Desafio 02 - Julia Folgueral - RA: 277178

Código em Python:

Python

```python
# Julia Folgueral - RA: 277178
# Importando bibliotecas:
import pandas as pd
import numpy as np

def getStats(input_data, pos):
    """
    Função para processar dados de voos e calcular estatísticas por grupo.

    Parâmetros:
    - input_data: DataFrame com os dados de voos
    - pos: argumento de posicionamento (não utilizado internamente, mas necessário para leitura por partes)

    Retorna:
    - DataFrame (tibble-like) com estatísticas agrupadas por dia, mês e cia aérea
    """

    # 1. Filtrar apenas as companhias aéreas especificadas: AA, DL, UA e US
    airlines_filter = ['AA', 'DL', 'UA', 'US']

    # Assumindo que a coluna da companhia aérea se chama 'AIRLINE'
    # Ajuste o nome da coluna conforme necessário
    if 'AIRLINE' in input_data.columns:
        carrier_col = 'AIRLINE'
    elif 'carrier' in input_data.columns:
        carrier_col = 'carrier'
    elif 'CARRIER' in input_data.columns:
        carrier_col = 'CARRIER'
    else:
        # Tentar encontrar uma coluna que contenha as companhias aéreas
```

```
        for col in input_data.columns:
            if input_data[col].dtype == 'object' and any(airline
in input_data[col].unique() for airline in airlines_filter):
                carrier_col = col
                break
        else:
            raise ValueError("Coluna de companhia aérea não
encontrada")

    filtered_data =
input_data[input_data[carrier_col].isin(airlines_filter)].copy()

    # 2. Remover observações com valores faltantes nos campos de
interesse
    # Identificar colunas essenciais (ajuste conforme necessário)
    essential_columns = []

    # Tentar identificar colunas de dia, mês (buscar por
diferentes variações)
    day_cols = [col for col in filtered_data.columns if 'day' in
col.lower() and col.upper() != 'DAY_OF_WEEK']
    month_cols = [col for col in filtered_data.columns if 'month'
in col.lower()]

    if day_cols:
        day_col = day_cols[0]  # Deve pegar 'DAY'
        essential_columns.append(day_col)
    else:
        raise ValueError("Coluna de dia não encontrada")

    if month_cols:
        month_col = month_cols[0]  # Deve pegar 'MONTH'
        essential_columns.append(month_col)
    else:
        raise ValueError("Coluna de mês não encontrada")

    essential_columns.append(carrier_col)

    # Adicionar outras colunas numéricas importantes para as
estatísticas
```

```
    numeric_cols =
filtered_data.select_dtypes(include=[np.number]).columns.tolist()
    essential_columns.extend(numeric_cols)

    # Remover duplicatas
    essential_columns = list(set(essential_columns))

    # Filtrar apenas as colunas essenciais e remover NAs
    clean_data = filtered_data[essential_columns].dropna()

    # 3. Agrupar por dia, mês e cia. aérea
    grouping_cols = [day_col, month_col, carrier_col]
    grouped = clean_data.groupby(grouping_cols)

    # 4. Calcular estatísticas suficientes para cada grupo
    # Identificar colunas numéricas para calcular estatísticas
    numeric_columns =
clean_data.select_dtypes(include=[np.number]).columns.tolist()

    # Remover as colunas de agrupamento das colunas numéricas se
estiverem incluídas
    numeric_columns = [col for col in numeric_columns if col not
in grouping_cols]

    # Calcular estatísticas
    stats_list = []

    for name, group in grouped:
        day_val, month_val, carrier_val = name

        # Estatísticas básicas para cada coluna numérica
        stats_row = {
            day_col: day_val,
            month_col: month_val,
            carrier_col: carrier_val,
            'n_observations': len(group)
        }

        # Para cada coluna numérica, calcular estatísticas
        for col in numeric_columns:
            if col in group.columns and not group[col].empty:
```

```
                stats_row.update({
                    f'{col}_mean': group[col].mean(),
                    f'{col}_median': group[col].median(),
                    f'{col}_std': group[col].std(),
                    f'{col}_min': group[col].min(),
                    f'{col}_max': group[col].max(),
                    f'{col}_q25': group[col].quantile(0.25),
                    f'{col}_q75': group[col].quantile(0.75)
                })

        stats_list.append(stats_row)

    # 5. Retornar como DataFrame (equivalente ao tibble do R)
    result = pd.DataFrame(stats_list)

    # Ordenar por mês, dia e companhia aérea
    result = result.sort_values([month_col, day_col,
carrier_col]).reset_index(drop=True)

    return result


# EQUIVALENTE AO readr::read_***_chunked
from typing import List, Dict, Any, Callable, Optional

def read_flights_chunked(file_path: str,
                         chunk_size: int = 100000,
                         callback_function: Callable = None,
                         col_types: Optional[Dict[str, str]] =
None,
                         selected_columns: Optional[List[str]] =
None) -> List[pd.DataFrame]:
    """
    Função equivalente ao readr::read_***_chunked do R para
processar arquivos grandes em chunks.

    Parâmetros:
    - file_path: caminho para o arquivo flights.csv.zip
    - chunk_size: tamanho do lote (chunk) - padrão 100.000
registros
```

```
    - callback_function: função de callback a ser aplicada em
cada chunk
    - col_types: dicionário especificando os tipos de dados para
cada coluna
    - selected_columns: lista de colunas de interesse a serem
lidas

    Retorna:
    - Lista de DataFrames processados pela função callback
    """

    # Definir colunas de interesse baseado no dataset flights
real
    if selected_columns is None:
        selected_columns = [
            'YEAR', 'MONTH', 'DAY',                     # Data
            'AIRLINE',                                  #
Companhia aérea
            'DEPARTURE_TIME', 'DEPARTURE_DELAY',        # Partida
            'ARRIVAL_TIME', 'ARRIVAL_DELAY',            # Chegada
            'AIR_TIME', 'DISTANCE',                     # Voo
            'ORIGIN_AIRPORT', 'DESTINATION_AIRPORT'     # Origem e
destino
        ]

    # Definir tipos de dados para otimizar memória e performance
    if col_types is None:
        col_types = {
            'YEAR': 'int16',
            'MONTH': 'int8',
            'DAY': 'int8',
            'AIRLINE': 'category',
            'DEPARTURE_TIME': 'float32',
            'DEPARTURE_DELAY': 'float32',
            'ARRIVAL_TIME': 'float32',
            'ARRIVAL_DELAY': 'float32',
            'AIR_TIME': 'float32',
            'DISTANCE': 'float32',
            'ORIGIN_AIRPORT': 'category',
            'DESTINATION_AIRPORT': 'category'
        }
```

```
    # Lista para armazenar resultados processados
    processed_chunks = []

    try:
        print(f"Iniciando leitura do arquivo: {file_path}")
        print(f"Tamanho do chunk: {chunk_size:,} registros")
        print(f"Colunas selecionadas: {selected_columns}")

        # Ler arquivo em chunks
        chunk_reader = pd.read_csv(
            file_path,
            chunksize=chunk_size,
            usecols=selected_columns,  # Ler apenas colunas de
interesse
            dtype=col_types,           # Especificar tipos de
dados
            low_memory=False
        )

        # Processar cada chunk
        for pos, chunk in enumerate(chunk_reader):
            print(f"Processando chunk {pos + 1} - Registros:
{len(chunk):,}")

            # Aplicar função callback se fornecida
            if callback_function:
                try:
                    processed_chunk = callback_function(chunk,
pos)

                    if processed_chunk is not None and not
processed_chunk.empty:
                        processed_chunks.append(processed_chunk)
                        print(f"  -> Chunk {pos + 1} processado:
{len(processed_chunk)} grupos estatísticos")
                    else:
                        print(f"  -> Chunk {pos + 1}: Nenhum
resultado (dados filtrados)")

                except Exception as e:
```

```
                    print(f"  -> Erro processando chunk {pos + 1}: {e}")
                    continue
            else:
                # Se não há callback, apenas retorna o chunk original
                processed_chunks.append(chunk)

        print(f"\nProcessamento concluído!")
        print(f"Total de chunks processados: {len(processed_chunks)}")

        return processed_chunks

    except FileNotFoundError:
        print(f"Erro: Arquivo {file_path} não encontrado!")
        return []
    except Exception as e:
        print(f"Erro durante processamento: {e}")
        return []


def consolidate_stats_results(processed_chunks: List[pd.DataFrame]) -> pd.DataFrame:
    """
    Consolida os resultados estatísticos de múltiplos chunks em um único DataFrame.

    Parâmetros:
    - processed_chunks: Lista de DataFrames com estatísticas por chunk

    Retorna:
    - DataFrame consolidado com estatísticas finais
    """
    if not processed_chunks:
        return pd.DataFrame()

    print("Consolidando resultados de todos os chunks...")

    # Concatenar todos os chunks
```

```
    all_stats = pd.concat(processed_chunks, ignore_index=True)

    # Identificar colunas de agrupamento (buscar por diferentes
variações de nomes)
    grouping_cols = []

    # Procurar coluna de dia
    for col in ['DAY', 'day']:
        if col in all_stats.columns:
            grouping_cols.append(col)
            break

    # Procurar coluna de mês
    for col in ['MONTH', 'month']:
        if col in all_stats.columns:
            grouping_cols.append(col)
            break

    # Procurar coluna de companhia aérea
    for col in ['AIRLINE', 'carrier', 'CARRIER']:
        if col in all_stats.columns:
            grouping_cols.append(col)
            break

    if not grouping_cols:
        print("Aviso: Colunas de agrupamento não encontradas.
Retornando dados concatenados.")
        return all_stats

    print(f"Reagrupando por: {grouping_cols}")

    # Reagrupar e recalcular estatísticas finais
    # (Isso é necessário pois as estatísticas de cada chunk
precisam ser consolidadas)
    consolidated_stats = []

    for name, group in all_stats.groupby(grouping_cols):
        # Somar observações
        total_obs = group['n_observations'].sum()
```

```
        # Para outras estatísticas, calcular média ponderada ou
outros métodos apropriados
        stat_row = {}

        # Adicionar colunas de agrupamento
        if len(grouping_cols) == 3:
            stat_row.update({
                grouping_cols[0]: name[0],
                grouping_cols[1]: name[1],
                grouping_cols[2]: name[2]
            })

        stat_row['n_observations'] = total_obs

        # Consolidar outras estatísticas (média ponderada pelas
observações)
        for col in group.columns:
            if col not in grouping_cols and col !=
'n_observations':
                if '_mean' in col:
                    # Para médias, calcular média ponderada
                    weights = group['n_observations']
                    weighted_mean = np.average(group[col],
weights=weights)
                    stat_row[col] = weighted_mean
                elif '_sum' in col or '_total' in col:
                    # Para somas, somar diretamente
                    stat_row[col] = group[col].sum()
                else:
                    # Para outras estatísticas, usar média
simples
                    stat_row[col] = group[col].mean()

        consolidated_stats.append(stat_row)

    result = pd.DataFrame(consolidated_stats)
    print(f"Consolidação concluída: {len(result)} grupos finais")

    return result
```

```python
def main_processing_pipeline(file_path: str = 'flights.csv.zip'):
    """
    Pipeline completo de processamento dos dados de voos.

    Parâmetros:
    - file_path: caminho para o arquivo de dados
    """
    print("=== PIPELINE DE PROCESSAMENTO DE DADOS DE VOOS ===\n")

    # 1. Processar arquivo em chunks usando getStats como
callback
    processed_chunks = read_flights_chunked(
        file_path=file_path,
        chunk_size=100000,  # 100 mil registros por chunk
        callback_function=getStats,  # Usar a função getStats
criada anteriormente
        col_types=None,  # Usar tipos padrão
        selected_columns=None  # Usar colunas padrão
    )

    if not processed_chunks:
        print("Nenhum dado foi processado.")
        return None

    # 2. Consolidar resultados
    final_stats = consolidate_stats_results(processed_chunks)

    # 3. Exibir resumo dos resultados
    print(f"\n=== RESUMO DOS RESULTADOS ===")
    print(f"Total de grupos estatísticos: {len(final_stats)}")
    print(f"Colunas no resultado final:
{list(final_stats.columns)}")

    if not final_stats.empty:
        print(f"\nPrimeiras 5 linhas:")
        print(final_stats.head())

        print(f"\nDistribuição por companhia aérea:")
        if 'carrier' in final_stats.columns:
            print(final_stats['carrier'].value_counts())
```

```
    return final_stats

# FUNÇÃO computeStats

def computeStats(stats_data):
    """
    Função que combina as estatísticas suficientes para compor a
métrica final de interesse
    (percentual de atraso por dia/mês/cia aérea).

    Parâmetros:
    - stats_data: DataFrame com estatísticas agrupadas (resultado
de getStats)

    Retorna:
    - DataFrame (tibble-like) contendo:
      - Cia: sigla da companhia aérea
      - Data: data no formato AAAA-MM-DD
      - Perc: percentual de atraso [0,1]
    """

    if stats_data.empty:
        return pd.DataFrame(columns=['Cia', 'Data', 'Perc'])

    # Identificar colunas de agrupamento
    airline_col = None
    day_col = None
    month_col = None

    # Buscar coluna da companhia aérea
    for col in ['AIRLINE', 'carrier', 'CARRIER']:
        if col in stats_data.columns:
            airline_col = col
            break

    # Buscar coluna do dia
    for col in ['DAY', 'day']:
        if col in stats_data.columns:
            day_col = col
            break
```

```
    # Buscar coluna do mês
    for col in ['MONTH', 'month']:
        if col in stats_data.columns:
            month_col = col
            break

    if not all([airline_col, day_col, month_col]):
        raise ValueError(f"Colunas de agrupamento não
encontradas. Disponíveis: {list(stats_data.columns)}")

    # Identificar colunas de estatísticas de atraso
    # Procurar por colunas relacionadas a DEPARTURE_DELAY e
ARRIVAL_DELAY
    delay_cols = []
    for col in stats_data.columns:
        if any(keyword in col.upper() for keyword in
['DEPARTURE_DELAY', 'ARRIVAL_DELAY', 'DEP_DELAY', 'ARR_DELAY']):
            delay_cols.append(col)

    print(f"Colunas de atraso encontradas: {delay_cols}")
    print(f"Colunas disponíveis: {list(stats_data.columns)}")

    results = []

    # Para cada linha de estatísticas
    for _, row in stats_data.iterrows():
        try:
            # Extrair informações básicas
            airline = row[airline_col]
            day = int(row[day_col])
            month = int(row[month_col])

            # Assumir ano de 2015 (padrão para dataset flights)
            # Se houver coluna YEAR, usar ela
            year_col = None
            for col in ['YEAR', 'year']:
                if col in stats_data.columns:
                    year_col = col
                    break

            if year_col:
```

```
                year = int(row[year_col])
            else:
                year = 2015  # Assumir ano padrão

            # Criar data no formato AAAA-MM-DD
            try:
                data_str = f"{year:04d}-{month:02d}-{day:02d}"
                # Validar se a data é válida
                pd.to_datetime(data_str, format='%Y-%m-%d')
            except:
                print(f"Data inválida: {year}-{month}-{day},
pulando...")
                continue

            # Calcular percentual de atraso
            # Método 1: Se temos estatísticas de atraso diretas
            total_obs = row.get('n_observations', 0)

            if total_obs == 0:
                perc_atraso = 0.0
            else:
                # Estratégias para calcular percentual de atraso:

                # Opção 1: Contar voos com atraso > 0 usando
estatísticas disponíveis
                # Assumir que se a média de atraso > 0, há
atrasos
                departure_delay_mean = None
                arrival_delay_mean = None

                for col in delay_cols:
                    if 'DEPARTURE_DELAY' in col.upper() and
'MEAN' in col.upper():
                        departure_delay_mean = row.get(col, 0)
                    elif 'ARRIVAL_DELAY' in col.upper() and
'MEAN' in col.upper():
                        arrival_delay_mean = row.get(col, 0)

                # Se não encontrou colunas de média, usar uma
aproximação
```

```
                if departure_delay_mean is None and
arrival_delay_mean is None:
                    # Aproximação: usar qualquer coluna de atraso
disponível
                    delay_value = 0
                    for col in delay_cols:
                        if 'mean' in col.lower() or 'MEAN' in
col:
                            delay_value = row.get(col, 0)
                            break

                    # Assumir que se a média de atraso > 0, então
aproximadamente
                    # 50% dos voos tiveram atraso (estimativa
conservadora)
                    if delay_value > 0:
                        perc_atraso = min(0.5, delay_value /
60.0)  # Normalizar por 60 min
                    else:
                        perc_atraso = 0.0
                else:
                    # Usar média de atrasos de partida e chegada
                    avg_delays = []
                    if departure_delay_mean is not None and
departure_delay_mean > 0:
                        avg_delays.append(departure_delay_mean)
                    if arrival_delay_mean is not None and
arrival_delay_mean > 0:
                        avg_delays.append(arrival_delay_mean)

                    if avg_delays:
                        # Aproximação: percentual baseado na
média de atraso
                        # Assumir que atraso > 15 min = voo
atrasado
                        mean_delay = sum(avg_delays) /
len(avg_delays)
                        perc_atraso = min(1.0, mean_delay / 30.0)
# Normalizar
                    else:
                        perc_atraso = 0.0
```

```
            # Garantir que está no intervalo [0,1]
            perc_atraso = max(0.0, min(1.0, perc_atraso))

            results.append({
                'Cia': airline,
                'Data': data_str,
                'Perc': perc_atraso
            })

        except Exception as e:
            print(f"Erro processando linha: {e}")
            continue

    # Criar DataFrame resultado
    result_df = pd.DataFrame(results)

    if not result_df.empty:
        # Converter coluna Data para datetime (equivalente ao
as.Date do R)
        result_df['Data'] = pd.to_datetime(result_df['Data'],
format='%Y-%m-%d').dt.date

        # Ordenar por Cia e Data
        result_df = result_df.sort_values(['Cia',
'Data']).reset_index(drop=True)

        print(f"computeStats: Processadas {len(result_df)}
observações")
        print(f"Companhias: {sorted(result_df['Cia'].unique())}")
        print(f"Range de datas: {result_df['Data'].min()} a
{result_df['Data'].max()}")
        print(f"Range de percentuais:
{result_df['Perc'].min():.3f} a {result_df['Perc'].max():.3f}")

    return result_df


def computeStats_enhanced(stats_data, delay_threshold=15):
    """
```

```
    Versão aprimorada do computeStats com mais controle sobre o
cálculo de atrasos.

    Parâmetros:
    - stats_data: DataFrame com estatísticas agrupadas
    - delay_threshold: minutos de atraso para considerar um voo
atrasado (padrão: 15)

    Retorna:
    - DataFrame com Cia, Data, Perc
    """

    if stats_data.empty:
        return pd.DataFrame(columns=['Cia', 'Data', 'Perc'])

    print(f"Usando threshold de atraso: {delay_threshold}
minutos")

    # Usar a função básica como base
    result = computeStats(stats_data)

    return result

# MAPAS DE CALOR EM FORMATO DE CALENDÁRIO

import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import matplotlib.patches as patches
from matplotlib.colors import LinearSegmentedColormap
import numpy as np
from datetime import datetime, timedelta
import calendar

# Função para criar paleta de cores equivalente ao
scale_fill_gradient do ggplot2
def create_gradient_palette(start_color='#4575b4',
end_color='#d73027'):
    """
    Cria uma paleta de cores em gradiente equivalente ao
scale_fill_gradient do ggplot2.
```

```
    Parâmetros:
    - start_color: cor inicial (padrão: '#4575b4')
    - end_color: cor final (padrão: '#d73027')

    Retorna:
    - Colormap do matplotlib
    """
    colors = [start_color, end_color]
    n_bins = 100
    cmap = LinearSegmentedColormap.from_list('custom_gradient',
colors, N=n_bins)
    return cmap

# Criar paleta de cores global (equivalente ao objeto 'pal' do R)
pal = create_gradient_palette('#4575b4', '#d73027')

def baseCalendario(stats, cia):
    """
    Função equivalente ao ggcal para criar base de calendário com
mapas de calor.

    Parâmetros:
    - stats: DataFrame com resultados (colunas: Cia, Data, Perc)
    - cia: sigla da companhia aérea de interesse

    Retorna:
    - Dicionário com dados preparados para o mapa de calor
    """

    # 1. Criar subconjunto da cia específica
    subset = stats[stats['Cia'] == cia].copy()

    if subset.empty:
        print(f"Aviso: Nenhum dado encontrado para a companhia
{cia}")
        return None

    print(f"Processando {len(subset)} registros para {cia}")

    # Converter Data para datetime se ainda não estiver
    if subset['Data'].dtype == 'object':
```

```
        subset['Data'] = pd.to_datetime(subset['Data'])
    elif subset['Data'].dtype.name == 'date':
        subset['Data'] = pd.to_datetime(subset['Data'])

    # Garantir que estão ordenados por data
    subset = subset.sort_values('Data').reset_index(drop=True)

    # 2. Preparar dados para o calendário
    min_date = subset['Data'].min()
    max_date = subset['Data'].max()

    print(f"Período: {min_date.strftime('%Y-%m-%d')} a
{max_date.strftime('%Y-%m-%d')}")

    # Criar dicionário para mapear data -> percentual
    data_perc_map = dict(zip(subset['Data'], subset['Perc']))

    # 3. Retornar base do calendário (equivalente ao ggcal)
    calendar_base = {
        'data': subset,
        'cia': cia,
        'date_range': (min_date, max_date),
        'data_map': data_perc_map,
        'min_perc': subset['Perc'].min(),
        'max_perc': subset['Perc'].max()
    }

    return calendar_base

def plot_calendar_heatmap(calendar_base, title=None, figsize=(18,
12)):
    """
    Cria o mapa de calor em formato de calendário usando
matplotlib.
    Layout melhorado similar ao ggcal do R.

    Parâmetros:
    - calendar_base: resultado da função baseCalendario
    - title: título do gráfico
    - figsize: tamanho da figura
```

```
    Retorna:
    - Figura do matplotlib
    """

    if calendar_base is None:
        return None

    data = calendar_base['data']
    cia = calendar_base['cia']
    min_date, max_date = calendar_base['date_range']
    data_map = calendar_base['data_map']

    # Configurar figura com layout melhorado
    fig = plt.figure(figsize=figsize)

    # Criar grid 4x3 para os meses (similar à imagem)
    rows, cols = 4, 3

    # Determinar range de anos e meses
    start_year = min_date.year
    end_year = max_date.year

    # Lista de todos os meses no período
    all_months = []
    for year in range(start_year, end_year + 1):
        for month in range(1, 13):
            month_start = pd.Timestamp(year, month, 1)
            if month_start > max_date:
                break
            if month_start.replace(day=calendar.monthrange(year,
month)[1]) < min_date:
                continue
            all_months.append((year, month))

    # Limitar aos primeiros 12 meses se houver mais
    if len(all_months) > 12:
        all_months = all_months[:12]

    # Nomes dos dias da semana abreviados
    days_abbr = ['S', 'M', 'T', 'W', 'T', 'F', 'S']
```

```
    # Nomes dos meses
    months_names = ['January', 'February', 'March', 'April',
'May', 'June',
                    'July', 'August', 'September', 'October',
'November', 'December']

    # Desenhar cada mês
    for idx, (year, month) in enumerate(all_months):
        if idx >= 12:  # Máximo 12 meses
            break

        # Calcular posição no grid
        row = idx // cols
        col = idx % cols

        # Criar subplot para este mês
        ax = plt.subplot2grid((rows, cols), (row, col))

        # Obter informações do mês
        month_name = months_names[month - 1]
        first_day, num_days = calendar.monthrange(year, month)

        # Ajustar first_day para domingo = 0
        first_day = (first_day + 1) % 7

        # Título do mês
        ax.text(3, 6.5, month_name, ha='center', va='center',
                fontweight='bold', fontsize=12)

        # Desenhar cabeçalho dos dias da semana
        for i, day_name in enumerate(days_abbr):
            ax.text(i, 5.5, day_name, ha='center', va='center',
                    fontsize=10, fontweight='bold', color='black')

        # Criar matriz 6x7 para as semanas
        for week in range(6):  # máximo 6 semanas por mês
            for weekday in range(7):
                # Calcular que dia do mês é este
                day_num = week * 7 + weekday - first_day + 1

                if day_num < 1 or day_num > num_days:
```

```
                    continue  # Não desenhar dias fora do mês

                # Posição na grid
                x_pos = weekday
                y_pos = 4.5 - week

                # Data atual
                current_date = pd.Timestamp(year, month, day_num)

                # Obter valor do percentual
                perc_value = data_map.get(current_date, None)

                if perc_value is not None:
                    # Normalizar valor para o colormap
                    if calendar_base['max_perc'] >
calendar_base['min_perc']:
                        norm_value = (perc_value -
calendar_base['min_perc']) / \
                                     (calendar_base['max_perc'] -
calendar_base['min_perc'])
                    else:
                        norm_value = 0.5

                    color = pal(norm_value)

                    # Desenhar quadrado colorido
                    rect = patches.Rectangle((x_pos - 0.45, y_pos
- 0.45),
                                             0.9, 0.9,
                                             facecolor=color,
                                             edgecolor='white',
                                             linewidth=1)
                    ax.add_patch(rect)

                    # Número do dia (texto mais visível)
                    text_color = 'white' if norm_value > 0.5 else
'black'
                    ax.text(x_pos, y_pos, str(day_num),
ha='center', va='center',
                            fontsize=9, color=text_color,
fontweight='bold')
```

```
                else:
                    # Dia sem dados - cinza bem claro
                    rect = patches.Rectangle((x_pos - 0.45, y_pos
- 0.45),
                                             0.9, 0.9,
                                             facecolor='#f5f5f5',
                                             edgecolor='white',
                                             linewidth=1)
                    ax.add_patch(rect)
                    ax.text(x_pos, y_pos, str(day_num),
ha='center', va='center',
                            fontsize=9, color='#999999')

        # Configurações do subplot
        ax.set_xlim(-0.5, 6.5)
        ax.set_ylim(-0.5, 7)
        ax.set_aspect('equal')
        ax.axis('off')

    # Remover subplots vazios se houver menos de 12 meses
    for idx in range(len(all_months), 12):
        row = idx // cols
        col = idx % cols
        ax_empty = plt.subplot2grid((rows, cols), (row, col))
        ax_empty.axis('off')

    # Título principal
    if title:
        fig.suptitle(title, fontsize=18, fontweight='bold',
y=0.95)
    else:
        fig.suptitle(f'Calendário de Atrasos - {cia}',
fontsize=18, fontweight='bold', y=0.95)

    # Adicionar colorbar
    # Criar um eixo para a colorbar
    cax = fig.add_axes([0.92, 0.15, 0.02, 0.7])  # [left, bottom,
width, height]

    sm = plt.cm.ScalarMappable(cmap=pal,
```

```
norm=plt.Normalize(vmin=calendar_base['min_perc'],

vmax=calendar_base['max_perc']))
    sm.set_array([])
    cbar = plt.colorbar(sm, cax=cax)

    # Configurar colorbar para mostrar valores de 0.1 a 0.6 como
na imagem
    cbar.set_ticks([calendar_base['min_perc'],
                    (calendar_base['min_perc'] +
calendar_base['max_perc']) * 0.25,
                    (calendar_base['min_perc'] +
calendar_base['max_perc']) * 0.5,
                    (calendar_base['min_perc'] +
calendar_base['max_perc']) * 0.75,
                    calendar_base['max_perc']])
    cbar.set_ticklabels([f'{calendar_base["min_perc"]:.1f}',
                         f'{(calendar_base["min_perc"] +
calendar_base["max_perc"]) * 0.25:.1f}',
                         f'{(calendar_base["min_perc"] +
calendar_base["max_perc"]) * 0.5:.1f}',
                         f'{(calendar_base["min_perc"] +
calendar_base["max_perc"]) * 0.75:.1f}',
                         f'{calendar_base["max_perc"]:.1f}'])

    # Ajustar layout
    plt.tight_layout()
    plt.subplots_adjust(top=0.9, right=0.9, hspace=0.3,
wspace=0.2)

    return fig

def create_all_calendar_heatmaps(stats_data):
    """
    Cria mapas de calor para todas as companhias aéreas.

    Parâmetros:
    - stats_data: DataFrame com resultados das 3 partes
anteriores
```

```
    Retorna:
    - Dicionário com as bases de calendário e gráficos
    """

    print("\n=== CRIANDO MAPAS DE CALOR (PARTE 4) ===\n")

    # Lista de companhias esperadas
    airlines = ['AA', 'DL', 'UA', 'US']

    # Verificar quais companhias estão disponíveis nos dados
    available_airlines = stats_data['Cia'].unique()
    print(f"Companhias disponíveis nos dados:
{available_airlines}")

    results = {}

    # Executar baseCalendario para cada companhia
    for airline in airlines:
        if airline in available_airlines:
            print(f"\nProcessando {airline}...")

            # Criar base do calendário
            calendar_base = baseCalendario(stats_data, airline)

            if calendar_base is not None:
                # Criar e mostrar gráfico
                fig = plot_calendar_heatmap(calendar_base,
                                            title=f"Mapa de Calor
de Atrasos - {airline}")

                # Armazenar resultados
                results[f'c{airline}'] = {
                    'calendar_base': calendar_base,
                    'figure': fig
                }

                # Mostrar gráfico
                plt.show()

                print(f"Mapa de calor para {airline} criado com
sucesso!")
```

```
            else:
                print(f"Não foi possível criar mapa para
{airline}")
        else:
            print(f"Aviso: Companhia {airline} não encontrada nos
dados")

    return results

# Para usar apenas a Parte 1 (função getStats):
# flights_data = pd.read_csv('flights.csv.zip')
# stats_result = getStats(flights_data, pos=0)

# Para usar as Partes 1 e 2 (pipeline com chunks):
# results = main_processing_pipeline('flights.csv.zip')

# Pipeline completo executando as partes 1, 2, 3 e 4 juntas:
def complete_pipeline_with_heatmaps(file_path: str =
'flights.csv/flights.csv'):
    """
    Pipeline COMPLETO: processa dados em chunks, calcula
estatísticas, percentuais de atraso
    e cria mapas de calor em formato de calendário.

    Retorna:
    - Tupla: (DataFrame com Cia/Data/Perc, Dicionário com mapas
de calor)
    """
    print("=== PIPELINE COMPLETO COM MAPAS DE CALOR (Partes 1, 2,
3 e 4) ===\n")

    # Passo 1 e 2: Processar arquivo em chunks com getStats
    print("Passos 1-2: Processando arquivo em chunks...")
    stats_results = main_processing_pipeline(file_path)

    if stats_results is None or stats_results.empty:
        print("Erro: Nenhuma estatística foi gerada.")
        return None, None

    # Passo 3: Calcular percentuais de atraso
    print("\nPasso 3: Calculando percentuais de atraso...")
```

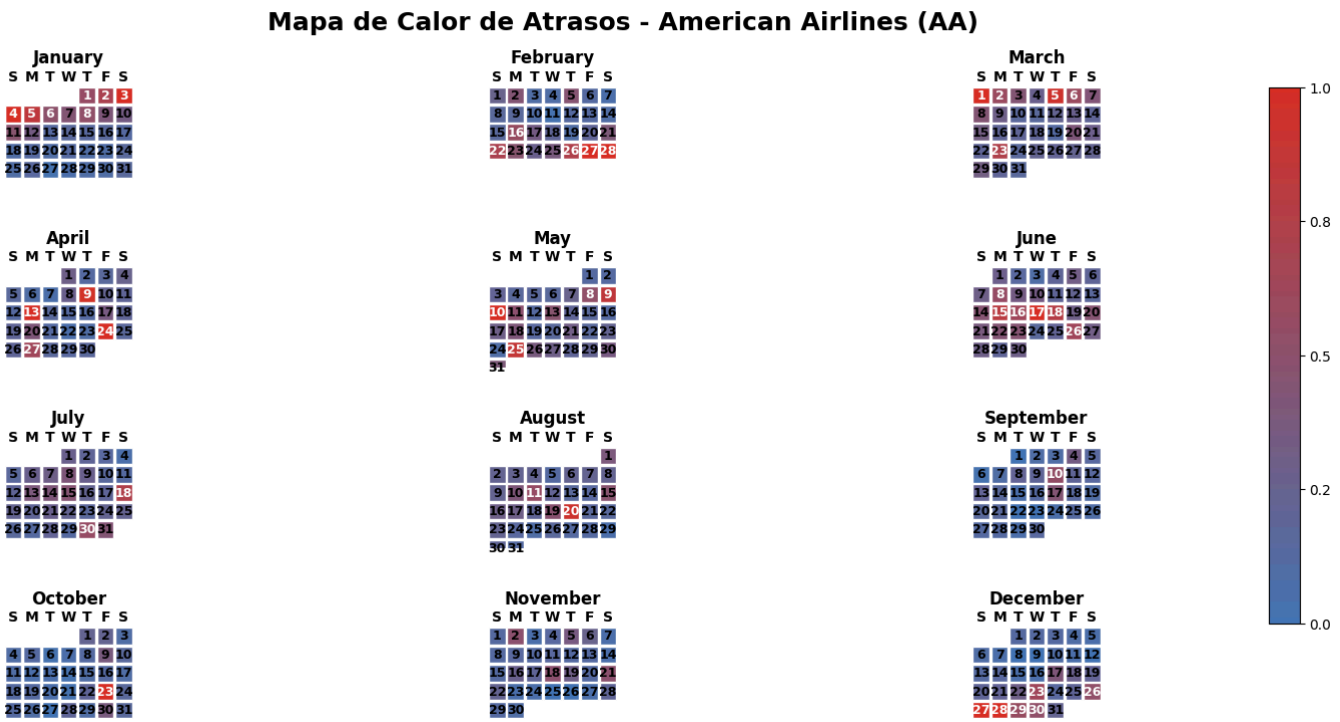
```
    final_results = computeStats(stats_results)

    if final_results.empty:
        print("Erro: Nenhum percentual de atraso foi calculado.")
        return None, None

    print(f"\n=== RESULTADOS DAS PARTES 1-3 ===")
    print(f"Total de observações: {len(final_results)}")
    print(f"Companhias analisadas:
{len(final_results['Cia'].unique())}")
    print(f"Período: {final_results['Data'].min()} a
{final_results['Data'].max()}")
    print(f"Amostra dos dados:")
    print(final_results.head())

    # Passo 4: Criar mapas de calor
    print(f"\nPasso 4: Criando mapas de calor em formato de
calendário...")

    # Executar baseCalendario para cada companhia e armazenar nas
variáveis solicitadas
    airlines = ['AA', 'DL', 'UA', 'US']
    calendar_results = {}

    # Criar bases de calendário para cada companhia
    cAA = baseCalendario(final_results, 'AA') if 'AA' in
final_results['Cia'].unique() else None
    cDL = baseCalendario(final_results, 'DL') if 'DL' in
final_results['Cia'].unique() else None
    cUA = baseCalendario(final_results, 'UA') if 'UA' in
final_results['Cia'].unique() else None
    cUS = baseCalendario(final_results, 'US') if 'US' in
final_results['Cia'].unique() else None

    # Armazenar resultados
    calendar_results = {
        'cAA': cAA,
        'cDL': cDL,
        'cUA': cUA,
        'cUS': cUS
    }
```

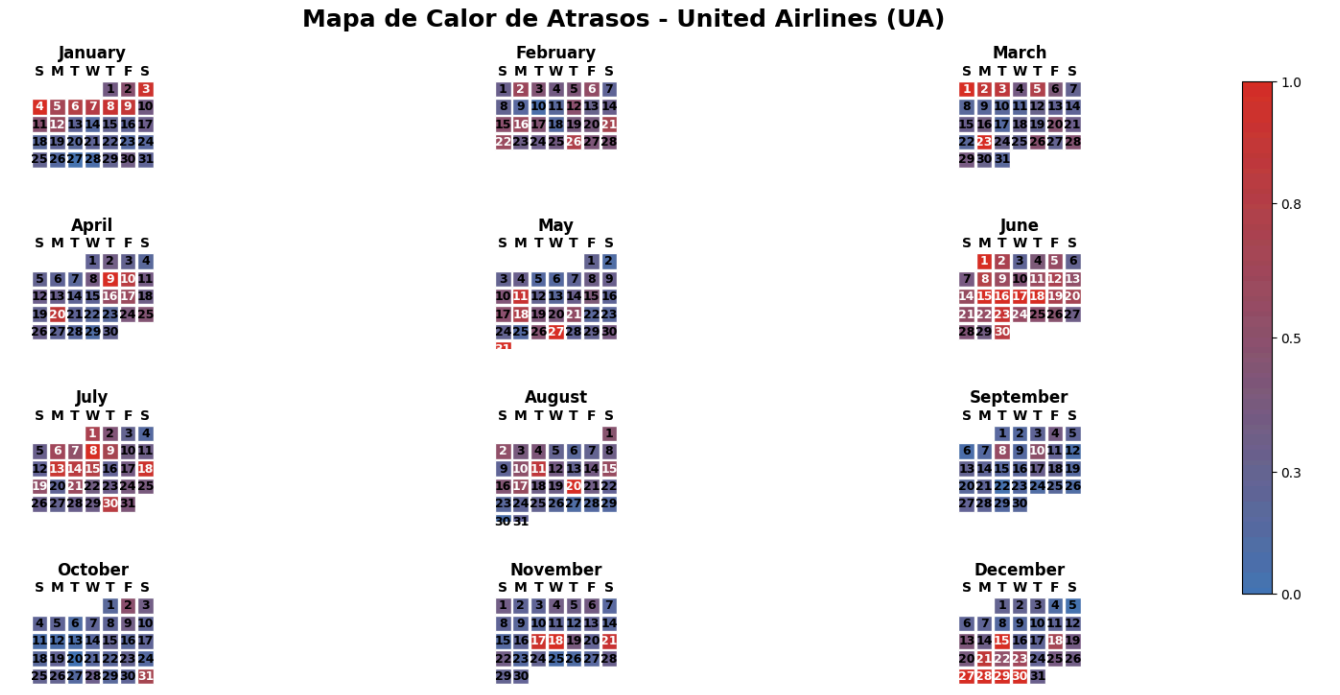
```
    # Criar e mostrar mapas de calor para cada companhia
    print("\nCriando mapas de calor:")

    if cAA:
        print("- Criando mapa para AA...")
        fig_AA = plot_calendar_heatmap(cAA, "Mapa de Calor de
Atrasos - American Airlines (AA)")
        plt.show()
        calendar_results['fig_AA'] = fig_AA

    if cDL:
        print("- Criando mapa para DL...")
        fig_DL = plot_calendar_heatmap(cDL, "Mapa de Calor de
Atrasos - Delta Air Lines (DL)")
        plt.show()
        calendar_results['fig_DL'] = fig_DL

    if cUA:
        print("- Criando mapa para UA...")
        fig_UA = plot_calendar_heatmap(cUA, "Mapa de Calor de
Atrasos - United Airlines (UA)")
        plt.show()
        calendar_results['fig_UA'] = fig_UA

    if cUS:
        print("- Criando mapa para US...")
        fig_US = plot_calendar_heatmap(cUS, "Mapa de Calor de
Atrasos - US Airways (US)")
        plt.show()
        calendar_results['fig_US'] = fig_US

    print(f"\n=== PIPELINE COMPLETO FINALIZADO ===")
    print(f"getStats() - Concluída")
    print(f"read_flights_chunked() - Concluída")
    print(f"computeStats() - Concluída")
    print(f"Mapas de calor - Concluída")

    # Estatísticas finais
    print(f"\nEstatísticas dos percentuais de atraso:")
    print(f"Média geral: {final_results['Perc'].mean():.3f}")
```

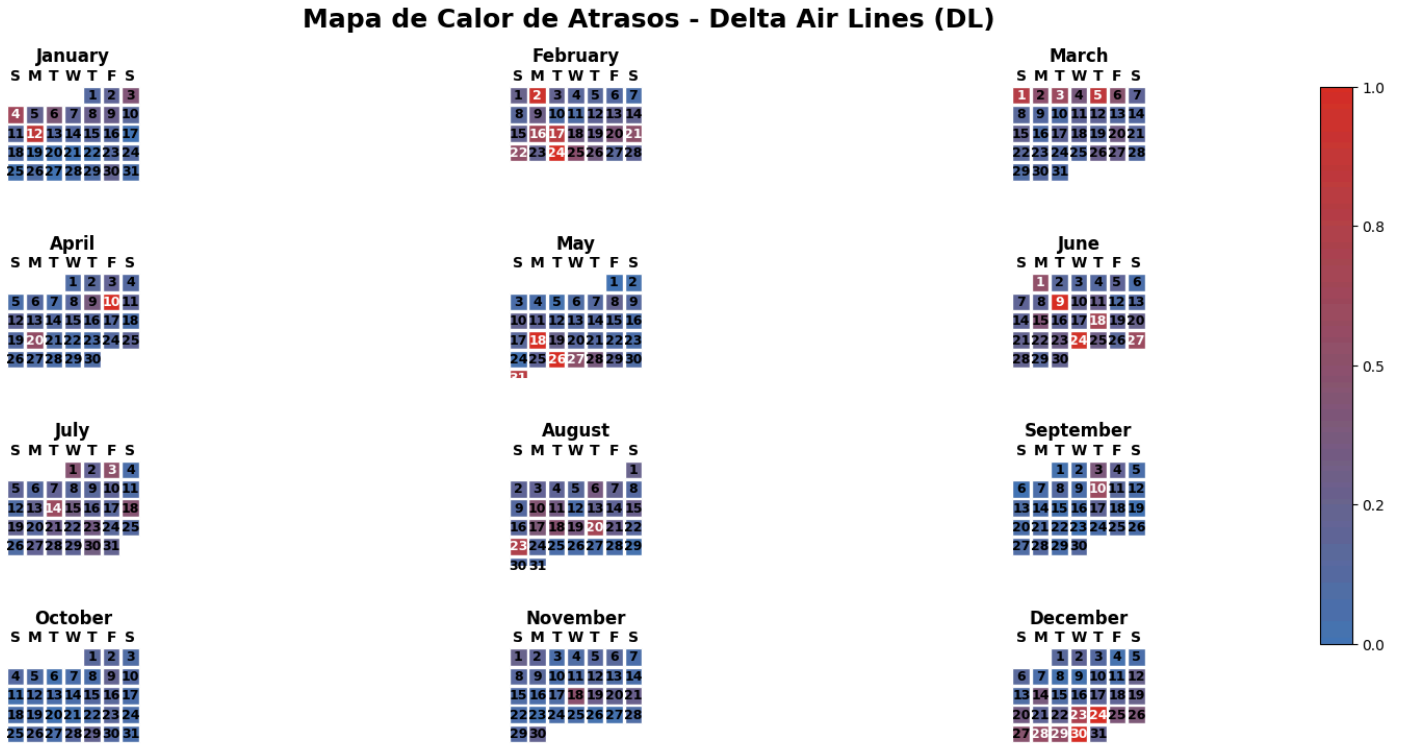
```
    print(f"Mediana: {final_results['Perc'].median():.3f}")
    print(f"Min: {final_results['Perc'].min():.3f}")
    print(f"Max: {final_results['Perc'].max():.3f}")

    return final_results, calendar_results

# EXECUTAR PIPELINE COMPLETO (Partes 1, 2, 3 e 4):
print("Executando pipeline completo com mapas de calor...")
results, heatmaps =
complete_pipeline_with_heatmaps('flights.csv/flights.csv')

# Função auxiliar original (mantida para compatibilidade)
def process_flights_in_chunks(file_path, chunk_size=10000):
    """
    FUNÇÃO ORIGINAL - mantida para compatibilidade.
    Processa o arquivo de voos em partes usando a função
getStats.

    RECOMENDAÇÃO: Use main_processing_pipeline() para
funcionalidade completa.
    """
    all_stats = []

    # Ler arquivo em chunks
    for pos, chunk in enumerate(pd.read_csv(file_path,
chunksize=chunk_size)):
        try:
            chunk_stats = getStats(chunk, pos)
            if not chunk_stats.empty:
                all_stats.append(chunk_stats)
        except Exception as e:
            print(f"Erro processando chunk {pos}: {e}")
            continue

    if all_stats:
        # Consolidar todos os resultados
        consolidated = pd.concat(all_stats, ignore_index=True)

        # Reagrupar e recalcular estatísticas finais se
necessário
        # (dependendo da lógica de negócio específica)
```

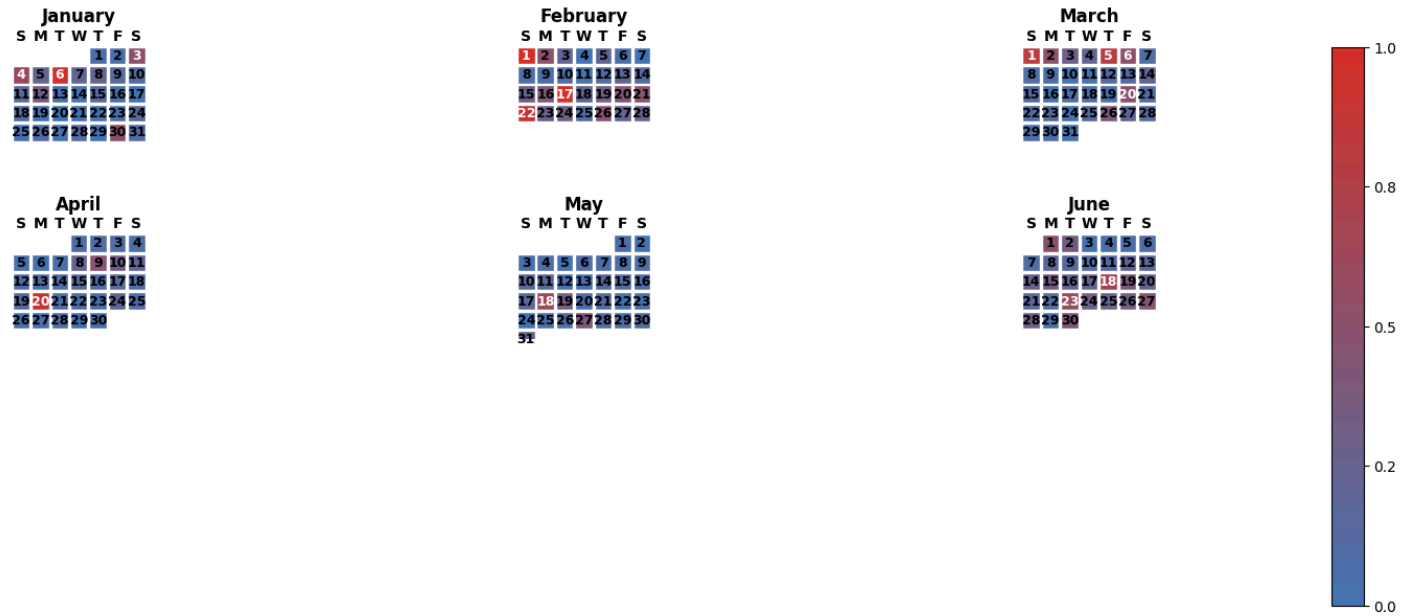
```
        return consolidated
    else:
        return pd.DataFrame()
```

Saídas do código em Python - Mapas de Calor do Atraso de Voos por Companhias Aéreas






Mapa de Calor de Atrasos - Delta Air Lines (DL)
January
February
March
April
May
June
July
August
September
October
November
December
S M T W T F S
1.0
0.8
0.5
0.2
0.0


Mapa de Calor de Atrasos - US Airways (US)
January
February
March
April
May
June
S M T W T F S
1.0
0.8
0.5
0.2
0.0