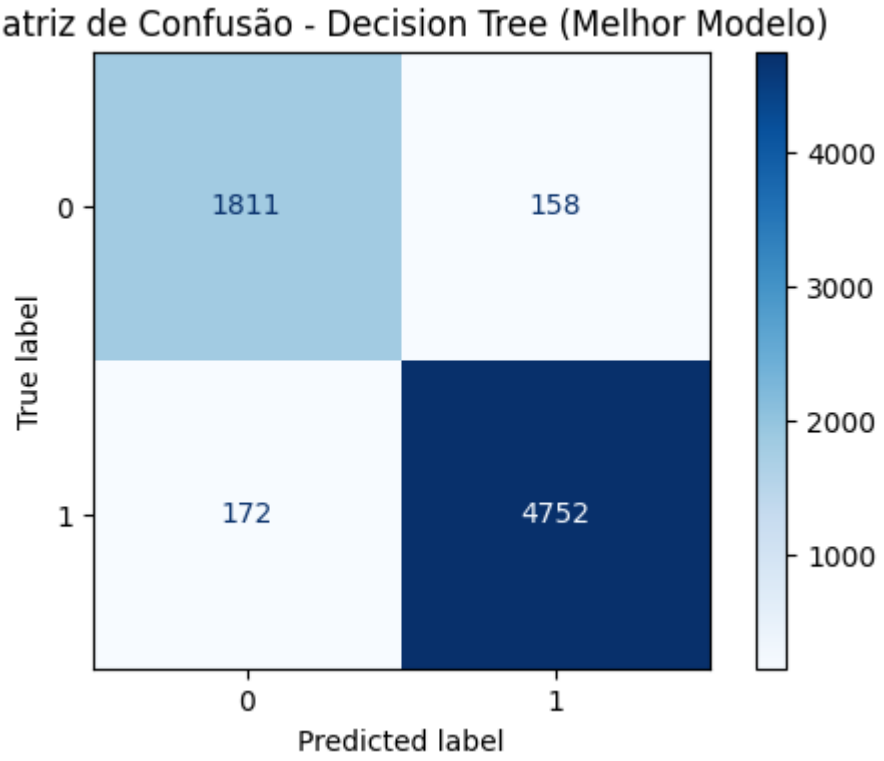
Desempenho no Conjunto de Teste

- A Decision Tree, após a otimização de hiperparâmetros, obteve acurácia de 0.9521, indicando que mais de 95% das instâncias foram corretamente classificadas. Com precisão de 0.9405 e recall de 0.9424, o F1-score de 0.9415 demonstra um bom equilíbrio entre a identificação correta de instâncias positivas e a prevenção de falsos alarmes.
- Por fim, o AUC de 0.9535 reforça a capacidade do modelo de distinguir efetivamente as classes em diferentes limiares de decisão.

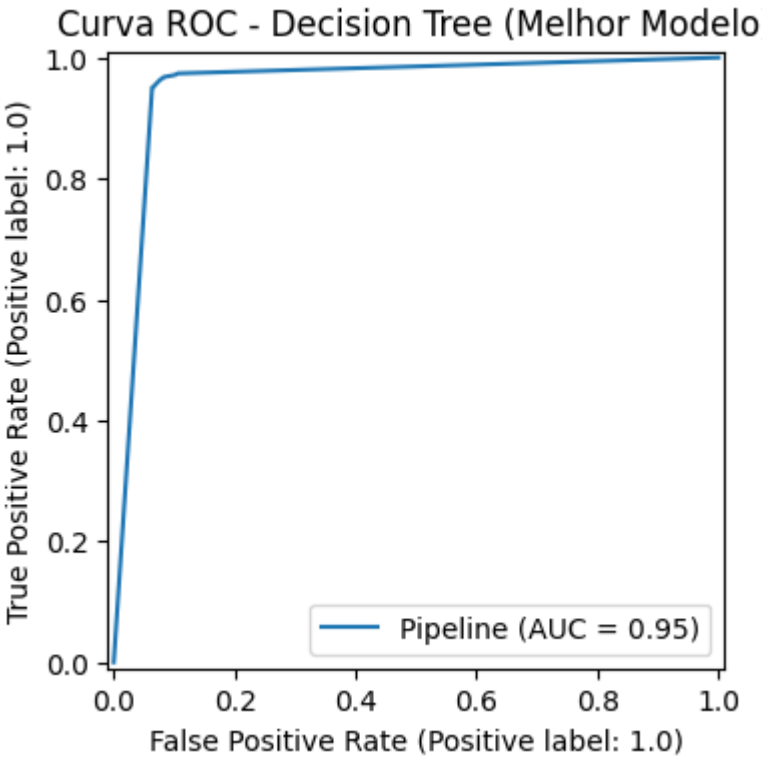
```
Desempenho no Conjunto de Teste:
Acurácia : 0.9521
Precisão : 0.9405
Recall   : 0.9424
F1-score : 0.9415
AUC      : 0.9535
```

Matriz de Confusão e Curva ROC (Decision Tree – Melhor Modelo)

- A matriz de confusão mostra que a árvore de decisão acertou 1811 instâncias da classe 0 e 4752 da classe 1, enquanto cometeu 158 falsos positivos e 172 falsos negativos. Isso indica um bom equilíbrio na classificação, com a maior parte das instâncias corretamente identificadas em ambas as classes.



- A Curva ROC, por sua vez, está próxima do canto superior esquerdo, refletindo alta sensibilidade (taxa de verdadeiros positivos) para baixos níveis de falsos positivos. O valor de AUC = 0.95 confirma a capacidade do modelo de separar eficazmente as duas classes em diferentes limiares de decisão, corroborando o bom desempenho já observado nas outras métricas.



iv. Multilayer perceptron

1. A seguir, destacam-se alguns pontos observados nos resultados:

b. **Arquitetura da Rede (hidden_layer_sizes)**

i. O modelo alcançou bons resultados tanto com uma única camada mais ampla quanto com múltiplas camadas menores, mostrando a flexibilidade do MLP em lidar com diferentes arquiteturas.

c. **Função de Ativação (activation)**

i. `'relu'` e `'tanh'` foram testadas. Ambas atingiram F1s próximos, embora `'relu'` seja ligeiramente mais frequente entre as melhores combinações.
ii. `'tanh'` também gerou bons resultados, especialmente em configurações com `'sgd'` e taxas de aprendizado mais altas (0.1).

d. **Algoritmo de Otimização (solver)**

i. `'adam'` obteve ótimos resultados, como esperado, os melhores modelos o utilizaram, assim, obtendo F1 elevados, por vezes acima de 0.95.
ii. `'sgd'` também apresentou desempenho competitivo, alcançando F1 acima de 0.95 em múltiplas rodadas.

e. **Taxa de Aprendizado (learning_rate_init)**

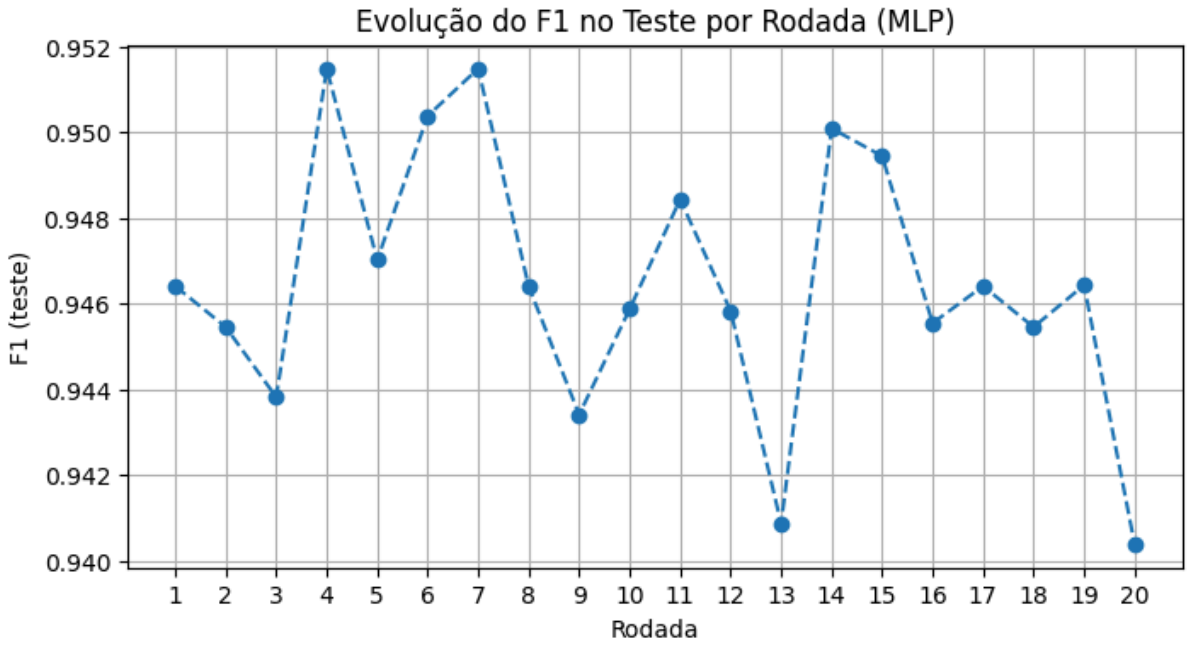
i. Foram testados valores como 0.001, 0.01 e 0.1.
ii. Taxas mais altas (0.1) podem acelerar o aprendizado, mas também correm risco de oscilações e não convergência perfeita. Apesar disso, geraram bons resultados em diversas configurações. Os melhores, como estimado, vieram de learning rates menores, como 0.001.

f. **Regularização (alpha)**

i. O valor de `0.01` apareceu com frequência nos melhores modelos, reforçando que uma regularização média já pode ajudar a evitar

overfitting, mantendo boa capacidade de aprendizado.

Evolução do F1 ao Longo das Rodadas e Melhor Configuração



O gráfico exibe as oscilações do F1 de teste em 20 rodadas de busca de hiperparâmetros para a rede MLP, variando aproximadamente de 0.940 a 0.952. Na **rodada 4**, o modelo obteve o melhor resultado, com F1 próximo de **0.951** no teste. Os hiperparâmetros responsáveis por esse pico de desempenho foram:

- **solver = 'adam'**
- **learning_rate_init = 0.001**
- **hidden_layer_sizes = (100, 100)**
- **alpha = 0.01**
- **activation = 'tanh'**

Essa combinação indica que duas camadas ocultas de 100 neurônios, aliadas a uma regularização leve (alpha = 0.01) e à função de ativação *tanh* (o que foi uma surpresa), conseguiram

capturar de forma eficaz a complexidade dos dados, mantendo boa estabilidade no treinamento por meio do *adam* e de uma taxa de aprendizado moderada (0.001).

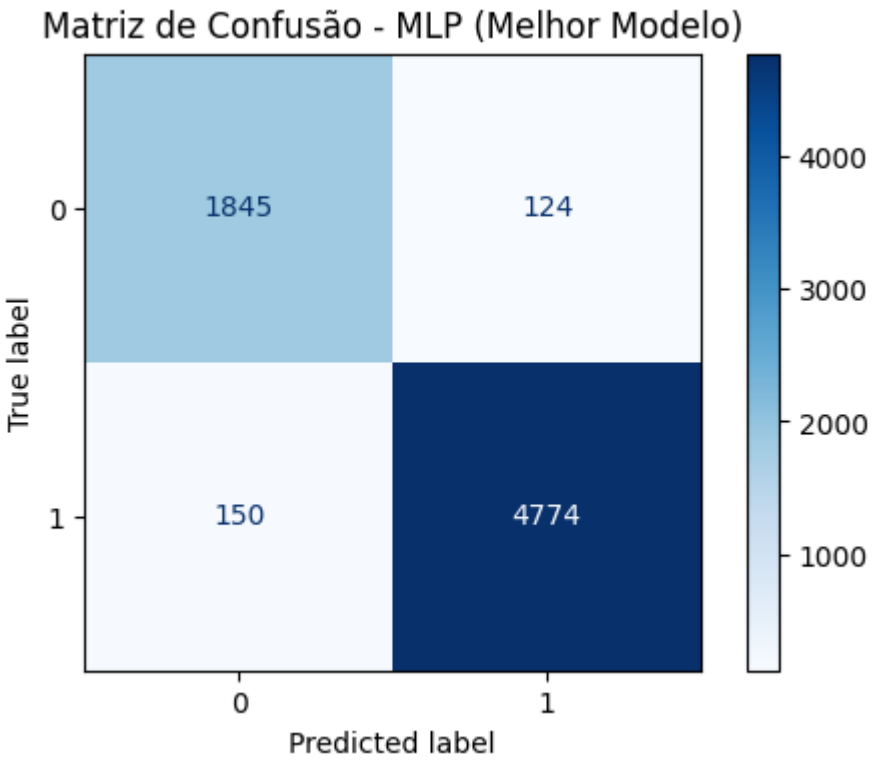
```
Melhor rodada: 4
F1 na melhor rodada: 0.9514906711821585
Melhores hiperparâmetros: {'mlp__solver': 'adam', 'mlp__learning_rate_init': 0.001, 'mlp__hidden_layer_sizes': (100, 100), 'mlp__alpha': 0.01, 'mlp__activation': 'tanh
```

Desempenho no Conjunto de Teste (MLP)

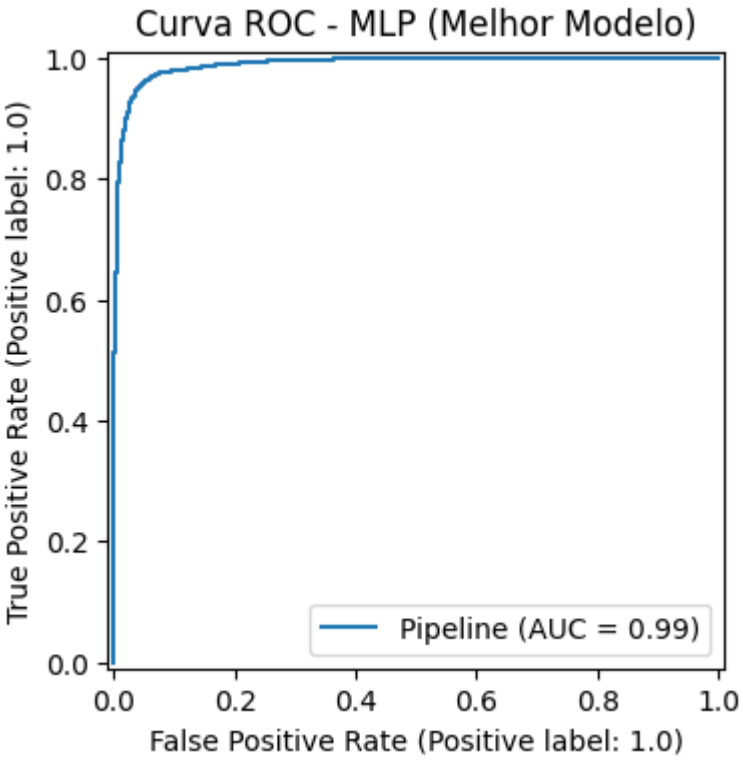
O MLP selecionado após a busca de hiperparâmetros atingiu **acurácia de 0.9602**, classificando corretamente cerca de 96% das instâncias. A **precisão de 0.9497** e o **recall de 0.9533** resultaram em um **F1-score de 0.9515**, demonstrando um equilíbrio sólido entre a identificação correta de instâncias positivas e a redução de falsos alarmes.

```
Desempenho no Conjunto de Teste:
Acurácia : 0.9602
Precisão : 0.9497
Recall   : 0.9533
F1-score : 0.9515
AUC      : 0.9897
```

A **matriz de confusão** confirma esse bom desempenho, com 1845 verdadeiros negativos e 4774 verdadeiros positivos, frente a apenas 124 falsos positivos e 150 falsos negativos.



Já a **Curva ROC** indica uma excelente separação das classes, com **AUC = 0.9897**, reforçando a capacidade do modelo em distinguir entre positivo e negativo em diversos limiares de decisão.



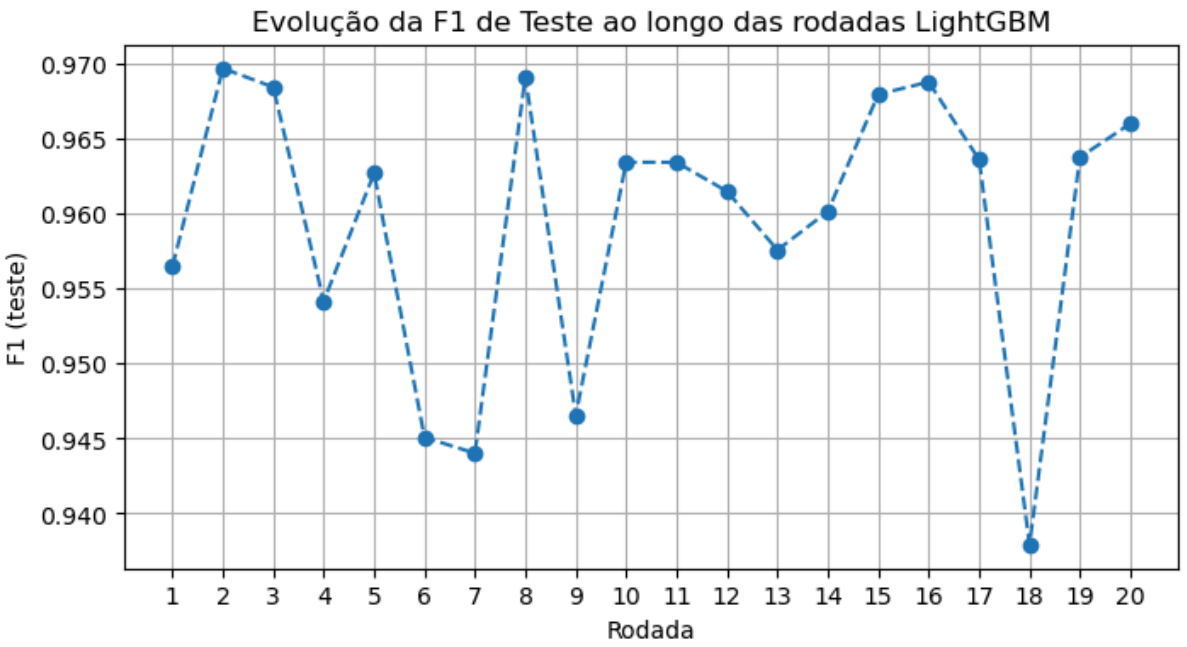
ii. Lightgbm

A seguir, destacam-se alguns pontos observados nos resultados:

F1-score na Validação Cruzada vs. F1 no Teste

Em geral, o F1 médio na validação cruzada ficou entre 0.945 e 0.955 , enquanto o F1 no teste variou aproximadamente de 0.935 a 0.969..

Evolução do F1 ao Longo das Rodadas e Melhor Configuração



Melhor Rodada: 2

Na rodada 2, o F1 no teste atingiu cerca de 0.969, valor mais alto dentre as 20 iterações. Os hiperparâmetros que levaram a esse desempenho foram:

- num_leaves = 40
- lgbm__n_estimators = 500
- max_depth = -1
- learning_rate = 0.1
- boosting_type = gbdt

Essa combinação sugere que a ausência de restrição na profundidade da árvore (max_depth = -1), juntamente com um número moderado de folhas (num_leaves = 40), proporcionou um bom equilíbrio entre a complexidade do modelo e sua capacidade de generalização, evitando o sobreajuste. O uso de 500 estimadores e um learning rate de 0.1 permitiram ao modelo aprender de forma eficiente, mantendo um alto desempenho sem perder a robustez em dados não vistos.

Interpretação dos Resultados

O modelo LightGBM conseguiu alcançar uma F1 elevada (0.9697) quando os parâmetros foram ajustados de forma a permitir maior capacidade de aprendizado sem comprometer a generalização. O critério 'gbdt' aparece na melhor configuração, mostrando que, para esses dados, o uso de gradiente boosting com uma taxa de aprendizado moderada (learning_rate = 0.1) trouxe bons resultados, favorecendo uma boa separação das classes sem ocorrer sobreajuste.

Desempenho no Conjunto de Teste

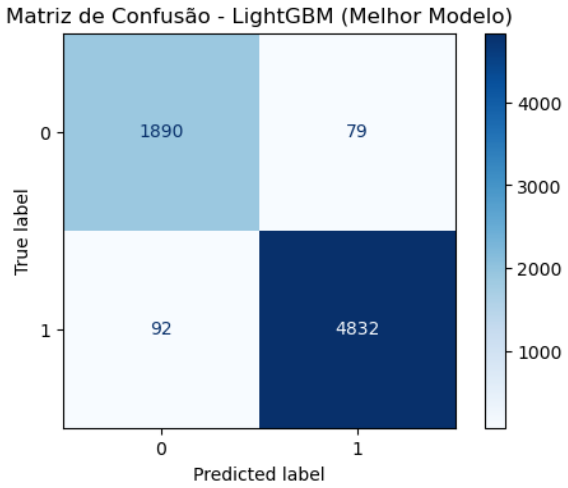
Após a otimização de hiperparâmetros, o modelo LightGBM obteve uma acurácia de 0.9752, indicando que mais de 97% das instâncias foram corretamente classificadas. Com precisão de 0.9687 e recall de 0.9706, o F1-score de 0.9697 demonstra um bom equilíbrio entre a identificação correta de instâncias positivas e a prevenção de falsos alarmes. Por fim, o AUC de 0.9963 reforça a capacidade do modelo de distinguir efetivamente as classes em diferentes limiares de decisão.

```
Desempenho no Conjunto de Teste:
Acurácia  : 0.9752
Precisão  : 0.9687
Recall    : 0.9706
F1-score  : 0.9697
AUC       : 0.9963
```

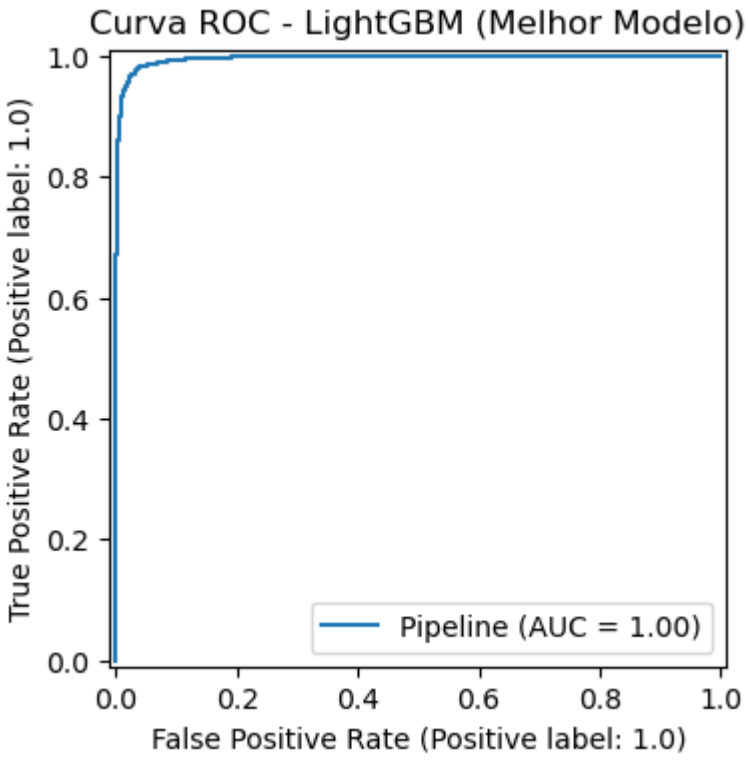
Matriz de Confusão e Curva ROC (LightGBM – Melhor Modelo)

A matriz de confusão mostra que o modelo LightGBM acertou 1890 instâncias da classe -1.0 e 4832 da classe 1.0, enquanto cometeu 79 falsos positivos e 92 falsos negativos. Isso indica um

bom equilíbrio na classificação, com a maior parte das instâncias corretamente identificadas em ambas as classes.



A Curva ROC, por sua vez, está próxima do canto superior esquerdo, refletindo alta sensibilidade (taxa de verdadeiros positivos) para baixos níveis de falsos positivos. O valor de AUC = 0.9963 confirma a capacidade do modelo de separar eficazmente as duas classes em diferentes limiares de decisão, corroborando o excelente desempenho já observado nas outras métricas.



iii. Comitê de redes neurais

Desempenho Consistente em Validação Cruzada

- O F1 médio na validação cruzada situou-se na faixa de 0.953 a 0.954 nas rodadas, mostrando que o comitê de MLPs manteve uma performance robusta mesmo com pequenas variações em *alpha* e no tipo de votação.

F1 no Teste

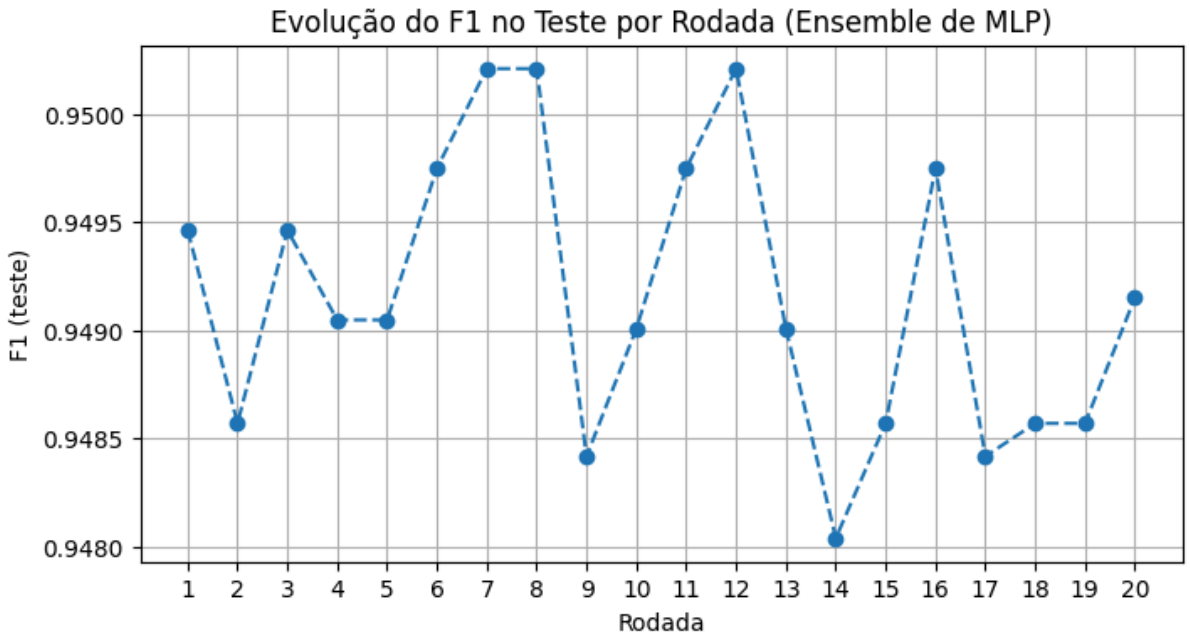
- No conjunto de teste, o F1 variou em torno de 0.948 a 0.950, com algumas configurações alcançando 0.9502 (por exemplo, Rodada 7/8 (foram iguais).
- A proximidade entre F1 de validação cruzada e de teste indica boa generalização, sem sinais fortes de overfitting.

Comparação Hard vs. Soft Voting

- Ambas as abordagens geraram resultados muito próximos, com *soft* voting ligeiramente superior em algumas rodadas (por exemplo, alcançando F1 no teste de 0.9502).
- No *soft voting*, as probabilidades de cada MLP são somadas para definir a classe, enquanto no *hard voting* ocorre uma votação majoritária. A vantagem do *soft voting* costuma aparecer quando as redes têm níveis de confiança distintos e complementares.

Impacto de alpha

- Em muitos cenários, valores de *alpha = 0.01* e *alpha = 0.001* foram os melhores, refletindo a necessidade de alguma regularização para controlar o sobreajuste, mas sem restringir demais a capacidade de aprendizado das redes.
- É comum que pequenas diferenças de *alpha* gerem resultados similares, especialmente quando o ensemble agrega várias redes.



O gráfico exibe a variação do F1 ao longo de 20 rodadas para o comitê de MLPs. A pontuação mantém-se em um patamar elevado, variando em torno de 0.948 a 0.950, com algumas rodadas apresentando picos mais altos.

- **Melhor Rodada (7)**: O F1 alcançou cerca de **0.9502** no teste, quando o ensemble operou em modo *soft voting* e os valores de regularização (*alpha*) foram configurados em **0.01** para `mlp2` e `mlp3`, e **0.001** para `mlp1`.

```
Melhor rodada: 7
F1 nessa rodada: 0.9502091224386189
Melhores hiperparâmetros: {'mlp_ensemble__voting': 'soft', 'mlp_ensemble__mlp3__alpha': 0.01, 'mlp_ensemble__mlp2__alpha': 0.01, 'mlp_ensemble__mlp1__alpha': 0.001}
```

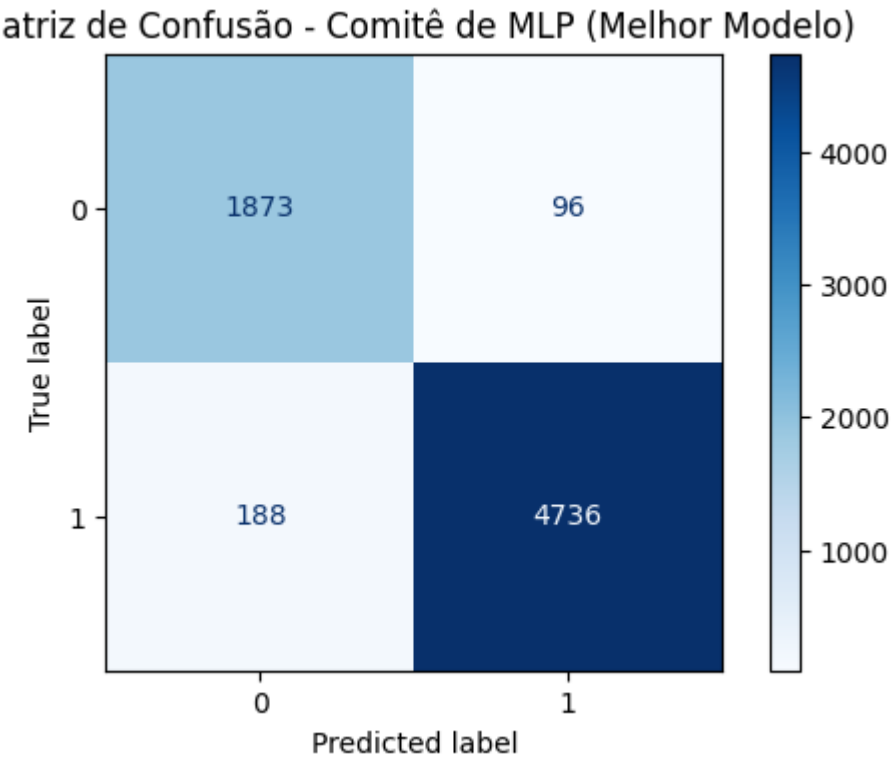
- Essa combinação sugere que regularizações mais fortes em duas das redes e uma regularização mais leve na terceira trouxeram um bom equilíbrio. O *soft voting* potencialmente aproveitou as diferentes probabilidades de saída dos três MLPs, resultando em um leve ganho de desempenho.
- Apesar de pequenas oscilações entre as rodadas, o ensemble manteve desempenho consistente em torno de F1

= 0.948–0.950, indicando que o comitê de MLPs é uma estratégia robusta para a tarefa, especialmente quando os hiperparâmetros de cada rede são afinados para oferecer uma mescla de vieses e variâncias distintas.

Desempenho do Comitê de MLP no Conjunto de Teste

```
Desempenho no Conjunto de Teste:
Acurácia : 0.9588
Precisão : 0.9445
Recall   : 0.9565
F1-score : 0.9502
AUC      : 0.9920
```

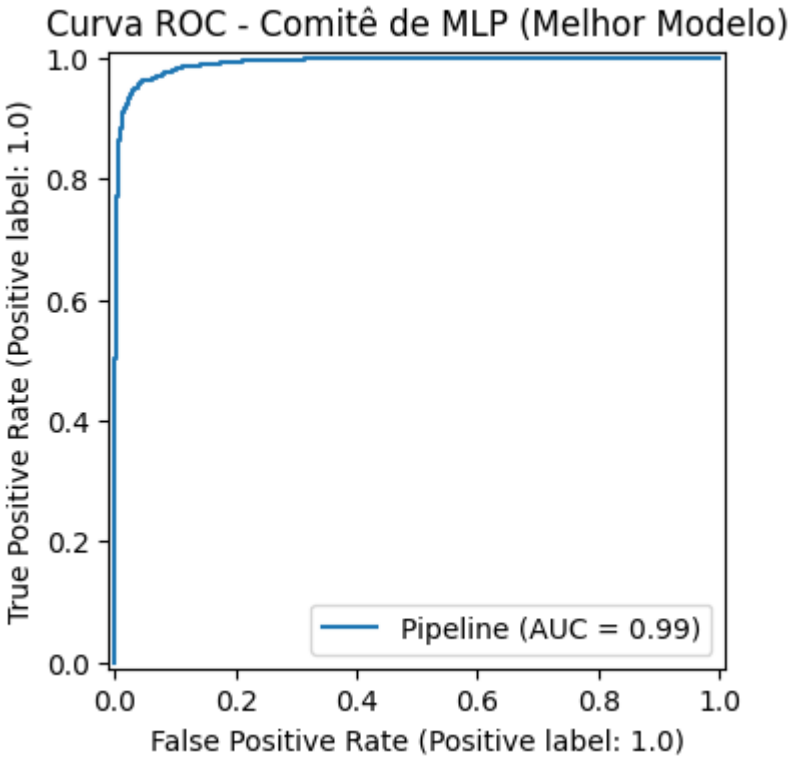
O comitê de redes neurais (MLPs) atingiu **acurácia de 0.9588**, refletindo que quase 96% das instâncias de teste foram classificadas corretamente. A **precisão de 0.9445** e o **recall de 0.9502** resultaram em um **F1-score de 0.9502**, evidenciando um bom equilíbrio entre a identificação correta de positivos (recall) e a redução de falsos alarmes (precisão).



A **matriz de confusão** mostra:

- 1873 verdadeiros negativos (classe 0 corretamente identificada).
- 4736 verdadeiros positivos (classe 1 corretamente identificada).
- 96 falsos positivos e 188 falsos negativos, que são números relativamente pequenos em comparação ao total de instâncias.

Já a **Curva ROC** próxima do canto superior esquerdo, com **AUC = 0.992**, confirma a alta capacidade discriminativa do ensemble, separando efetivamente as classes em diferentes limiares de decisão. Em conjunto, esses resultados indicam que a abordagem de comitê de MLPs, após a otimização de hiperparâmetros, apresenta excelente performance em termos de acurácia, equilíbrio entre precisão e recall, e poder de discriminação (AUC).



iv. SVM

F1-score na Validação Cruzada vs. F1 no Teste

Em geral, o F1 médio na validação cruzada ficou entre 0.934 e 0.955, enquanto o F1 no teste variou aproximadamente de 0.935 a 0.949. Esse comportamento sugere que, em geral, o modelo conseguiu generalizar bem para dados não vistos, com variações que refletem o ajuste fino dos hiperparâmetros em cada rodada.

Evolução do F1 ao Longo das Rodadas e Melhor Configuração

Durante as 20 rodadas, o F1 apresentou variações, mas na rodada 12 foi alcançado o maior F1 no teste, de 0.9496, o melhor desempenho entre as iterações.

Melhor Rodada: 12

Na rodada 12, o F1 no teste atingiu cerca de 0.9496, o valor mais alto dentre as 20 iterações. Os hiperparâmetros que levaram a esse desempenho foram:

kernel = 'poly'

gamma = 'scale'

degree = 3

class_weight = None

C = 10

Essa combinação de hiperparâmetros sugere que o uso do kernel polinomial ('poly') permitiu ao modelo capturar de forma eficaz a complexidade dos dados, ajustando-se bem às variações não lineares. A escolha do valor para **svm__gamma** ('scale') garantiu que o modelo não ficasse excessivamente sensível a pontos distantes, mantendo o equilíbrio no aprendizado. O grau do polinômio foi ajustado para **3**, o que parece ter sido um valor ótimo para modelar a relação entre as variáveis de forma não linear, sem gerar um modelo excessivamente complexo. O parâmetro **svm__C** foi definido como 10, o que forneceu uma boa regularização, controlando o overfitting e permitindo que o modelo generalizasse bem para dados não vistos. A ausência de ponderação para as classes (**svm__class_weight** = None) indicou que as classes estavam balanceadas ou que não era necessário ajustar os pesos para lidar com desbalanceamento.

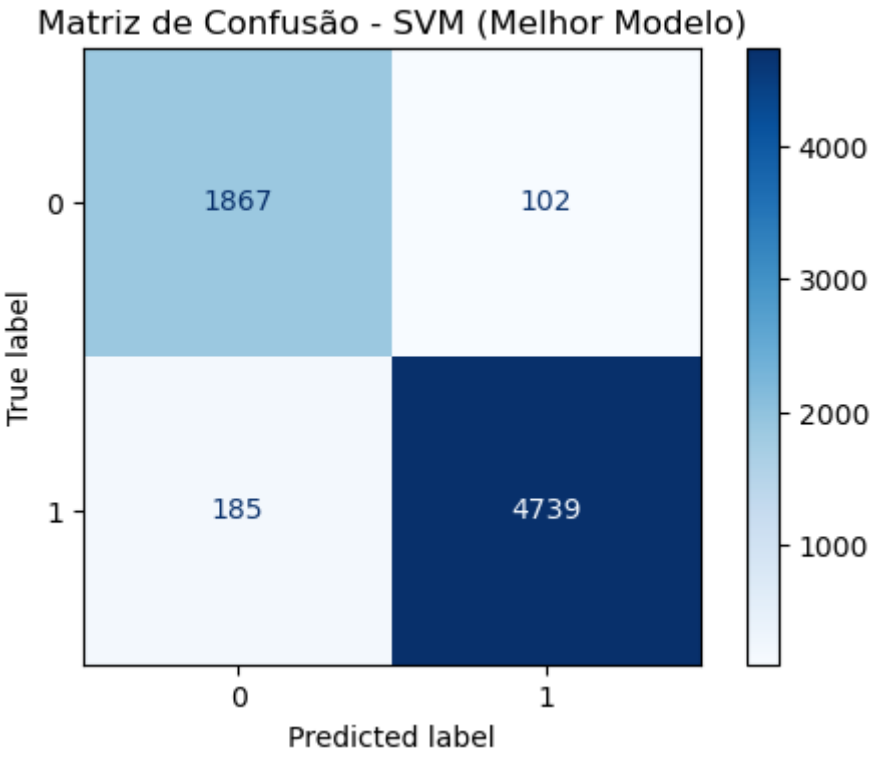
Com esses parâmetros, o modelo alcançou um bom equilíbrio entre capacidade de aprendizado e generalização, demonstrado pelo F1 elevado no teste.

Desempenho no Conjunto de Teste

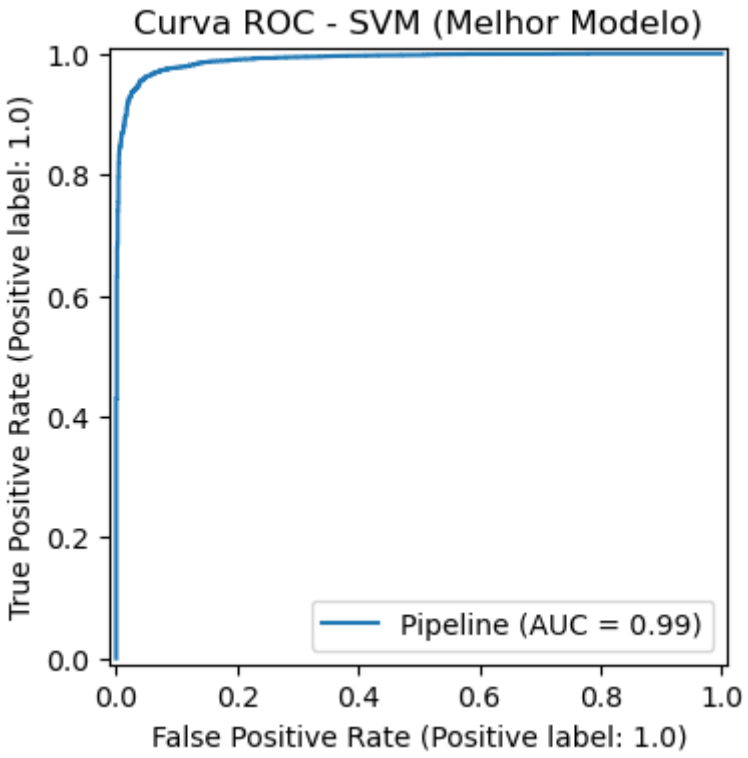
Após a otimização de hiperparâmetros, o modelo SVM obteve uma **acurácia de 0.9584**, indicando que mais de 95% das instâncias foram corretamente classificadas. Com **precisão de 0.9444** e **recall de 0.9553**, o **F1-score de 0.9496** demonstra um bom equilíbrio entre a identificação correta de instâncias positivas e a prevenção de falsos alarmes. O **AUC de 0.9903** reforça a capacidade do modelo de distinguir efetivamente as classes em diferentes limiares de decisão.

Matriz de Confusão e Curva ROC (SVM – Melhor Modelo)

A matriz de confusão mostra que o modelo SVM acertou **1867 instâncias** da classe **-1.0** e **4739 instâncias** da classe **1.0**, enquanto cometeu **102 falsos positivos** e **185 falsos negativos**. Isso indica um bom equilíbrio na classificação, com a maior parte das instâncias corretamente identificadas em ambas as classes.



A **Curva ROC**, por sua vez, está próxima do canto superior esquerdo, refletindo alta sensibilidade (taxa de verdadeiros positivos) para baixos níveis de falsos positivos. O valor de **AUC = 0.9903** confirma a capacidade do modelo de separar eficazmente as duas classes em diferentes limiares de decisão, corroborando o excelente desempenho já observado nas outras métricas.



v. LVQ

Ao longo de 20 rodadas, o modelo LQV apresentou resultados não tão elevados de F1, variando de aproximadamente **0.82 a 0.915 no teste**. Embora o desempenho tenha sido relativamente estável, algumas configurações se destacaram por proporcionar **melhor equilíbrio entre precisão e recall**.

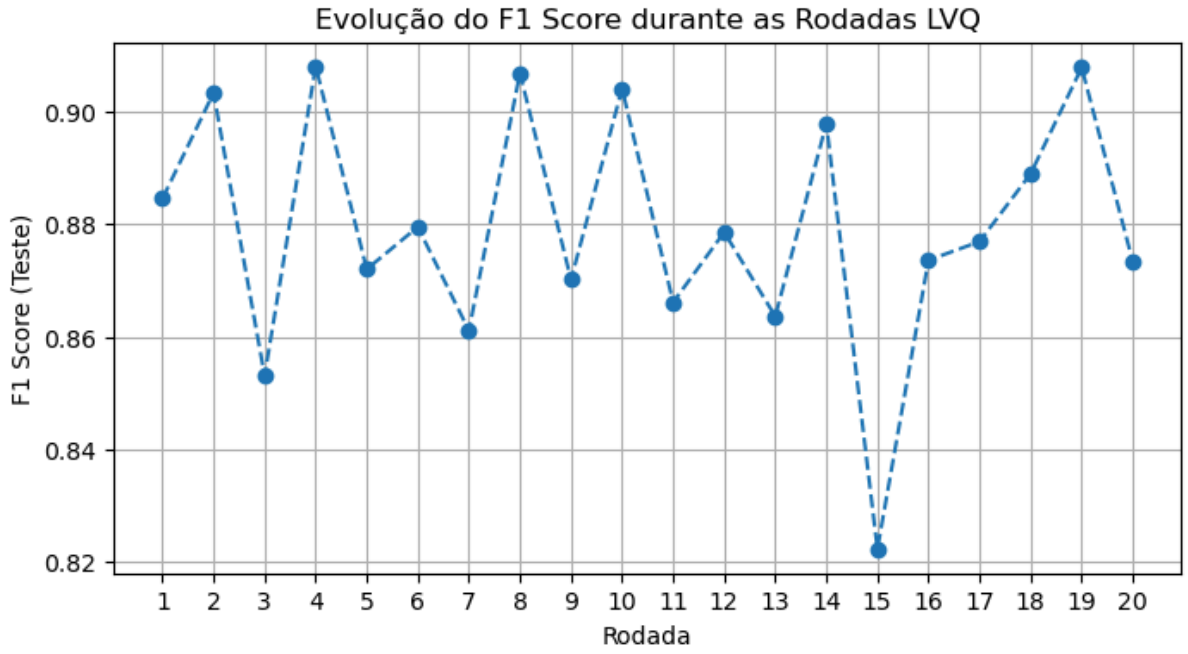
Melhor Rodada: 4

Na **rodada 4**, o modelo alcançou seu melhor desempenho, com um **F1 no teste de aproximadamente 0.908**, o mais alto entre todas as iterações. Os hiperparâmetros responsáveis por esse resultado foram:

- **n_codebooks** = 20

- **learning_rate** = 0.053
- **epochs** = 2

Essa combinação sugere que um **número moderado de codebooks** (*n_codebooks = 20*) permitiu ao modelo capturar melhor a estrutura dos dados sem excessiva complexidade. O **learning rate de 0.053** garantiu um ajuste gradual e eficaz dos pesos, evitando oscilações abruptas no aprendizado. Além disso, a configuração de **apenas 2 épocas de treinamento** mostrou-se suficiente para a convergência do modelo, prevenindo sobreajuste e permitindo uma **boa generalização em dados não vistos**.

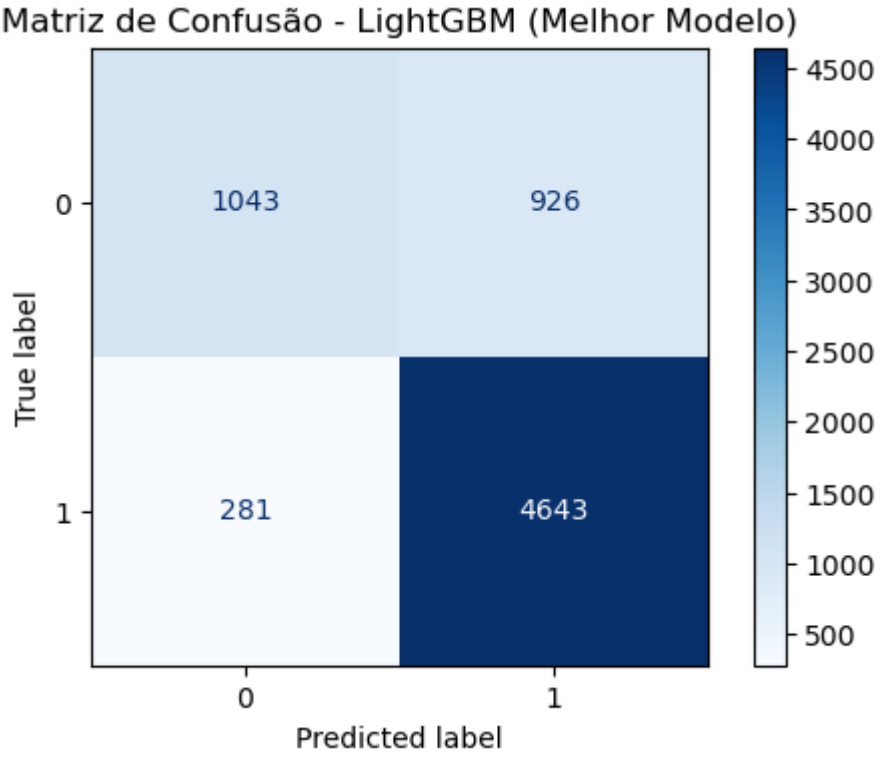


Desempenho no Conjunto de Teste - LVQ

O modelo **LVQ** obteve uma **acurácia de 0.8249**, indicando que cerca de **82% das instâncias** foram corretamente classificadas. Com **precisão de 0.8107** e **recall de 0.7363**, o **F1-score de 0.7592** revela um **desempenho moderado**, com uma leve tendência a minimizar falsos positivos.

Matriz de Confusão e Análise das Classes

A **matriz de confusão** mostra que o modelo corretamente classificou **1043 instâncias da classe -1.0** e **4643 da classe 1.0**, mas apresentou **926 falsos positivos e 281 falsos negativos**. Isso indica um **desempenho desigual entre as classes**, com a classe **1.0** sendo melhor identificada (**recall de 0.94**) do que a classe **-1.0** (**recall de 0.53**).



O **relatório de classificação** reforça essa diferença: enquanto a **classe 1.0** apresenta **F1-score de 0.88**, a **classe -1.0** obteve apenas **0.63**, evidenciando que o modelo enfrenta dificuldades em identificar corretamente as instâncias negativas. Esse comportamento pode indicar a necessidade de **ajustes nos hiperparâmetros** ou de técnicas como **balanceamento de classes**, para garantir um desempenho mais uniforme entre as categorias.

vi. **Stacking**

Ao longo de 20 rodadas, o *StackingClassifier* combinando KNN, Random Forest e LightGBM com uma Logistic Regression como estimador final apresentou resultados bastante elevados de F1 (variando de aproximadamente 0.960 a 0.964 no teste). Alguns destaques:

g. **Melhores Configurações:**

i. **n_estimators (Random Forest):** valores entre 50 e 150 funcionaram bem.
ii. **num_leaves (LightGBM):** tipicamente 50 ou 70, indicando que um número moderado de folhas equilibra profundidade e generalização.
iii. **n_neighbors (KNN):** variou entre 2 e 20, mostrando que tanto vizinhos próximos quanto mais numerosos podem ser úteis dependendo da combinação com os outros modelos.
iv. **final_estimator (Logistic Regression):** O melhor modelo utilizou 0.1, padrão que foi observado nos outros testes

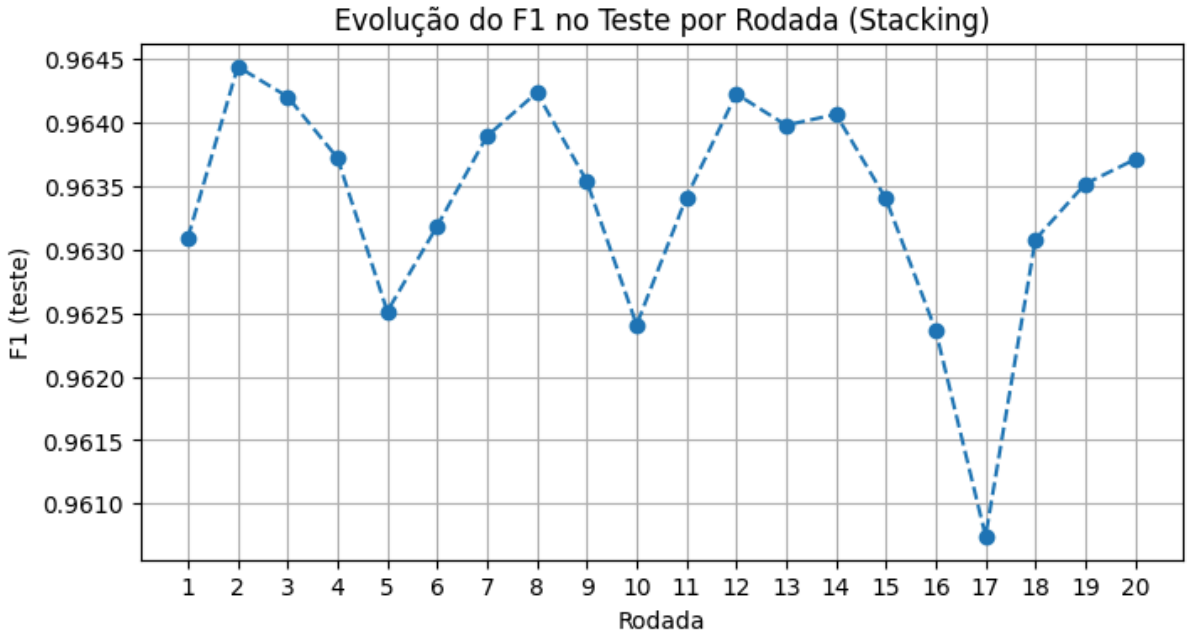
h. **Desempenho Consistente:**

A maioria das rodadas obteve F1 acima de 0.962 no teste, evidenciando a robustez do *StackingClassifier*. Mesmo com pequenas variações nos hiperparâmetros, o modelo manteve resultados próximos, sugerindo que a combinação de algoritmos base é bastante complementar.

Evolução do F1 no Teste (StackingClassifier)

O gráfico apresenta a variação do F1 no teste ao longo de 20 rodadas para o StackingClassifier. Em geral, o desempenho

manteve-se em um patamar elevado, entre **0.960** e **0.965**, evidenciando a robustez do método de empilhamento.



Melhor Rodada (2): Registrou-se um **F1 ≈ 0.9644**, com hiperparâmetros que incluíam:

- **Random Forest** com 150 árvores (n_estimators=150).
- **LightGBM** com num_leaves=50.
- **KNN** com n_neighbors=16.
- **Regressão Logística** (final_estimator) usando penalty='l2' e C=0.1.

```
Melhor rodada: 2
F1 nessa rodada: 0.9644414170091202
Melhores hiperparâmetros: {'stack__rf__n_estimators': 150, 'stack__lgbm__num_leaves': 50, 'stack__knn__n_neighbors': 16, 'stack__final_estimator__penalty': 'l2', 'stack__final_estimator__C': 0.1}
```

Esse resultado mostra que combinar diferentes modelos base (KNN, RF, LGBM) e ajustar cuidadosamente a camada final de regressão logística pode elevar ainda mais o desempenho. As flutuações de F1 entre as rodadas são típicas de buscas aleatórias, mas a consistência em valores altos reforça a eficácia do stacking para a tarefa em questão.

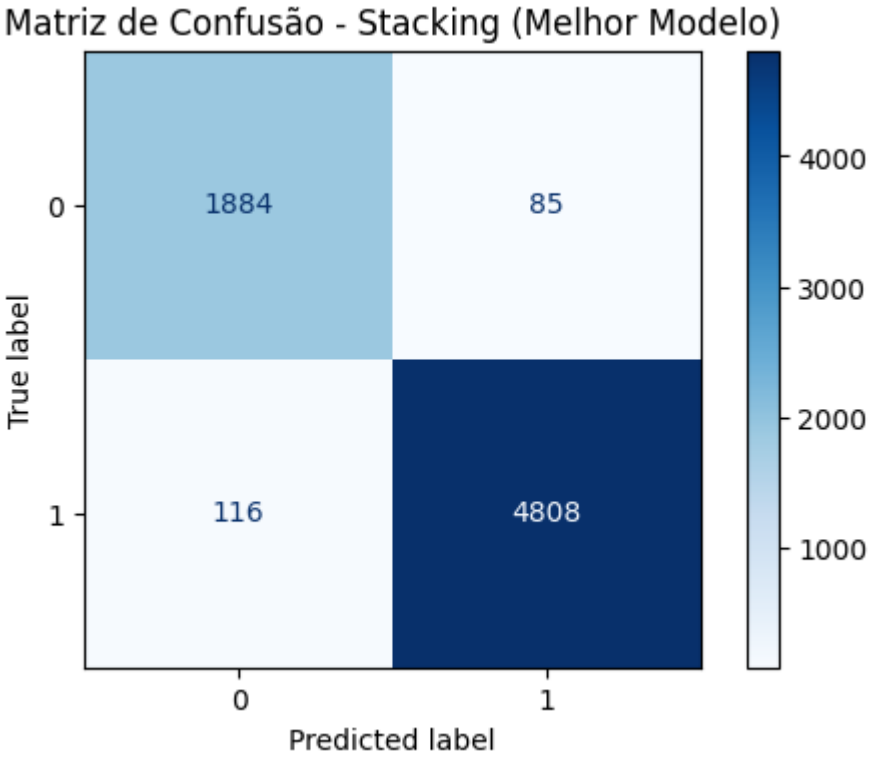
Desempenho no Conjunto de Teste (StackingClassifier)

O melhor modelo de stacking obteve **acurácia de 0.9708**, demonstrando que cerca de 97% das instâncias de teste foram classificadas corretamente. Com **precisão de 0.9623** e **recall de 0.9666**, o **F1-score** atingiu **0.9644**, evidenciando um ótimo equilíbrio entre a identificação correta de positivos e a redução de falsos alarmes. A **AUC = 0.9951** reflete a alta capacidade discriminativa do modelo em diferentes limiares de decisão.

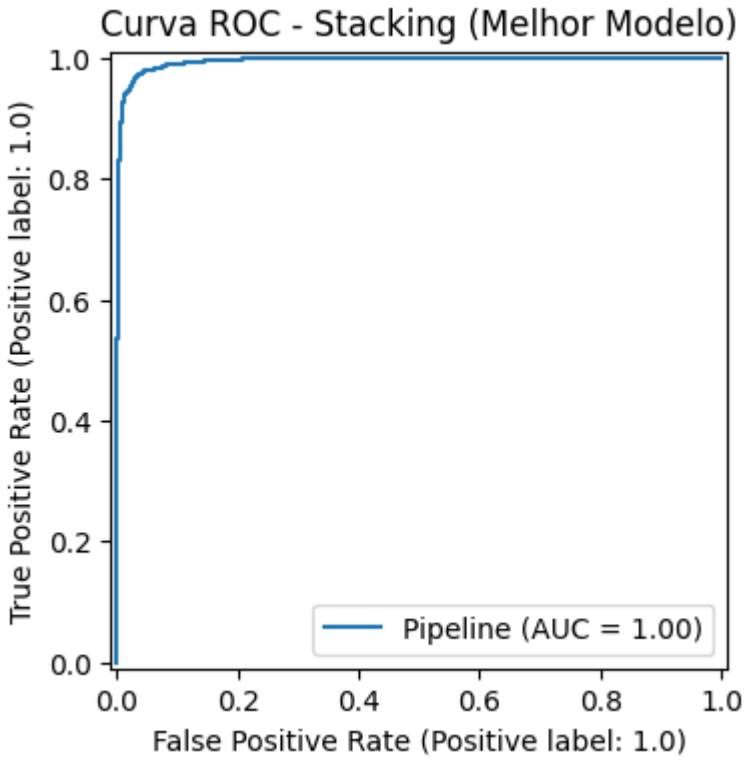
```
Desempenho no Conjunto de Teste:
Acurácia : 0.9708
Precisão : 0.9623
Recall   : 0.9666
F1-score : 0.9644
AUC      : 0.9951
```

A **matriz de confusão** reforça esse desempenho:

- 1884 instâncias da classe 0 foram corretamente classificadas, contra apenas 85 falsos positivos.
- 4808 instâncias da classe 1 foram corretamente reconhecidas, com 116 falsos negativos.



Já a **curva ROC** está muito próxima do canto superior esquerdo, sinalizando alta sensibilidade para baixas taxas de falsos positivos. Em conjunto, esses resultados confirmam a eficácia do método de empilhamento na tarefa de classificação, unindo a força de diferentes estimadores base e uma camada final de regressão logística bem ajustada.



i. XGBoost

Impacto dos Hiperparâmetros

- Taxa de aprendizado (learning rate): A maioria das melhores rodadas apresentou valores entre 0.05 e 0.2, o que mostra que taxas muito baixas podem levar a convergência mais lenta, enquanto taxas mais altas ajudaram o modelo a aprender mais rapidamente sem comprometer o desempenho.
- Número de estimadores (n_estimators): Geralmente entre 150 e 300, com a tendência de melhores resultados na faixa de 250-300, indicando que mais árvores contribuíram para melhorar o desempenho.
- Profundidade máxima das árvores (max_depth): A maioria dos modelos teve max_depth entre 7 e 9, sugerindo que árvores muito profundas não são necessárias e podem até aumentar o risco de overfitting.
- Gamma e colsample_bytree: Gamma variou entre 0 e 0.3, e colsample_bytree entre 0.6 e 1.0, indicando que a seleção de

features e a regularização têm impacto, mas não um padrão fixo para todas as rodadas.

O F1-score na validação cruzada variou entre 0.9607 e 0.9656, mostrando boa estabilidade. No conjunto de teste, os valores foram de 0.9566 a 0.9705, com algumas oscilações maiores.

- Melhor resultado no teste: 0.9705 (Rodada 3)
- Pior resultado no teste: 0.9566 (Rodada 12)
- Maior diferença entre Validação e Teste: Rodada 14, onde o F1-score caiu de 0.9647 (validação) para 0.9594 (teste), possivelmente por overfitting.
- A maioria das rodadas manteve o F1-score no teste acima de 0.96, o que indica um modelo consistente e bem ajustado.

Melhor Rodada (3): A melhor rodada (em termos de F1-score no conjunto de teste) foi a Rodada 3, com 0.9705, com os seguintes parâmetros:

- n_estimators: 250;
- max_depth: 9;
- subsample: 0.8;
- learning_rate: 0.2;
- gamma: 0.1;
- colsample_bytree: 0.6.

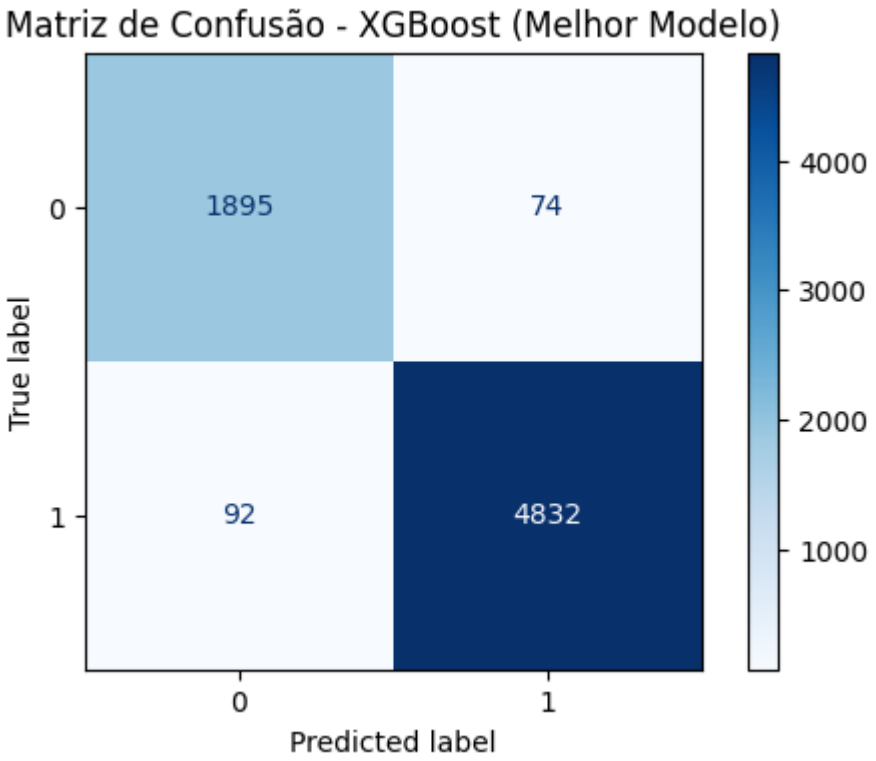
Desempenho no conjunto de teste:

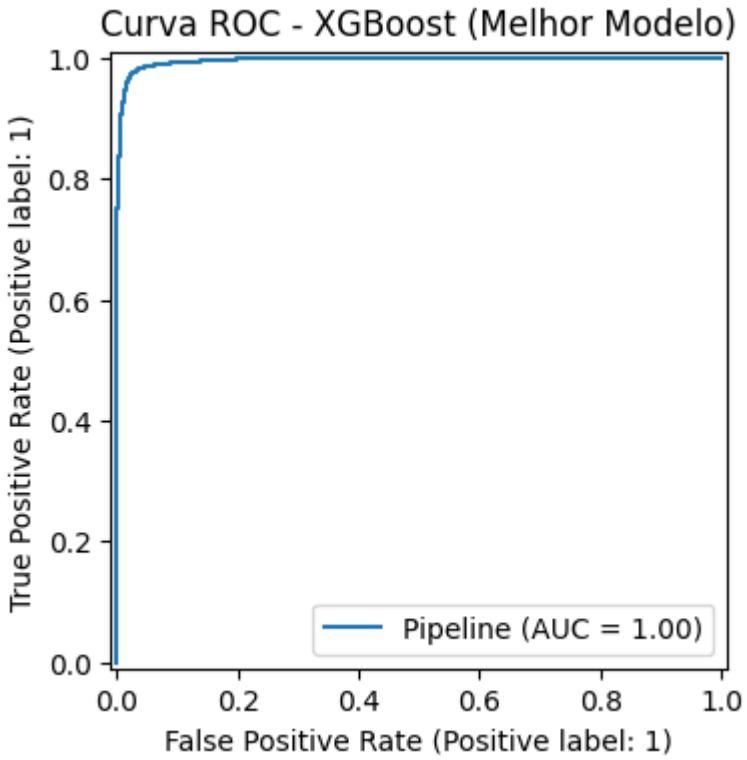
O desempenho do modelo XGBoost no conjunto de teste é altamente satisfatório, apresentando resultados robustos em diversas métricas. A **acurácia é de 97.59%**, indicando que o modelo acertou a grande maioria das previsões. A precisão de 96.93% e o recall de 97.19% refletem uma boa capacidade de identificar corretamente tanto as classes positivas quanto negativas, com um equilíbrio entre os dois. O F1-score de 97.06%

confirma a harmonia entre precisão e recall, representando a qualidade geral do modelo.

```
Desempenho no Conjunto de Teste:
Acurácia  : 0.9759
Precisão  : 0.9693
Recall    : 0.9719
F1-score  : 0.9706
AUC       : 0.9961
```

A AUC de 99.61% destaca a excelente capacidade do modelo em distinguir entre as classes, com um desempenho excepcional na área sob a curva ROC.





Melhor Rodada (3): A melhor rodada (em termos de F1-score no conjunto de teste) foi a Rodada 3, com 0.9705, com os seguintes parâmetros:

- n_estimators: 250;
- max_depth: 9;
- subsample: 0.8;
- learning_rate: 0.2;
- gamma: 0.1;
- colsample_bytree: 0.6.

Desempenho no conjunto de teste

6. Comparação e Análise

Após a análise comparativa dos diversos modelos testados, fica claro que o **StackingClassifier** e o **XGBoost** se destacaram como os melhores modelos para a tarefa. Esse primeiro modelo, que combina estimadores base heterogêneos (KNN, Random Forest e LightGBM) com uma camada final de

regressão logística, apresentou desempenho consistente e robusto ao longo das 20 rodadas de busca de hiperparâmetros, o mesmo pode ser dito para o XGBoost, com as melhores métricas.

No conjunto de teste, o modelo de stacking obteve:

- **Acurácia de 0.9708**: Cerca de 97% das instâncias foram classificadas corretamente.
- **F1-score de 0.9644**: Indicando um excelente equilíbrio entre precisão e recall.
- **AUC de 0.9951 (tão alta que no gráfico é apresentada como 1.00)**: Demonstrando uma alta capacidade de discriminação entre as classes.

No conjunto de teste, o modelo de XGBoost obteve:

- **Acurácia de 0.9759**: Cerca de 97,59% das instâncias foram classificadas corretamente.
- **F1-score de 0.9706**: Indicando um excelente equilíbrio entre precisão e recall.
- **AUC de 0.9961 (tão alta que no gráfico é apresentada como 1.00)**: Demonstrando uma alta capacidade de discriminação entre as classes.

Esses índices superiores, aliados à consistência dos resultados na validação cruzada, evidenciam que a estratégia de empilhamento potencializa as vantagens dos modelos individuais e minimiza suas fraquezas, além da viabilidade fantástica do XGBoost, que aliada ao treinamento leve, obteve resultados impecáveis, especialmente pelo fato de que a abordagem de stacking demorou 3h para ser executada, enquanto o XGBoost apenas minutos.