Guia Completo para Apresentação: Análise de Dados de Dengue

Introdução: Análise de Dados de Dengue no Contexto da Pós-Graduação

Prezados professores e colegas, boa noite.

É com grande satisfação que nós apresentamos o resultado de um trabalho desenvolvido no âmbito da nossa pós-graduação, focado na análise de dados epidemiológicos de dengue. Este estudo visa demonstrar a aplicação prática de técnicas de análise de dados para extrair insights valiosos de conjuntos de dados complexos, contribuindo para uma compreensão mais aprofundada da dinâmica da doença no Brasil.

Nosso ponto de partida foi um robusto conjunto de dados sobre casos de dengue, obtido diretamente do portal **DataSUS**, uma fonte oficial e de grande relevância para a saúde pública brasileira. Especificamente, utilizamos a base de dados disponível em https://opendatasus.saude.gov.br/gl/dataset/arboviroses-dengue.
Para garantir a correta interpretação e manipulação dessas informações, consultamos o **catálogo de dados** oficial, que detalha cada variável e seu significado, acessível em https://s3.sa-east-1.amazonaws.com/ckan.saude.gov.br/SINAN/Dengue/dic_dados_dengue.pdf.

Ao longo desta apresentação, guiaremos vocês pelas principais etapas do nosso trabalho, que foram estruturadas e executadas em um ambiente de notebook Jupyter. Abordaremos inicialmente a **Apresentação do Dataset**, onde detalharemos a estrutura da base de dados, as variáveis selecionadas e a justificativa para suas escolhas, bem como o processo de pré-processamento e normalização dos dados para torná-los aptos à análise. Em seguida, mergulharemos na **Análise dos Dados**, explorando visualizações como histogramas e mapas de calor para identificar padrões, distribuições e correlações entre as variáveis. Além disso, discutiremos a aplicação de testes estatísticos, como o Qui-quadrado, para verificar associações entre variáveis categóricas, e a construção de modelos de Machine Learning para predição de casos de dengue, incluindo o tratamento de desbalanceamento de classes. Nosso objetivo é

não apenas apresentar os resultados técnicos, mas também discutir as implicações e o potencial dessas análises para o campo da saúde pública.

Esperamos que esta apresentação seja esclarecedora e inspire novas discussões sobre o uso de dados para combater a dengue e outras arboviroses. Muito obrigado.

Explicação Detalhada do Notebook Jupyter - Análise de Casos de Dengue

Este documento detalha os comandos e o propósito de cada etapa do notebook Jupyter `notebook_dengue.ipynb`, preparado para uma apresentação ao vivo.

1. Introdução e Carregamento de Dados

Propósito:

Esta seção inicial define o objetivo do projeto e carrega o conjunto de dados principal para análise.

Comandos:

Importação de Bibliotecas

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

- `import pandas as pd`: Importa a biblioteca Pandas, essencial para manipulação e análise de dados tabulares (DataFrames).
- `import numpy as np`: Importa a biblioteca NumPy, utilizada para operações numéricas de alta performance, especialmente com arrays.
- `import matplotlib.pyplot as plt`: Importa a biblioteca Matplotlib, usada para a criação de gráficos e visualizações estáticas.
- `import seaborn as sns`: Importa a biblioteca Seaborn, que é construída sobre o Matplotlib e oferece uma interface de alto nível para criar gráficos estatísticos atraentes e informativos.

Configuração de Exibição e Carregamento do Dataset

```
pd.set_option(\'display.max_columns\', None)  # Mostra todas as colunas
pd.set_option(\'display.expand_frame_repr\', False)

df = pd.read_csv(\"dados_dengue.csv\")
df.head()
```

- `pd.set_option(\'display.max_columns\', None)` : Configura o Pandas para exibir todas as colunas de um DataFrame, evitando que colunas sejam omitidas na visualização.
- `pd.set_option(\'display.expand_frame_repr\', False)` : Evita que o DataFrame seja quebrado em várias linhas ao ser exibido, melhorando a legibilidade.
- `df = pd.read_csv(\"dados_dengue.csv\")` : Carrega o arquivo CSV `dados_dengue.csv` em um DataFrame do Pandas, atribuindo-o à variável `df`. Este é o dataset principal que será analisado.
- `df.head()` : Exibe as primeiras 5 linhas do DataFrame `df`. Isso é útil para uma rápida inspeção dos dados e de sua estrutura após o carregamento.

2. Análise das Características da Base de Dados

Propósito:

Esta seção explora as características básicas do dataset, como dimensões, estatísticas descritivas e tipos de dados, para entender melhor a estrutura e o conteúdo dos dados.

Comandos:

Dimensões do DataFrame

```
df.shape
```

- `df.shape` : Retorna uma tupla representando as dimensões do DataFrame (número de linhas, número de colunas). É fundamental para saber o tamanho do dataset.

Estatísticas Descritivas

```
df.describe()
```

- `df.describe()`: Gera estatísticas descritivas das colunas numéricas do DataFrame, como contagem, média, desvio padrão, valores mínimo e máximo, e quartis. Ajuda a ter uma visão geral da distribuição dos dados.

Informações do DataFrame

```
df.info()
```

- `df.info()`: Exibe um resumo conciso do DataFrame, incluindo o número de entradas não nulas em cada coluna, o tipo de dado de cada coluna e o uso de memória. Essencial para identificar colunas com valores ausentes e verificar os tipos de dados.

Tipos de Dados por Coluna

```
for coluna in df.columns:
    tipo = df[coluna].dtypes
    print(f\'{coluna}: {tipo}\')
```

- `for coluna in df.columns:`: Itera sobre o nome de cada coluna no DataFrame.
- `tipo = df[coluna].dtypes`: Obtém o tipo de dado da coluna atual.
- `print(f\'{coluna}: {tipo}\')`: Imprime o nome da coluna e seu respectivo tipo de dado. Isso permite uma verificação explícita dos tipos de dados de todas as colunas.

3. Seleção de Colunas e Análise Inicial da Variável Target

Propósito:

Nesta etapa, são selecionadas as colunas mais relevantes para a análise, com foco na variável `CLASSI_FIN` (classificação final) como o alvo principal, e é feita uma primeira inspeção de seus valores.

Comandos:

Definição de Sintomas

```
sintomas = [\'FEBRE\', \'MIALGIA\', \'CEFALEIA\', \'EXANTEMA\', \'VOMITO\',
\'NAUSEA\', \'DOR_COSTAS\', \'CONJUNTVIT\', \'ARTRITE\', \'ARTRALGIA\',
\'PETEQUIA_N\', \'LACO\', \'DOR_RETRO\']
```

- **`sintomas = [...]`**: Cria uma lista contendo os nomes das colunas que representam os sintomas da dengue. Esta lista será usada para facilitar a seleção e manipulação dessas colunas.

Seleção de Colunas Relevantes

```
df = df[[\'CLASSI_FIN\', \'ID_MUNICIP\', \'FEBRE\', \'MIALGIA\', \'CEFALEIA\',
\'EXANTEMA\', \'VOMITO\', \'NAUSEA\', \'DOR_COSTAS\', \'CONJUNTVIT\',
\'ARTRITE\', \'ARTRALGIA\', \'PETEQUIA_N\', \'LACO\', \'DOR_RETRO\']]
df.head()
```

- **`df = df[...]`**: Filtra o DataFrame `df`, mantendo apenas as colunas especificadas na lista. Isso reduz a complexidade do dataset, focando nas variáveis consideradas mais importantes para o problema.
- **`df.head()`**: Exibe as primeiras linhas do DataFrame após a seleção das colunas, confirmando que apenas as colunas desejadas foram mantidas.

Contagem de Valores da Variável Target

```
df[\'CLASSI_FIN\'].value_counts()
```

- : Conta a frequência de cada valor único na coluna `CLASSI_FIN`. Isso é crucial para entender a distribuição da variável alvo (seja ela desbalanceada ou não) e os diferentes tipos de classificação final de dengue presentes no dataset.

4. Análise de Dados e Visualização

Propósito:

Esta seção foca na visualização da distribuição das variáveis numéricas e na análise da correlação entre as variáveis, utilizando gráficos e mapas de calor.

Comandos:

Distribuição de Variáveis Numéricas

```
 colunas_numericas = df.select_dtypes(include=[\'float64\', \'int64\']).columns
plt.figure(figsize=(20, 15))

for i, coluna in enumerate(colunas_numericas, 1):
    plt.subplot(4, 5, i)  # Ajuste o layout conforme o número de colunas
    sns.histplot(df[coluna], kde=True, bins=30, color=\'orange\') # type:
ignore
    plt.title(f\'Distribuição de {coluna}\'')
    plt.xlabel(df[coluna].name)
    plt.ylabel(\'Frequência\')

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

- **`colunas_numericas = df.select_dtypes(include=[\'float64\', \'int64\']).columns`** : Seleciona os nomes de todas as colunas que possuem tipos de dados numéricos (float64 ou int64).
- **`plt.figure(figsize=(20, 15))`** : Cria uma figura para os gráficos com um tamanho específico (20 polegadas de largura por 15 polegadas de altura), garantindo que os gráficos sejam bem dimensionados.
- **`for i, coluna in enumerate(colunas_numericas, 1):`** : Itera sobre cada coluna numérica, atribuindo um índice `i` e o nome da `coluna`.
- **`plt.subplot(4, 5, i)`** : Cria uma grade de subplots (4 linhas, 5 colunas) e seleciona o `i`-ésimo subplot para o gráfico atual. Isso permite exibir múltiplos histogramas em uma única figura.
- **`sns.histplot(df[coluna], kde=True, bins=30, color=\'orange\')`** : Gera um histograma para a coluna atual, mostrando a distribuição de frequência dos valores. `kde=True` adiciona uma estimativa de densidade de kernel, `bins=30` define o número de barras e `color=\'orange\'` define a cor do gráfico.
- **`plt.title(...)`**, **`plt.xlabel(...)`**, **`plt.ylabel(...)`** : Define o título do subplot e os rótulos dos eixos X e Y, tornando o gráfico compreensível.
- **`plt.tight_layout()`** : Ajusta automaticamente os parâmetros do subplot para que os gráficos se encaixem na área da figura sem sobreposição.
- **`plt.show()`** : Exibe a figura com todos os histogramas gerados.

Matriz de Correlação (Mapa de Calor)

```
 plt.figure(figsize=(14, 12)) #ajusta o tamanho

sns.heatmap(df.corr(),
            cmap=\'coolwarm\',       # Define o esquema de cores (azul para
correlações negativas, vermelho para positivas)
            annot=True,            # Adicionar valores numéricos
            fmt=\".2f\",             # Formato com 2 casas decimais
            linewidths=0.5,        # Adicionar linhas entre células
            cbar_kws={\'shrink\': 0.8}  # Ajustar barra de cores
            )

plt.xticks(rotation=45, ha=\'right\')  # Coloca os nomes do campos do eixo x 45
graus
plt.yticks(rotation=0)  # Coloca os nomes do campos do eixo y na horizontal
plt.tight_layout()       # Ajustar layout automaticamente, evitar que título ou
legendas sejam cortados nas bordas
plt.title(\'Matriz de Correlação entre Variáveis\', fontsize=14)
plt.show()
```

- `plt.figure(figsize=(14, 12))` : Define o tamanho da figura para o mapa de calor.
- `sns.heatmap(df.corr(), ...)` : Gera um mapa de calor da matriz de correlação do DataFrame. `df.corr()` calcula a correlação de Pearson entre todas as colunas numéricas.
  - `cmap=\'coolwarm\'` : Define o esquema de cores, onde `coolwarm` usa tons de azul para correlações negativas e vermelho para positivas.
  - `annot=True` : Exibe os valores de correlação em cada célula do mapa de calor.
  - `fmt=\".2f\"` : Formata os valores de correlação com duas casas decimais.
  - `linewidths=0.5` : Adiciona linhas entre as células para melhor separação visual.
  - `cbar_kws={\'shrink\': 0.8}` : Ajusta o tamanho da barra de cores lateral.
- `plt.xticks(rotation=45, ha=\'right\')` : Rotaciona os rótulos do eixo X em 45 graus para evitar sobreposição, especialmente com muitos nomes de colunas.
- `plt.yticks(rotation=0)` : Mantém os rótulos do eixo Y na horizontal.
- `plt.tight_layout()` : Ajusta o layout para evitar cortes de títulos ou legendas.
- `plt.title(\'Matriz de Correlação entre Variáveis\', fontsize=14)` : Define o título principal do mapa de calor.

- `plt.show()` : Exibe o mapa de calor.

5. Teste Qui-quadrado de Independência

Propósito:

Como a correlação de Pearson não é adequada para variáveis categóricas, esta seção aplica o teste Qui-quadrado de independência para verificar a associação estatística entre as variáveis de sintomas (categóricas) e a variável alvo `CLASSI_FIN` .

Comandos:

Importação e Aplicação do Teste Qui-quadrado

```
from scipy.stats import chi2_contingency

alpha = 0.05
for coluna in df.columns:
    tabela = pd.crosstab(df[coluna], df[\'CLASSI_FIN\'])
    chi2, p, dof, expected = chi2_contingency(tabela)
    # Compare the p-value to the significance level
    print(f\"p-valor `{coluna}`: {p}\")
```

- `from scipy.stats import chi2_contingency` : Importa a função `chi2_contingency` da biblioteca SciPy, que realiza o teste Qui-quadrado de independência.
- `alpha = 0.05` : Define o nível de significância (alfa) para o teste. Um p-valor menor que `alpha` indica uma associação estatística significativa.
- `for coluna in df.columns:` : Itera sobre cada coluna do DataFrame.
- `tabela = pd.crosstab(df[coluna], df[\'CLASSI_FIN\'])` : Cria uma tabela de contingência (cruzamento de frequências) entre a coluna atual e a coluna `CLASSI_FIN` . Esta tabela é a entrada para o teste Qui-quadrado.
- `chi2, p, dof, expected = chi2_contingency(tabela)` : Executa o teste Qui-quadrado. Retorna o valor da estatística Qui-quadrado (`chi2`), o p-valor (`p`), os graus de liberdade (`dof`) e as frequências esperadas (`expected`).
- `print(f\"p-valor : {p}\")` : Imprime o p-valor para cada coluna. Um p-valor baixo (geralmente < 0.05) sugere que a coluna está estatisticamente relacionada com `CLASSI_FIN` .

6. Normalização e Pré-processamento de Dados

Propósito:

Esta seção trata da limpeza e transformação dos dados, incluindo a remoção de valores nulos e a padronização de valores para preparar o dataset para a modelagem.

Comandos:

Verificação de Valores Nulos

```
df.isnull().sum()
```

- `df.isnull().sum()` : Retorna a contagem de valores nulos para cada coluna do DataFrame. Essencial para identificar a presença e a extensão de dados ausentes.

Remoção de Linhas com Valores Nulos na Variável Target

```
df = df.dropna()
print(df.isnull().sum())
print(df.shape)
```

- **`df = df.dropna()`** : Remove todas as linhas do DataFrame que contêm pelo menos um valor nulo. A decisão de remover nulos na variável target é justificada para manter a integridade dos dados categóricos e evitar variações indesejadas na predição.
- **`print(df.isnull().sum())`** : Verifica novamente a contagem de nulos após a remoção, confirmando que não há mais valores ausentes.
- **`print(df.shape)`** : Exibe as novas dimensões do DataFrame após a remoção das linhas, mostrando quantas linhas foram afetadas.

Conversão de Tipos de Dados para Inteiro

```
for coluna in df.columns:
    df[coluna] = df[coluna].fillna(0).astype(int)
df.head()
```

- **`for coluna in df.columns:`** : Itera sobre todas as colunas do DataFrame.
- **`df[coluna] = df[coluna].fillna(0).astype(int)`** : Preenche quaisquer valores nulos remanescentes (embora `dropna()` já tenha sido usado, esta linha

garante que não haja nulos antes da conversão) com 0 e, em seguida, converte o tipo de dado da coluna para inteiro. Isso padroniza os dados para uso em modelos de machine learning.

- `df.head()` : Exibe as primeiras linhas do DataFrame para verificar a conversão dos tipos de dados.

Normalização de Valores de Sintomas (2 para 0)

```
for sintoma in sintomas:
    df.loc[df[sintoma] == 2, sintoma] = 0
df.head()
```

- `for sintoma in sintomas:` : Itera sobre a lista de colunas de sintomas previamente definida.
- `df.loc[df[sintoma] == 2, sintoma] = 0` : Localiza todas as ocorrências do valor `2` (que representa 'Não' para sintomas) e as substitui por `0`, padronizando os valores booleanos para `1` (Sim) e `0` (Não).
- `df.head()` : Exibe as primeiras linhas do DataFrame para verificar a normalização dos valores dos sintomas.

Normalização da Variável Target `CLASSI_FIN`

```
df.loc[df[\"CLASSI_FIN\"] == 10, \"CLASSI_FIN\"] = 1
df.loc[df[\"CLASSI_FIN\"] == 11, \"CLASSI_FIN\"] = 1
df.loc[df[\"CLASSI_FIN\"] == 12, \"CLASSI_FIN\"] = 1
df.loc[df[\"CLASSI_FIN\"] == 8, \"CLASSI_FIN\"] = 0
df.head(10)
```

- `df.loc[df[\"CLASSI_FIN\"] == 10, \"CLASSI_FIN\"] = 1` : Substitui o valor `10` (Dengue) na coluna `CLASSI_FIN` por `1`.
- `df.loc[df[\"CLASSI_FIN\"] == 11, \"CLASSI_FIN\"] = 1` : Substitui o valor `11` (Dengue com sinais de alarme) por `1` .
- `df.loc[df[\"CLASSI_FIN\"] == 12, \"CLASSI_FIN\"] = 1` : Substitui o valor `12` (Dengue grave) por `1` .
- `df.loc[df[\"CLASSI_FIN\"] == 8, \"CLASSI_FIN\"] = 0` : Substitui o valor `8` (Descartado) por `0` .
- **Propósito**: Esta etapa transforma a variável `CLASSI_FIN` em uma variável binária (`1` para casos de dengue, `0` para casos descartados), tornando-a

adequada para modelos de classificação binária.

- `df.head(10)` : Exibe as primeiras 10 linhas do DataFrame para verificar a normalização da variável target.

7. Treino e Avaliação dos Modelos

Propósito:

Esta seção prepara os dados para a modelagem, divide o dataset em conjuntos de treino e teste, treina diferentes modelos de classificação e avalia seu desempenho.

Comandos:

Separação da Base de Treino e Teste

```
from sklearn.model_selection import train_test_split

x = df[[\"ID_MUNICIP\", \"FEBRE\", \"MIALGIA\", \"CEFALEIA\", \"VOMITO\"]]
y = df[\"CLASSI_FIN\"]

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

print(\"Total base de treino: \", len(x_train))
print(\"Total base de teste: \", len(y_test))
```

- `from sklearn.model_selection import train_test_split` : Importa a função `train_test_split` do Scikit-learn, usada para dividir arrays ou matrizes em subconjuntos aleatórios de treino e teste.
- `x = df[...]` : Define as variáveis preditoras (features) `x` , que são as colunas selecionadas que serão usadas para prever a classificação da dengue.
- `y = df[\"CLASSI_FIN\"]` : Define a variável alvo (target) `y` , que é a coluna `CLASSI_FIN` já normalizada.
- `x_train, x_test, y_train, y_test = train_test_split(...)` : Divide os dados em quatro conjuntos:
  - `x_train` : Features para treino.
  - `x_test` : Features para teste.
  - `y_train` : Target para treino.
  - `y_test` : Target para teste.

    - `test_size=0.2`: Indica que 20% dos dados serão usados para o conjunto de teste e 80% para o treino.
    - `stratify=y`: Garante que a proporção das classes da variável `y` seja a mesma nos conjuntos de treino e teste. Isso é crucial para datasets desbalanceados.
    - `random_state=42`: Define uma semente para a aleatoriedade, garantindo que a divisão dos dados seja a mesma em execuções futuras, o que torna o resultado reproduzível.
- `print(...)`: Exibe o número total de amostras nos conjuntos de treino e teste.

Treinamento e Avaliação de Múltiplos Modelos

```
 from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (accuracy_score, classification_report,
confusion_matrix,
                             roc_curve, auc, precision_recall_curve)

models = {
    \"Logistic Regression\": LogisticRegression(max_iter=1000,
random_state=42),
    \"Random Forest\": RandomForestClassifier(n_estimators=100,
random_state=42),
    \"Gradient Boosting\": GradientBoostingClassifier(n_estimators=100,
random_state=42),
    \"KNN\": KNeighborsClassifier(n_neighbors=5)
}

resultados = {}
for nome, modelo in models.items():
    print(f\"\\nTreinando modelo: {nome}\")

    modelo.fit(x_train, y_train)

    y_pred = modelo.predict(x_test)

    # Calculando métricas
    acuracia = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, output_dict=True)

    # Armazenando resultados
    resultados[nome] = {
        \"acuracia\": acuracia,
        \"report\": report,
        \"modelo\": modelo,
        \"predicoes\": y_pred
    }

    print(f\"Acurácia: {acuracia:.4f}\")
    print(\"\\nClassification Report:\")
    print(classification_report(y_test, y_pred))
```

- **Importação de Modelos e Métricas**: Importa as classes dos modelos de machine learning (`LogisticRegression` , `RandomForestClassifier` , `GradientBoostingClassifier` , `KNeighborsClassifier`) e as funções de métricas de avaliação (`accuracy_score` , `classification_report` , etc.) do Scikit-learn.
- **`models = {...}`** : Cria um dicionário onde as chaves são os nomes dos modelos e os valores são as instâncias dos modelos com seus respectivos parâmetros. Isso permite iterar facilmente sobre diferentes algoritmos.
    - **`LogisticRegression`** : Um modelo linear para classificação binária.

    - `RandomForestClassifier` : Um modelo de ensemble baseado em árvores de decisão, conhecido por sua robustez.
    - `GradientBoostingClassifier` : Outro modelo de ensemble que constrói árvores de decisão de decisão de forma sequencial, corrigindo erros das árvores anteriores.
    - `KNeighborsClassifier` : Um classificador baseado na proximidade dos pontos de dados (k-vizinhos mais próximos).
- `for nome, modelo in models.items():` : Itera sobre cada modelo no dicionário `models` .
- `modelo.fit(x_train, y_train)` : Treina o modelo atual usando os dados de treino (`x_train` e `y_train`).
- `y_pred = modelo.predict(x_test)` : Realiza previsões no conjunto de teste (`x_test`) usando o modelo treinado.
- **Cálculo e Armazenamento de Métricas**: Calcula a acurácia (`accuracy_score`) e gera um relatório de classificação (`classification_report`) para cada modelo. Os resultados são armazenados no dicionário `resultados` .
- `print(...)` : Exibe a acurácia e o relatório de classificação para cada modelo, fornecendo uma visão detalhada do desempenho.

Salvando o Modelo Treinado (KNN)

```
import pickle

with open(\"model.pkl\", \"wb\") as arquivo:
    pickle.dump(resultados[\"KNN\"][\"modelo\"], arquivo)
```

- `import pickle` : Importa a biblioteca `pickle` , que permite serializar e desserializar objetos Python. Isso é útil para salvar modelos treinados e carregá-los posteriormente sem precisar retreiná-los.
- `with open(\"model.pkl\", \"wb\") as arquivo:` : Abre um arquivo chamado `model.pkl` em modo de escrita binária (`\"wb\"`).
- `pickle.dump(resultados[\"KNN\"][\"modelo\"], arquivo)` : Salva o modelo KNN (que está armazenado no dicionário `resultados` sob a chave \"KNN\" e subchave \"modelo\") no arquivo `model.pkl` .

8. Tratamento de Desbalanceamento de Classes com SMOTE

Propósito:

Esta seção aborda o problema do desbalanceamento de classes (onde uma classe tem muito mais amostras que a outra), que pode levar a modelos com alta acurácia, mas baixa precisão para a classe minoritária. O SMOTE (Synthetic Minority Over-sampling Technique) é aplicado para balancear as classes.

Comandos:

Aplicação do SMOTE

```
from imblearn.over_sampling import SMOTE

# Aplicar SMOTE para oversampling da classe minoritária
oversample = SMOTE()
x_train_os, y_train_os = oversample.fit_resample(x_train, y_train)

print(\"Total base de treino: \", len(x_train))
print(\"Total base de teste: \", len(y_test))

print(\"Total base de treino oversampling: \", len(x_train_os))
print(\"Total base de teste oversampling: \", len(y_train_os))
```

- **`from imblearn.over_sampling import SMOTE`** : Importa a classe `SMOTE` da biblioteca `imbalanced-learn`, que é especializada em lidar com datasets desbalanceados.
- **`oversample = SMOTE()`** : Cria uma instância do SMOTE.
- **`x_train_os, y_train_os = oversample.fit_resample(x_train, y_train)`** : Aplica o SMOTE ao conjunto de treino. Ele gera novas amostras sintéticas para a classe minoritária, balanceando a distribuição das classes. Os novos conjuntos de treino balanceados são `x_train_os` e `y_train_os`.
- **`print(...)`** : Exibe o número de amostras antes e depois do oversampling, demonstrando o efeito do SMOTE no balanceamento das classes.

Retreinamento e Avaliação dos Modelos com Dados Balanceados

```
 from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (accuracy_score, classification_report,
confusion_matrix,
                             roc_curve, auc, precision_recall_curve)

models = {
    \"Logistic Regression\": LogisticRegression(max_iter=1000,
random_state=42),
    \"Random Forest\": RandomForestClassifier(n_estimators=100,
random_state=42),
    \"Gradient Boosting\": GradientBoostingClassifier(n_estimators=100,
random_state=42),
    \"KNN\": KNeighborsClassifier(n_neighbors=5)
}

resultados = {}
for nome, modelo in models.items():
    print(f\"\\nTreinando modelo: {nome}\")

    modelo.fit(x_train_os, y_train_os)

    y_pred = modelo.predict(x_test)

    # Calculando métricas
    acuracia = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, output_dict=True)

    # Armazenando resultados
    resultados[nome] = {
        \"acuracia\": acuracia,
        \"report\": report,
        \"modelo\": modelo,
        \"predicoes\": y_pred
    }

    print(f\"Acurácia: {acuracia:.4f}\")
    print(\"\\nClassification Report:\")
    print(classification_report(y_test, y_pred))
```

- **Propósito**: Após o balanceamento das classes com SMOTE, os modelos são retreinados com os dados balanceados (`x_train_os`, `y_train_os`) e avaliados novamente no conjunto de teste original (`x_test`, `y_test`). Isso permite verificar se o oversampling melhorou o desempenho dos modelos, especialmente a precisão para a classe minoritária.
- **Comandos**: Os comandos são os mesmos da seção anterior de treinamento e avaliação de modelos, mas agora utilizando os dados de treino balanceados pelo SMOTE.