LightGBM - Modelo final

Jupyter Notebook

In [1]:

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GroupShuffleSplit
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    roc_auc_score, roc_curve,
    precision_recall_curve, average_precision_score,
    accuracy_score, confusion_matrix, classification_report
)
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from sklearn.utils import shuffle
import joblib
import lightgbm as lgb
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings('ignore', category=UserWarning)

RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

caminho = "/Users/renanmoura/Documents/mestrado/PE-AI/data/dados.xlsx"
df = pd.read_excel(caminho)
print("Shape original:", df.shape)

target_col = "PreEclampsia"

display(df.head())
```

```
Shape original: (571, 59)
```

In [2]:

```
# MAPAS
map_raca = {"Branco": 1, "Pardo": 2, "Preto": 3}
map_boolean = {
    "Sim": 1, "YES": 1, "SIM": 1, "TRUE": 1,
    "Nao": 0, "NAO": 0, "Não": 0, "NÃO": 0, "FALSE": 0
}
map_hist_diabetes = {
    "Não": 0, "NAO": 0, "NÃO": 0, "Nao": 0,
    "1º grau": 3, "1º GRAU": 3, "1 GRAU": 3,
    "2º grau": 2, "2º GRAU": 2, "2 GRAU": 2,
    "3º grau": 1, "3º GRAU": 1, "3 GRAU": 1
}

# FEATURES
input_features = [
    "idade", "imc", "diabetes", "hipertensao",
    "origemRacial", "historicoFamiliarDiabetes", "TipoDiabetes",
    "mediaIP","perdasGestacionais", "peso",
    "idadeGestacional", "idadeGestacionalCorrigida", "pesoFetal",
    "percentilArteriaUterina", "percentilArtUmbilical",
    "percentilPeso","circunferenciaAbdominal"
]

df_processed = df.copy()

# IDADE POR PACIENTE

if "paciente_id" in df_processed.columns:
    paciente_ids = df_processed["paciente_id"]
else:
    df_processed["paciente_id_temp"] = df_processed[
        ["dataNascimento","origemRacial","imc"]
    ].astype(str).agg("_".join, axis=1)
    paciente_ids = df_processed["paciente_id_temp"]

df_processed["paciente_id_base"] = paciente_ids

data_referencia = pd.to_datetime("2025-12-02")

paciente_to_nasc = {}
for pid in paciente_ids.unique():
    nasc = pd.to_datetime(
```

```
        df_processed.loc[paciente_ids == pid, "dataNascimento"],
        errors="coerce"
    ).dropna()
    paciente_to_nasc[pid] = nasc.mode().iloc[0] if len(nasc) else None

def calc_idade(d):
    if pd.isna(d):
        return 28
    return np.clip((data_referencia - d).days / 365.25, 15, 50)

df_processed["idade"] = df_processed["paciente_id_base"].map(
    lambda x: calc_idade(paciente_to_nasc.get(x))
)

# GESTAÇÕES POR DATA

if "Data" not in df_processed.columns:
    raise ValueError("Coluna 'Data' (data da consulta) é obrigatória")

df_processed["Data"] = pd.to_datetime(
    df_processed["Data"], errors="coerce", dayfirst=True
)

df_processed = df_processed.sort_values(
    ["paciente_id_base","Data"]
).reset_index(drop=True)

MAX_GAP = 270   # dias (~9 meses)

episodios = []

for pid, grupo in df_processed.groupby("paciente_id_base"):
    datas = grupo["Data"].values
    ep = 1

    for i in range(len(grupo)):
        if i == 0:
            episodios.append(f"{pid}")
            continue

        gap = (datas[i] - datas[i-1]).astype('timedelta64[D]').astype(int)
        if gap > MAX_GAP:
            ep += 1

        suf = "" if ep == 1 else chr(ord("A") + ep - 2)
        episodios.append(f"{pid}{suf}")

df_processed["PacienteIdEpisodio"] = episodios

# CONVERSÕES CATEGÓRICAS

if "origemRacial" in df_processed.columns:
    df_processed["origemRacial"] = (
        df_processed["origemRacial"]
        .astype(str).str.strip()
        .map(map_raca)
        .astype(float)
    )

for col in ["diabetes","hipertensao"]:
    if col in df_processed.columns:
        df_processed[col] = (
            df_processed[col]
            .astype(str).str.strip()
            .map(map_boolean)
            .astype(float)
        )

if "historicoFamiliarDiabetes" in df_processed.columns:
    df_processed["historicoFamiliarDiabetes"] = (
        df_processed["historicoFamiliarDiabetes"]
        .astype(str).str.strip()
        .replace(map_hist_diabetes)
        .astype(float)
    )

if "TipoDiabetes" in df_processed.columns:
    df_processed["TipoDiabetes"] = (
        df_processed["TipoDiabetes"]
        .astype(str).str.strip()
        .replace({
            "Diabetes Gestacional": 1,
            "Tipo 1": 2,
            "Tipo 2": 3
        })
    )
    df_processed["TipoDiabetes"] = pd.to_numeric(
```

```
        df_processed["TipoDiabetes"], errors="coerce"
    ).fillna(0)

# DADOS OBSTÉTRICOS

if "perdasGestacionais" in df_processed.columns:
    df_processed["perdasGestacionais"] = (
        pd.to_numeric(df_processed["perdasGestacionais"], errors="coerce")
        .fillna(0)
    )

if "mediaIP" in df_processed.columns:
    df_processed["mediaIP"] = np.where(
        df_processed["mediaIP"] >= 1.3,
        df_processed["mediaIP"] * 1.3,
        df_processed["mediaIP"]
    )

# peso = peso - pesoFetal

peso_mae_kg = pd.to_numeric(df_processed["peso"], errors="coerce").fillna(0)
peso_feto_g = pd.to_numeric(df_processed["pesoFetal"], errors="coerce").fillna(0)

peso_feto_kg = peso_feto_g / 1000.0

df_processed["peso"] = (peso_mae_kg - peso_feto_kg).clip(lower=35)

# GARANTIR NUMÉRICO + NAN

for col in input_features:
    if col not in df_processed.columns:
        print(f" Criando {col}=0")
        df_processed[col] = 0

    df_processed[col] = (
        pd.to_numeric(df_processed[col], errors="coerce")
        .fillna(0)
        .astype(float)
    )

# LIMITES CLÍNICOS

def aplicar_limites_realistas(X, feature_names):
    limites = {
        "idade": (15, 50),
        "peso": (35, 150),
        "imc": (15, 50),
        "pesoFetal": (0, 5000)
    }
    X = X.copy()
    for f,(lo,hi) in limites.items():
        if f in feature_names:
            X[f] = X[f].clip(lo,hi)
    return X

df_processed[input_features] = aplicar_limites_realistas(
    df_processed[input_features],
    input_features
)

# ALVO

alvo = (
    df_processed[target_col]
    .replace({True:1,False:0})
    .astype(str).str.upper()
    .replace({"TRUE":1,"FALSE":0})
)

alvo = pd.to_numeric(alvo, errors="coerce")
df_processed[target_col] = alvo.astype("Int64")

df_processed = df_processed[~df_processed[target_col].isna()]
df_processed[target_col] = df_processed[target_col].astype(int)

# ESTATÍSTICAS FINAIS


print("\n=== ESTATÍSTICAS FINAIS ===")

for c in ["idade","peso","imc"]:
    s = df_processed[c]
    print(f"{c}: min={s.min():.1f}, max={s.max():.1f}, mean={s.mean():.1f}")

print(
    f"\nShape final: {df_processed.shape}"
    f"\nGestações únicas: {df_processed['PacienteIdEpisodio'].nunique()}"
```

```
    f"\nSem NaNs nas features"
)

print("Target:", df_processed[target_col].value_counts().to_dict())
```

```
=== ESTATÍSTICAS FINAIS ===
idade: min=15.0, max=43.6, mean=32.0
peso: min=35.0, max=119.1, mean=73.0
imc: min=15.0, max=43.0, mean=28.3

Shape final: (151, 63)
Gestações únicas: 146
Sem NaNs nas features
Target: {0: 127, 1: 24}
```

In [3]:

```
# Preparar dados
X = df_processed[input_features].copy()
y = df_processed[target_col].copy()
groups = df_processed["PacienteIdEpisodio"].values


# Split por paciente
gss = GroupShuffleSplit(n_splits=1, test_size=0.20, random_state=RANDOM_STATE)
train_idx, test_idx = next(gss.split(X, y, groups=groups))

X_train = X.iloc[train_idx].reset_index(drop=True)
X_test = X.iloc[test_idx].reset_index(drop=True)
y_train = y.iloc[train_idx].reset_index(drop=True)
y_test = y.iloc[test_idx].reset_index(drop=True)

print(f"Treino: {X_train.shape} | Teste: {X_test.shape}")
print(f"Pacientes treino: {df_processed['PacienteId'].iloc[train_idx].nunique()}")
print(f"Pacientes teste: {df_processed['PacienteId'].iloc[test_idx].nunique()}")
print(f"\nDistribuição y_train:\n{y_train.value_counts()}")
print(f"\nDistribuição y_test:\n{y_test.value_counts()}")
```

```
Treino: (121, 17) | Teste: (30, 17)
Pacientes treino: 121
Pacientes teste: 30

Distribuição y_train:
PreEclampsia
0    100
1     21
Name: count, dtype: int64

Distribuição y_test:
PreEclampsia
0    27
1     3
Name: count, dtype: int64
```

In [4]:

```
sm = SMOTE(random_state=RANDOM_STATE)
X_train_smote, y_train_smote = sm.fit_resample(X_train, y_train)

print(f"Shape após SMOTE: {X_train_smote.shape}")
print(f"Distribuição pós-SMOTE:\n{pd.Series(y_train_smote).value_counts()}")
```

```
Shape após SMOTE: (200, 17)
Distribuição pós-SMOTE:
PreEclampsia
0    100
1    100
Name: count, dtype: int64
```

In [5]:

```
X_np = X_train_smote.values
y_np = y_train_smote.values.astype(float)

cols_binarias = ["diabetes", "hipertensao"]

cont_cols = [c for c in input_features if c not in cols_binarias]
cont_idx = [input_features.index(c) for c in cont_cols]

# Função de augmentação com ruído gaussiano
def augment_gaussian_noise(X_np, y_np, factor=0.3, noise_std=0.02, random_state=None):
    rng = np.random.RandomState(random_state)
    n_new = int(len(X_np) * factor)
    if n_new == 0:
        return X_np.copy(), y_np.copy()
    idx = rng.randint(0, len(X_np), size=n_new)
    X_new = X_np[idx].copy()

    # ruído só nas colunas contínuas
    noise = rng.normal(0, noise_std, size=X_new[:, cont_idx].shape)
    X_new[:, cont_idx] += noise

    y_new = y_np[idx]
    return X_new, y_new

Xg, yg = augment_gaussian_noise(
    X_np, y_np, factor=0.4, noise_std=0.02, random_state=RANDOM_STATE
)

X_aug = np.vstack([X_np, Xg])
y_aug = np.concatenate([y_np, yg])
X_final, y_final = shuffle(X_aug, y_aug, random_state=RANDOM_STATE)


X_final_df = pd.DataFrame(X_final, columns=input_features)


X_final_df = aplicar_limites_realistas(X_final_df, input_features)


for col in cols_binarias:
    if col in X_final_df.columns:
        X_final_df[col] = X_final_df[col].clip(0, 1)

X_final = X_final_df.values

print(f"Shape final após augmentation: {X_final.shape}")
```

```
Shape final após augmentation: (280, 17)
```

In [6]:

```
X_train_val, X_test_final, y_train_val, y_test_final = train_test_split(
    X_final, y_final, test_size=0.15, stratify=np.round(y_final), random_state=RANDOM_STATE
)

X_train_final, X_val, y_train_final, y_val = train_test_split(
    X_train_val, y_train_val, test_size=0.15, stratify=np.round(y_train_val), random_state=RANDOM_STATE
)

print(f"Treino final: {X_train_final.shape}")
print(f"Validação: {X_val.shape}")
print(f"Teste final: {X_test_final.shape}")
```

```
Treino final: (202, 17)
Validação: (36, 17)
Teste final: (42, 17)
```

In [7]:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_final)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test_final)


X_train_scaled = pd.DataFrame(X_train_scaled, columns=input_features)
X_val_scaled = pd.DataFrame(X_val_scaled, columns=input_features)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=input_features)

# Suprimir warnings
import warnings
warnings.filterwarnings('ignore', category=UserWarning)


lgb_model = lgb.LGBMClassifier(
```

```
    objective='binary',
    boosting_type='gbdt',
    n_estimators=1000,
    learning_rate=0.05,
    num_leaves=20,
    max_depth=5,
    min_child_samples=30,
    min_child_weight=0.001,
    subsample=0.7,
    colsample_bytree=0.7,
    colsample_bylevel=0.7,
    reg_alpha=1.0,
    reg_lambda=2.0,
    random_state=RANDOM_STATE,
    n_jobs=-1,
    verbosity=-1,
    force_row_wise=True,
    metric='binary_logloss'
)


lgb_model.fit(
    X_train_scaled,
    y_train_final,
    eval_set=[(X_val_scaled, y_val)],
    eval_metric='binary_logloss',
    callbacks=[
        lgb.early_stopping(stopping_rounds=100, verbose=False),
        lgb.log_evaluation(period=0, show_stdv=False)
    ]
)

print("LightGBM treinado com sucesso!")

class ManualPipeline:
    def __init__(self, scaler, model, feature_names):
        self.scaler = scaler
        self.model = model
        self.feature_names = feature_names

    def predict_proba(self, X):
        X_df = pd.DataFrame(X, columns=self.feature_names)
        X_scaled = self.scaler.transform(X_df)
        return self.model.predict_proba(X_scaled)

    def predict(self, X):
        X_df = pd.DataFrame(X, columns=self.feature_names)
        X_scaled = self.scaler.transform(X_df)
        return self.model.predict(X_scaled)

trained_model = ManualPipeline(scaler, lgb_model, input_features)

# importância das features
if hasattr(lgb_model, 'feature_importances_'):
    feature_importance = lgb_model.feature_importances_
    importance_df = pd.DataFrame({
        'feature': input_features,
        'importance': feature_importance
    }).sort_values('importance', ascending=False)

    print("\nTop 10 features mais importantes:")
    print(importance_df.head(10))
```

```
LightGBM treinado com sucesso!

Top 10 features mais importantes:
                   feature  importance
1                      imc         477
9                    peso          342
0                    idade         259
3              hipertensao         141
6             TipoDiabetes         127
14  percentilArtUmbilical         111
4             origemRacial         110
7                  mediaIP          89
12              pesoFetal           87
10       idadeGestacional           83
```

In [8]:

```
print("Antes do augmentation:")
print(f"X_train_smote shape: {X_train_smote.shape}")
print(f"Distribuição y_train_smote: {pd.Series(y_train_smote).value_counts()}")

print("\nApós augmentation:")
print(f"X_final shape: {X_final.shape}")
print(f"Distribuição y_final: {pd.Series(y_final).value_counts()}")
```

```
Antes do augmentation:
X_train_smote shape: (200, 17)
Distribuição y_train_smote: PreEclampsia
0    100
1    100
Name: count, dtype: int64

Após augmentation:
X_final shape: (280, 17)
Distribuição y_final: 0.0    146
1.0    134
Name: count, dtype: int64
```

In [9]:

```
from sklearn.metrics import precision_score, recall_score, f1_score

y_proba = lgb_model.predict_proba(X_test_scaled)[:, 1]
y_pred = (y_proba >= 0.5).astype(int)
auc = roc_auc_score(y_test_final, y_proba)
accuracy = accuracy_score(y_test_final, y_pred)
precision = precision_score(y_test_final, y_pred)
recall = recall_score(y_test_final, y_pred)
f1 = f1_score(y_test_final, y_pred)

print(f"Métricas calculadas:")
print(f"AUC: {auc:.4f}")
print(f"Acurácia: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")

bundle_lgbm = {
    "model": lgb_model,
    "scaler": scaler,
    "input_features": input_features,
    "target": target_col,
    "map_raca": map_raca,
    "map_boolean": map_boolean,
    "map_hist_diabetes": map_hist_diabetes,
    "performance": {
        "auc": auc,
        "accuracy": accuracy,
        "precision": precision,
        "recall": recall,
        "f1": f1
    },
    "version": "2.0_otimizado_final"
}

# Salvar
bundle_path = "/Users/renanmoura/Documents/mestrado/PE-AI/models/model_lgbm_bundle.pkl"
joblib.dump(bundle_lgbm, bundle_path)

print("Bundle salvo com sucesso!")
print(f"Features no modelo: {len(input_features)}")
```

```
Métricas calculadas:
AUC: 0.9886
Acurácia: 0.9524
Precision: 0.9500
Recall: 0.9500
F1-Score: 0.9500
Bundle salvo com sucesso!
Features no modelo: 17
```

In [16]:

```
import warnings
warnings.filterwarnings('ignore', category=UserWarning)

y_proba = trained_model.predict_proba(X_test_final)[:, 1]
y_pred = (y_proba >= 0.5).astype(int)
y_test_int = np.round(y_test_final).astype(int)

print(f"Matriz de Confusão:\n{confusion_matrix(y_test_int, y_pred)}")
```

```
Matriz de Confusão:
[[21  1]
 [ 1 19]]
```

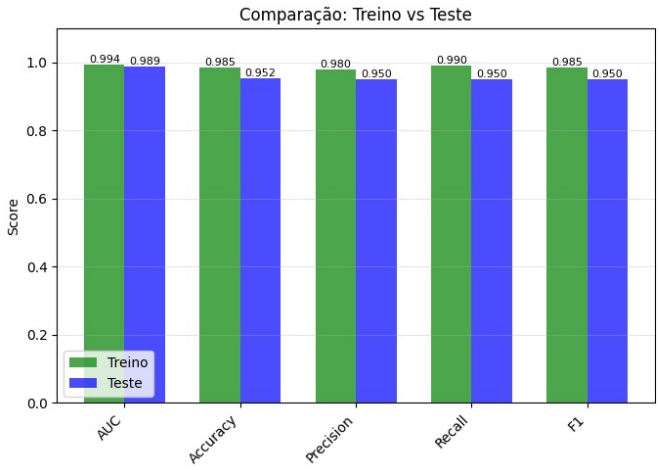
In [11]:

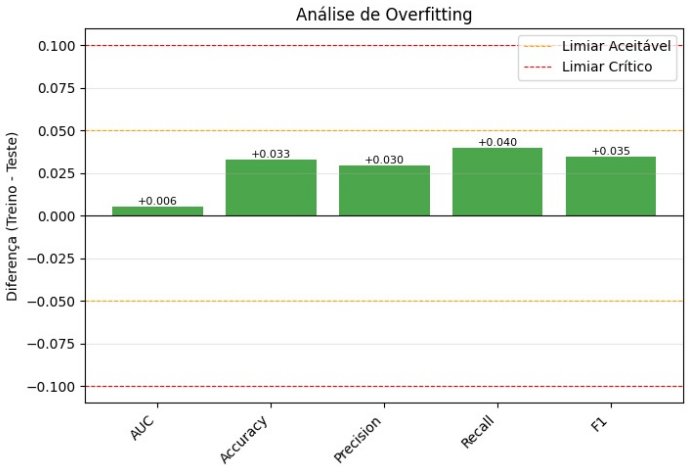
```
out_dir = "imagens"
os.makedirs(out_dir, exist_ok=True)

# %% Cell 9b - Análise de Overfitting (Treino vs Teste)
print("=" * 70)
print("ANÁLISE DE OVERFITTING: TREINO VS TESTE")
print("=" * 70)

# Predições no conjunto de TREINO
y_proba_train = trained_model.predict_proba(X_train_final)[:, 1]
y_pred_train = (y_proba_train >= 0.5).astype(int)
y_train_int = np.round(y_train_final).astype(int)

# Predições no conjunto de TESTE
y_proba_test = trained_model.predict_proba(X_test_final)[:, 1]
y_pred_test = (y_proba_test >= 0.5).astype(int)
y_test_int = np.round(y_test_final).astype(int)

# Calcular métricas para TREINO
train_auc = roc_auc_score(y_train_int, y_proba_train)
train_accuracy = accuracy_score(y_train_int, y_pred_train)
train_precision = precision_score(y_train_int, y_pred_train, zero_division=0)
train_recall = recall_score(y_train_int, y_pred_train, zero_division=0)
train_f1 = f1_score(y_train_int, y_pred_train, zero_division=0)

# Calcular métricas para TESTE
test_auc = roc_auc_score(y_test_int, y_proba_test)
test_accuracy = accuracy_score(y_test_int, y_pred_test)
test_precision = precision_score(y_test_int, y_pred_test, zero_division=0)
test_recall = recall_score(y_test_int, y_pred_test, zero_division=0)
test_f1 = f1_score(y_test_int, y_pred_test, zero_division=0)

# Calcular diferenças
diff_auc = train_auc - test_auc
diff_accuracy = train_accuracy - test_accuracy
diff_precision = train_precision - test_precision
diff_recall = train_recall - test_recall
diff_f1 = train_f1 - test_f1

# Exibir resultados
print("\n MÉTRICAS COMPARATIVAS:")
print("\n" + "-" * 70)
print(f"{'Métrica':<15} {'Treino':<12} {'Teste':<12} {'Diferença':<12} {'Status'}")
print("-" * 70)

metrics_data = [
    ("AUC-ROC", train_auc, test_auc, diff_auc),
    ("Acurácia", train_accuracy, test_accuracy, diff_accuracy),
    ("Precision", train_precision, test_precision, diff_precision),
    ("Recall", train_recall, test_recall, diff_recall),
    ("F1-Score", train_f1, test_f1, diff_f1)
]

for metric_name, train_val, test_val, diff in metrics_data:
    # Determinar status
    if abs(diff) < 0.05:
        status = "Excelente"
    elif abs(diff) < 0.10:
        status = "Aceitável"
    else:
        status = "Overfitting"

    print(f"{metric_name:<15} {train_val:>7.4f}     {test_val:>7.4f}     {diff:>+7.4f}     {status}")

print("-" * 70)

# Análise geral
print("\n DIAGNÓSTICO:")
overfitting_count = sum(1 for _, _, _, diff in metrics_data if abs(diff) > 0.10)
```

```
if overfitting_count == 0:
    print("MODELO SAUDÁVEL: Não há sinais significativos de overfitting")
    print("   O modelo está generalizando bem para dados não vistos.")
elif overfitting_count <= 2:
    print("OVERFITTING MODERADO: Algumas métricas mostram diferenças")
else:
    print("OVERFITTING SEVERO")


# Visualização gráfica
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Gráfico 1: Comparação das métricas
ax = axes[0]
metrics_names = ['AUC', 'Accuracy', 'Precision', 'Recall', 'F1']
train_values = [train_auc, train_accuracy, train_precision, train_recall, train_f1]
test_values = [test_auc, test_accuracy, test_precision, test_recall, test_f1]

x = np.arange(len(metrics_names))
width = 0.35

bars1 = ax.bar(x - width/2, train_values, width, label='Treino', color='green', alpha=0.7)
bars2 = ax.bar(x + width/2, test_values, width, label='Teste', color='blue', alpha=0.7)

ax.set_ylabel('Score')
ax.set_title('Comparação: Treino vs Teste')
ax.set_xticks(x)
ax.set_xticklabels(metrics_names, rotation=45, ha='right')
ax.legend()
ax.grid(axis='y', alpha=0.3)
ax.set_ylim([0, 1.1])

# Adicionar valores nas barras
for bars in [bars1, bars2]:
    for bar in bars:
        height = bar.get_height()
        ax.text(bar.get_x() + bar.get_width()/2., height,
                f'{height:.3f}',
                ha='center', va='bottom', fontsize=8)

# Gráfico 2: Diferenças (Overfitting)
ax = axes[1]
differences = [diff_auc, diff_accuracy, diff_precision, diff_recall, diff_f1]
colors = ['red' if abs(d) > 0.10 else 'orange' if abs(d) > 0.05 else 'green'
          for d in differences]

bars = ax.bar(metrics_names, differences, color=colors, alpha=0.7)
ax.axhline(y=0, color='black', linestyle='-', linewidth=0.8)
ax.axhline(y=0.05, color='orange', linestyle='--', linewidth=0.8, label='Limiar Aceitável')
ax.axhline(y=-0.05, color='orange', linestyle='--', linewidth=0.8)
ax.axhline(y=0.10, color='red', linestyle='--', linewidth=0.8, label='Limiar Crítico')
ax.axhline(y=-0.10, color='red', linestyle='--', linewidth=0.8)

ax.set_ylabel('Diferença (Treino - Teste)')
ax.set_title('Análise de Overfitting')
ax.set_xticklabels(metrics_names, rotation=45, ha='right')
ax.legend()
ax.grid(axis='y', alpha=0.3)

# Adicionar valores nas barras
for bar, diff in zip(bars, differences):
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2., height,
            f'{diff:+.3f}',
            ha='center', va='bottom' if height > 0 else 'top', fontsize=8)

plt.tight_layout()
plt.savefig(os.path.join(out_dir, "analise_overfitting.png"), dpi=300, bbox_inches="tight")
plt.show()
```

```
======================================================================
ANÁLISE DE OVERFITTING: TREINO VS TESTE
======================================================================

 MÉTRICAS COMPARATIVAS:

----------------------------------------------------------------------
Métrica         Treino       Teste        Diferença    Status
----------------------------------------------------------------------
AUC-ROC          0.9942      0.9886     +0.0055     Excelente
Acurácia         0.9851      0.9524     +0.0328     Excelente
Precision        0.9796      0.9500     +0.0296     Excelente
Recall           0.9897      0.9500     +0.0397     Excelente
F1-Score         0.9846      0.9500     +0.0346     Excelente
----------------------------------------------------------------------

 DIAGNÓSTICO:
```

```
MODELO SAUDÁVEL: Não há sinais significativos de overfitting
   O modelo está generalizando bem para dados não vistos.
```



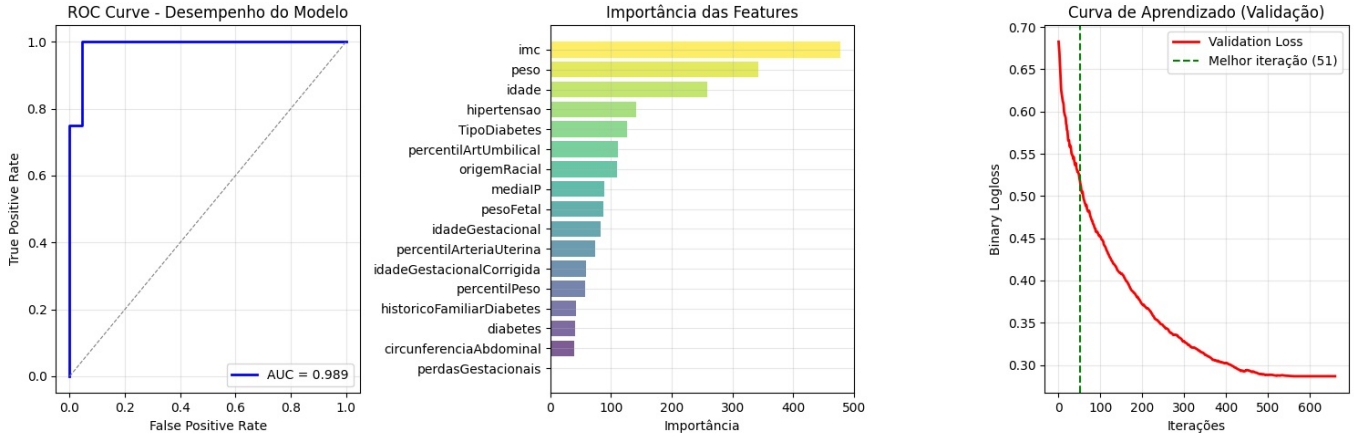


In [12]:

```
import os

out_dir = "imagens"
os.makedirs(out_dir, exist_ok=True)

print("=== GRÁFICOS ===")

# ROC, importância, curva de aprendizado
fpr, tpr, thresholds = roc_curve(y_test_final, y_proba)

plt.figure(figsize=(15, 5))

# 1) ROC
plt.subplot(1, 3, 1)
plt.plot(fpr, tpr, label=f"AUC = {auc:.3f}", linewidth=2, color='blue')
plt.plot([0,1],[0,1],'--', linewidth=0.8, color='gray')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Desempenho do Modelo")
plt.legend(loc="lower right")
plt.grid(alpha=0.3)

# 2) Importância
importance_df = pd.DataFrame({
    'feature': input_features,
    'importance': lgb_model.feature_importances_
}).sort_values('importance', ascending=True)

plt.subplot(1, 3, 2)
colors = plt.cm.viridis(np.linspace(0, 1, len(importance_df)))
plt.barh(importance_df['feature'], importance_df['importance'], color=colors, alpha=0.7)
plt.xlabel('Importância')
plt.title('Importância das Features')
plt.grid(alpha=0.3)

# 3) Curva de aprendizado
if hasattr(lgb_model, 'evals_result_'):
    evals_result = lgb_model.evals_result_
    val_loss = evals_result['valid_0']['binary_logloss']

    plt.subplot(1, 3, 3)
    plt.plot(val_loss, label='Validation Loss', linewidth=2, color='red')
    plt.axvline(x=51, color='green', linestyle='--', label='Melhor iteração (51)')
    plt.xlabel('Iterações')
    plt.ylabel('Binary Logloss')
    plt.title('Curva de Aprendizado (Validação)')
    plt.legend()
    plt.grid(alpha=0.3)

plt.tight_layout()
plt.savefig(os.path.join(out_dir, "avaliacao_lightgbm.png"), dpi=300, bbox_inches="tight")
plt.show()

# ----------------- segunda figura -----------------

plt.figure(figsize=(12, 4))

# Distribuição das probabilidades
plt.subplot(1, 2, 1)
prob_class_0 = y_proba[y_test_final == 0]
```

```
prob_class_1 = y_proba[y_test_final == 1]

plt.hist(prob_class_0, alpha=0.7, label='Sem Pré-eclâmpsia', bins=10, color='blue')
plt.hist(prob_class_1, alpha=0.7, label='Com Pré-eclâmpsia', bins=10, color='red')
plt.xlabel('Probabilidade Predita')
plt.ylabel('Frequência')
plt.title('Distribuição das Probabilidades')
plt.legend()
plt.grid(alpha=0.3)

# Matriz de confusão
plt.subplot(1, 2, 2)
cm = confusion_matrix(y_test_final, y_pred)
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Matriz de Confusão')
plt.colorbar()
tick_marks = np.arange(2)
plt.xticks(tick_marks, ['Sem PE', 'Com PE'])
plt.yticks(tick_marks, ['Sem PE', 'Com PE'])

thresh = cm.max() / 2.
for i, j in np.ndindex(cm.shape):
    plt.text(j, i, format(cm[i, j], 'd'),
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.savefig(os.path.join(out_dir, "probabilidades_matriz_confusao.png"), dpi=300, bbox_inches="tight")
plt.show()
```

```
=== GRÁFICOS ===
```



In [13]:

```
warnings.filterwarnings('ignore', category=UserWarning)

# Dados de exemplo
dados_teste_api = {
  "idade": 39,
  "imc": 34.0,
  "diabetes": 1,
  "hipertensao": 1,
  "origemRacial": "Pardo",
```

```
  "historicoFamiliarDiabetes": "1º grau",
  "TipoDiabetes": "Tipo 2",
  "mediaIP": 1.5,
  "perdasGestacionais": 2,
  "peso": 98,
  "idadeGestacional": 30,
  "idadeGestacionalCorrigida": 30,
  "pesoFetal": 1100,
  "percentilArteriaUterina": 85,
  "percentilArtUmbilical": 60,
  "percentilPeso": 15,
  "circunferenciaAbdominal": 235
}


def preprocess_input_data(input_data):
    input_df = pd.DataFrame([input_data])

    # 1) Raça
    if "origemRacial" in input_df.columns:
        input_df["origemRacial"] = (
            input_df["origemRacial"]
            .astype(str).str.strip()
            .replace(map_raca)
        )

    cols_bool_api = ["diabetes", "hipertensao"]
    for col in cols_bool_api:
        if col in input_df.columns:
            input_df[col] = (
                input_df[col]
                .replace(map_boolean)
                .astype(float)
            )

    if "historicoFamiliarDiabetes" in input_df.columns:
        input_df["historicoFamiliarDiabetes"] = (
            input_df["historicoFamiliarDiabetes"]
            .astype(str).str.strip()
            .replace(map_hist_diabetes)
            .astype(float)
        )


    if "TipoDiabetes" in input_df.columns:
        map_tipo_diabetes = {
            "Diabetes Gestacional": 1,
            "Tipo 1": 2,
            "Tipo 2": 3
        }
        input_df["TipoDiabetes"] = (
            input_df["TipoDiabetes"]
            .astype(str).str.strip()
            .replace(map_tipo_diabetes)
        )
        input_df["TipoDiabetes"] = pd.to_numeric(
            input_df["TipoDiabetes"], errors="coerce"
        ).fillna(0.0)


    for feature in input_features:
        if feature not in input_df.columns:
            input_df[feature] = 0

    for col in input_features:
        input_df[col] = pd.to_numeric(input_df[col], errors="coerce").fillna(0.0)

    return input_df[input_features]


dados_processados = preprocess_input_data(dados_teste_api)
probabilidade = trained_model.predict_proba(dados_processados)[0, 1]

print(f"Probabilidade de Pré-eclâmpsia: {probabilidade:.3f}")
print(f"Predição: {'ALTO RISCO' if probabilidade > 0.5 else 'baixo risco'}")
```

```
Probabilidade de Pré-eclâmpsia: 0.834
Predição: ALTO RISCO
```