Entregáveis 3.2 e 3.3. Relatório para avaliação da qualidade de CDA e CDPV

Índice

Índice das Figuras

1. Introdução

O presente relatório tem como principal objetivo avaliar a qualidade dos resultados produzidos pelo PyCCD, comparando-os com uma base de dados de referência (BDR) disponibilizada pela Direção-Geral do Território (DGT). Esta base de dados é um ficheiro shapefile que contém quebras de vegetação identificadas por analistas, delimitadas por buffers no tile T29TNE, que abrange a região centro de Portugal Continental.

O PyCCD produz, como saída principal, um ficheiro em formato parquet, que contém os resultados da análise de séries temporais para cada pixel. Cada linha deste ficheiro representa um segmento temporal identificado pelo PyCCD, o que significa que um único pixel pode estar associado a vários segmentos e, consequentemente, a várias linhas no ficheiro. A partir do ficheiro parquet gerado pelo PyCCD, é possível construir a Carta de Datas de Alteração (CDA) em formato raster. Esta carta representa, para cada pixel, a data da última alteração detetada no comportamento do NDVI (Índice de Vegetação da Diferença Normalizada), representando tanto aumentos como perdas de vegetação. Com base na mesma estrutura de dados, é possível também produzir a Carta de Datas de Perda de Vegetação (CDPV), em formato raster, ao analisar a diferença de magnitudes de NDVI antes e depois da quebra, permitindo assim identificar as perdas de vegetação.

A avaliação da qualidade do PyCCD é realizada por meio de um processo de validação, que compara as quebras de vegetação detetadas pelo PyCCD com as registadas na base de referência. A validação considera uma margem de tolerância temporal de ±60 dias, definindo o intervalo máximo admissível entre a data de quebra indicada pelo PyCCD e a data de quebra indicada pela BDR. Se as duas datas coincidirem dentro deste intervalo, é considerada uma correspondência válida entre os dois métodos.

Com base nesta correspondência, são calculadas métricas quantitativas que permitem avaliar o desempenho e a fiabilidade do PyCCD, designadamente:

- Verdadeiros Positivos (VP): Quebras identificadas tanto pelo PyCCD como pela BDR, dentro da margem de tolerância estabelecida;
- Verdadeiros Negativos (VN): Períodos em que nenhuma quebra foi detetada por nenhum dos métodos, confirmando a ausência de mudanças.
- Falsos Positivos (FP): Quebras detetadas pelo PyCCD, mas que não têm correspondência com nenhuma quebra registada pela BDR;
- Falsos Negativos (FN): Quebras assinaladas pela BDR que não foram detetadas pelo PyCCD;

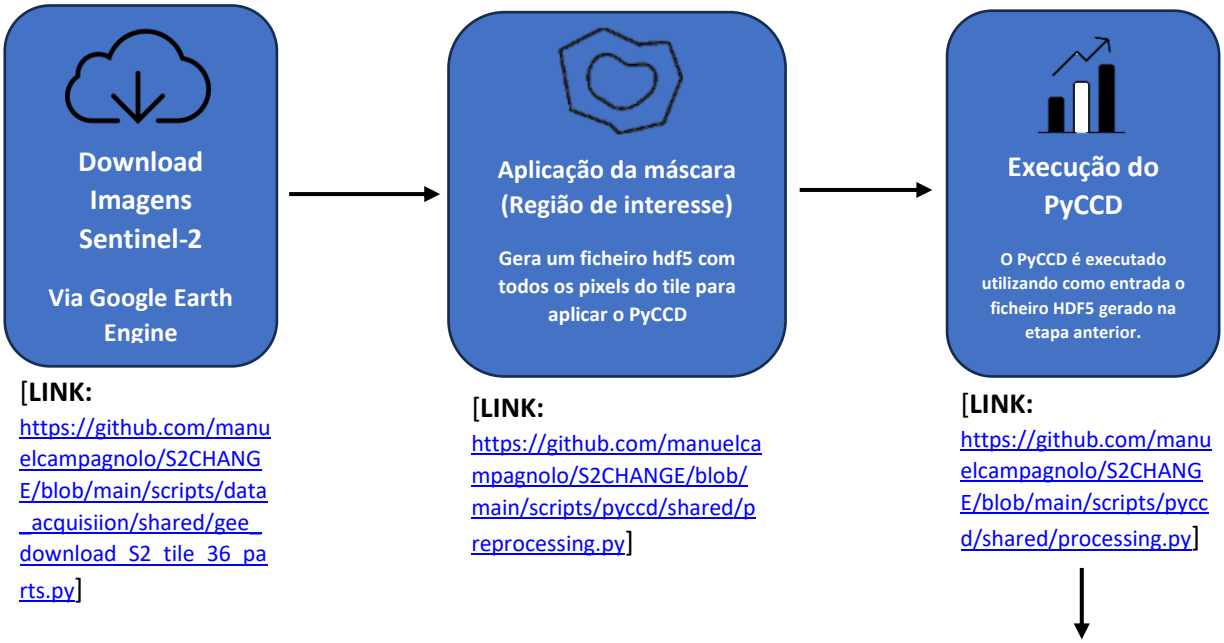
Com base nas métricas acima, são calculados os seguintes indicadores de desempenho:

- F1-score – Capacidade de o modelo identificar corretamente as quebras reais e de evitar deteções incorretas;
- Erro de Omissão – Percentagem de quebras reais que não foram detetadas pelo PyCCD (associado aos falsos negativos);
- Erro de Comissão – Percentagem de quebras assinaladas pelo PyCCD que não correspondem a eventos reais (associado aos falsos positivos).

Este conjunto de métricas permite uma avaliação robusta do desempenho do PyCCD, identificando tanto a sua capacidade de detetar quebras reais como a sua tendência para gerar erros.

2. Pipeline do PyCCD

Foi elaborado o diagrama a seguir, que apresenta de forma clara as principais etapas do pipeline de processamento do PyCCD, incluindo os links correspondentes do GitHub para cada parte do processo:



[**LINK:** https://github.com/manuelcampagnolo/S2CHANGE/blob/main/scripts/data_acquisiion/shared/gee_download_S2_tile_36_parts.py]

[**LINK:** https://github.com/manuelcampagnolo/S2CHANGE/blob/main/scripts/pyccd/shared/preprocessing.py]

[**LINK:** https://github.com/manuelcampagnolo/S2CHANGE/blob/main/scripts/pyccd/shared/processing.py]

[**LINK:** https://github.com/manuelcampagnolo/S2CHANGE/blob/main/scripts/pyccd/shared/avaliacao_exatidao_pyccd.py]

[**LINK:** https://github.com/manuelcampagnolo/S2CHANGE/blob/main/scripts/pyccd/local/main.py https://github.com/manuelcampagnolo/S2CHANGE/blob/main/scripts/pyccd/shared/ccd_results_filter.py]

→ Esta parte do código ainda está em desenvolvimento.

3. Carta de Datas de Perdas de Vegetação (CDPV)

A Carta de Datas de Perda de Vegetação (CDPV) tem como principal objetivo identificar com precisão as datas em que ocorrem perdas significativas de cobertura vegetal, com base na análise temporal do NDVI. A CDPV, tal como a Carta de Datas de Alteração (CDA), é obtida a partir dos ficheiros parquet, gerados pelo PyCCD, que armazenam os segmentos individualmente para cada pixel. Contudo, enquanto a CDA abrange todas as alterações detetadas — tanto aumentos quanto reduções na vegetação —, a CDPV foca-se exclusivamente nos eventos em que há uma diminuição do NDVI, após uma alteração identificada. Para calcular a magnitude do NDVI entre os segmentos, é utilizado o valor final de cada segmento ajustado com o valor inicial do segmento subsequente, sendo essa diferença calculada com base nos coeficientes que ajustam os segmentos.

Abaixo, encontra-se o código em Python que calcula a magnitude do NDVI entre os segmentos de um mesmo pixel, processo que ainda está em fase de desenvolvimento:

```python
DAYS_IN_YEAR = 365.25
NODATA_VALUE = -9999

def calcular_magnitude_entre_segmentos(df):
    df_resultado = pd.DataFrame()

    for (x, y), group in df.groupby(['x_coord', 'y_coord']):
        group_predicted_values = []
        group_ndvi_magnitudes = []

        for _, row in group.iterrows():
            intercept = row['intercept_values']
            coef = row['coeficientes']  # vetor com 7 coeficientes
            tStart, tEnd = int(row['tStart']), int(row['tEnd'])

            start_date = datetime.datetime.utcfromtimestamp(tStart / 1000)
            end_date = datetime.datetime.utcfromtimestamp(tEnd / 1000)

            days = np.arange(start_date.toordinal(), end_date.toordinal() + 1)

            # Cálculo dos valores do segmento ajustado
            ndvi = (
                intercept +
                coef[0] * days +
                coef[1]*np.cos(1 * 2 * np.pi * days / DAYS_IN_YEAR) +
                coef[2]*np.sin(1 * 2 * np.pi * days / DAYS_IN_YEAR) +
                coef[3]*np.cos(2 * 2 * np.pi * days / DAYS_IN_YEAR) +
                coef[4]*np.sin(2 * 2 * np.pi * days / DAYS_IN_YEAR) +
                coef[5]*np.cos(3 * 2 * np.pi * days / DAYS_IN_YEAR) +
                coef[6]*np.sin(3 * 2 * np.pi * days / DAYS_IN_YEAR)
            )

            group_predicted_values.append(ndvi)

        # Calcular as magnitudes de NDVI entre os segmentos do grupo
        for i in range(len(group_predicted_values) - 1):
            last_val = group_predicted_values[i][-1]  # Último valor do segmento atual
            next_val = group_predicted_values[i + 1][0]  # Primeiro valor do próximo segmento
            group_ndvi_magnitudes.append(int(last_val - next_val))

        # Se não houver mais segmentos, adicionar o valor -9999 no último
        if group_ndvi_magnitudes:
            group_ndvi_magnitudes.append(NODATA_VALUE)

        if len(group_ndvi_magnitudes) < len(group):
            group_ndvi_magnitudes += [np.nan] * (len(group) - len(group_ndvi_magnitudes))

        group["predicted_values"] = group_predicted_values
        group["ndvi_magnitude"] = group_ndvi_magnitudes

        df_resultado = pd.concat([df_resultado, group])

    return df_resultado
```

Figura 1 – Excerto do código em python utilizado para a construção da CDPV, baseado na estrutura dos ficheiros parquet gerados pelo PyCCD.

4. Parâmetros do PyCCD

Os resultados avaliados foram produzidos com o PyCCD configurado com os seguintes parâmetros:

- `'ALPHA': 2`
- `'CHISQUAREPROB': 0.999`
- `'LASSO_MAX_ITER': 1000`
- Período de execução do PyCCD**:** de abril de 2017 até 31 de dezembro de 2021.

Estes parâmetros podem ser ajustados conforme necessário no ficheiro parameters.py, disponível no repositório GitHub do projeto, acessível através do seguinte caminho: https://github.com/manuelcampagnolo/S2CHANGE/blob/main/scripts/pyccd/ccd/parameters.py

5. Inputs da validação

A validação dos resultados do PyCCD é realizada com base em três conjuntos de dados de entrada:

a) Ficheiros parquet gerados pelo PyCCD

- Cada linha dos ficheiros contém:
  - Localização geográfica do pixel (x_coord, y_coord) (coordenadas em EPSG:32629);
  - Datas de início (`tStart`) e fim (`tEnd`) do segmento (em milissegundos);
  - Data da quebra (`tBreak`), quando aplicável (em milissegundos);
  - Coeficientes de ajuste aplicados ao segmento;
  - Probabilidade de mudança (changeProb): Este valor representa a confiança do modelo de que houve uma alteração significativa no comportamento do NDVI dentro do segmento:
    - changeProb = 100 indica uma certeza de que ocorreu uma mudança, com alta probabilidade de alteração no comportamento do NDVI.
    - changeProb = 0 indica ausência de alteração, ou seja, o modelo não identificou nenhuma mudança significativa no comportamento do NDVI.

Index	4
tBreak	1619481600000
tEnd	1619049600000
tStart	1500854400000
changeProb	100
x_coord	499995
y_coord	4390205
coeficientes	[-5.69563053e-01 8.93928921e+02 -1.5043127… -4.93139459e+01 1.30788534e+02 6.4159287…
intercept_values	426785

Figura 2 – Exemplo de um segmento presente nos ficheiros parquets.

b) Ficheiro da base de dados de referência

- O ficheiro da BDR deverá conter as seguintes colunas obrigatórias para garantir a correta execução do código de validação:
    - buffer_ID
    - data_0
    - data_1
    - data_2
    - data_3
    - classe_0
    - classe_1
    - classe_2
    - classe_3
    - tipo_1
    - tipo_2
    - área
    - classe2018
    - classe2019
    - classe2020
    - classe2021
    - altera

Nota: As colunas data_0, data_1, data_2 e data_3 representam datas de referência fornecidas por analistas com base na interpretação visual dos dados. Estas datas servem como base temporal para avaliar a precisão da deteção automática de mudanças realizadas pelo PyCCD.

c) Parâmetros de validação temporal

- `dt_ini = '2018-09-12'`: Data inicial da janela de validação.
- `dt_end = '2021-09-30'`: Data final da janela de validação.
- `theta = 60`: Tolerância temporal (em dias). Define o intervalo máximo entre a data da quebra identificada pelo PyCCD e a quebra indicada pela BDR para que ambas sejam consideradas coincidentes.

```
# --------------------------------
#      PARAMETROS DA VALIDACAO
# --------------------------------
# datas do filtro das datas da analise (DGT 300)
dt_ini = '2018-09-12' # data inicial
dt_end = '2021-09-30' # data final
# Margem de tolerancia entre a quebra do Modelo e do Analista
theta = 60 # +/- theta dias de diferenca
# banda a filtrar com base na magnitude
bandFilter = None #nao implementado ainda - nao mexer

FOLDER_PARQUET = r'C:\Users\scaetano\Downloads\T29TNE'
BDR_DGT = r'C:\Users\Public\Documents\BDR_300_artigo\BDR_CCDC_TNE_Adjusted.shp'
runValidation(FOLDER_PARQUET, BDR_DGT, dt_ini, dt_end, bandFilter, theta)
```

Figura 3 – Inputs do código da validação dos resultados do PyCCD.

6. Outputs da validação

a) Ficheiro de validação (.csv):

- Criado automaticamente após a execução do código. É guardado dentro da pasta de input, no subdiretório criado com o nome "accuracy_assessment".
- Cada linha corresponde a um segmento dos ficheiros parquet, onde cada segmento é comparado com a BDR.
- O resultado da comparação é registado no ficheiro csv, indicando a correspondência (ou não) entre as quebras detetadas pelo PyCCD e as identificadas pela BDR.

b) Métricas de desempenho apresentadas no final do processo:

- **F1-Score:** Resume a precisão e a capacidade de deteção do modelo.
- **Erro de Omissão:** Indica a proporção de quebras reais que o modelo não conseguiu detetar.
- **Erro de Comissão:** Representa a proporção de quebras indicadas pelo modelo que não correspondem a alterações verificadas.

```
In [2]: runfile('C:/Users/scaetano/Desktop/S2CHANGE/scripts/pyccd/shared/a
Desktop/S2CHANGE/scripts/pyccd/shared')
A correr validação dos resultados do ccd...
Métricas de validação para ficheiro:
s2_images-NDVI_XX999YM1NOBS6LDA2ITER1000_START20170408_END20211230_ROIDGT
F1-score = 82.8%
Omission error = 15.28%
Commission error = 19.03%
```

Figura 4 – Métricas do desempenho do PyCCD.

7. Resultados e conclusões

O PyCCD apresentou um desempenho satisfatório na deteção de alterações, com um F1-Score de 82.8%, refletindo uma boa capacidade de identificar padrões de mudança ao longo do tempo. A validação foi realizada com base na Carta de Datas de Alteração (CDA), o que significa que foram consideradas todas as alterações identificadas, independentemente de serem associadas a perdas ou ganhos de vegetação. Assim, a avaliação não se limitou apenas às perdas, como seria o caso da CDPV, mas incluiu todas as alterações detetadas pelo PyCCD.

É importante salientar que a base de dados de referência utilizada, disponibilizada pela DGT, apresenta uma limitação temporal, uma vez que cobre apenas até ao final de 2021. Como o processamento dos dados pelo PyCCD se estende até ao final de 2024, será necessário complementar esta validação com fontes adicionais e atualizadas, de modo a garantir uma avaliação contínua e precisa do desempenho do PyCCD ao longo de todo o período analisado (e.g BDR's da Navigator e do ICNF).

Adicionalmente, encontra-se em desenvolvimento a Carta de Datas de Perda de Vegetação (CDPV) que isola exclusivamente os eventos associados à perda de vegetação. O código para a criação e representação espacial da CDPV ainda está em fase de testes, mas será facilmente adaptável ao processo de validação já implementado, que atualmente contempla todas as alterações detetadas pelo PyCCD.

8. Anexos

O código utilizado para a validação encontra-se disponível no repositório GitHub do projeto, acessível através do seguinte link: https://github.com/manuelcampagnolo/S2CHANGE/blob/main/scripts/pyccd/shared/avaliacao_exatidao_pyccd.py

```
import os
from datetime import datetime
import pandas as pd
import numpy as np
import geopandas as gpd
import csv

import warnings
warnings.filterwarnings('ignore')


def inferDelimiter(pathDF):
  with open(pathDF, 'r') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.readline())
    return dialect.delimiter

def convertDate(data):
  """Retorna ano, mês e dia a partir de data no formato YYYY-MM-DD"""
  data = data.split('-')
  y = int(data[0])
  m = int(data[1])
  d = int(data[2])
  return y,m,d

def filterDate(pathDF, dataI, dataF,bandFilter, mag = None):
    """
    Reduz o número de linhas do data frame de entrada, removendo as linhas fora do período de análise e
    para o limite estabelecido de magnitude máxima.
    Entrada:
        pathDF: caminho do Data Frame do CCDC
        dataI: String com a data inicial na forma = 'AAAA-MM-DD' (e.g. a data inicial dos analistas nos pontos DGT 300)
        dataF: String com a data final na forma = 'AAAA-MM-DD' (e.g. a data final dos analistas nos pontos DGT 300)
        bandFilter: String com a banda para a qual se deseja filtrar os dados. A esta banda é aplicado o criterio do mag.
        mag: Número com o limite da magnitude, e.g 0 só serão utilizadas as linhas com magnitudo menor ou igual a zero
    Saída:
        Data Frame filtrado
```

```
    """
    # Data Frame CCDC
    if pathDF.endswith('.csv'):
        delimiter = inferDelimiter(pathDF)
        df = pd.read_csv(pathDF, delimiter = delimiter)
    if pathDF.endswith('.pkl'):
        df = pd.read_pickle(pathDF)

    for dtCol in df.columns:
        if 'tBreak' in dtCol or 'tEnd' in dtCol or 'tStart' in dtCol:
            mask = df.loc[:, dtCol] == 0
            df[dtCol] = pd.to_datetime(df[dtCol], unit = 'ms')
            df.loc[mask, dtCol] = np.nan
        elif 'End_S' in dtCol:
            df[dtCol] = pd.to_datetime(df[dtCol]) # Esta coluna inicialmente esta em formato texto
    df.rename(columns={ 'Unnamed: 0':'IDCCDC'}, inplace=True)

    if mag != None:
        # caso haja magnitude limite, colocar tudo como NAT que seja acima deste limite
        df.loc[df[bandFilter] > mag, 'tBreak'] = pd.to_datetime(np.nan)
        df = df.copy()
    else:
        df = df.copy()

    # filtro das datas
    yi, mi, diai = convertDate(dataI)
    fltInicial = datetime(yi, mi, diai)
    yf, mf, diaf = convertDate(dataF)
    fltFinal = datetime(yf, mf, diaf)

    # 1 Adiciona a coluna com a menor data de start do fit
    df['startMin'] = df.groupby(['coord_ccdc'])['tStart'].transform('min')

    # 2 Adiciona o número de breaks existentes num grupo de IDCCDC, independente de fltInicial e
fltFinal
    df['numBreak'] = np.ceil(df.groupby(['coord_ccdc'])['changeProb'].transform('sum'))

    # Colocar Nat nas probabilidades fracionadas
    df.loc[((df.changeProb > 0) & (df.changeProb < 1)), 'tBreak'] = pd.to_datetime(np.nan)

    # 3 Verifica se se os breaks estão dentro do período de análise e transforma em NaT todos os que
não estão
    df['breaks_in_tmask'] = (~df.tBreak.isnull()).astype(int)
    df.loc[(df['tBreak'] <= fltInicial) | (df['tBreak'] >= fltFinal), 'breaks_in_tmask'] = 0
    df.loc[(df['tBreak'] <= fltInicial) | (df['tBreak'] >= fltFinal), 'tBreak'] = np.nan

    # Mascaras necessárias
```

```
    # a) Verifica os breaks NaT para as linhas com mais de 1 break
    mask = pd.Series(np.zeros(len(df),dtype=bool),index = df.index)
    mask.loc[(df.tBreak.isnull()) & (df.numBreak > 1)]= True #cond3

    # b) Verifica nas linhas de 1 break e sejam nulos qual é aquele que tem o início da série,
    #pois caso esteja fora da data de análise deve ser eliminado
    nmask = pd.Series(np.zeros(len(df),dtype=bool),index = df.index)
    nmask.loc[(df.tBreak.isnull()) & (df.numBreak == 1) & (df.breaks_in_tmask == 0) & (df.tStart ==
df.startMin)]= True

    # Aplica as mascaras acima e gera um novo DF
    subset_Filtro = df[((mask == False) & (nmask == False))].copy()

    # c) Calcula quantos linhas há por IDCCDC e caso ainda existam 2 significa que o break está dentro
do período de análise e o fit final, sem break
    # deve ser eliminado
    smask = pd.Series(np.zeros(len(subset_Filtro),dtype=bool),index = subset_Filtro.index)
    smask.loc[(subset_Filtro.groupby(['coord_ccdc'])['IDCCDC'].transform('count') == 2) &
(subset_Filtro.changeProb == 0) & (df.numBreak == 1)] = True
    subset_Filtro = subset_Filtro[(smask == False)].copy()

    # d) Para os IDCCDC que apresentam linhas com probabilidade fracionada, mantem esta linha, no
caso de todas estarem fora do período de análise
    pmask = pd.Series(np.zeros(len(df),dtype=bool),index = df.index)
    pmask.loc[~((df.changeProb > 0) & (df.changeProb < 1) & (df.tBreak.isnull()) &
(df.groupby(['coord_ccdc'])['tBreak'].transform('count') == 0))]=True
    subset_Filtro = pd.concat([subset_Filtro,df[pmask == False]])#subset_Filtro.append(df[pmask ==
False])

    # e) Para os IDCCDC que tem mais de um break e todos estao fora do periodo e devemos manter o
fit final
    fmask = pd.Series(np.zeros(len(df),dtype=bool),index = df.index)
    fmask.loc[((df.changeProb == 0) & (df.numBreak > 1) & (df.tBreak.isnull()) &
(df.groupby(['coord_ccdc'])['tBreak'].transform('count') == 0))]=True
    subset_Filtro = pd.concat([subset_Filtro,df[fmask]])#subset_Filtro.append(df[fmask])


    return subset_Filtro

def spatialJoin(pathPoligonosDGT, dfCCDC):
  """
  Realizar o spatial join entre o dataframe do CCDC e os poligonos com alteracoes identificadas pela
DGT
  Entrada:
   - pathPoligonosDGT: String com o caminho completo dos poligonos desenhados pela DGT
   - pathDataFrameCCDC: Data Frame filtrado do CCDC
   Saida:
```

```
  """
  # 1) ABRIR OS ARQUIVOS
  ## Poligonos DGT
  gdfVal = gpd.read_file(pathPoligonosDGT)
  gdfVal.to_crs(crs = 'EPSG:3763', inplace = True) # Originalmente eles estao em WGS84 29N
converte para ETRS
  ## Pontos ISA

  # 2) CONVERTER O DF PARA GEO DF
  gdfCCDC = gpd.GeoDataFrame(dfCCDC, geometry = gpd.points_from_xy(dfCCDC.longitude,
dfCCDC.latitude), crs=32629) # old csvs - crs=4326
  gdfCCDC.to_crs(crs=4326, inplace=True)

  ## criar a bordadura
  ###idBord = identity.copy() # cria uma copia do identity gerado acima
  idBord = gdfVal.copy()
  idBord['geometry'] = idBord.geometry.buffer(-10) # reduz a geometria em 10 metros
  idBord.drop(list(idBord.columns)[:-1], axis = 1, inplace = True) #remove todas as colunas menos a
da geometria
  idBord['bordadura'] = 1 # cria uma nova coluna para poder identificar a borda dura
  ## novo identity para termos a area da borda dura

  ###identity = gpd.overlay(identity, idBord, how='identity')
  identity = gpd.overlay(gdfVal, idBord, how = 'identity')

  # Como o poligono inicial nao tinha a coluna de bordadura, há feições onde
  # temos 1 e Nulos, com a linha abaixo invertemos o campo onde era Nullo passa a True
  # e onde era 1 passa para False, ou 1 e 0
  identity.bordadura = identity.bordadura.isnull()
  # Convertemos o resultado para WGS84
  identity.to_crs(crs = 'EPSG: 4326', inplace = True)


  ## As datas da DGT estao no formato (20200103) e precisam ser convertidas
  for dataCol in ['data_0', 'data_1', 'data_2', 'data_3']:
     # primeiro converter para datetime
     maskZero = pd.Series(np.zeros(len(identity),dtype=bool))
     erro = identity[dataCol].isnull()
     identity.loc[erro, dataCol] = 0
     # converter tudo para inteiros e onde for 0 indicar 1970
     identity[dataCol] = identity[dataCol].astype(int)
     maskZero = identity.loc[:, dataCol] == 0
     identity.loc[maskZero, dataCol] = 19700101
     # converter para datetime
     identity[dataCol] = pd.to_datetime(identity[dataCol], format = '%Y%m%d')
     identity.loc[maskZero, dataCol] = np.nan
```

```
 # 4) SPATIAL JOIN ENTRE OS CENTROIDES DO CCDC COM OS BUFFERS DE 200 METROS
 subset = gpd.sjoin(gdfCCDC, identity, how='inner')
 subset.reset_index(inplace = True)
 subset['buffer_ID'] = subset.buffer_ID.astype('int')


 #Descobrir quais linhas precisam ser duplicadas.
 #Pressupondo que não é possível ter informação da 'data_3' sem existir a 'data_1'
 #é possível filtrar e verificar a negação de quais dados são nulos e depois somar
 #o reultado.
 #0 = False False: não há data_1 e nem data_3
 #1 = True  False: existe data_1 e não data_3
 #2 = True  True: existem data_1 e Data_3

 cond = ~subset.filter(items=['data_1', 'data_3']).isnull()
 subset['analistas'] = cond.sum(axis=1)
 subset.loc[subset['analistas'] == 0, 'exists_event'] = False # Analista nao identificou nada
 subset.loc[subset['analistas'] > 0, 'exists_event'] = True # Analista identificou alteracao


 #CRIA UM DF TEMPORARIO PARA COPIAR AS LINHAS ONDE EXISTEM A 'DATA_3' E INSERE ESTA
DATA NO CAMPO 'DATA1_Z'
 #DEPOIS ADICIONA ISTO AO DATA FRAME ORIGINAL

 subset['data1_z'] = ''
 # criar coluna para as datas anteriores
 # subset['data0_z'] = ''
 subset['nome'] = '' # teste para nomear os analistas
 subset['tipo'] = ''
 subset['classeAnterior'] = ''
 subset['classeAtual'] = ''
 dfTemp = pd.DataFrame(columns = subset.columns)
 for row in subset.itertuples():
   # verifica se há duas datas e duplica a linha
   if row.analistas == 2:
      dfTemp = pd.concat([dfTemp,
subset[subset.index==row.Index]],ignore_index=False)#dfTemp.append(subset[subset.index ==
row.Index], ignore_index=False)
 dfTemp.data1_z = dfTemp.data_3
 # capturar o valor da data_2
 # defTemp.data0_z = dfTemp.data_2
 dfTemp.nome = 'B' # teste para nomear os analistas
 dfTemp.tipo = dfTemp.tipo_2
 dfTemp.classeAtual = dfTemp.classe_3
 dfTemp.classeAnterior = dfTemp.classe_2
```

```
  subset.data1_z = subset.data_1
  # capturar o valor da data_0
  # subset.data0_z = subset.data_0
  subset.nome = 'A' # teste para nomear os analistas
  subset.tipo = subset.tipo_1
  subset.classeAtual = subset.classe_1
  subset.classeAnterior = subset.classe_0

  subset = pd.concat([subset, dfTemp],ignore_index=False)#subset.append(dfTemp,
ignore_index=False)

  # Contagem do numero de breaks
  subset['Valid_breaks'] = np.ceil(subset.groupby(['coord_ccdc',
'nome'])['changeProb'].transform('sum'))

  # COLUNA DO DELTA MIN
  subset['delta_min'] = (subset.data1_z - subset.tBreak).dt.days
  subset.drop(['data_1', 'data_3', 'tipo_1', 'tipo_2','classe_0', 'classe_1','classe_2', 'classe_3'], axis = 1,
inplace = True)

  # verificar quais colunas tem magnitude de indices
  mags = [ t for t in subset.columns if 'magnitude' in t and not 'B' in t]
  ordem = [ 'coord_ccdc','buffer_ID', 'IDCCDC', 'altera', 'changeProb'] + mags + ['tBreak', 'data1_z',
       'bordadura', 'classe2018', 'classe2019', 'classe2020','classe2021', 'classeAnterior','tipo',
       'classeAtual', 'analistas', 'nome', 'exists_event', 'Valid_breaks' , 'delta_min', 'geometry']


  return subset[ordem], subset


def preprocessCsvS2(csv_s2, end_of_series):
  """
  Does a pre-processing of the csv containing detection results to ensure it has the necessary
columns
  and coherent values for the validation procedure.
  Args:
    csv_s2: a pandas dataframe obtained after reading the csv file containing ccd detection results;
    end_of_series: date of the last image in the series - a string in the form YYYY-mm-dd.
  Returns:
    Pre-processed dataframe.
  """

  csv_s2 = csv_s2.copy()
  from ast import literal_eval
  #do some processing on the csv
  # Selecionar as colunas a explodir e as dos coeficientes
  tabExplode = []
```

```
  tabCoefs = []
  for c in csv_s2.columns:
    if 'coefs' in c or 'magnitude' in c or 'rmse' in c:
      tabExplode.append(c)
    if 'coefs' in c:
      tabCoefs.append(c)
  tabExplode = tabExplode + ['changeProb', 'tBreak', 'tEnd', 'tStart']

  #convert from string of list to list
  for col in tabExplode:
    try:
      csv_s2[col] = csv_s2[col].apply(literal_eval)
    except: #sometimes CCDC returns 'Infinity' or 'NaN' as a rmse value, which results in literal_eval
not working
      #csv_s2[col] = csv_s2[col].apply(lambda x: x.replace('Infinity','9999999'))
      #csv_s2[col] = csv_s2[col].apply(lambda x: x.replace('NaN','-9999999'))
      csv_s2[col] = csv_s2[col].apply(literal_eval)
  #convert lat long separated by comma to separated by point
  #csv_s2['Lat'] = csv_s2['Lat'].apply(lambda x: x.replace(",","."))
  #csv_s2['Lon'] = csv_s2['Lon'].apply(lambda x: x.replace(",","."))

  #explode
  csv_s2 = csv_s2.explode(tabExplode)

  csv_s2['End_S'] = end_of_series
  csv_s2['coord_ccdc'] = list(zip(csv_s2.Lat, csv_s2.Lon))
  csv_s2['Dist_Point'] = -1#''
  csv_s2['Point_Val'] = -1#''

  #convert date columns from float to int
  for col in ['tBreak', 'tEnd', 'tStart']:
    csv_s2[col] = csv_s2[col].astype('int64')

  csv_s2.rename(columns={'Lat':'latitude','Lon':'longitude'}, inplace=True)

  return csv_s2


def preprocessParquetS2(parquet_directory, end_of_series):
  """
  Does a pre-processing of a directory containing parquet files with detection results to ensure it has
the necessary columns
  and coherent values for the validation procedure.
  Args:
    parquet_directory: path to a directory containing parquet files with ccd detection results;
    end_of_series: date of the last image in the series - a string in the form YYYY-mm-dd.
  Returns:
```

```
  Pre-processed dataframe.
 """

 column_names = ['tBreak', 'tEnd', 'tStart', 'changeProb', 'x_coord', 'y_coord', 'coeficientes']
 main_df = pd.DataFrame(columns=column_names)

 for file in os.listdir(parquet_directory):
  if file.endswith('.parquet'):
   file_path = os.path.join(parquet_directory, file)
   temp_df = pd.read_parquet(file_path)
   temp_df = temp_df[column_names].copy()
   main_df = pd.concat([main_df, temp_df], ignore_index=True)

 main_df.reset_index(drop=True, inplace=True)

 main_df.rename(columns={'x_coord':'longitude','y_coord':'latitude'}, inplace=True)
 main_df['End_S'] = end_of_series
 main_df['coord_ccdc'] = list(zip(main_df.latitude, main_df.longitude))
 main_df['Dist_Point'] = -1
 main_df['Point_Val'] = -1

 return main_df


# função de validação do data frame
def valPol(df, theta):
 """
 Esta função recebe o geodataframe gerado no spatialJoin() e contabiliza as métricas de positivos e
negativos.
 A Saída é a matriz com os cálculos e um dicinário com as métricas contabilizadas.

 """

 # transforma a coluna de delta min para valor absoluto e cria uma nova coluna com o mínimo delta
min por ponto
 df.reset_index(inplace = True)
 original_delta_min = df['delta_min'].copy()
 df['delta_min'] = abs(df['delta_min'].fillna(99999)) # substitui os nullos para evitar que sejam os
minimos
 df['Min_delta_min'] = df.groupby(['coord_ccdc', 'nome'])['delta_min'].transform('min') # calcula o
valor minimo por ponto
 df['delta_min'] = abs(original_delta_min) # retorna o valor absoluto da coluna original
 df['Min_delta_min'] = df['Min_delta_min'].replace(99999,np.nan) # substitui os 99999 por nullos

 bf = df.copy()
```

```
  bf['Valid_breaks'] = bf.groupby(['coord_ccdc', 'nome']).transform('count')[['tBreak']] # verifica os
breaks validos por pontos
  # SE O TBREAK FOR OBJETO ELE JAMAIS SERA NULO, CONVERTER PARA DATA.
  bf.tBreak = pd.to_datetime(bf.tBreak)
  bf.tStart = pd.to_datetime(bf.tStart)
  bf.tEnd = pd.to_datetime(bf.tEnd)
  bf.analistas = bf.analistas.astype(int)
  bf.exists_event = bf.exists_event.astype(int)
  bf.buffer_ID = bf.buffer_ID.astype(int)
  bf.IDCCDC = bf.IDCCDC.astype(int)

  ## ALGUMAS MASCARAS INICIAIS NECESSARIAS
  # mascara dos breaks a mais que analistas ainda em reformulacao

  # PARA O CASO DE TER SOMENTE UM BREAK FP E DOIS ANALISTAS PARA NAO TER DUPLICACAO
  mask = pd.Series(np.zeros(len(bf),dtype=bool), index= bf.index)
  mask.loc[(bf.analistas == 2) & (bf.Valid_breaks < bf.analistas) ] = True #& (bf.delta_min > theta)

  bf.loc[mask, 'Min_delta_min'] = bf.loc[mask].groupby(['coord_ccdc'])['delta_min'].transform('min')

  # Contabilizar
  # colocar todos os VP (delta_min <=31)
  #VP
  bf.loc[( (bf.delta_min <= theta) & (~bf.tBreak.isnull()) & (bf.analistas > 0) ), 'VP'] = 1
  # #FP
  # # sem a condição da magnitude ou (changeProb ==1) serao selecionados os que devem ser
negativos
  # bf.loc[( (bf.analistas == 0) & (bf.ndvi_magnitude != 0) & (~bf.tBreak.isnull())), 'FP' ] = 1 #FP puro
  # bf.loc[( (bf.delta_min > theta) & (bf.ndvi_magnitude != 0) & ( (bf.delta_min == bf.Min_delta_min)
& (~bf.Min_delta_min.isnull()) ) )  , 'FP' ] = 1
  # bf.loc[( (bf.delta_min > theta) & (bf.ndvi_magnitude != 0) & (bf.analistas == 1)  ) &
(~bf.tBreak.isnull()), 'FP' ] = 1
  #FP
  # sem a condição da magnitude ou (changeProb ==1) serao selecionados os que devem ser
negativos
  bf.loc[( (bf.analistas == 0) & (~bf.tBreak.isnull())), 'FP' ] = 1 #FP puro
  bf.loc[( (bf.delta_min > theta) & ( (bf.delta_min == bf.Min_delta_min) &
(~bf.Min_delta_min.isnull()) ) )  , 'FP' ] = 1
  bf.loc[( (bf.delta_min > theta) & (bf.analistas == 1)  ) & (~bf.tBreak.isnull()), 'FP' ] = 1
  #FN
  bf.loc[( (bf.analistas > 0)  & (bf.tBreak.isnull()) ), 'FN' ] = 1 # FN puro
  # falsos negativos que precisam ser contabilizado para os FPs
  bf.loc[(bf.analistas == 1) & (bf.Valid_breaks == 1) & (bf.FP == 1), 'FN'] = 1 # parece funcionar
  bf.loc[(bf.analistas == 2) & (bf.Valid_breaks == 3) & (bf.FP == 1) , 'FN'] = 1

  #VN
  bf.loc[( (bf.analistas == 0) & (bf.tBreak.isnull()) ), 'VN' ] = 1
```

```
  # converter os NaN para 0
  bf[['VP', 'FP', 'FN', 'VN']] = bf[['VP', 'FP', 'FN', 'VN']].fillna(0)

  # verificar os breaks que nao foram classificados
  # para isso gero uma coluna total onde somo todas as metricas, as linhas onde ha 0 nao foram
classificadas
  bf['total'] = bf.VP + bf.FP +bf.FN + bf.VN
  mask = pd.Series(np.zeros(len(bf),dtype=bool), index= bf.index) #mascara
  # agrupar por coordenada e t break, assim as somente os breaks que nao foram validados para
nenhum analista terao valor 0
  mask.loc[(bf.groupby(['coord_ccdc','tBreak'])['total'].transform('sum')==0) & (bf.analistas == 2) &
(bf.Valid_breaks > bf.analistas)] = True
  # neste grupo selecionado devo procurar aquele que tem menor distancia para um analista e
classificar como FP
  mask2 = bf[mask].groupby(['coord_ccdc'])['delta_min'].transform('min') == bf.delta_min[mask]
  # agora classificar os candidatos que atendem as duas mascaras
  bf.loc[(mask & mask2), ['FP']] = 1

  # Ajuste FN
  # se for na célula anterior isso contará para o total e a mascara anterior não será feita em alguns
pontos onde deve ser feita
  bf.loc[((bf.FP ==1) & (bf.analistas == 1) & (bf.delta_min == bf.Min_delta_min) & (bf.Valid_breaks ==
2))   , 'FN' ] = 1
  bf.loc[((bf.FP ==1) & (bf.analistas == 1) & (bf.delta_min == bf.Min_delta_min) & (bf.Valid_breaks ==
3))   , 'FN' ] = 1
  bf.loc[(bf.analistas == 2) & (bf.Valid_breaks == 1) & (bf.VP == 0), 'FN'] = 1
  bf.loc[(bf.analistas == 2) & (bf.Valid_breaks == 2) & (bf.FP == 1), 'FN'] = 1
  #return bf
  # Bloco para corrigir o problema de quando as duas datas DGT estão mais próximas do mesmo
break
  # listar as coordenadas que tem o problema com mesmo break classificado
  listCoord = list(bf.coord_ccdc[(bf.groupby(['coord_ccdc','tBreak'])['total'].transform('sum') == 0) &
(bf.analistas == 2) & (bf.Valid_breaks == 2)])
  #return listCoord
  # dividir o data frame em dois para poder limpar as linhas com problema
  bf_filter = bf.loc[~bf.coord_ccdc.isin(listCoord)].copy()
  # limpeza
  bf_remove_lines = bf.loc[bf.coord_ccdc.isin(listCoord)].copy()
  # zerar todas as métricas para poder recalcular
  bf_remove_lines.loc[:, ['VP','VN','FP', 'FN']] = 0
  #return bf_remove_lines
  bf_removed = bf_remove_lines.groupby(['buffer_ID','IDCCDC']).apply(testeRemove).copy() #
função de remoção
  #return bf_removed
  try:
```

```
    bf_removed = bf_removed.drop(columns=['buffer_ID','IDCCDC']).reset_index() # evitar problema
de indece dup.
  except:
    pass
  # Agora teremos somente duas linhas por ponto que são obrigatóriamente FP ou VP
  #VP
  bf_removed.loc[( (bf_removed.delta_min <= theta) ), 'VP'] = 1
  #FP, FN
  bf_removed.loc[( (bf_removed.delta_min > theta) ), ['FP', 'FN']] = 1
  # unir os dois dfs novamente
  bf_final = pd.concat([bf_filter, bf_removed])#bf_filter.append(bf_removed)

  # remover aqueles que nao possuem metrica
  bf_final = bf_final[(bf_final.VP > 0) | (bf_final.FP > 0) | (bf_final.FN > 0) | (bf_final.VN > 0) ].copy()
  # remover aqueles que apresentam as classes especificas
  bf_final = bf_final[~(bf_final.tipo.isin(['Agricultura','Agua']))].copy()


  # verificar quais colunas tem magnitude de indices
  mags = [ t for t in bf_final.columns if 'magnitude' in t and not 'B' in t]
  # colunas para retornar um DF mais limpo
  c = ['buffer_ID', 'IDCCDC', 'coord_ccdc', 'changeProb'] + mags + ['tBreak',
      'data1_z', 'analistas', 'nome', 'exists_event', 'Valid_breaks',
      'delta_min', 'Min_delta_min', 'VP', 'FP', 'FN', 'VN'] #geometry
  # também poderá retornar o DF todo classificado, em processo.
  return bf_final[c], bf_final

# # # função para realizar a limpeza de linhas indesejadas
def testeRemove(groupedby):
  min_delta_min = groupedby['Min_delta_min'].min()
  #remove rows only if there is more than 1 row per point, the number of analyst dates is not zero
and min_delta_min is greater than zero.
  if len(groupedby) > 1 and groupedby.analistas.min() > 0 and min_delta_min >= 0:
    # Updated section with check on matching rows
    # Add a check to see if there are any rows matching the condition
    matching_rows = groupedby.loc[groupedby['delta_min']==min_delta_min][['tBreak','data1_z']]

    if len(matching_rows) > 0:  # Only proceed if matching rows
      Bj, Ai = matching_rows.values[0]
      mask = ((groupedby['tBreak'] == Bj) | (groupedby['data1_z'] == Ai)) &
(groupedby['delta_min']!=min_delta_min)
      groupedby = groupedby[~mask]

    # original
    # Bj, Ai = groupedby.loc[groupedby['delta_min']==min_delta_min][['tBreak','data1_z']].values[0]
    # #remove rows that contain Ai or Bj (other than the row with the min_delta_min)
```

```
    # mask = ((groupedby['tBreak'] == Bj) | (groupedby['data1_z'] == Ai)) &
(groupedby['delta_min']!=min_delta_min)
    # groupedby = groupedby[~mask]

  return groupedby

def runValidation(FOLDER_PARQUET, BDR_DGT, dt_ini, dt_end, bandFilter, theta):
    """
    Corre a validação dos resultados da deteção realizando spatial join
    com a base de dados de referência.
    Imprime as métricas de validação e gera ficheiro csv VAL.

    Args:
      FOLDER_PARQUET: path to folder containing CCDC results in parquet files, will save results;
      BDR_DGT: caminho para o ficheiro da base de dados de validação;
      dt_ini: data inicial do período de referência dos analistas (str YYYY-mm-dd);
      df_end: data final do período de referência dos analistas (str YYYY-mm-dd);
      bandFIlter: não implementado ainda;
      theta: margem de tolerância da validação, em dias (int);

    Returns:
      None (imprime métricas e gera ficheiro csv)

    """

    print('A correr validação dos resultados do ccd...')
    #pegar data do fim da serie temporal (ultima imagem)

    single_file = os.listdir(FOLDER_PARQUET)[0]
    reference_index = single_file.find('END')
    end_of_series = single_file[reference_index + 3 : reference_index + 11]
    year, month, day = [end_of_series[:4], end_of_series[4:6], end_of_series[6:]]
    end_of_series = f"{year}-{month}-{day}"

    results_path = os.path.join(FOLDER_PARQUET, "accuracy_assessment")
    if not os.path.exists(results_path):
      os.makedirs(results_path)

    #correr pre-processamento
    csv_s2 = preprocessParquetS2(FOLDER_PARQUET, end_of_series)
    csv_s2['changeProb'] = csv_s2['changeProb'] / 100
    csv_preprocessed_path = os.path.join(results_path, 'pre_proc.csv')
    csv_s2.to_csv(csv_preprocessed_path)

    """## Filtrar datas
    Limitar análise ao período considerado pelos analistas DGT
    """
```

```
    #correr filtro de datas
    ccdcFiltro = filterDate(csv_preprocessed_path, dt_ini, dt_end, bandFilter)
    """## Spatial join
    Faz join dos pontos do csv com a informação de referencia da DGT (300 buffers). É associada aos pontos a informação da validação - data de alteração, tipo, classes, etc.
    """
    #gdfVal = gpd.read_file(BDR_DGT)
    #gdfVal.to_crs(crs = 'EPSG:3763', inplace = True)
    #executa o join
    ccdcVal, ccdcVal_T = spatialJoin(BDR_DGT, ccdcFiltro)
    """## Validação
    Faz a validação da deteção - compara resultado do modelo (ccd) com dados de referência DGT
    """
    #faz a validação da deteção
    DF_FINAL, DF_FINAL_T = valPol(ccdcVal_T, theta) #funcoes.valPol
    """**Resultados da validação**"""
    #delimita análise apenas para pontos referentes a transições entre Pinheiro Bravo e Eucalipto para Superfície sem vegetação, herbáceas e matos
    #elimina também pontos da bordadura
    df_aux = DF_FINAL_T.copy()
    df_aux = df_aux.loc[(df_aux.altera=="Sem Alteracao")|((df_aux.altera=="Com Alteracao")&(df_aux.classeAnterior.isin(['Pinheiro bravo','Eucalipto']))&(df_aux.classeAtual.isin(['Superficie sem vegetacao escura','Superficie sem vegetacao clara','Vegetacao herbacea espontanea','Matos'])))]
    df_aux = df_aux.loc[df_aux.bordadura==0]
    #imprime f1-score, erro e omissão e erro de comissão
    cm = df_aux.FP.sum()/(df_aux.FP.sum()+df_aux.VP.sum())
    om = df_aux.FN.sum()/(df_aux.FN.sum()+df_aux.VP.sum())
    f1 = 2*(1-om)*(1-cm)/(2-om-cm)
    print("Métricas de validação para ficheiro:")

    group_name = single_file.split("_rank_")[0]

    print(group_name)
    print('F1-score = {}%'.format(round(100*f1,2)))
    print('Omission error = {}%'.format(round(100*om,2)))
    print('Commission error = {}%'.format(round(100*cm,2)))

    DF_FINAL_T.to_csv(os.path.join(results_path, f'VAL_{group_name}.csv'), index=False)
#%%
# ---------------------------------
#     PARAMETROS DA VALIDAS□□O
# ---------------------------------
# datas do filtro das datas da an□lise (DGT 300)
########### N□o alterar ################
dt_ini = '2017-10-01' # data inicial
dt_end = '2023-12-31' # data final
```

```
# dt_end = '2023-12-29' # data final
# Margem de tolerรขncia entre a quebra do Modelo e do Analista
theta = 60 # +/- theta dias de diferenca
# banda a filtrar com base na magnitude
bandFilter = None #nรฃo implementado ainda - nรฃo mexer

# FOLDER_PARQUET = r'C:\Users\scaetano\Downloads\BDR_300_artigo'
FOLDER_PARQUET = r'C:\Users\Public\Documents\outputs_ROI\tabular\T29TNE_0.999'
BDR_DGT = r'C:\Users\scaetano\Downloads\BDR_MIX_TNE\BDR_MIX_TNE_new2.shp'
# BDR_DGT = r'D:\BDR_CCDC_TNE_Adjusted.shp'
runValidation(FOLDER_PARQUET, BDR_DGT, dt_ini, dt_end, bandFilter, theta)
```