Classificador De Inadimplência

Carolina Santiago de Medeiros - DRE 122053305

1 Código Trabalho 1

1.1 Definindo os DataFrames de treinamento e de teste

[ ]:
```
import pandas as pd

df_train = pd.read_csv('Dataset/conjunto_de_treinamento.csv')
df_train.drop(columns=['id_solicitante'], inplace=True)

df_test = pd.read_csv('Dataset/conjunto_de_teste.csv')
df_test.drop(columns=['id_solicitante'], inplace=True)
```

1.2 Pré-processamento dos dados

1.2.1 Removendo features que atrapalham o resultado final (olhar seção de Análise de Dados)

[ ]:
```
df_train.drop(columns=['valor_patrimonio_pessoal',␣
 ↪'renda_mensal_regular', 'renda_extra',␣
 ↪'meses_no_trabalho','grau_instrucao_companheiro',␣
 ↪'profissao_companheiro', 'estado_onde_trabalha',␣
 ↪'produto_solicitado', 'possui_outros_cartoes',␣
 ↪'possui_telefone_celular', 'grau_instrucao'], inplace=True)
df_test.drop(columns=['valor_patrimonio_pessoal',␣
 ↪'renda_mensal_regular', 'renda_extra',␣
 ↪'meses_no_trabalho','grau_instrucao_companheiro',␣
 ↪'profissao_companheiro', 'estado_onde_trabalha',␣
 ↪'produto_solicitado', 'possui_outros_cartoes',␣
 ↪'possui_telefone_celular', 'grau_instrucao'], inplace=True)
```

1.3 Atribuindo valores numéricos a variáveis nominais

1.3.1 Variáveis binárias:

- Sim: 1, Não: 0
- Sexo - Feminino: 1, Masculino: 0

[ ]:
```
def var_binarias(df):
    # Variáveis de sim ou não
```

```
    yn_cols = ['possui_telefone_residencial',␣
↪'vinculo_formal_com_empresa', 'possui_telefone_trabalho']

    for col in yn_cols:
        df[col] = df[col].map({'N': 0, 'Y': 1})

    df['sexo'] = df['sexo'].map({'M': 0, 'F': 1, 'N': None})
    df['forma_envio_solicitacao'] = df['forma_envio_solicitacao'].
↪map({'internet': 0, 'correio': 1, 'presencial': 2})

    return df

df_train = var_binarias(df_train)
df_test = var_binarias(df_test)
```

1.3.2 Formatação das variáveis de UF (estados)

- Atribui um valor de 0 a 26 para cada sigla.
- Ex. 'RJ': 18

[ ]:
```
def estados_format(df, column):
    # Variáveis de UF
    return df[column].map({'AC': 0, 'AL': 1, 'AM': 2, 'AP': 3, 'BA':␣
↪4, 'CE': 5, 'DF': 6, 'ES': 7, 'GO': 8, 'MA': 9, 'MG': 10, 'MS': 11,␣
↪'MT': 12, 'PA': 13, 'PB': 14, 'PE': 15, 'PI': 16, 'PR': 17, 'RJ':␣
↪18, 'RN': 19, 'RO': 20, 'RR': 21, 'RS': 22, 'SC': 23, 'SE': 24,␣
↪'SP': 25, 'TO': 26})

df_train['estado_onde_nasceu'] = estados_format(df_train,␣
↪'estado_onde_nasceu')
df_train['estado_onde_reside'] = estados_format(df_train,␣
↪'estado_onde_reside')

df_test['estado_onde_nasceu'] = estados_format(df_test,␣
↪'estado_onde_nasceu')
df_test['estado_onde_reside'] = estados_format(df_test,␣
↪'estado_onde_reside')
```

1.3.3 Transforma tipos 'str' em 'int'

- Algumas colunas estão preenchidas por valores numéricos, porém em formato de 'string'.
- A célula a seguir transforma esses valores em 'int'.
- Para colunas que possuem valores NaN, a função os transforma em "-1" (decisão arbitrária) antes de passar para int.

[ ]:
```
def tel_format(df, column):
    df[column].replace(' ', -1, inplace=True)
```

```
    return df[column].map(int)

df_train['codigo_area_telefone_trabalho'] = tel_format(df_train,␣
↪'codigo_area_telefone_trabalho')
df_train['codigo_area_telefone_residencial'] = tel_format(df_train,␣
↪'codigo_area_telefone_residencial')

df_test['codigo_area_telefone_trabalho'] = tel_format(df_test,␣
↪'codigo_area_telefone_trabalho')
df_test['codigo_area_telefone_residencial'] = tel_format(df_test,␣
↪'codigo_area_telefone_residencial')
```

1.4 Completando colunas com valores faltando (null)

- As colunas nominais que restaram são preenchidas com '-1' em espaços em branco.
- Algumas colunas, como 'grau_instrucao_companheiro', foram excluídas simplesmente por possuir uma grande quantidade de valores faltando (>10000)

[ ]:
```
for i in range(len(df_train.columns)):
    if df_train[df_train.columns[i]].isnull().sum() > 0:
        print(df_train.columns[i], df_train[df_train.columns[i]].
↪isnull().sum())
        df_train[df_train.columns[i]].fillna(df_train[df_train.
↪columns[i]].mean(), inplace=True)


for i in range(len(df_test.columns)):
    if df_test[df_test.columns[i]].isnull().sum() > 0:
        print(df_test.columns[i], df_test[df_test.columns[i]].
↪isnull().sum())
        df_test[df_test.columns[i]].fillna(df_test[df_test.
↪columns[i]].mean(), inplace=True)
```

1.4.1 Output:

Para o df_train:

- sexo 32
- estado_onde_nasceu 822
- tipo_residencia 536
- meses_na_residencia 1450
- profissao 3097
- ocupacao 2978
- profissao_companheiro 11514
- grau_instrucao_companheiro 12860

Para o df_teste:

- sexo 8
- estado_onde_nasceu 210
- tipo_residencia 125
- meses_na_residencia 362
- profissao 762
- ocupacao 690
- profissao_companheiro 2887
- grau_instrucao_companheiro 3210

1.5 Treinamento do Modelo com Naive Bayes

1.5.1 Introdução ao Naive Bayes

O algoritmo Naive Bayes é um método de aprendizado de máquina baseado na aplicação do teorema de Bayes, com a suposição ingênua (daí o nome "Naive") de independência condicional entre as características. É amplamente utilizado para problemas de classificação, especialmente quando temos dados categóricos ou discretos.

1.5.2 Separação dos Dados

Antes de treinar o modelo, realizamos a preparação dos dados, que envolve a separação do conjunto de treinamento e teste. Os dados são divididos em duas partes: a matriz de características (X_train e X_test), que contém as variáveis independentes, e o vetor de variável alvo (y_train), que contém as classes ou valores a serem previstos.

1.5.3 Treinamento

Para criar o modelo Naive Bayes, importamos a classe `GaussianNB` do módulo `sklearn.naive_bayes`. Em seguida, criamos uma instância do modelo chamada `model` usando `GaussianNB()`.

A seguir, o modelo é treinado com os dados de treinamento através do método `fit(X_train, y_train)`. O modelo usa os dados de treinamento para aprender as probabilidades das classes e a distribuição dos recursos para fazer previsões.

1.5.4 Previsões

Uma vez que o modelo está treinado, realizamos as previsões em um conjunto de teste (`X_test`) usando o método `predict(X_test)`. As previsões resultantes são armazenadas na variável `y_pred` e `predictions`, que será utilizada posteriormente, na seção Resultados.

[ ]:
```
from sklearn.naive_bayes import GaussianNB

# Separar os dados em treino e teste
X_train = df_train.iloc[:, :-1]
y_train = df_train.iloc[:, -1]
X_test = df_test.iloc[:, :]

# Criar o modelo Naive Bayes
```

```
model = GaussianNB()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
predictions = y_pred
```

1.6 Análise dos Dados

Nesta seção, apresentamos duas funções que auxiliaram na análise dos dados antes do treinamento do modelo de Naive Bayes.

1.6.1 Função compare_feature_impact

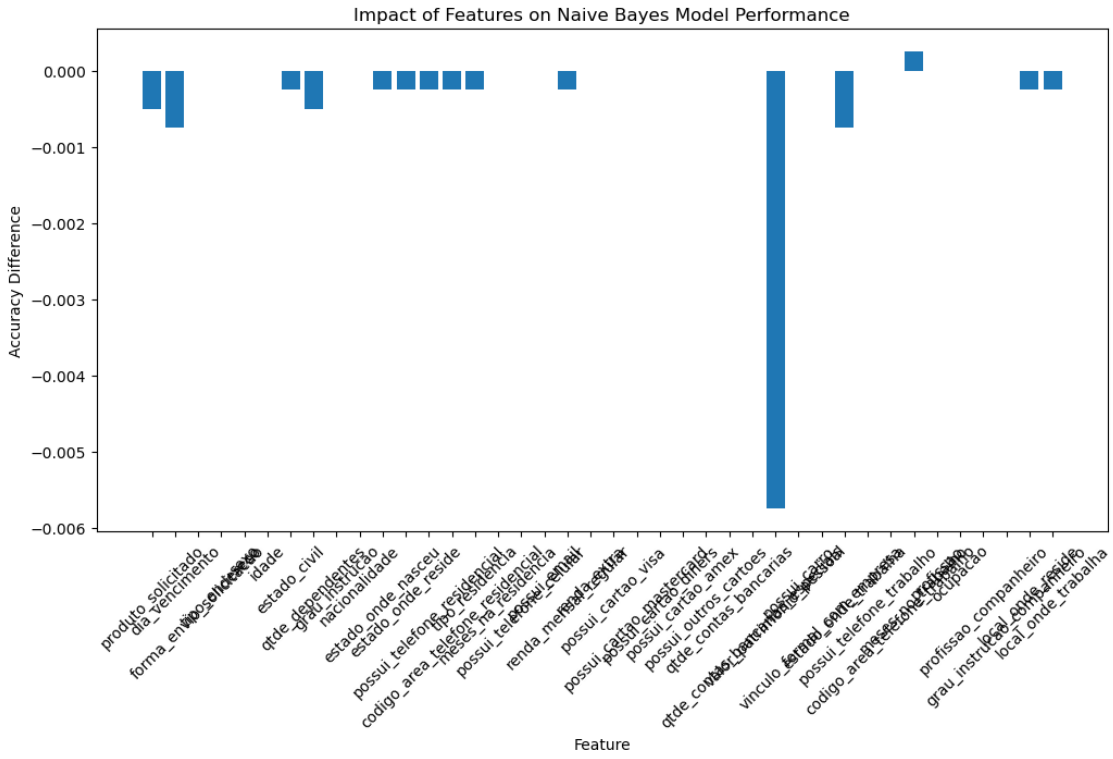
A função `compare_feature_impact(df, target_column)` é responsável por avaliar o "impacto" de cada feature (variável) nos resultados do modelo Naive Bayes. Primeiramente, o modelo Naive Bayes é treinado de forma `isolada` (dentro da função) com todas as features do conjunto de treino e sua acurácia é calculada no conjunto de teste como uma linha de base. Em seguida, a função itera sobre cada feature, treina o modelo sem ela e calcula a nova acurácia. A diferença entre a acurácia do modelo completo e a acurácia sem a feature é armazenada em um dicionário chamado `feature_impact`.

Além disso, a função cria uma lista chamada `features_to_remove` para armazenar as features cuja remoção resulta em uma diminuição significativa da acurácia (diferença menor ou igual a -0.005). A lista é classificada com base nas diferenças de acurácia para fornecer uma indicação das features que podem ser menos relevantes.

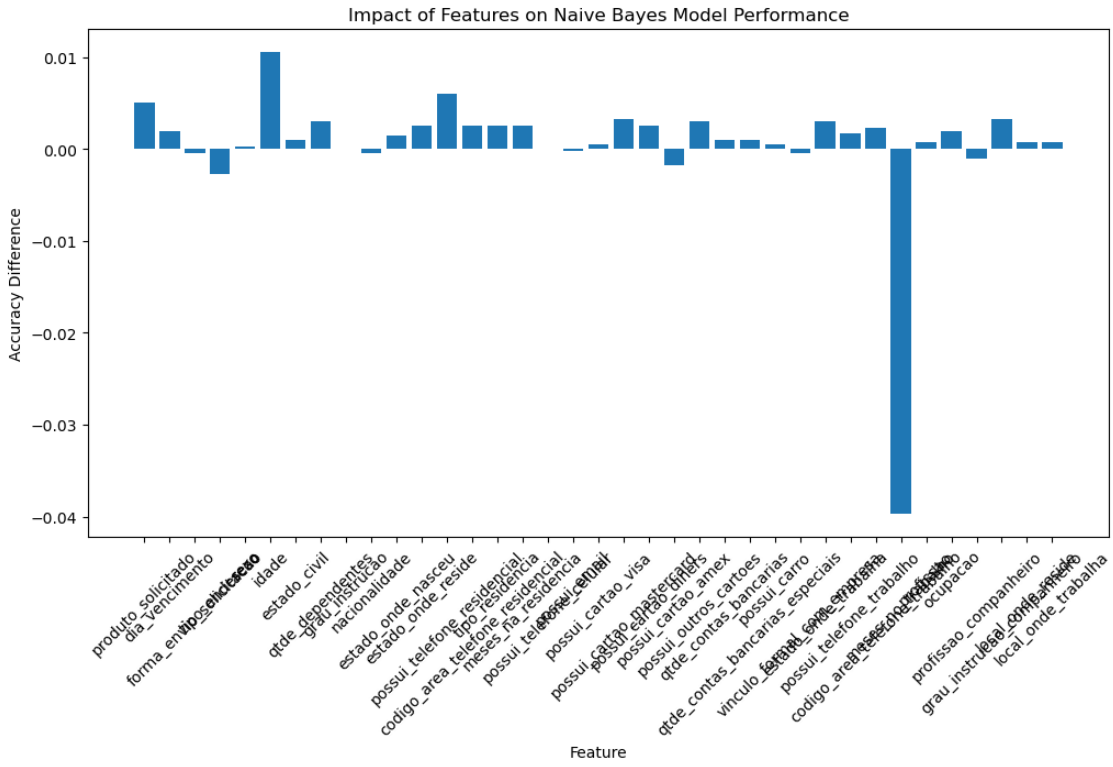
Por fim, a função gera um gráfico de barras para visualizar o impacto de cada feature na acurácia do modelo e imprime a lista de features a serem removidas.

O método mais utilizado para o treinamento eficiente do modelo foi a análise dos gráficos gerados pelas funções dessa seção, e a partir disso decidir quais features estavam atrapalhando o resultado final.

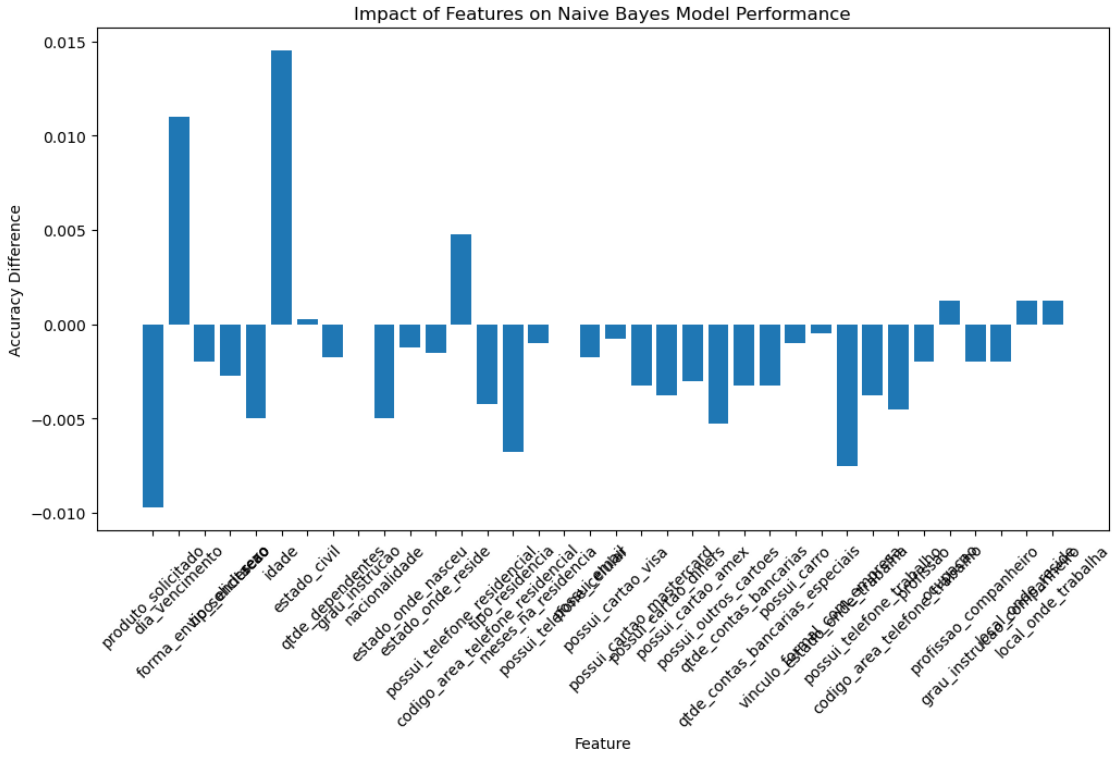
Para a realização dessas decisões, a função `compare_feature_impact` foi utilizada para gerar um gráfico baseado no DataFrame de treinamento com todas as suas features originais. Foi observado (Figura) que a feature 'valor_patrimonio_pessoal' era a principal em termos de desvio da acuracia, e portanto foi a primeira a ser removida. A acurácia medida utilizando todas as features e 20% dos dados do DataFrame de treino como dados de teste foi de 49,3%.



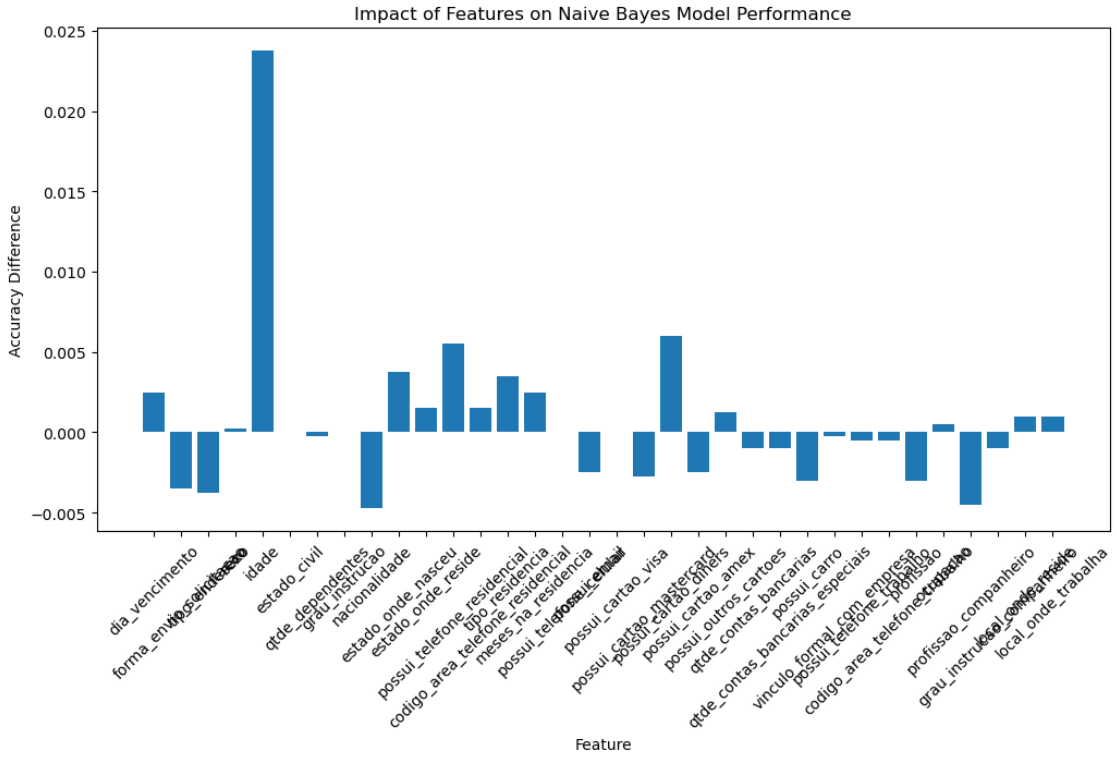
Após a remoção de 'valor_patrimonio_pessoal', a acurácia subiu para 51.05% e a função `compare_feature_impact` sugeriu a remoção das features 'renda_mensal_regular' e 'renda_extra'. O parâmetro de diferença de acurácias menor ou igual a -0.005 provou ser muito eficiente pois, a cada remoção, o gráfico aparentava se estabilizar ainda mais. A figura abaixo apresenta o gráfico depois da remoção das categorias de renda.



Com isso, torna-se óbvia a necessidade da remoção de 'meses_no_trabalho'. A nova acurácia sobe para 56.25%, mas ainda variando entre 54.9% até 57.7%. A seguir, o gráfico de impacto após esse passo.



Seguindo com a análise, a função sugere a remoção de 5 features, mas foi decidido que seriam removidas apenas as duas com diferença de acurácia de maiores módulos. Nesse caso, foram as categorias 'produto_solicitado' e 'estado_onde_trabalha'. A acurácia subiu para uma média de 56.5%, e o gráfico apresentou extrema estabilização entre as variáveis.

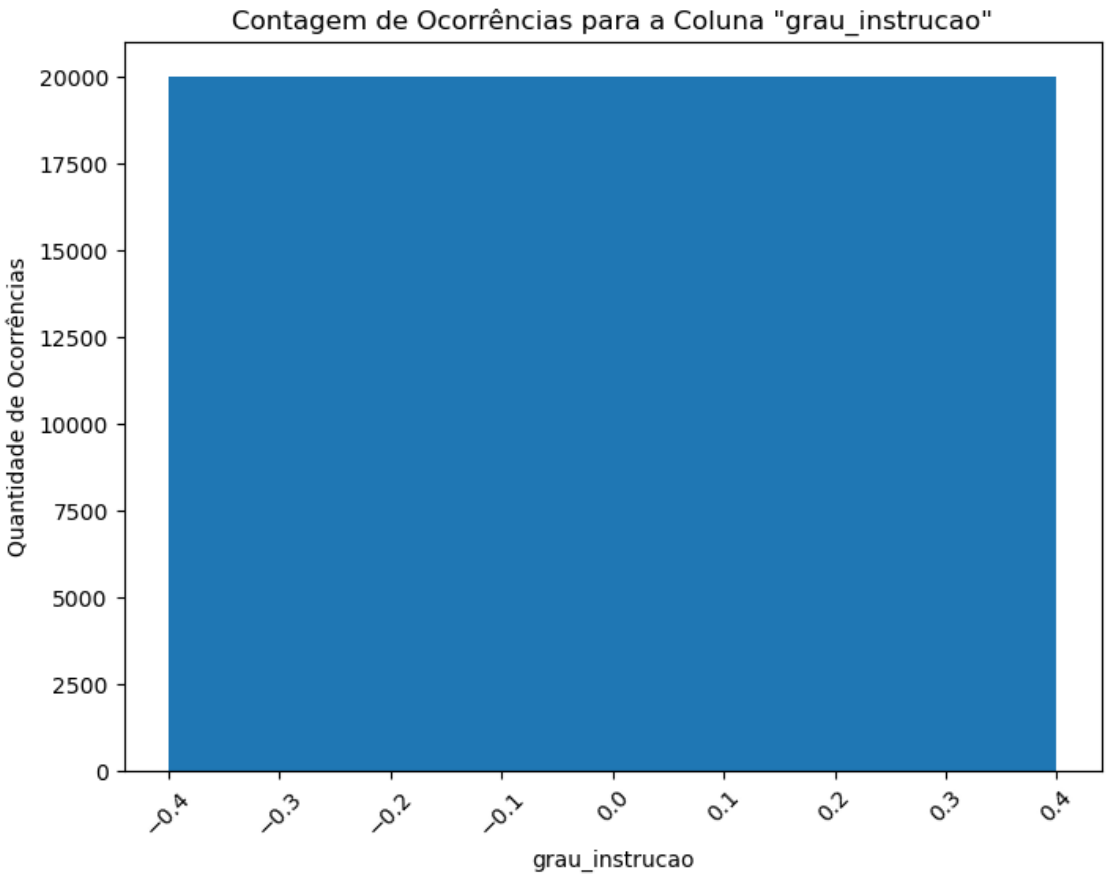


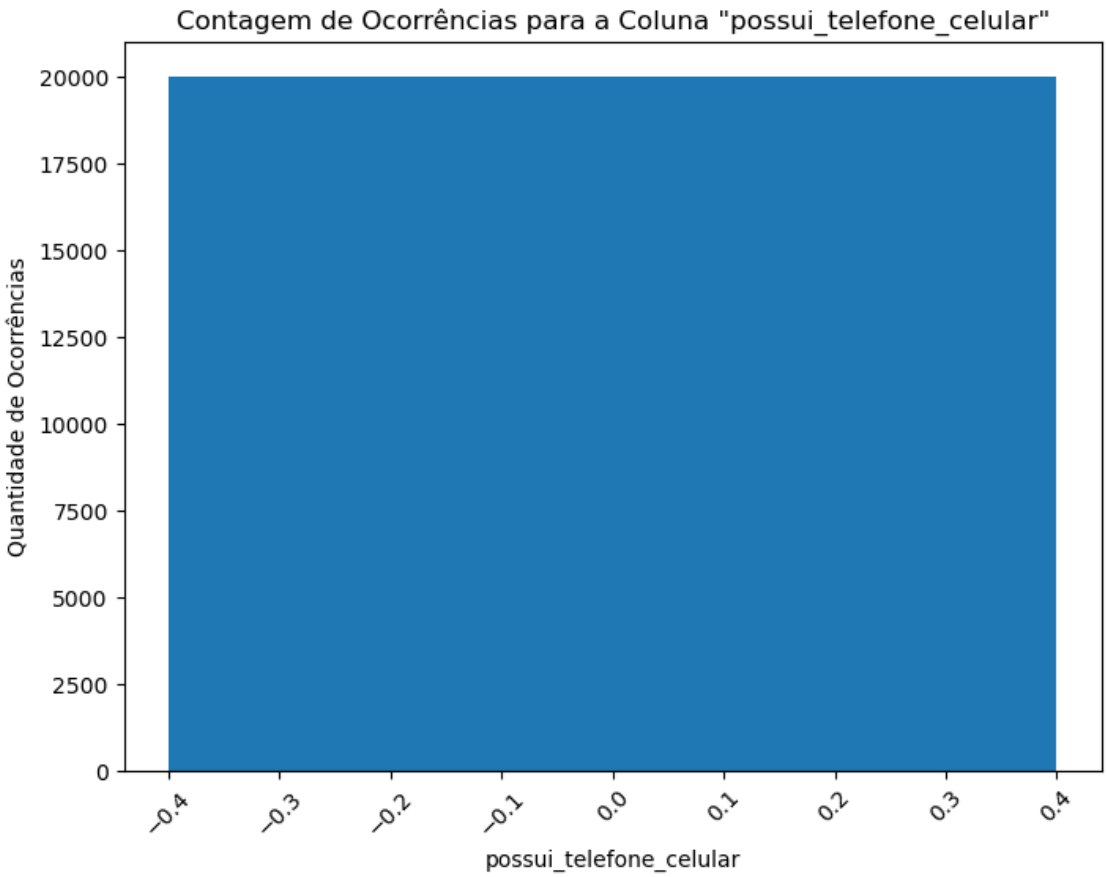
1.6.2 Função generate_bar_charts

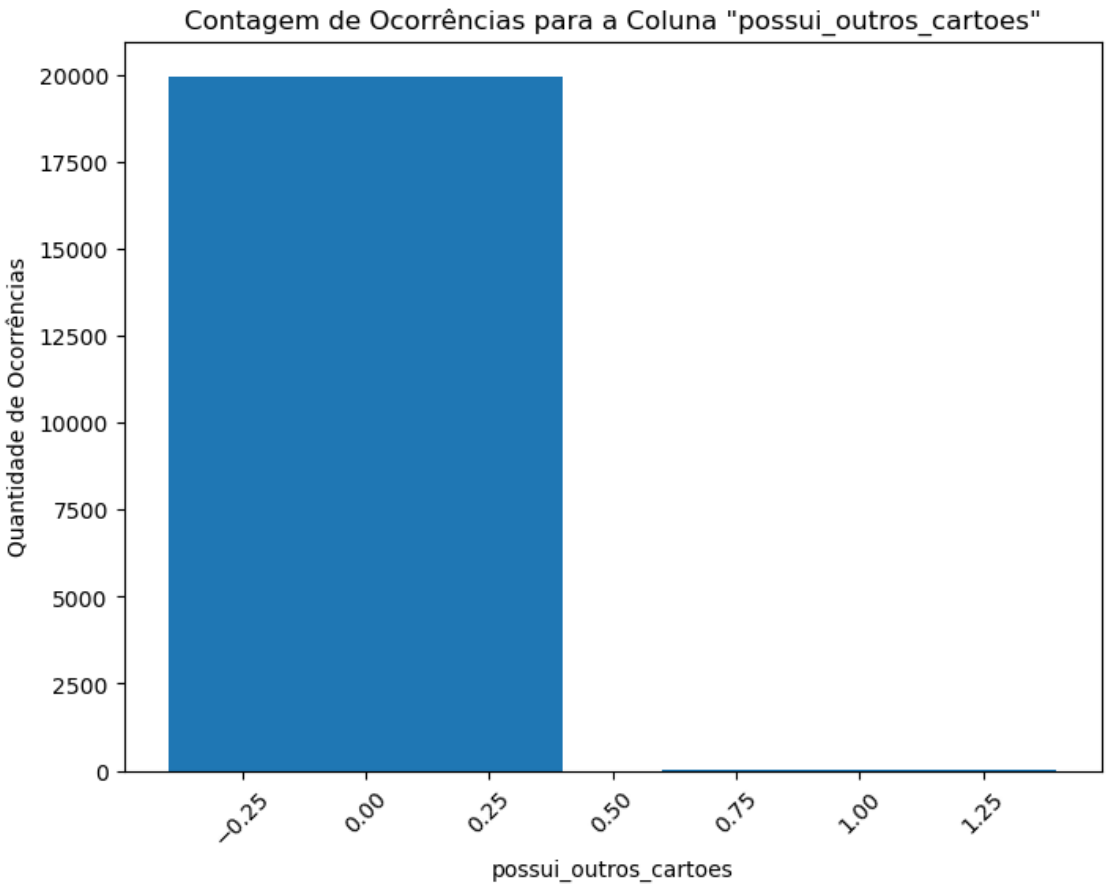
A função `generate_bar_charts(df)` é responsável por gerar gráficos de barras para todas as colunas do DataFrame fornecido. Ela itera sobre cada coluna e verifica o tipo de dados. Se a coluna for do tipo categórico, a função conta o número de ocorrências de cada valor na coluna e cria um gráfico de barras para visualizar a distribuição dos dados.

O dicionário `bar_charts` é criado para armazenar os gráficos de barras de cada coluna. Cada gráfico é plotado usando a biblioteca `matplotlib.pyplot`.

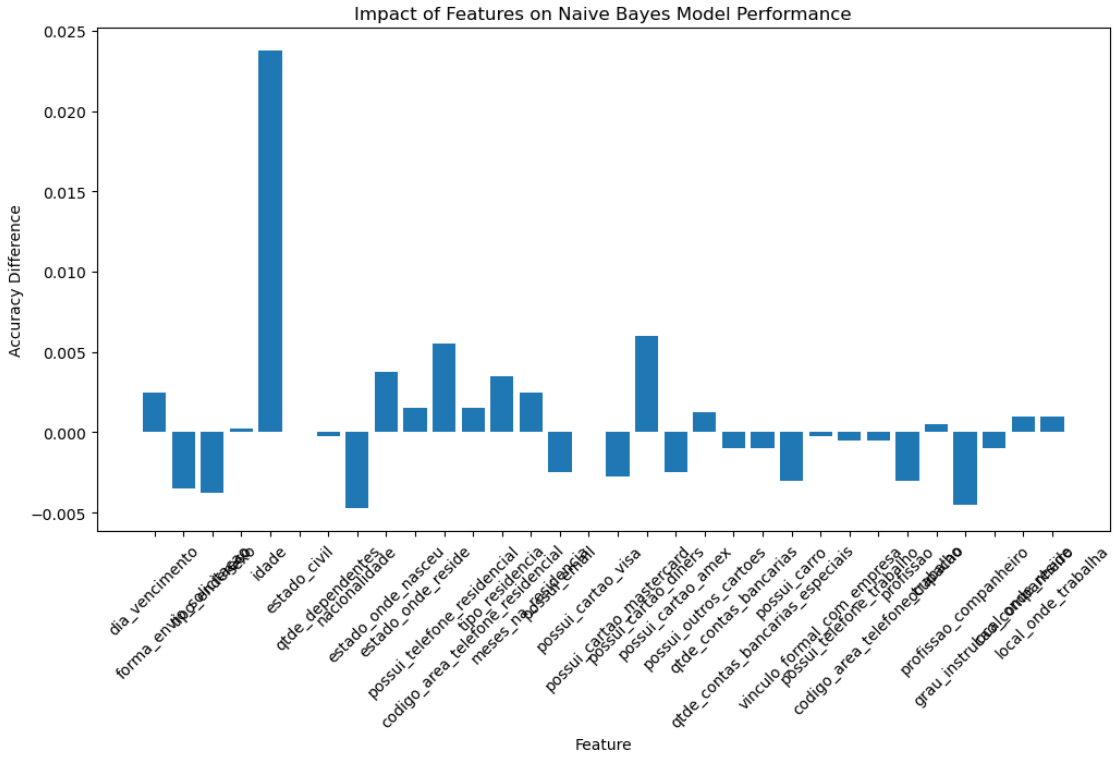
A segunda etapa do Pré-Processamento de Dados consiste na análise dos gráficos gerados, para avaliar se todas as variáveis estão de acordo com o modelo. Foi notado que as variáveis 'possui_outros_cartoes', 'possui_telefone_celular' e 'grau_instrucao' deveriam ser removidas.


Contagem de Ocorrências para a Coluna "grau_instrucao"
Quantidade de Ocorrências
20000
17500
15000
12500
10000
7500
5000
2500
0
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
grau_instrucao


Contagem de Ocorrências para a Coluna "possui_telefone_celular"
Quantidade de Ocorrências
20000
17500
15000
12500
10000
7500
5000
2500
0
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
possui_telefone_celular



Após essa etapa, foi observa uma acurácia média de 56.67% e um gráfico de impacto das variaveis no seguinte formato, sem muitas mudanças:



[ ]:
```
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


def compare_feature_impact(df, target_column):
    # Prepare the feature matrix (X) and target variable (y)
    X = df.drop(target_column, axis=1)
    y = df[target_column]

    # Split the dataset into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y,␣
↪test_size=0.2, random_state=42)

    # Create and train the Naive Bayes model with all features
    naive_bayes_model_with_all_features = GaussianNB()
    naive_bayes_model_with_all_features.fit(X_train, y_train)

    # Get the baseline accuracy with all features included
    y_pred_with_all_features = naive_bayes_model_with_all_features.
↪predict(X_test)
    baseline_accuracy = accuracy_score(y_test,␣
↪y_pred_with_all_features)
```

```
    # Create a dictionary to store the feature names and their␣
↪corresponding accuracy differences
    feature_impact = {}

    # Create an empty list to store features with a small accuracy␣
↪difference
    features_to_remove = []

    # Iterate over each feature and compare the model performance␣
↪with and without the feature
    for feature in X.columns:
        # Train the Naive Bayes model without the current feature
        X_train_without_feature = X_train.drop(feature, axis=1)
        X_test_without_feature = X_test.drop(feature, axis=1)

        naive_bayes_model_without_feature = GaussianNB()
        naive_bayes_model_without_feature.
↪fit(X_train_without_feature, y_train)

        # Get the accuracy without the feature
        y_pred_without_feature = naive_bayes_model_without_feature.
↪predict(X_test_without_feature)
        accuracy_without_feature = accuracy_score(y_test,␣
↪y_pred_without_feature)

        # Calculate the accuracy difference compared to the baseline
        feature_impact[feature] = baseline_accuracy -␣
↪accuracy_without_feature

        # Check if the accuracy difference is smaller than -0.005 and␣
↪add the feature to the list
        if feature_impact[feature] <= -0.005:
            features_to_remove.append(feature)
    features_to_remove.sort(key=lambda x: feature_impact[x])


    # Create a bar plot showing the impact of each feature on the␣
↪model's performance
    plt.figure(figsize=(12, 6))
    plt.bar(feature_impact.keys(), feature_impact.values())
    plt.xlabel('Feature')
    plt.ylabel('Accuracy Difference')
    plt.title('Impact of Features on Naive Bayes Model Performance')
    plt.xticks(rotation=45)
    plt.show()
```

```
    # Print the list of features with accuracy difference smaller␣
↪than -0.005
    print("Features to Remove:")
    print(features_to_remove)



compare_feature_impact(df_train, 'inadimplente')
```

[ ]:
```
import matplotlib.pyplot as plt

def generate_bar_charts(df):
    # Cria um dicionário para armazenar os gráficos de barras de cada␣
↪coluna
    bar_charts = {}

    # Itera sobre todas as colunas do DataFrame
    for column in df.columns:
        # Verifica o tipo da coluna (objeto é tratado como categórico)

        # Conta o número de ocorrências de cada valor na coluna
        value_counts = df[column].value_counts()

        # Cria um gráfico de barras
        plt.figure(figsize=(8, 6))
        plt.bar(value_counts.index, value_counts.values)
        plt.xlabel(column)
        plt.ylabel('Quantidade de Ocorrências')
        plt.title(f'Contagem de Ocorrências para a Coluna "{column}"')
        plt.xticks(rotation=45)
        plt.show()

        # Adiciona o gráfico ao dicionário
        bar_charts[column] = plt

    return bar_charts

generate_bar_charts(df_train)
```

1.7 Resultados

Seção utilizada para salvar os resultados num arquivo “predictions.csv”.

[ ]:
```
df_test = pd.read_csv('Dataset/conjunto_de_teste.csv')

prediction_file = pd.DataFrame(predictions, columns=['inadimplente'])
```

```
prediction_file = pd.concat([df_test['id_solicitante'],␣
↪prediction_file], axis=1)
prediction_file = prediction_file.to_csv('results/predictions.csv',␣
↪index=False)


prediction_file = pd.read_csv('results/predictions.csv')
prediction_file.shape
```