ACID Evolution

Python Codes

Main Code

```python
import os
import excavator
import constants
import pandas as pd
import pickle
import time
import datetime
import sys
import git
import nltk
nltk.download('punkt')
nltk.download('punkt_tab')

'''
This script goes to each repo and mines commits and commit messages and then get the defect category
'''
def getBranchName(path):
    try:
        repo = git.Repo(path)
        default_branch_ref = repo.git.symbolic_ref('refs/remotes/origin/HEAD')
        default_branch = default_branch_ref.replace('refs/remotes/origin/', '')
        return default_branch
    except Exception as e:
        print(f"Error selecting Branch repo {path}: {e}")
        return None

def giveTimeStamp():
  tsObj = time.time()
  strToret = datetime.datetime.fromtimestamp(tsObj).strftime('%Y-%m-%d %H:%M:%S')
  return strToret

if __name__=='__main__':

    t1 = time.time()
    print('Started at:', giveTimeStamp())
    print('*'*100)

    flag_arg = sys.argv[sys.argv.index("--flag-arg") + 1]
    if flag_arg == '-x':
       orgName='EXTRA'
       print('ACID will now run on extra testing repos')
       out_fil_nam = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY.PKL'
       out_csv_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY_CATEG_OUTPUT_FINAL.csv'
       out_pkl_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY_CATEG_OUTPUT_FINAL.PKL'
    elif flag_arg == "-replication":
        orgName = 'PIPR-replication'
        print('ACID will now run on PIPr Replication repos')
        out_fil_nam = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY.PKL'
        out_csv_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY_CATEG_OUTPUT_FINAL.csv'
        out_pkl_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY_CATEG_OUTPUT_FINAL.PKL'
    elif flag_arg == "-t":
        orgName='TEST'
        print('ACID will now run on default testing repos')
        out_fil_nam = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/TEST2_ONLY.PKL'
        out_csv_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/TEST2_ONLY_CATEG_OUTPUT_FINAL.csv'
        out_pkl_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/TEST2_ONLY_CATEG_OUTPUT_FINAL.PKL'
    else:
        orgName = flag_arg
        print(f'ACID will now run on {flag_arg} repos')
        output_location = os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
        out_fil_nam     = output_location + f'/{flag_arg}_ONLY.PKL'
        out_csv_fil     = output_location + f'/{flag_arg}_CATEG_OUTPUT_FINAL.csv'
        out_pkl_fil     = output_location + f'/{flag_arg}_ONLY_CATEG_OUTPUT_FINAL.PKL'

    if orgName == 'EXTRA' or orgName == "TEST" or orgName == "PIPR-replication":
        csv_replication = None
        csv_default     = None
    else:
        csv_replication = os.path.abspath(sys.argv[sys.argv.index("--csv-replication") + 1])
        csv_default     = os.path.abspath(sys.argv[sys.argv.index("--csv-default") + 1])

    script_dir = os.path.dirname(os.path.abspath(__file__))
    pathRepo = script_dir + "/" + constants.DATASET_DIR + "/" + orgName + "/"
    fileName = pathRepo + "/" + constants.REPO_FILE_LIST
    elgibleRepos = excavator.getEligibleProjects(fileName)
    dic   = {}
    categ = []
    for proj_ in elgibleRepos:
        try:
            if proj_ == constants.REPO_FILE_LIST:
                continue
            path_proj = pathRepo + proj_
            branchName = getBranchName(path_proj)
            per_proj_commit_dict, per_proj_full_defect_list = excavator.runMiner(orgName, proj_, branchName,
csv_file_path=csv_replication, csv_default=csv_default)
            categ = categ + per_proj_full_defect_list
            # print proj_ , len(per_proj_full_defect_list)
            print('Finished analyzing:', proj_)
            dic[proj_] = (per_proj_commit_dict, per_proj_full_defect_list)
            # print(dic[proj_])
        except Exception as e:
            print(e)
        print('='*50)

    all_proj_df = pd.DataFrame(categ)
    all_proj_df.to_csv(out_csv_fil, header=['HASH','CATEG','REPO','TIME'], index=False)

    with open(out_pkl_fil, 'wb') as fp_:
        pickle.dump(dic, fp_)
    print('*'*100)
    print('Ended at:', giveTimeStamp())
    print('*'*100)
    t2 = time.time()
    time_diff = round( (t2 - t1 ) / 60, 5)
    print("Duration: {} minutes".format(time_diff))
    print('*'*100)
```

Main Concurrent Code

```python
import os
import excavator
import constants
import pandas as pd
import pickle
import time
import datetime
import sys
import git
import concurrent.futures
import nltk
nltk.download('punkt')
nltk.download('punkt_tab')

'''
This script goes to each repo and mines commits and commit messages and then get the defect category
'''
def getBranchName(path):
    try:
        repo = git.Repo(path)
        default_branch_ref = repo.git.symbolic_ref('refs/remotes/origin/HEAD')
        default_branch = default_branch_ref.replace('refs/remotes/origin/', '')
        return default_branch
    except Exception as e:
        print(f"Error selecting Branch repo {path}: {e}")
        return None

def giveTimeStamp():
  tsObj = time.time()
  strToret = datetime.datetime.fromtimestamp(tsObj).strftime('%Y-%m-%d %H:%M:%S')
  return strToret

if __name__=='__main__':

    t1 = time.time()
    print('Started at:', giveTimeStamp())
    print('*'*100)

    flag_arg = sys.argv[sys.argv.index("--flag-arg") + 1]
    if flag_arg == '-x':
       orgName='EXTRA'
       print('ACID will now run on extra testing repos')
       out_fil_nam = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY.PKL'
       out_csv_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY_CATEG_OUTPUT_FINAL.csv'
       out_pkl_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY_CATEG_OUTPUT_FINAL.PKL'
    elif flag_arg == "-replication":
        orgName = 'PIPR-replication'
        print('ACID will now run on PIPr Replication repos')
        out_fil_nam = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY.PKL'
        out_csv_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY_CATEG_OUTPUT_FINAL.csv'
        out_pkl_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY_CATEG_OUTPUT_FINAL.PKL'
    elif flag_arg == "-t":
        orgName='TEST'
        print('ACID will now run on default testing repos')
        out_fil_nam = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/TEST2_ONLY.PKL'
        out_csv_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/TEST2_ONLY_CATEG_OUTPUT_FINAL.csv'
        out_pkl_fil = '/home/aluno/ACID-dataset/ARTIFACT/OUTPUT/TEST2_ONLY_CATEG_OUTPUT_FINAL.PKL'
    else:
        orgName = flag_arg
        print(f'ACID will now run on {flag_arg} repos')
        output_location = os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
        out_fil_nam     = output_location + f'/{flag_arg}_ONLY.PKL'
        out_csv_fil     = output_location + f'/{flag_arg}_CATEG_OUTPUT_FINAL.csv'
        out_pkl_fil     = output_location + f'/{flag_arg}_ONLY_CATEG_OUTPUT_FINAL.PKL'
    def process_project(orgName, proj_, pathRepo, dic, categ, csv_replication, csv_default):
     try:
        if proj_ == constants.REPO_FILE_LIST: return
        path_proj = pathRepo + proj_
        branchName = getBranchName(path_proj)

        if branchName is None:
            raise Exception(f"Branch name not found for project {proj_}")

        per_proj_commit_dict, per_proj_full_defect_list = excavator.runMiner(orgName, proj_, branchName, csv_replication, csv_default)

        categ += per_proj_full_defect_list
        dic[proj_] = (per_proj_commit_dict, per_proj_full_defect_list)

        print(f'Finished analyzing: {proj_}')
        print("="*50)
     except Exception as e:
        print(f"Error processing project {proj_}: {e}")

    def run_in_parallel(orgName, elgibleRepos, pathRepo, dic, categ, csv_replication, csv_default):
     with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = [executor.submit(process_project, orgName, proj_, pathRepo, dic, categ, csv_replication, csv_default) for proj_ in elgibleRepos]

        # Wait for all tasks to complete and handle exceptions
        for future in concurrent.futures.as_completed(futures):
            try:
                future.result()  # Retrieve the result of the task (or raise any exceptions)
            except Exception as e:
                print(f"Task raised an exception: {e}")

    if orgName == 'EXTRA' or orgName == "TEST" or orgName == "PIPR-replication":
        csv_replication = None
        csv_default     = None
    else:
        csv_replication = os.path.abspath(sys.argv[sys.argv.index("--csv-replication") + 1])
        csv_default     = os.path.abspath(sys.argv[sys.argv.index("--csv-default") + 1])

    script_dir = os.path.dirname(os.path.abspath(__file__))
    pathRepo = script_dir + "/" + constants.DATASET_DIR + "/" + orgName + "/"
    fileName = pathRepo + "/" + constants.REPO_FILE_LIST
    elgibleRepos = excavator.getEligibleProjects(fileName)
    dic   = {}
    categ = []
    run_in_parallel(orgName, elgibleRepos, pathRepo, dic, categ, csv_replication, csv_default)

    all_proj_df = pd.DataFrame(categ)
    all_proj_df.to_csv(out_csv_fil, header=['HASH','CATEG','REPO','TIME'], index=False)

    with open(out_pkl_fil, 'wb') as fp_:
        pickle.dump(dic, fp_)
    print('*'*100)
    print('Ended at:', giveTimeStamp())
    print('*'*100)
    t2 = time.time()
    time_diff = round( (t2 - t1 ) / 60, 5)
    print("Duration: {} minutes".format(time_diff))
    print('*'*100)
```

Excavator Code

```
from    nltk.tokenize  import sent_tokenize
from    git            import Repo
import  os
import  csv
import  numpy as np
import  sys
import  subprocess
import  constants
import  classifier
import  ast
csv.field_size_limit(sys.maxsize)


def getEligibleProjects(fileNameParam):
  repo_list = []
  with open(fileNameParam, constants.FILE_READ_MODE) as f:
    reader = csv.reader(f)
    for row in reader:
      repo_list.append(row[0])
  return repo_list

def getPuppetFilesOfRepo(repo_dir_absolute_path):
    pp_, non_pp = [], []
    for root_, dirs, files_ in os.walk(repo_dir_absolute_path):
      for file_ in files_:
        full_p_file = os.path.join(root_, file_)
        if((os.path.exists(full_p_file)) and (constants.AST_PATH not in full_p_file) ):
#             if (full_p_file.endswith(constants.PP_EXTENSION)):
          if any(full_p_file.endswith(ext) for ext in constants.IAC_FILES):
            pp_.append(full_p_file)
    return pp_

def getRelPathOfFiles(all_pp_param, repo_dir_absolute_path):
  common_path = repo_dir_absolute_path
  files_relative_paths = [os.path.relpath(path, common_path) for path in all_pp_param]
  return files_relative_paths

def getPuppRelatedCommits(repo_dir_absolute_path, ppListinRepo, branchName=constants.MASTER_BRANCH):
  mappedPuppetList=[]
  track_exec_cnt = 0
  repo_  = Repo(repo_dir_absolute_path)
  all_commits = list(repo_.iter_commits(branchName))
  for each_commit in all_commits:
    track_exec_cnt = track_exec_cnt + 1

    cmd_of_interrest1 = constants.CHANGE_DIR_CMD + repo_dir_absolute_path + " ; "
    cmd_of_interrest2 = constants.GIT_COMM_CMD_1 + str(each_commit)  +  constants.GIT_COMM_CMD_2
    cmd_of_interrest = cmd_of_interrest1 + cmd_of_interrest2
    commit_of_interest  = str(subprocess.check_output([constants.BASH_CMD, constants.BASH_FLAG, 
cmd_of_interrest])) #in Python 3 subprocess.check_output returns byte

    for ppFile in ppListinRepo:
      if ppFile in commit_of_interest:
        file_with_path = os.path.join(repo_dir_absolute_path, ppFile)
        mapped_tuple = (file_with_path, each_commit)
        mappedPuppetList.append(mapped_tuple)

  return mappedPuppetList

def IacRelatedCommits(repo_dir_absolute_path, iac_list_repo, branchName=constants.MASTER_BRANCH):
  mapped_iac_list = []
  track_exec_cnt  = 0
  repo_  = Repo(repo_dir_absolute_path)
  all_commits = list(repo_.iter_commits(branchName))
  for each_commit in all_commits:
    track_exec_cnt = track_exec_cnt + 1

    cmd_of_interrest1 = constants.CHANGE_DIR_CMD + repo_dir_absolute_path + " ; "
    cmd_of_interrest2 = constants.GIT_COMM_CMD_1 + str(each_commit)  +  constants.GIT_COMM_CMD_2
    cmd_of_interrest = cmd_of_interrest1 + cmd_of_interrest2
    commit_of_interest  = str(subprocess.check_output([constants.BASH_CMD, constants.BASH_FLAG, 
cmd_of_interrest])) #in Python 3 subprocess.check_output returns byte

    for iac_file in iac_list_repo:
      if iac_file in commit_of_interest:
```

```python
        file_with_path = os.path.join(repo_dir_absolute_path, iac_file)
        mapped_tuple = (file_with_path, each_commit)
        mapped_iac_list.append(mapped_tuple)

  return mapped_iac_list

def getDiffStr(repo_path_p, commit_hash_p, file_p):
    cdCommand = f"{constants.CHANGE_DIR_CMD} {repo_path_p} ; "
    theFile = os.path.relpath(file_p, repo_path_p)

    diffCommand = f"{constants.GIT_SHOW_CMD} {commit_hash_p} -- {theFile}"
    command2Run = cdCommand + diffCommand

    diff_output = subprocess.check_output(
        [constants.BASH_CMD, constants.BASH_FLAG, command2Run]
    ).decode('utf-8')
    return diff_output

def makeDepParsingMessage(m_, i_):
    upper, lower  = 0, 0
    lower = i_ - constants.STR_LIST_BOUNDS
    upper = i_ + constants.STR_LIST_BOUNDS
    if upper > len(m_):
      upper = - 1
    if lower < 0:
      lower = 0
    return constants.WHITE_SPACE.join(m_[i_ - constants.STR_LIST_BOUNDS : i_ + constants.STR_LIST_BOUNDS])

def processMessage(indi_comm_mess):
    splitted_messages = []
#   original
#    if ('*' in indi_comm_mess):
    if ('*' in indi_comm_mess) or (';' in indi_comm_mess):
      splitted_messages = indi_comm_mess.split('*')
      splitted_messages = indi_comm_mess.split(';')
    else:
      splitted_messages = sent_tokenize(indi_comm_mess)
    return splitted_messages

def analyzeCommit(repo_path_param, iac_commits_mapping):
  verbose = False   # For oracle dataset it is True (later), otherwise it is False
  pupp_bug_list = []
  all_commit_file_dict  = {}
  all_defect_categ_list = []
  hash_tracker = []
  for tuple_ in iac_commits_mapping:

    file_ = tuple_[0]
    commit_ = tuple_[1]
    msg_commit =  commit_.message

    msg_commit = msg_commit.replace('\n', constants.WHITE_SPACE)
    msg_commit = msg_commit.replace(',',  ';')
    msg_commit = msg_commit.replace('\t', constants.WHITE_SPACE)
    msg_commit = msg_commit.replace('&',  ';')
    msg_commit = msg_commit.replace('#',  constants.WHITE_SPACE)
    msg_commit = msg_commit.replace('=',  constants.WHITE_SPACE)

    commit_hash = commit_.hexsha

    # '''
    # for testing purpose , uncomment only for tool accuracy purpose
    # '''
    # if commit_hash in constants.ORACLE_HASH_CHECKLIST:
    #   verbose = True
    # else:
    #   verbose = False
    # '''
    # '''

    timestamp_commit = commit_.committed_datetime
    str_time_commit  = timestamp_commit.strftime(constants.DATE_TIME_FORMAT) ## date with time

    #### categorization zone
    per_commit_defect_categ_list = []
    if (commit_hash not in hash_tracker):
      bug_status, index_status = classifier.detectBuggyCommit(msg_commit, verbose)
      # print bug_status
      #if commit_hash == "1f03639bcddb66031b16ed6cfdf91f2bbdeca6c8":
        #bug_status = True
```

```python
   if (bug_status) or (classifier.detectRevertedCommit(msg_commit) ):
    processed_message = processMessage(msg_commit)
    # each commit has multiple messages, need to merge them together in one list here, not in classifier
    for tokenized_msg in processed_message:
      diff_content_str = getDiffStr(repo_path_param, commit_hash, file_)
      bug_categ = classifier.detectCateg(tokenized_msg, diff_content_str, verbose)
      # bug_categ = classifier.detectCategHashFounder(tokenized_msg, diff_content_str, verbose, hash=commit_hash)
      if len(bug_categ) == 0:
       per_commit_defect_categ_list.append(constants.BUGGY_COMMIT)
      else:
       per_commit_defect_categ_list += bug_categ
   else:
    per_commit_defect_categ_list  = [constants.NO_DEFECT_CATEG]

   bug_categ_list = np.unique(per_commit_defect_categ_list)
   '''
   for testing purpose , uncomment only for tool accuracy purpose
   '''
   # if verbose:
   #   print bug_categ_list
   '''
   '''
   if (len(bug_categ_list) > 0):
    for bug_categ_ in bug_categ_list:
      tup_ = (commit_hash, bug_categ_, file_, str_time_commit)
      all_defect_categ_list.append(tup_)
      # -- my debug
      #if (tup_[1] != constants.NO_DEFECT_CATEG):
      #  print(tup_[1])
      # ---
      # print tup_[0], tup_[1], tup_[2], tup_[3]
      # print '-'*25
   else:
    tup_ = (commit_hash, constants.NO_DEFECT_CATEG, file_, str_time_commit)
    all_defect_categ_list.append(tup_)

   hash_tracker.append(commit_hash)
  #### file to hash mapping zone
  if commit_hash not in all_commit_file_dict:
   all_commit_file_dict[commit_hash] = [file_]
  else:
   all_commit_file_dict[commit_hash]  = all_commit_file_dict[commit_hash] + [file_]

 return all_commit_file_dict, all_defect_categ_list

def getIacFilesOfRepo(repo_id, csv_file_path = constants.CSV_REPLICATION, csv_default = constants.CSV_DEFAULT_PATH):
  with open(csv_file_path, 'r', encoding='utf-8') as csv_file:
    reader = csv.DictReader(csv_file)
    for row in reader:
      if row['id'] == str(repo_id):
        # Converter strings de listas para listas reais
        iac_paths = ast.literal_eval(row['iac_paths'])
        related_files = ast.literal_eval(row['related_files'])
        result = iac_paths + related_files
        result = [item.replace(csv_default + "/" + repo_id + '/', '') for item in result]
        return result
  return None, None

def getId(path):
 return path.split('/')[-1]

def runMiner(orgParamName, repo_name_param, branchParam, csv_file_path = None, csv_default = None):
 script_dir = os.path.dirname(os.path.abspath(__file__))
 repo_path   = script_dir + "/" + constants.DATASET_DIR + "/" + orgParamName + "/" + repo_name_param
 repo_branch = branchParam

 repo_id = getId(repo_path)
 all_iac_files_in_repo = getIacFilesOfRepo(repo_id, csv_file_path, csv_default)
 iac_commits_in_repo = IacRelatedCommits(repo_path, all_iac_files_in_repo, repo_branch)
 commit_file_dict, categ_defect_list = analyzeCommit(repo_path, iac_commits_in_repo)

 return commit_file_dict, categ_defect_list



def dumpContentIntoFile(strP, fileP):
 fileToWrite = open( fileP, constants.FILE_WRITE_MODE)
 fileToWrite.write(strP )
```

```
    fileToWrite.close()
    return str(os.stat(fileP).st_size)

```

Difference Parser Code

```
#reff: https://github.com/cscorley/whatthepatch
import whatthepatch

import constants

from fuzzywuzzy import fuzz

import re

# [(1, None, '# == Class cdh4::pig'), (None, 1, '# == Class cdh::pig'), (2, 2, '#'), (3, None, '# Installs and configures Apache Pig.'), (None, 3, '# Installs and configures Apache Pig and Pig DataFu.'), (4, 4, '#'), (5, None, 'class cdh4::pig {'), (6, None, "  package { 'pig':"), (7, None, "    ensure => 'installed',"), (8, None, '  }'), (None, 5, 'class cdh::pig('), (None, 6, "    $pig_properties_template = 'cdh/pig/pig.properties.erb',"), (None, 7, "    $log4j_template = 'cdh/pig/log4j.properties.erb',"), (None, 8, ')'), (None, 9, '{'), (None, 10, '    # cdh::pig requires hadoop client and configs are installed.'), (None, 11, "    Class['cdh::hadoop'] -> Class['cdh::pig']"), (9, 12, ''), (10, None, "  file { '/etc/pig/conf/pig.properties':"), (11, None, "    content => template('cdh4/pig/pig.properties.erb'),"), (12, None, "    require => Package['pig'],"), (13, None, '  }'), (None, 13, "    package { 'pig':"), (None, 14, "        ensure => 'installed',"), (None, 15, '    }'), (None, 16, "    package { 'pig-udf-datafu':"), (None, 17, "        ensure => 'installed',"), (None, 18, '    }'), (None, 19, ''), (None, 20, '    $config_directory = "/etc/pig/conf.${cdh::hadoop::cluster_name}"'), (None, 21, '    # Create the $cluster_name based $config_directory.'), (None, 22, '    file { $config_directory:'), (None, 23, "        ensure  => 'directory',"), (None, 24, "        require => Package['pig'],"), (None, 25, '    }'), (None, 26, "    cdh::alternative { 'pig-conf':"), (None, 27, "        link    => '/etc/pig/conf',"), (None, 28, '        path    => $config_directory,'), (None, 29, '    }'), (None, 30, ''), (None, 31, '    file { "${config_directory}/pig.properties":'), (None, 32, '        content => template($pig_properties_template),'), (None, 33, "        require => Package['pig'],"), (None, 34, '    }'), (None, 35, '    file { "${config_directory}/log4j.properties":'), (None, 36, '        content => template($log4j_template),'), (None, 37, "        require => Package['pig'],"), (None, 38, '    }'), (14, 39, '}')]

def parseTheDiff(diff_text):
    parse_out_dict = {}
    diff_mess_str = str(diff_text) ## changes for Python 3 migration
    for diff_ in whatthepatch.parse_patch(diff_mess_str):
        all_changes_line_by_line = diff_[1] ## diff_ is a tuple, changes is idnetified by the second index
        line_numbers_added, line_numbers_deleted = [], []
        add_dic, del_dic = {}, {}
        parse_out_dict   = {}
        for change_tuple in all_changes_line_by_line:
            if (change_tuple[0] != None ):
                line_numbers_added.append(change_tuple[0])
                add_dic[change_tuple[0]] = change_tuple[2]
            if (change_tuple[1] != None ):
                line_numbers_deleted.append(change_tuple[1])
                del_dic[change_tuple[1]] = change_tuple[2]
        lines_changed = list(set(line_numbers_added).intersection(line_numbers_deleted))
        for line_number in lines_changed:
            if ((line_number in add_dic) and (line_number in del_dic)):
                parse_out_dict[line_number] = [ del_dic[line_number], add_dic[line_number]] ## <removed content, added cotnent>
        #print parse_out_dict
    return parse_out_dict

def filterTextList(txt_lis):
    return_list = []
    return_list = [x_.lower() for x_ in txt_lis if  constants.HASH_SYMBOL not in x_ ]
    return_list = [x_.replace(constants.TAB, '') for x_ in return_list ]
    return_list = [x_.replace(constants.NEWLINE, '') for x_ in return_list ]
    return_list = [x_ for x_ in return_list if len(x_) > 1 ]
    return return_list

def getAddDelLines(diff_mess):
    added_text, deleted_text = [], []
    diff_mess_str = str(diff_mess)
    try:
        for diff_ in whatthepatch.parse_patch(diff_mess_str):
            all_changes_line_by_line = getattr(diff_, "changes", None)

            if all_changes_line_by_line:
                for change in all_changes_line_by_line:
                    if change.new is not None:
                        added_text.append(change.line)

                    if change.old is not None:
                        deleted_text.append(change.line)

```

```python
    except Exception as e:
        print(f"[ERROR] Error when processing diff: {e}")

    return added_text, deleted_text

def getSpecialConfigDict(text_str_list, splitter):
    dic2ret = {}
    for x_ in text_str_list:
        if (splitter in x_):
            _key_ = x_.replace(constants.WHITE_SPACE, '').split(splitter)[0]
            _val_ = x_.replace(constants.WHITE_SPACE, '').split(splitter)[-1]
            if _key_ not in dic2ret:
                dic2ret[_key_] = _val_
    # print text_str_list
    # print dic2ret
    return dic2ret

def filterConfig(oldValue):
    oldValue = oldValue.replace(",","")
    oldValue = oldValue.replace("'","")
    oldValue = oldValue.replace(";","")
    val_     = oldValue.replace(">","")

    return val_

def getConfigChangeCnt(start_dict, end_dict):
    tracker = 0
    track_list = []
    val_track_list = []
    for k_, v_ in start_dict.items():
        if (k_ in end_dict ) and (k_ not in track_list) and (v_ not in val_track_list) and (len(v_) > 1):
            oldValue     = end_dict[k_]
            newValue     = v_
            # need more pre processign ugh
            oldValue = filterConfig(oldValue)
            newValue = filterConfig(newValue)
            if newValue != oldValue:
                # print k_
                # print oldValue, newValue
                tracker = tracker + 1
        track_list.append(k_)
        val_track_list.append(v_)
    # print '>'*5
    return tracker

def checkDiffForConfigDefects(diff_text):
    added_text , deleted_text = [], []
    final_flag = False
    added_text, deleted_text = getAddDelLines(diff_text)
    added_text   = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)
    config_change_tracker = 0

    valu_assi_dict_addi = getSpecialConfigDict(added_text, constants.VAR_SIGN)
    valu_assi_dict_deli = getSpecialConfigDict(deleted_text, constants.VAR_SIGN)

    attr_assi_dict_addi = getSpecialConfigDict(added_text, constants.ATTR_SIGN)
    attr_assi_dict_deli = getSpecialConfigDict(deleted_text, constants.ATTR_SIGN)

    # config_change_tracker = getConfigChangeCnt(valu_assi_dict_addi, valu_assi_dict_deli) +
getConfigChangeCnt(valu_assi_dict_deli, valu_assi_dict_addi) + getConfigChangeCnt(attr_assi_dict_addi, attr_assi_dict_deli) +
getConfigChangeCnt( attr_assi_dict_deli, attr_assi_dict_addi)
    config_change_tracker = getConfigChangeCnt(valu_assi_dict_addi, valu_assi_dict_deli) +
getConfigChangeCnt(attr_assi_dict_addi, attr_assi_dict_deli)

    if config_change_tracker > 0 :
        final_flag = True


    return final_flag

def checkDiffForDepDefects(diff_text):
    added_text , deleted_text = [], []
    final_flag, final_flag_1, final_flag_2 = False , False, False
    added_text, deleted_text = getAddDelLines(diff_text)
    added_text   = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)
    added_text   = [x_ for x_ in added_text if constants.VAR_SIGN not in x_ ]
    added_text   = [x_ for x_ in added_text if constants.ATTR_SIGN not in x_ ]

    deleted_text   = [x_ for x_ in deleted_text if constants.VAR_SIGN not in x_ ]
    deleted_text   = [x_ for x_ in deleted_text if constants.ATTR_SIGN not in x_ ]
    # print added_text, deleted_text
    added_text   = [z_ for z_ in added_text if any(x_ in z_ for x_ in constants.diff_depen_code_elems ) ]
    deleted_text = [z_ for z_ in deleted_text if any(x_ in z_ for x_ in constants.diff_depen_code_elems ) ]

    if (len(added_text) > 0 ) and (len(deleted_text) > 0 ) :
       final_flag = True

    return final_flag

import re

def has_comment(line):
    # ignore if it's a string
    if re.fullmatch(r'\s*["\'].*["\']\s*', line.strip()):
        return False

    # line's entire comment
    if re.search(r'^\s*(#|//).+', line):
        return True

    # inline comment
    if re.search(r'[^"\']*(#|//).+', line):
        return True

    # blocks comment
    if re.search(r'/\*.*?\*/', line, re.DOTALL):
        return True

    return False

def checkDiffForDocDefects(diff_text):
    lines_changed = []
    final_flag = False
    diff_mess_str = str(diff_text) ## changes for Python 3 migration
    for diff_ in whatthepatch.parse_patch(diff_mess_str):
        all_changes_line_by_line = diff_[1]
        line_numbers_added, line_numbers_deleted = [], []
        if all_changes_line_by_line is not None:
            for change_tuple in all_changes_line_by_line:
                content = change_tuple[2]
                content = content.replace(constants.WHITE_SPACE, '')
                if change_tuple[0] is not None and has_comment(content):
                    line_numbers_added.append(content)
                if change_tuple[1] is not None and has_comment(content):
                    line_numbers_deleted.append(content)
        lines_changed = list(set(line_numbers_added).intersection(line_numbers_deleted))
        # print lines_changed
    lines_changed = [x_ for x_ in lines_changed if len(x_) > 1 ]
    if len(lines_changed) > 0:
        final_flag = True
    return final_flag

def checkDiffForNetwork(diff_text):
    added_text, deleted_text = [], []
    final_flag = False
    added_text, deleted_text = getAddDelLines(diff_text)

    added_text = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)

    added_text = [line for line in added_text if any(keyword in line for keyword in constants.diff_network_elems)]
    deleted_text = [line for line in deleted_text if any(keyword in line for keyword in constants.diff_network_elems)]

    if added_text or deleted_text:
        final_flag = True

    return final_flag

def checkDiffForCredentials(diff_text):
    added_text, deleted_text = [], []
    final_flag = False
    added_text, deleted_text = getAddDelLines(diff_text)

    added_text = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)

    added_text = [line for line in added_text if any(keyword in line for keyword in constants.diff_credentials_kw_list)]
    deleted_text = [line for line in deleted_text if any(keyword in line for keyword in constants.diff_credentials_kw_list)]

    if added_text or deleted_text:
        final_flag = True

    return final_flag

def checkDiffForLogicDefects(diff_text):
    added_text , deleted_text = [], []
    final_flag, final_flag_1, final_flag_2 = False , False, False
    added_text, deleted_text = getAddDelLines(diff_text)
    added_text   = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)
    added_text   = [x_ for x_ in added_text if constants.VAR_SIGN not in x_ ]
    added_text   = [x_ for x_ in added_text if constants.ATTR_SIGN not in x_ ]

    deleted_text   = [x_ for x_ in deleted_text if constants.VAR_SIGN not in x_ ]
    deleted_text   = [x_ for x_ in deleted_text if constants.ATTR_SIGN not in x_ ]
    # print added_text, deleted_text
    added_text   = [z_ for z_ in added_text if any(x_ in z_ for x_ in constants.diff_logic_code_elems ) ]
    deleted_text = [z_ for z_ in deleted_text if any(x_ in z_ for x_ in constants.diff_logic_code_elems ) ]

    if (len(added_text) > 0 ) or (len(deleted_text) > 0 ) :
       final_flag = True
    return final_flag

def checkDiffForSecurityDefects(diff_text):
    final_flag = False
    added_text , deleted_text = [], []

    added_text, deleted_text = getAddDelLines(diff_text)
    added_text   = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)
    added_text   = [x_ for x_ in added_text if constants.VAR_SIGN  in x_ ]
    added_text   = [x_ for x_ in added_text if constants.ATTR_SIGN  in x_ ]

    deleted_text = [x_ for x_ in deleted_text if constants.VAR_SIGN  in x_ ]
    deleted_text = [x_ for x_ in deleted_text if constants.ATTR_SIGN  in x_ ]

    added_text   = [x_.split(constants.VAR_SIGN)[0].replace(constants.WHITE_SPACE, '') for x_ in added_text]
    added_text   = [x_.split(constants.ATTR_SIGN )[0].replace(constants.WHITE_SPACE, '') for x_ in added_text]

    deleted_text   = [x_.split(constants.VAR_SIGN)[0].replace(constants.WHITE_SPACE, '') for x_ in deleted_text]

    deleted_text   = [x_.split(constants.ATTR_SIGN )[0].replace(constants.WHITE_SPACE, '') for x_ in deleted_text]

    added_text   = [z_ for z_ in added_text if any(x_ in z_ for x_ in constants.diff_secu_code_elems ) ]
    deleted_text = [z_ for z_ in deleted_text if any(x_ in z_ for x_ in constants.diff_secu_code_elems ) ]
    # print added_text, deleted_text
    if (len(added_text) > 0) or (len(deleted_text) > 0):
        final_flag = True
    return final_flag

def checkDiffForServiceDefects(diff_text):
    final_flag = False
    added_text , deleted_text = [], []

    added_text, deleted_text = getAddDelLines(diff_text)
    added_text   = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)
    added_text   = [x_ for x_ in added_text if constants.VAR_SIGN  not in x_ ]
    added_text   = [x_.lower() for x_ in added_text if constants.ATTR_SIGN not in x_ ]

    deleted_text = [x_ for x_ in deleted_text if constants.VAR_SIGN not in x_ ]
    deleted_text = [x_.lower() for x_ in deleted_text if constants.ATTR_SIGN not in x_ ]

    added_text   = [z_ for z_ in added_text if any(x_ in z_ for x_ in constants.diff_service_code_elems) ]
    deleted_text = [z_ for z_ in deleted_text if any(x_ in z_ for x_ in constants.diff_service_code_elems) ]

    if (len(added_text) > 0 ) and (len(deleted_text) > 0 ) :
       final_flag = True
    return final_flag

def matchStringsFuzzily(add_str_lis, del_str_lis):
    # takes two sring as input, returns levesjhteins's ratio, reff: https://www.datacamp.com/community/tutorials/fuzzy-string-python
    add_str = constants.WHITE_SPACE.join(add_str_lis)
    del_str = constants.WHITE_SPACE.join(del_str_lis)
    lower_add_str = add_str.lower()
    lower_del_str = del_str.lower()
    lev_str_ratio = fuzz.token_sort_ratio( lower_add_str, lower_del_str  ) ## this is levenshteien ratio, in a
sorted manner
    return lev_str_ratio


def checkDiffForSyntaxDefects(diff_text):
    final_flag = False
    added_text , deleted_text = [], []
    attr_added_text , attr_deleted_text = [], []
    var_added_text , var_deleted_text = [], []

    added_text, deleted_text = getAddDelLines(diff_text)
    added_text   = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)

    '''
    look for variable name change
    '''
    attr_added_text   = [x_.lower() for x_ in added_text if constants.ATTR_SIGN in x_ ]
    var_added_text    = [x_.lower().replace(constants.WHITE_SPACE, '') for x_ in added_text if
constants.VAR_SIGN in x_ ]

    attr_deleted_text = [x_.lower() for x_ in deleted_text if constants.ATTR_SIGN in x_ ]
    var_deleted_text  = [x_.lower().replace(constants.WHITE_SPACE, '') for x_ in deleted_text if
constants.VAR_SIGN in x_ ]
    '''
    Now compare
    '''

    # if (len(added_text)) and (len(deleted_text)): ## wrong logic
    if ((len(attr_added_text)) == (len(attr_deleted_text))) or (len(var_added_text) == len(var_deleted_text) )
: ## right logic , same number of additions and deletiosn for variables
        final_flag = True
    elif ( (matchStringsFuzzily(attr_added_text, attr_deleted_text) > constants.lev_cutoff ) or
(matchStringsFuzzily(var_added_text, var_deleted_text) > constants.lev_cutoff ) ):
        # Why does the original author uses minus?
        # final_flag - True
        final_flag = True


    return final_flag



def checkDiffForIdempotenceDefects(diff_text):
    final_flag = False
    added_text , deleted_text = [], []

    added_text, deleted_text = getAddDelLines(diff_text)
    added_text   = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)

    added_text = [x_ for x_ in added_text if constants.diff_idem_code_elem in x_ ]
    if (len(added_text) == 1) or (len(deleted_text) > constants.diff_idem_removal_cnt):
       final_flag = True

    return final_flag

def checkDiffForIdemWithAttr(diff_text):
    final_flag = False
    flag_list  = []
    added_text , deleted_text = [], []

    added_text, deleted_text = getAddDelLines(diff_text)
    added_text   = filterTextList(added_text)
    deleted_text = filterTextList(deleted_text)

    if(len(deleted_text) < len(added_text)):
        for text_ in added_text:
            for elem in constants.diff_extra_idem_elems:
                if elem in text_:
                    flag_list.append(True)
    if (len(flag_list) > 0):
        final_flag = True
    return final_flag

```

Classifier Code

```python
import constants
import diff_parser
import re
import spacy
spacy_engine = spacy.load(constants.SPACY_ENG_DICT)
from nltk.stem.porter import *
stemmerObj = PorterStemmer()



def checkForNum(str_par):
    return any(char_.isdigit() for char_ in str_par)

def filterCommitMessage(msg_par):
    temp_msg_    = msg_par.lower()
    splitted_msg = temp_msg_.split(constants.WHITE_SPACE)
    splitted_msg = [stemmerObj.stem(x_) for x_ in splitted_msg] ##porter stemming , x_ is a string
    splitted_msg = [x_ for x_ in splitted_msg if len(x_) > 1 ]  ## remove special characterers , x_ is a string
    # splitted_msg = [x_ for x_ in splitted_msg if x_.isalnum() ]  ## remove special characterers , x_ is a string
    filtered_msg = [x_ for x_ in splitted_msg if checkForNum(x_) == False ] ## remove alphanumeric characters , x_ is a string

    return filtered_msg

def doDepAnalysis(msg_par):
    msg_to_analyze = []
    filtered_msg = filterCommitMessage(msg_par)
    unicode_msg_ = constants.WHITE_SPACE.join(filtered_msg)
    try:
        unicode_msg  = str(unicode_msg_, constants.UTF_ENCODING)
    except:
        unicode_msg = unicode_msg_
    # print unicode_msg
    spacy_doc = spacy_engine(unicode_msg)
    for token in spacy_doc:
        if (token.dep_ == constants.ROOT_TOKEN):
            for x_ in token.children:
                msg_to_analyze.append(x_.text)
    return constants.WHITE_SPACE.join(msg_to_analyze)


def doTempCleanUp(msg_str):
    msg_ = msg_str.replace(constants.CLOSE_KW, constants.WHITE_SPACE)
    msg_ = msg_.replace(constants.MERGE_KW, constants.WHITE_SPACE)
    msg_ = msg_.replace(constants.DFLT_KW, constants.WHITE_SPACE)

    return msg_

def detectBuggyCommit(msg_, verboseFlag = False):
    flag2ret  = False
    index2ret = 0
    msg_ = msg_.lower()

    if (constants.IDEM_XTRA_KW in msg_) or (constants.SYNTAX_XTRA_KW2 in msg_):
        msg_ = doTempCleanUp(msg_)

    if(any(x_ in msg_ for x_ in constants.prem_bug_kw_list)) and ( constants.DFLT_KW not in msg_) and (constants.MERGE_KW not in msg_) :
        str2see = [y_ for y_ in constants.prem_bug_kw_list][0]
        index2ret = msg_.find( str2see  )
        flag2ret = True
    return flag2ret, index2ret

def detectRevertedCommit(msg_):
    flag2ret  = False
    msg_ = msg_.lower()
    revert_matches = re.findall(constants.REVERT_REGEX, msg_)
    if(len(revert_matches) > 0):
        flag2ret = True
    return flag2ret

def categ_check(key_words=[], msg=None, diff_function=None, diff=None, classification=constants.NO_DEFECT_CATEG):
    if any(kw in msg for kw in key_words) or (diff_function != None and diff_function(diff)):
        return classification
    return constants.NO_DEFECT_CATEG
'''
```

```
detectCateg takes a sentence and a diff from a commit message as input , and return a defect category (single value)
'''
def detectCateg(msg_, diff_, verboseFlag=False,hash=None):
    temp_msg_ = '' ## for oracle dataset
    defect_categ_to_ret = set()

    if (len(diff_) > 0):
        temp_msg_list   = filterCommitMessage(msg_) # for extra false negative rules
        temp_msg_       = constants.WHITE_SPACE.join(temp_msg_list) # for extra false negative rules
        msg_            = doDepAnalysis(msg_) ## depnding on results, this extra step of dependnecy parsing may change


        for classification, key_words, diff_function in constants.CLASSIFICATION_PARSE:
            categ_check_classification = categ_check(msg=msg_, diff=diff_, key_words=key_words, diff_function=diff_function, classification=classification)
            if categ_check_classification != constants.NO_DEFECT_CATEG and categ_check_classification != constants.BUGGY_COMMIT:
                defect_categ_to_ret.add(categ_check_classification)

        # These are the extra rules that the author selected

        # extra rule for idempotence
        if ( constants.IDEM_XTRA_KW in temp_msg_ ) and ( constants.EXTRA_FIX_KEYWORD in temp_msg_ ) or any(_ in temp_msg_ for _ in constants.idem_defect_kw_list):
            defect_categ_to_ret.add(constants.IDEM_DEFECT_CATEG)

        # extra rule for conditional
        if (( constants.LOGIC_XTRA_KW1 in temp_msg_ ) or ( constants.LOGIC_XTRA_KW2 in temp_msg_ ) or ( constants.LOGIC_XTRA_KW3 in temp_msg_ ) ) and ( constants.EXTRA_FIX_KEYWORD in temp_msg_ ):
            defect_categ_to_ret.add(constants.CONDI_DEFECT_CATEG )

        # extra rule for syntax
        if any(kw in temp_msg_ for kw in constants.syxtax_xtra_kw_list) and (( constants.EXTRA_FIX_KEYWORD in temp_msg_ )  ):
            defect_categ_to_ret.add(constants.SYNTAX_DEFECT_CATEG)

        # extra rule for doc
        if ( constants.DOC_XTRA_KW in temp_msg_   ) and ( constants.EXTRA_FIX_KEYWORD in temp_msg_ ):
            defect_categ_to_ret.add(constants.DOC_DEFECT_CATEG)

        if any(kw in temp_msg_ for kw in constants.dep_xtra_kw_list) and (( constants.EXTRA_FIX_KEYWORD in temp_msg_ )  ):
            defect_categ_to_ret.add(constants.DEP_DEFECT_CATEG)

        if any(kw in temp_msg_ for kw in constants.resource_xtra_kw_list) and (( constants.EXTRA_FIX_KEYWORD in temp_msg_ )  ):
            defect_categ_to_ret.add(constants.SERVICE_RESOURCE_DEFECT_CATEG)

        if any(kw in temp_msg_ for kw in constants.network_extra_kw_list) and (( constants.EXTRA_FIX_KEYWORD in temp_msg_ )  ):
            defect_categ_to_ret.add(constants.NETWORK_DEFECT_CATEG)

        if any(kw in temp_msg_ for kw in constants.storage_extra_kw_list) and (( constants.EXTRA_FIX_KEYWORD in temp_msg_ )  ):
            defect_categ_to_ret.add(constants.STORAGE_DEFECT_CATEG)

        if any(kw in temp_msg_ for kw in constants.credentials_extra_kw_list) and (( constants.EXTRA_FIX_KEYWORD in temp_msg_ )  ):
            defect_categ_to_ret.add(constants.CREDENTIALS_DEFECT_CATEG)

    return list(defect_categ_to_ret)
```

Constants Code

```
import diff_parser

DATASET_DIR  = 'dataset'
REPO_FILE_LIST = 'eligible_repos.csv'
MASTER_BRANCH  = 'main'
FILE_READ_MODE = 'r'
AST_PATH = 'EXTRA_AST'
PP_EXTENSION = '.pp'
IAC_FILES = [
    "Pulumi.yaml", "Pulumi.yml", "cdk.json", "cdktf.json",
    ".py", ".go", ".js", ".ts", ".java", ".tf",
    ".cs", ".fs", ".vb", ".cpp", ".kt", ".php", ".rb", ".swift", ".abap", ".edn"
]
DATE_TIME_FORMAT = "%Y-%m-%dT%H-%M-%S"
WHITE_SPACE  = ' '
```

```python
TAB = '\t'
NEWLINE = '\n'
HASH_SYMBOL = '#'
comments_elems = ['#', '//', '/*', '*/']
CSV_REPLICATION = '/home/aluno/ACID-dataset/ARTIFACT/IaC_Defect_Categ_Revamp/replication/iac_time_period.csv'
CSV_DEFAULT_PATH = '/home/aluno/filtered-repositories-iac-criteria/criteria4/'

CHANGE_DIR_CMD = 'cd '
GIT_COMM_CMD_1 = "git show --name-status "
GIT_COMM_CMD_2 = " | awk
'/(Pulumi\\.yaml|Pulumi\\.yml|cdk\\.json|cdktf\\.json|\\.py|\\.go|\\.js|\\.ts|\\.java|\\.tf|\\.cs|\\.fs|\\.vb|\\.cpp|\\.kt|\\.php|\\.rb|\
 {print $2}'"
BASH_CMD = 'bash'
BASH_FLAG = '-c'
GIT_SHOW_CMD = "git show"
GIT_DIFF_CMD = " git diff"
HG_REV_SPECL_CMD   = " ; hg log -p -r "

ENCODING = 'utf8'
UTF_ENCODING = 'utf-8'
FILE_WRITE_MODE = 'w'
SPACY_ENG_DICT  = 'en_core_web_sm'
ROOT_TOKEN = 'ROOT'

STR_LIST_BOUNDS  = 3 # tri-grams
NO_DEFECT_CATEG        = 'NO_DEFECT'
BUGGY_COMMIT           = 'BUGGY_COMMIT'
prem_bug_kw_list      = ['error', 'bug', 'fix', 'issu', 'mistake', 'incorrect', 'fault', 'defect', 'flaw',
'solve' ]

CONFIG_DEFECT_CATEG    = 'CONFIG_DATA_DEFECT'
config_defect_kw_list       = ['value', 'config', 'option',  'setting', 'hiera', 'data']

DEP_DEFECT_CATEG       = 'DEP_DEFECT'
dep_defect_kw_list    =     ['requir', 'depend', 'relation', 'order', 'sync', 'compatibil', 'ensur',
'inherit']
dep_defect_kw_list    +=    ['version', 'deprecat', 'packag', 'path', 'modul', 'upgrad', 'updat']
dep_xtra_kw_list       = ['module']

# Retirado import por causa de port em network
# These are used on diff_parser
VAR_SIGN = '='
ATTR_SIGN = '=>'
diff_depen_code_elems = ['->' , '::', 'include', 'packag', 'exec', 'require', 'import', 'version']

DOC_DEFECT_CATEG      = 'DOC_DEFECT'
# origal author doc_defect_kw_list    =     ['doc', 'comment', 'spec', 'license', 'copyright', 'notice',
'header', 'readme']
# changed made removing 'spec' and header
doc_defect_kw_list    =     ['doc', 'comment', 'licens', 'copyright', 'notic', 'readm']
doc_defect_kw_list    +=    ['descript']

IDEM_DEFECT_CATEG     = 'IDEM_DEFECT'
idem_defect_kw_list   =     ['idempot']
idem_defect_kw_list   +=    ['determin']

diff_idem_code_elem    = 'class'
diff_idem_removal_cnt  = 10

CONDI_DEFECT_CATEG     = 'CONDITIONAL_DEFECT'
logic_defect_kw_list  =     ['logic', 'condition', 'bool']

diff_logic_code_elems = ['if' , 'unless', 'els', 'case']
diff_logic_code_elems+= ['while', 'elif']

SECU_DEFECT_CATEG           = 'SECURITY_DEFECT'
secu_defect_kw_list         =     ['vulner', 'ssl', 'secr', 'authent', 'password', 'security', 'cve']
# secu_defect_kw_list         +=    ['cert', 'firewall', 'encrypt', 'protect']
# adding access to control
secu_defect_kw_list         +=    ['cert', 'firewall', 'encrypt', 'protect', 'access']

diff_secu_code_elems        = ['tls', 'cert', 'cred', 'ssl', 'password', 'pass', 'pwd']

NETWORK_DEFECT_CATEG        = 'CD_NETWORK_DEFECT'
# removing address that was from the original author
# network_defect_kw_list      = ['network', 'address', 'port', 'tcp', 'dhcp', 'ssh', 'gateway', 'connect']
network_defect_kw_list      = ['network', 'port', 'tcp', 'dhcp', 'ssh', 'gateway', 'connect']
network_defect_kw_list     += ['rout']
diff_network_elems          = ['url', 'vpc', 'subnet', 'endpoint']
network_extra_kw_list       = ['gateway']
```

```python
# ip tem que sair por causa de descrIPt em doc

STORAGE_DEFECT_CATEG        = 'CD_STORAGE_DEFECT'
storage_defect_kw_list      = ['sql', 'db', 'databas', 'disk']
storage_extra_kw_list       = ['disk']
# retirar database por causa de data em configuration

CACHE_DEFECT_CATEG          = 'CD_CACHE_DEFECT'
cache_defect_kw_list        = ['cach', 'memory', 'buffer', 'evict', 'ttl']

CREDENTIALS_DEFECT_CATEG    = 'CD_CREDENTIAL_DEFECT'
credentials_defect_kw_list  = ['polic', 'credential', 'iam', 'role', 'token', 'user', 'usernam', 'password']
credentials_extra_kw_list   = ['polic']
diff_credentials_kw_list    = ['polic', 'credential']

FILE_SYSTEM_DEFECT_CATEG    = 'CD_FILE_SYSTEM_DEFECT'
file_system_defect_kw_list  = ['file', 'permiss']

SYNTAX_DEFECT_CATEG    = 'SYNTAX_DEFECT'
syntax_defect_kw_list    = ['compil', 'lint', 'warn', 'typo', 'spell', 'indent', 'regex', 'duplicat', 'variabl', 'whitespac']
syntax_defect_kw_list   += ['type', 'format', 'naming', 'casing', 'styl', 'comma', 'pattern', 'quot']
# retirar name por causa de username

SERVICE_RESOURCE_DEFECT_CATEG        = 'SERVICE_RESOURCE_DEFECT'
resource_defect_kw_list     = ['servic', 'server', 'location', 'resourc', 'provi', 'cluster']
resource_xtra_kw_list       = ['kube','cloud']
diff_service_code_elems     = ['service']

SERVICE_PANIC_DEFECT_CATEG          = 'SERVICE_PANIC_DEFECT'
panic_defect_kw_list = ['check', 'deploy', 'reboot', 'build', 'mount', 'kernel', 'extran', 'bypass']

CLASSIFICATION_PARSE = [
    (CONDI_DEFECT_CATEG,            logic_defect_kw_list,       diff_parser.checkDiffForLogicDefects),
    (IDEM_DEFECT_CATEG,             idem_defect_kw_list,        None),
    (DOC_DEFECT_CATEG,              doc_defect_kw_list,         diff_parser.checkDiffForDocDefects),
    (SYNTAX_DEFECT_CATEG,           syntax_defect_kw_list,      None),
    (SECU_DEFECT_CATEG,             secu_defect_kw_list,        diff_parser.checkDiffForSecurityDefects),
    (DEP_DEFECT_CATEG,              dep_defect_kw_list,         diff_parser.checkDiffForDepDefects),
    (CONFIG_DEFECT_CATEG,           config_defect_kw_list,      diff_parser.checkDiffForConfigDefects),
    (NETWORK_DEFECT_CATEG,          network_defect_kw_list,     diff_parser.checkDiffForNetwork),
    (STORAGE_DEFECT_CATEG,          storage_defect_kw_list,     None),
    (CACHE_DEFECT_CATEG,            cache_defect_kw_list,       None),
    (FILE_SYSTEM_DEFECT_CATEG,      file_system_defect_kw_list, None),
    (CREDENTIALS_DEFECT_CATEG,      credentials_defect_kw_list, diff_parser.checkDiffForCredentials),
    (SERVICE_RESOURCE_DEFECT_CATEG, resource_defect_kw_list,    diff_parser.checkDiffForServiceDefects),
    (SERVICE_PANIC_DEFECT_CATEG,    panic_defect_kw_list,       None)
]

EXTRA_FIX_KEYWORD     = 'fix'
EXTRA_BUG_KEYWORD     = 'bug'

DFLT_KW         = 'default'
CLOSE_KW        = 'closes-bug'
MERGE_KW        = 'merge'
REVERT_KW       = 'revert'
REVERT_REGEX    = r'^revert.*\".*\"'

IDEM_XTRA_KW    = 'idempot' # for detectBuggyCommit()

LOGIC_XTRA_KW1  = 'condit'
LOGIC_XTRA_KW2  = 'logic'
LOGIC_XTRA_KW3  = 'bool'

SYNTAX_XTRA_KW1 = 'lint'
SYNTAX_XTRA_KW2 = 'typo'
SYNTAX_XTRA_KW4 = 'syntax'
syxtax_xtra_kw_list = ['lint', 'typo', 'syntax','type']
DOC_XTRA_KW     = 'notice'
# DEPEND_XTRA_KW  = 'override'
# NETWORK_XTRA_KW = 'provis'

diff_config_code_elems = ['hiera' , 'hash', 'parameter']

diff_extra_idem_elems  = ['ensure', 'unless', 'creates', 'replace']

lev_cutoff = 75

'''
Oracle dataset work
```

```
'''
ORACLE_HASH_CHECKLIST = ['75e460ab929a76e9e4a8d42740a529b3a476e952',
                         '9a5a540738f887f87886ae4f9f52d5ade1b26bc7',
                         '0d834093814b3d184eff36b2835530a847ee6421',
                         '854e0e7b9fc339dc56bf3e2b3de7107c3f35b835',
                         'a7dedf197a24bf8a3fad00d1d1f58eede2f43057',
                         '114536ef2e7c569300019844e0ca57d278e27791'
                        ]
```

Criteria Code

```python
import os
import sys
import subprocess
import pandas as pd
from concurrent.futures import ThreadPoolExecutor

iac_extensions = [".tf", "Pulumi.yaml", "Pulumi.yml", "cdk.json", "cdktf.json", ".edn"]

def is_not_fork(repo_path):
    config_file = os.path.join(repo_path, ".git", "config")
    if not os.path.exists(config_file):
        print(f"[WARNING] {repo_path}: .git/config não encontrado.")
        return None
    with open(config_file, "r") as f:
        is_fork = "fork = true" not in f.read()
        print(f"[INFO] {repo_path}: Fork? {'Não' if is_fork else 'Sim'}")
        return is_fork

def iac_percentage(repo_path):
    total_files = 0
    iac_files = 0
    iac_directories = set()
    for root, _, files in os.walk(repo_path):
        has_iac = any(file_.endswith(ext) for file_ in files for ext in iac_extensions)
        if has_iac:
            iac_directories.add(root)
        total_files += len(files)
    for iac_dir in iac_directories:
        iac_files += sum(len(files) for _, _, files in os.walk(iac_dir))
    percentage = (iac_files / total_files) * 100 if total_files > 0 else 0
    print(f"[INFO] {repo_path}: IaC = {iac_files}/{total_files} arquivos ({percentage:.2f}%)")
    return percentage

def commits_per_month(repo_path):
    result = subprocess.run(
        ["git", "log", "--date=format:%Y-%m", "--pretty=format:%ad"],
        cwd=repo_path,
        stdout=subprocess.PIPE,
        text=True
    )
    dates = result.stdout.splitlines()
    unique_months = set(dates)
    cpm = len(dates) / len(unique_months) if unique_months else 0
    print(f"[INFO] {repo_path}: {len(dates)} commits em {len(unique_months)} meses = {cpm:.2f} commits/mês")
    return cpm

def num_contributors(repo_path):
    result = subprocess.run(
        ["git", "log", "--pretty=format:%ae"],
        cwd=repo_path,
        stdout=subprocess.PIPE,
        text=True
    )
    all_emails = set(result.stdout.splitlines())
    filtered = {email for email in all_emails if not email.endswith("@github.com")}
    print(f"[INFO] {repo_path}: {len(filtered)} contribuidores (sem bots/github)")
    return len(filtered)

def analyze_repo(repo, dataset_dir, input_dir, output_dir, filters):
    repo_path = os.path.join(dataset_dir, repo)
    if not os.path.isdir(os.path.join(repo_path, ".git")):
        print(f"[WARNING] {repo_path} não é um repositório Git válido. Ignorando.")
        return {"repo": repo, "status": "Not a Git repo"}

    input_repos = os.listdir(input_dir) if input_dir else []
    is_input = repo in input_repos

    results = {"repo": repo}
    print(f"[INFO] Analisando repositório: {repo}")

    if filters["--fork"]:
        results["is_not_fork"] = is_not_fork(repo_path)
    if filters["--iac-percentage"]:
        results["iac_percentage"] = iac_percentage(repo_path)
    if filters["--commits-per-month"]:
        results["commits_per_month"] = commits_per_month(repo_path)
    if filters["--num-contributors"]:
        results["num_contributors"] = num_contributors(repo_path)

    passed = True
    if filters["--fork"] and not results.get("is_not_fork", False):
        passed = False
    if filters["--iac-percentage"] and (results.get("iac_percentage") or 0) < 11:
        passed = False
    if filters["--commits-per-month"] and (results.get("commits_per_month") or 0) < 2:
        passed = False
    if filters["--num-contributors"] and (results.get("num_contributors") or 0) < 10:
        passed = False

    results["passed"] = passed
    print(f"[INFO] {repo}: {'PASSOU' if passed else 'NÃO passou'} nos filtros.")

    if passed and is_input:
        target_path = os.path.join(output_dir, repo)
        try:
            if not os.path.exists(target_path):
                os.symlink(os.path.abspath(repo_path), target_path, target_is_directory=True)
                results["link_created"] = True
                print(f"[INFO] Link simbólico criado para {repo}")
            else:
                results["link_created"] = False
                print(f"[INFO] Link simbólico já existia para {repo}")
        except Exception as e:
            results["link_created"] = False
            results["error"] = str(e)
            print(f"[ERROR] Falha ao criar link para {repo}: {e}")
    return results

if __name__=="__main__":
    if "--dataset" not in sys.argv or "--output" not in sys.argv:
        print("Usage: python3 criterias.py --dataset path --input path --output path [--fork] [--iac-percentage] [--commits-per-month] [--num-contributors] [--csv path/to/file.csv]")
        sys.exit(1)

    dataset_dir = os.path.abspath(sys.argv[sys.argv.index("--dataset") + 1])
    output_dir = os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
    input_dir = os.path.abspath(sys.argv[sys.argv.index("--input") + 1]) if "--input" in sys.argv else None
    os.makedirs(output_dir, exist_ok=True)

    # Define caminho do CSV
    if "--csv" in sys.argv:
        csv_path = os.path.abspath(sys.argv[sys.argv.index("--csv") + 1])
    else:
        csv_path = os.path.join(output_dir, "criterias_results.csv")

    filters = {
        "--fork": "--fork" in sys.argv,
        "--iac-percentage": "--iac-percentage" in sys.argv,
        "--commits-per-month": "--commits-per-month" in sys.argv,
        "--num-contributors": "--num-contributors" in sys.argv
    }

    repos = os.listdir(dataset_dir)
    print(f"[INFO] Iniciando análise de {len(repos)} repositórios...")

    results = []
    with ThreadPoolExecutor() as executor:
        futures = [executor.submit(analyze_repo, repo, dataset_dir, input_dir, output_dir, filters) for repo in repos]

        for i, future in enumerate(futures, 1):
            result = future.result()
            results.append(result)
            print(f"[INFO] {i}/{len(repos)} repositórios processados")

    # Salva CSV
    fieldnames = set()
    new_df = pd.DataFrame(results)

    if os.path.exists(csv_path):
        print(f"[INFO] CSV já existe, atualizando resultados: {csv_path}")

        # Carrega CSV existente
        existing_df = pd.read_csv(csv_path)
        # Junta com base na coluna "repo", mantendo dados anteriores e atualizando os novos
        existing_df["repo"] = existing_df["repo"].astype(str)
        new_df["repo"] = new_df["repo"].astype(str)
        merged_df = pd.merge(existing_df, new_df, on="repo", how="outer", suffixes=('', '_new'))

        # Atualiza os campos com os dados novos (colunas *_new), se existirem
        for col in new_df.columns:
            if col != "repo":
                new_col = col + "_new"
                if new_col in merged_df.columns:
                    merged_df[col] = merged_df[new_col].combine_first(merged_df[col])
                    merged_df.drop(columns=[new_col], inplace=True)

        # Salva de volta no CSV
        merged_df.to_csv(csv_path, index=False)
        print(f"[INFO] Resultados atualizados no CSV.")
    else:
        print(f"[INFO] Criando novo CSV em: {csv_path}")
        new_df.to_csv(csv_path, index=False)
        print(f"[INFO] {len(results)} repositórios registrados no novo CSV.")
```

Criteria Frequency Code

```python
import os
import csv
from concurrent.futures import ThreadPoolExecutor
import sys


def classify_technology_in_directory(repo_path):
    """
    Classify the technology based on files in the repository directory.
    Priority: Pulumi > Terraform > AWS CDK
    """
    for root, _, files in os.walk(repo_path):
        for f in files:
            if f.endswith("Pulumi.yaml") or f.endswith("Pulumi.yml"):
                return "Pulumi"
            elif f.endswith("cdktf.json") or f.endswith(".tf"):
                return "Terraform"
            elif f.endswith("cdk.json"):
                return "AWS CDK"
            elif f.endswith(".edn"):
                return "NUBANK"
    return "NOTFOUND"


def process_criteria(criteria_dir, output_dir):
    """
    Process a single criteria directory to classify technologies.
    """
    technology_counts = {"Pulumi": 0, "Terraform": 0, "AWS CDK": 0, "NUBANK": 0, "NOTFOUND": 0}
    results = []

    # Iterate over the subdirectories in the criteria directory
    for repo_id in os.listdir(criteria_dir):
        repo_path = os.path.join(criteria_dir, repo_id)

        # Only process if it's a directory
        if os.path.isdir(repo_path):
            tech_classification = classify_technology_in_directory(repo_path)
            technology_counts[tech_classification] += 1
            results.append([repo_id, tech_classification])

    # Write results to a CSV
    output_csv = os.path.join(output_dir, f"{os.path.basename(criteria_dir.rstrip('/'))}_output.csv")
    os.makedirs(output_dir, exist_ok=True)
    with open(output_csv, "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["ID", "Technology"])
        writer.writerows(results)

    # Print counts
    string = f"Counts for {criteria_dir}:\n"
    for tech, count in technology_counts.items():
        string += f"{tech}: {count}\n"
    print(string)

    return output_csv


def process_directories_in_parallel(criteria_dirs, output_dir):
    """
    Process all criteria directories in parallel.
    """
    with ThreadPoolExecutor() as executor:
        futures = [
            executor.submit(process_criteria, criteria_dir, output_dir)
            for criteria_dir in criteria_dirs
        ]
        for future in futures:
            print(f"Output CSV generated: {future.result()}")


if __name__ == "__main__":
    if not '--input' in sys.argv or not '--output' in sys.argv:
        print("Usage: python3 criterias-frequency.py --input path1,path2,path3,path4 --output path")
        sys.exit(1)

    criteria_dirs = [os.path.abspath(path) for path in sys.argv[sys.argv.index('--input') + 1].split(',')]
    print(f"Executing the frequency of the following paths: {criteria_dirs}")
    output_dir = os.path.abspath(sys.argv[sys.argv.index('--output') + 1])

    process_directories_in_parallel(criteria_dirs, output_dir)
```

Related Files Finder Code

```python
import os
import csv
import json
from concurrent.futures import ThreadPoolExecutor
import sys

csv.field_size_limit(sys.maxsize)

# Definição dos arquivos e extensões permitidas por tecnologia
IAC_FILES = {
    "Pulumi": {
        "patterns": ["Pulumi.yaml", "Pulumi.yml"],
        "extensions": [".js", ".ts", ".py", ".go", ".cs", ".fs", ".vb", ".java"]
    },
    "Terraform": {
        "patterns": ["cdktf.json", ".tf"],
        "extensions": [".ts", ".py", ".java", ".cs", ".go"]
    },
    "AWS CDK": {
        "patterns": ["cdk.json"],
        "extensions": [".cpp", ".go", ".java", ".js", ".kt", ".cs", ".ts", ".php", ".py", ".rb", ".rs",
".swift", ".abap"]
    },
    "NUBANK":   {
        "patterns": [".edn"],
        "extensions": [".edn"]
    }
}

def process_directory(parent_dir, subdir_name):
    """Processa um diretório pai e identifica os arquivos IaC e vizinhos compatíveis."""
    subdir_path = os.path.join(parent_dir, subdir_name)
    iac_data = {
        "id": subdir_name,
        "iac_type": None,
        "iac_paths": [],
        "related_files": []
    }

    print(f"[DEBUG] Processando diretório: {subdir_path}")

    # Percorre arquivos no diretório pai
    for dirpath, _, filenames in os.walk(subdir_path):
        for iac_type, details in IAC_FILES.items():
            found_iac_files = [
                f for f in filenames
                if any(f == pattern or f.endswith(pattern) for pattern in details["patterns"])
            ]
            if found_iac_files:
                # Define o tipo de IaC identificado e adiciona seus arquivos
                iac_data["iac_type"] = iac_type
                iac_data["iac_paths"].extend(
                    [os.path.join(dirpath, f) for f in found_iac_files]
                )

                # Captura arquivos vizinhos compatíveis apenas com a tecnologia correspondente
                allowed_extensions = details["extensions"]
                neighbor_files = [
                    os.path.join(dirpath, f)
                    for f in filenames
                    if os.path.splitext(f)[1] in allowed_extensions
                ]
                iac_data["related_files"].extend(neighbor_files)

    print(f"[DEBUG] Resultados para ID '{subdir_name}': {iac_data}")
    return iac_data

def find_iac_files_with_neighbors_parallel(root_dir, MAX_THREADS=8):
    """Procura arquivos IaC e seus vizinhos usando paralelização."""
    iac_results = []
    parent_dirs = [d for d in os.listdir(root_dir) if os.path.isdir(os.path.join(root_dir, d))]

    print(f"[DEBUG] Diretórios identificados: {parent_dirs}")

    with ThreadPoolExecutor(max_workers=MAX_THREADS) as executor:
        tasks = [executor.submit(process_directory, root_dir, subdir) for subdir in parent_dirs]

        for future in tasks:
            iac_results.append(future.result())

    print(f"[DEBUG] Total de diretórios processados: {len(iac_results)}")
    return iac_results

def save_to_csv(data, output_file):
    """Salva os resultados em um arquivo CSV."""
    print(f"[DEBUG] Salvando resultados no arquivo: {output_file}")
    with open(output_file, mode="w", newline="", encoding="utf-8") as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=["id", "iac_type", "iac_paths", "related_files"])
        writer.writeheader()
        for entry in data:
            writer.writerow({
                "id": entry["id"],
                "iac_type": entry["iac_type"] if entry["iac_type"] else "None",
                "iac_paths": json.dumps(entry["iac_paths"]),
                "related_files": json.dumps(entry["related_files"])
            })
    print(f"[DEBUG] Resultados salvos com sucesso em {output_file}")

if __name__ == "__main__":
    if "--input" not in sys.argv or "--output" not in sys.argv:
        print("Usage: python3 1-related-files-generator.py --input path --output path -t number_threads")
        sys.exit(1)

    root_dir = os.path.abspath(sys.argv[sys.argv.index("--input") + 1])
    output = os.path.abspath(sys.argv[sys.argv.index("--output") + 1])

    if "-t" in sys.argv:
        n_threads = int(sys.argv[sys.argv.index("-t") + 1])
        iac_data = find_iac_files_with_neighbors_parallel(root_dir, n_threads)
    else:
        iac_data = find_iac_files_with_neighbors_parallel(root_dir)

    save_to_csv(iac_data, output)
```

Commits Counter Code

```python
import os
import subprocess
import csv
import sys
from concurrent.futures import ThreadPoolExecutor, as_completed

csv.field_size_limit(sys.maxsize)

def is_git_repo(path):
    return os.path.exists(os.path.join(path, ".git"))

def count_commits_for_files(repo_path, file_paths):
    """
    Conta o número de commits relacionados a uma lista de arquivos em um repositório Git.
    """
    unique_commits = set()
    for file_path in file_paths:
        relative_path = os.path.relpath(file_path, repo_path)
        cmd = ["git", "-C", repo_path, "log", "--pretty=%H", "--", relative_path]
        try:
            result = subprocess.run(cmd, capture_output=True, text=True, check=True)
            commits = result.stdout.strip().split("\n")
            unique_commits.update(commits)
        except subprocess.CalledProcessError:
            print(f"[ERROR] Falha ao executar git log para arquivo {relative_path} em {repo_path}")
            continue
    return len(unique_commits)

def count_total_commits(repo_path):
    """
    Conta o total de commits do repositório.
    """
    cmd = ["git", "-C", repo_path, "log", "--pretty=%H"]
    try:
        result = subprocess.run(cmd, capture_output=True, text=True, check=True)
        return len(result.stdout.strip().split("\n"))
    except subprocess.CalledProcessError:
        print(f"[ERROR] Falha ao executar git log no repositório {repo_path}")
        return None

def process_repository_row(row, dataset_dir):
    """
    Processa uma linha do CSV (um repositório) e retorna a linha com os campos de commits preenchidos.
    """
    repo_id = row["id"]
    try:
        iac_paths = eval(row["iac_paths"])
        related_files = eval(row["related_files"])
    except Exception as e:
        print(f"[ERROR] Erro ao processar paths do repositório {repo_id}: {e}")
        row["commit_count"] = ""
        row["total_commit_count"] = ""
        return row

    iac_paths = [path for path in iac_paths if path]
    related_files = [path for path in related_files if path]

    if not iac_paths and not related_files:
        print(f"[INFO] Ignorando repositório {repo_id} porque não há arquivos válidos.")
        row["commit_count"] = ""
        row["total_commit_count"] = ""
        return row

    repo_path = os.path.join(dataset_dir, repo_id)

    if not is_git_repo(repo_path):
        print(f"[WARNING] Diretório '{repo_path}' não é um repositório Git.")
        row["commit_count"] = ""
        row["total_commit_count"] = ""
        return row

    file_paths = iac_paths + related_files
    commit_count = count_commits_for_files(repo_path, file_paths)
    total_commits = count_total_commits(repo_path)

    row["commit_count"] = commit_count
    row["total_commit_count"] = total_commits if total_commits is not None else ""
    return row

def process_repositories_and_commits(input_csv, output_csv, dataset_dir):
    """
    Processa o CSV de entrada de forma concorrente, contando commits relacionados aos arquivos IaC
    e salvando o resultado em um CSV de saída.
    """
    with open(input_csv, mode="r") as infile, open(output_csv, mode="w", newline="") as outfile:
        reader = csv.DictReader(infile)
        fieldnames = reader.fieldnames + ["commit_count", "total_commit_count"]
        writer = csv.DictWriter(outfile, fieldnames=fieldnames)
        writer.writeheader()

        rows = list(reader)

        with ThreadPoolExecutor(max_workers=8) as executor:
            futures = {executor.submit(process_repository_row, row, dataset_dir): row for row in rows}

            for future in as_completed(futures):
                result_row = future.result()
                writer.writerow(result_row)

# Entry point
if __name__ == "__main__":
    if "--input" not in sys.argv or "--output" not in sys.argv or "--dataset-dir" not in sys.argv:
        print("Usage: python3 2-commits-count.py --input path --output path --dataset-dir path")
        sys.exit(1)

    input_path = os.path.abspath(sys.argv[sys.argv.index("--input") + 1])
    output = os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
    dataset_dir = os.path.abspath(sys.argv[sys.argv.index("--dataset-dir") + 1])
    process_repositories_and_commits(input_path, output, dataset_dir)
```

Time Period Code

```python
import os
import subprocess
import csv
from datetime import datetime
import sys
from concurrent.futures import ThreadPoolExecutor, as_completed

csv.field_size_limit(sys.maxsize)

def get_commit_time_period(repo_path, file_paths):
    if not os.path.exists(os.path.join(repo_path, ".git")):
        print(f"[WARNING] Diretório '{repo_path}' não é um repositório Git. Ignorando.")
        return None, None

    commit_dates = []
    for file_path in file_paths:
        relative_path = os.path.relpath(file_path, repo_path)
        cmd = ["git", "-C", repo_path, "log", "--pretty=%ci", "--", relative_path]
        try:
            result = subprocess.run(cmd, capture_output=True, text=True, check=True)
            dates = result.stdout.strip().split("\n")
            commit_dates.extend(dates)
        except subprocess.CalledProcessError as e:
            print(f"[ERROR] git log falhou para '{relative_path}' em '{repo_path}': {e}")
            return None, None

    valid_dates = []
    for date in commit_dates:
        try:
            valid_dates.append(datetime.strptime(date, "%Y-%m-%d %H:%M:%S %z"))
        except ValueError as e:
            print(f"[WARNING] Data inválida '{date}' em {repo_path}: {e}")
            continue

    if not valid_dates:
        print(f"[WARNING] Nenhuma data válida em '{repo_path}'.")
        return None, None

    return min(valid_dates), max(valid_dates)


def process_row(row, dataset_dir):
    repo_id = row["id"]
    iac_paths = eval(row["iac_paths"])
    repo_path = os.path.join(dataset_dir, repo_id)

    try:
        oldest_commit, newest_commit = get_commit_time_period(repo_path, iac_paths)

        if oldest_commit and newest_commit:
            commit_time_period = (newest_commit - oldest_commit).days
            print(f"[INFO] {repo_id}: Período de commits = {commit_time_period} dias")
        else:
            commit_time_period = None
            print(f"[INFO] {repo_id}: Sem dados de commit válidos.")

        row["oldest_commit"] = oldest_commit.strftime("%Y-%m-%d %H:%M:%S %z") if oldest_commit else "N/A"
        row["newest_commit"] = newest_commit.strftime("%Y-%m-%d %H:%M:%S %z") if newest_commit else "N/A"
        row["commit_time_period"] = commit_time_period

    except Exception as e:
        print(f"[ERROR] Erro ao processar '{repo_id}': {e}")
        row["oldest_commit"] = "N/A"
        row["newest_commit"] = "N/A"
        row["commit_time_period"] = "N/A"

    return row


def process_time_period(input_csv, output_csv, dataset_dir):
    with open(input_csv, mode="r") as infile:
        reader = csv.DictReader(infile)
        fieldnames = reader.fieldnames + ["oldest_commit", "newest_commit", "commit_time_period"]
        rows = list(reader)

    print(f"[INFO] Iniciando processamento de {len(rows)} repositórios...")

    results = []
    with ThreadPoolExecutor() as executor:
        future_to_row = {executor.submit(process_row, row, dataset_dir): row for row in rows}
        for i, future in enumerate(as_completed(future_to_row), 1):
            result = future.result()
            results.append(result)
            print(f"[INFO] Processados {i}/{len(rows)} repositórios")

    with open(output_csv, mode="w", newline="") as outfile:
        writer = csv.DictWriter(outfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(results)

    print(f"[DONE] Todos os repositórios foram processados com sucesso.")
    print(f"[DONE] Resultados salvos em: {output_csv}")


if __name__ == "__main__":
    if not "--input" in sys.argv or not "--output" in sys.argv or not "--dataset-dir" in sys.argv:
        print("Usage: python3 3-time-period.py --input path --output path --dataset-dir path")
        sys.exit(1)

    input_path = os.path.abspath(sys.argv[sys.argv.index("--input") + 1])
    output = os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
    dataset_dir = os.path.abspath(sys.argv[sys.argv.index("--dataset-dir") + 1])
    process_time_period(input_path, output, dataset_dir)
```

Analyzer Code

```python
import csv
from datetime import datetime
import os
import sys
import traceback

csv.field_size_limit(sys.maxsize)

def analyze_csv(file_path, output_csv):
    results = {
        "TOTAL": {"repos": 0, "commits": 0, "iac_files": 0, "iac_commits": 0, "time_period": None},
        "Terraform": {"repos": 0, "commits": 0, "iac_files": 0, "iac_commits": 0, "time_period": None},
        "Pulumi": {"repos": 0, "commits": 0, "iac_files": 0, "iac_commits": 0, "time_period": None},
        "AWS CDK": {"repos": 0, "commits": 0, "iac_files": 0, "iac_commits": 0, "time_period": None},
        "NUBANK": {"repos": 0, "commits": 0, "iac_files": 0, "iac_commits": 0, "time_period": None}
    }

    time_periods = {"TOTAL": [], "Terraform": [], "Pulumi": [], "AWS CDK": [], "NUBANK": []}

    with open(file_path, newline='', encoding='utf-8') as csvfile:
        reader = csv.DictReader(csvfile)
        for i, row in enumerate(reader, 1):
            try:
                iac_paths = row["iac_paths"].strip("[]").split(", ")
                related_files = row["related_files"].strip("[]").split(", ")

                iac_files_count = len(iac_paths) if iac_paths[0] else 0
                related_commits_count = len(related_files) if related_files[0] else 0

                total_commits = int(row["total_commit_count"])
                repo_name = row["iac_type"]
                first_commit_date = datetime.strptime(row["oldest_commit"], "%Y-%m-%d %H:%M:%S %z")
                last_commit_date = datetime.strptime(row["newest_commit"], "%Y-%m-%d %H:%M:%S %z")

                category = "TOTAL"
                if "terraform" == repo_name.lower():
                    category = "Terraform"
                elif "pulumi" == repo_name.lower():
                    category = "Pulumi"
                elif "aws cdk" == repo_name.lower():
                    category = "AWS CDK"
                elif "nubank" == repo_name.lower():
                    category = "NUBANK"

                print(f"[INFO] Linha {i}: {repo_name}")

                # Atualizar métricas
                results[category]["repos"] += 1
                results[category]["commits"] += total_commits
                results[category]["iac_files"] += iac_files_count
                results[category]["iac_commits"] += related_commits_count
                time_periods[category].append(first_commit_date)
                time_periods[category].append(last_commit_date)

                # Atualizar TOTAL
                results["TOTAL"]["repos"] += 1
                results["TOTAL"]["commits"] += total_commits
                results["TOTAL"]["iac_files"] += iac_files_count
                results["TOTAL"]["iac_commits"] += related_commits_count
                time_periods["TOTAL"].append(first_commit_date)
                time_periods["TOTAL"].append(last_commit_date)
            except Exception as e:
                print(f"[ERROR] Erro na linha {i}: {e}")
                traceback.print_exc()
                continue

    # Calcular o período de tempo para cada categoria
    for category, times in time_periods.items():
        if times:
            results[category]["time_period"] = f"{min(times).strftime('%Y-%m-%d')} - {max(times).strftime('%Y-%m-%d')}"


    # Escrever resultados no CSV
    with open(output_csv, mode='w', newline='', encoding='utf-8') as csvfile:
        fieldnames = ["category", "type", "repos", "total_commits", "iac_files", "iac_commits", "time_period"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

        writer.writeheader()
        for category, data in results.items():
            writer.writerow({
                "category": category,
                "type": "GIT" if category != "TOTAL" else "",
                "repos": data["repos"],
                "total_commits": data["commits"],
                "iac_files": data["iac_files"],
                "iac_commits": data["iac_commits"],
                "time_period": data["time_period"] if data["time_period"] else "N/A"
            })


if __name__ == "__main__":
    if "--input" not in sys.argv or "--output" not in sys.argv:
        print("Usage: python3 4-analyze.py --input path --output path")
        sys.exit(1)

    input_csv = os.path.abspath(sys.argv[sys.argv.index("--input") + 1])
    output_csv = os.path.abspath(sys.argv[sys.argv.index("--output") + 1])

    analyze_csv(input_csv, output_csv)

    print(f"Resultados armazenados em: {output_csv}")
```

Script Shell

Repository Cloner Code

```bash
#!/bin/bash

# Initial configurations
default_dir="dataset"
log_file="clone_logs.csv"
repos_file="repos_list.txt" # File with the list of repositories
threads=10 # Number of threads
start_line=2 # Default starting line

```

```bash
function usage() {
  echo "Usage: $0 [-d directory] [-f repos_file] [-s start_line] [-t threads] [-c credential]"
  echo "Options:"
  echo "  -d  Destination directory for cloning repositories (default: dataset)"
  echo "  -f  File containing the list of repositories (one per line) (default: repos_list.txt)"
  echo "  -s  Starting line to continue cloning (default: 2)"
  echo "  -t  Number of simultaneous clones (default: 10)"
  echo "  -c  Credential type: ssh or token"
  exit 1
}

# Process arguments
while getopts "d:f:s:t:c:" opt; do
  case "$opt" in
    d) target_dir="$OPTARG" ;;
    f) repos_file="$OPTARG" ;;
    s) start_line="$OPTARG" ;;
    t) threads="$OPTARG" ;;
    c) credential_type="$OPTARG" ;;
    *) usage ;;
  esac
done

[[ -z "$target_dir" ]] && target_dir="$default_dir"
[[ -z "$credential_type" ]] && credential_type="ssh"

# Initial validations
if [[ ! -f "$repos_file" ]]; then
  echo "Error: Repositories file ($repos_file) not found."
  exit 1
fi

if [[ "$credential_type" != "ssh" && "$credential_type" != "token" ]]; then
  echo "Error: Invalid credential type. Use 'ssh' or 'token'."
  exit 1
fi

# The following command can prevent parallelism errors
# It increases the limit of open files
ulimit -n 4096

mkdir -p "$target_dir"
echo "Destination directory created at ($target_dir)"

if [[ ! -f "$log_file" ]]; then
  echo "Repository,Status,Message" > "$log_file"
fi
echo "Log file created at ($log_file)"

clone_repo() {
  local repo_url="$1"
  local repo_name=$(basename -s .git "$repo_url")

  if [[ "$credential_type" == "token" && "$repo_url" == https://* ]]; then
    repo_url=$(echo "$repo_url" | sed "s|https://|https://$github_token@|")
  fi

  echo "Cloning $repo_name..."

  if git clone "$repo_url" "$target_dir/$repo_name" &>/dev/null; then
    echo "$repo_url,Success," >> "$log_file"
    echo "[OK] $repo_name cloned successfully."
  else
    echo "$repo_url,Error,Failed to clone" >> "$log_file"
    echo "[ERROR] Failed to clone $repo_name."
  fi
}

export -f clone_repo
export credential_type
github_token=""

if [[ "$credential_type" == "token" ]]; then
  echo "Enter the GitHub token: "
  read -s github_token
  export github_token
fi
export target_dir
export log_file

```

```bash
# Process repositories in parallel
repos_to_clone=$(tail -n +$start_line "$repos_file")
echo "$repos_to_clone" | xargs -P $threads -n 1 -I {} bash -c 'clone_repo "$1"' _ {}

mkdir -p csv
mv "$log_file" "csv/$log_file"

find "$target_dir" -type d -exec chmod +x {} \;

echo "Process completed. Logs saved in csv/$log_file."
```

Criteria Applier Code

```bash
#!/bin/bash

log_file="criterias.log"
default_dir="dataset"
python_cmd="python3"

function usage() {
  echo "Usage: $0 [-d directory] [-p python_interpreter]"
  echo "Options:"
  echo "  -d  Specify the directory of the repositories cloned (default: dataset)"
  echo "  -p  Specify the Python interpreter (default: python3)"
  exit 1
}

while getopts "d:p:" opt; do
  case "$opt" in
    d) target_dir="$OPTARG" ;;
    p) python_cmd="$OPTARG" ;;
    *) usage ;;
  esac
done

if [[ -z "$target_dir" ]]; then
  target_dir="$default_dir"
fi

echo "Execution of the apply criterias script started" > $log_file

function handle_error() {
  echo "[ERROR] $1. Exiting." | tee -a $log_file
  exit 1
}

mkdir -p csv/criterias-output || handle_error "Failed to create CSV directories"

echo "Creating criterias directories..." | tee -a $log_file
mkdir -p criterias/criteria1 criterias/criteria2 criterias/criteria3 criterias/criteria4 || handle_error
"Failed to create criterias directories"

echo "Running the first filtering process..." | tee -a $log_file
$python_cmd replication/criterias.py --dataset $target_dir --input $target_dir --output criterias/criteria1 --
iac-percentage --csv csv/criterias-output/criterias_results.csv 2>&1 | tee -a $log_file || handle_error "First filtering
process failed"

echo "Running the second filtering process..." | tee -a $log_file
$python_cmd replication/criterias.py --dataset $target_dir --input criterias/criteria1 --output
criterias/criteria2 --fork --csv csv/criterias-output/criterias_results.csv 2>&1 | tee -a $log_file || handle_error "Second
filtering process failed"

echo "Running the third filtering process..." | tee -a $log_file
$python_cmd replication/criterias.py --dataset $target_dir --input criterias/criteria2 --output
criterias/criteria3 --commits-per-month --csv csv/criterias-output/criterias_results.csv 2>&1 | tee -a $log_file ||
handle_error "Third filtering process failed"

echo "Running the fourth filtering process..." | tee -a $log_file
$python_cmd replication/criterias.py --dataset $target_dir --input criterias/criteria3 --output
criterias/criteria4 --num-contributors --csv csv/criterias-output/criterias_results.csv 2>&1 | tee -a $log_file || handle_error
"Fourth filtering process failed"

echo "Creating CSV directories..." | tee -a $log_file
mkdir -p csv/criterias-output/criterias-frequency || handle_error "Failed to create CSV directories"

echo "Generating the CSV with related files..." | tee -a $log_file
$python_cmd replication/1-related-files-generator.py --input $target_dir --output csv/criterias-
output/csv1_files_with_neighbors.csv 2>&1 | tee -a $log_file || handle_error "CSV related files generation failed"

echo "Generating the CSV with the commits summary..." | tee -a $log_file
```

```bash
$python_cmd replication/2-commits-count.py --input csv/criterias-output/csv1_files_with_neighbors.csv --output csv/criterias-output/csv2_iac_commits_summary.csv --dataset-dir $target_dir 2>&1 | tee -a $log_file || handle_error "Commits summary CSV generation failed"

echo "Generating the CSV with the time period..." | tee -a $log_file
$python_cmd replication/3-time-period.py --input csv/criterias-output/csv2_iac_commits_summary.csv --output csv/criterias-output/csv3_iac_criterias_output.csv --dataset-dir $target_dir 2>&1 | tee -a $log_file || handle_error "Time period CSV generation failed"

echo "Generating the CSV with frequency..." | tee -a $log_file
$python_cmd replication/4-analyze.py --input csv/criterias-output/csv3_iac_criterias_output.csv --output csv/criterias-output/csv4_iac_output_frequency.csv 2>&1 | tee -a $log_file || handle_error "Frequency CSV generation failed"

echo "Generating criterias frequency csv..." | tee -a $log_file
$python_cmd replication/criteria-frequency.py --input criterias/criteria1,criterias/criteria2,criterias/criteria3,criterias/criteria4,$target_dir --output csv/criterias-output/criterias-frequency 2>&1 | tee -a $log_file || handle_error "Criterias frequency CSV generation failed"

echo "Criterias execution completed. Logs saved to $log_file." | tee -a $log_file
```

ACID Runner Code

```bash
#!/bin/bash

log_file="run_acid.log"
source_dir="dataset"
flag_arg="REPLICATION"
target_dir="ACID/dataset/$flag_arg"
output_dir="csv/acid-output"
script_to_run="ACID/main.py"
python_cmd="python3"

function usage() {
  echo "Usage: $0 [-c] [-p <python_interpreter>]"
  echo "Options:"
  echo "  -c    Use 'main-concurrent.py' instead of 'main.py'"
  echo "  -p    Specify the Python interpreter (default is 'python3')"
  exit 1
}

echo "Starting run-acid.sh" | tee -a "$log_file"

while getopts "cp:" opt; do
  case "$opt" in
    c)
      script_to_run="ACID/main-concurrent.py"
      echo "[INFO] Using concurrent script: $script_to_run" | tee -a "$log_file"
      ;;
    p)
      python_cmd="$OPTARG"
      echo "[INFO] Using Python interpreter: $python_cmd" | tee -a "$log_file"
      ;;
    *)
      usage
      ;;
  esac
done

echo "[INFO] Creating directories..." | tee -a "$log_file"
mkdir -p "ACID/dataset" | tee -a "$log_file"
mkdir -p "$target_dir"  | tee -a "$log_file"
mkdir -p "$output_dir"  | tee -a "$log_file"

echo "[INFO] Creating symbolic links from $source_dir to $target_dir..." | tee -a "$log_file"
for dir in "$source_dir"/*; do
  if [[ -d "$dir" ]]; then
    dir_abs_path=$(realpath "$dir")
    dir_name=$(basename "$dir")

    ln -s "$dir_abs_path" "$target_dir/$dir_name" 2>>"$log_file"

    echo "[INFO] Link created: $target_dir/$dir_name -> $dir_abs_path" | tee -a "$log_file"
  fi
done

echo "[INFO] Generating eligible repositories CSV at $target_dir/eligible_repos.csv..." | tee -a "$log_file"
ls "$target_dir" > "$target_dir/eligible_repos.csv" 2>>"$log_file"
if [[ $? -eq 0 ]]; then
  echo "[SUCCESS] CSV generated successfully." | tee -a "$log_file"
else
```

```
  echo "[ERROR] Failed to generate CSV." | tee -a "$log_file"
  exit 1
fi

echo "[INFO] Running $script_to_run with $python_cmd..." | tee -a "$log_file"
$python_cmd "$script_to_run" --flag-arg $flag_arg --csv-replication csv/criterias-output/csv3_iac_criterias_output.csv --csv-default "$source_dir" --output "$output_dir" 2>>"$log_file"
if [[ $? -eq 0 ]]; then
  echo "[SUCCESS] $script_to_run executed successfully." | tee -a "$log_file"
else
  echo "[ERROR] $script_to_run failed." | tee -a "$log_file"
  exit 1
fi

echo "[INFO] Script completed. Logs saved to $log_file." | tee -a "$log_file"
```