**Python code:**

import json

import sqlite3

import nltk

from nltk.cluster.util import cosine\_distance

from stop\_words import get\_stop\_words

import numpy

import re

import boto3

from botocore.client import Config

import websocket

import \_thread

import time

from io import BytesIO

import pandas as pd

import json

import sys

from nltk.tokenize import punkt

from nltk.tokenize.punkt import PunktSentenceTokenizer

Domain = *""*

BRD\_configFileName = *"../config/Rule\_BRD.txt"*

# Config contents

config = None;

Path = *""*

# Output excell

# Requirements dataframe

requirements\_file\_name = *"../data/Requirements.xlsx"*

requirements\_sheet\_name = *""*.join((Domain,*"Requirements"*))

requirements\_df = None

# Domain/UseCase dataframe

domain\_file\_name = *"../data/Domain.xlsx"*

domain\_sheet\_name = *""*.join((Domain,*"Domain"*))

domain\_df = None

# DataElements dataframe

dataelements\_file\_name =*"../data/DataElements.xlsx"*

dataelements\_sheet\_name =*""*.join((Domain,*"Dataelements"*))

dataelements\_df = None

def **split\_sentences**(text):

*""" Split text into sentences.*

*"""*

sentence\_delimiters = re.compile(*u'[\\[\\]\n.!?]'*)

sentences = sentence\_delimiters.split(text)

return sentences

def **split\_into\_tokens**(text):

*""" Split text into tokens.*

*"""*

tokens = nltk.word\_tokenize(text)

return tokens

def **POS\_tagging**(text):

*""" Generate Part of speech tagging of the text.*

*"""*

POSofText = nltk.tag.pos\_tag(text)

return POSofText

def **keyword\_tagging**(tag,tagtext,text):

*""" Tag the text matching keywords.*

*"""*

if (text.lower().find(tagtext.lower()) != -1):

return text[text.lower().find(tagtext.lower()):text.lower().find(tagtext.lower())+len(tagtext)]

else:

return *'UNKNOWN'*

#function not used

def **regex\_tagging**(tag,regex,text):

*""" Tag the text matching REGEX.*

*"""*

p = re.compile(regex, re.IGNORECASE)

matchtext = p.findall(text)

regex\_list=[]

if (len(matchtext)>0):

for regword in matchtext:

regex\_list.append(regword)

return regex\_list

def **BRD\_chunk\_tagging**(tag,chunk,text):

*""" Tag the text using chunking."""*

parsed\_cp = nltk.RegexpParser(chunk)

pos\_cp = parsed\_cp.parse(text)

#pos\_cp.draw()

chunk\_list=[]

for root in pos\_cp:

if isinstance(root, nltk.tree.Tree):

if root.label() == tag:

chunk\_word = *''*

for child\_root in root:

chunk\_word = chunk\_word +*' '*+ child\_root[0]

chunk\_list.append(chunk\_word)

return chunk\_list

def **augument\_SpResponse**(responsejson,updateType,text,tag): # update classified text and tag in entities of response json

*""" Update the output JSON with augumented classifications.*

*"""*

# print("augument\_response response: "+str(responsejson)+"updateType: "+updateType+"text or words: "+text+"tag in rule brd: "+tag )

if(updateType == *'keyword'*):

if not any(d.get(*'text'*, None) == text for d in responsejson[*'Keywords'*]):

responsejson[*'Keywords'*].append({*"User"*:text})

else:

if not any(d.get(*'text'*, None) == text for d in responsejson[*'Entities'*]) :

responsejson[*'Entities'*].append({*"type"*:tag,*"text"*:text})

#print(responsejson)

return responsejson

def **classify\_BRD\_text**(text, config, DOC\_TYPE):

*""" Perform augumented classification of the text for BRD specifically for getting the output with action.*

*"""*

#will be used for storing initial value of response json, this is from nlu earlier

with open(Path+*'../config/output\_format\_BRD.json'*) as f:

responsejson = json.load(f)

tokens = split\_into\_tokens(text)

postags = POS\_tagging(tokens)

#print("POS tags for sentence "+str(tokens)+" is "+ str(postags))

configjson = json.loads(config)#Rule\_BRD.txt

no\_of\_items = 0

for step in configjson[*'configuration'*][*'classification'*][*'stages'*][*'steps'*]:

if(step[*'type'*] == *'chunking'*):

for chunk in step[*'chunk'*]:

if(chunk[*"tag"*]==DOC\_TYPE):

tag=chunk[*"tag"*]

chunktags = BRD\_chunk\_tagging(*'ACTION'*,chunk[*'pattern'*],postags) #overrite happens if there are two chunks of same tags

if(len(chunktags)>0):

for words in chunktags:

responsejson= augument\_SpResponse(responsejson,*'ACTION'*,words,tag)

else:

print(*'UNKNOWN STEP'*)

return responsejson

stopWords = get\_stop\_words(*'english'*)

# List of words to be ignored for text similarity

stopWords.extend([*"The"*,*"This"*,*"That"*,*"."*,*"!"*,*"?"*,*"deals"*,*"use"*,*"case"*,*"function"*])

def **compute\_text\_similarity**(text1tags, text2tags):

*""" Compute text similarity using cosine*

*"""*

#stemming is the process for reducing inflected (or sometimes derived) words to their stem, base or root form

tokens\_text1 = []

tokens\_text2 = []

stemmer = nltk.stem.porter.PorterStemmer()#.WordNetLemmatizer()

*'''*

*sentences\_text1 = split\_sentences(text1)*

*sentences\_text2 = split\_sentences(text2)*

*#print("sentence 1",sentences\_text1)*

*#print("sentence 2",sentences\_text2)*

*#for tags in text1tags:*

*#pass*

*for element in text1tags:*

*tokens\_text1.extend(split\_into\_tokens(element))*

*for element in text2tags:*

*tokens\_text2.extend(split\_into\_tokens(element))*

*for sentence in sentences\_text1:*

*tokenstemp = split\_into\_tokens(sentence.lower())*

*tokens\_text1.extend(tokenstemp)*

*for sentence in sentences\_text2:*

*tokenstemp = split\_into\_tokens(sentence.lower())*

*tokens\_text2.extend(tokenstemp)*

*if (len(text1tags) > 0):*

*tokens\_text1.extend(text1tags)*

*if (len(text2tags) > 0):*

*tokens\_text2.extend(text2tags)*

*'''*

for element in text1tags:

tokens\_text1.extend(split\_into\_tokens(element))

for element in text2tags:

tokens\_text2.extend(split\_into\_tokens(element))

tokens1Filtered = [stemmer.stem(x) for x in tokens\_text1 if x not in stopWords]

tokens2Filtered = [stemmer.stem(x) for x in tokens\_text2 if x not in stopWords]

# remove duplicate tokens

tokens1Filtered = set(tokens1Filtered)

tokens2Filtered = set(tokens2Filtered)

tokensList=[]

text1vector = []

text2vector = []

if len(tokens1Filtered) < len(tokens2Filtered):

tokensList = tokens1Filtered

else:

tokensList = tokens2Filtered

for token in tokensList:

if token in tokens1Filtered:

text1vector.append(1)

else:

text1vector.append(0)

if token in tokens2Filtered:

text2vector.append(1)

else:

text2vector.append(0)

cosine\_similarity = 1-cosine\_distance(text1vector,text2vector)

if numpy.isnan(cosine\_similarity):

cosine\_similarity = 0

return cosine\_similarity

def **load\_artifacts**():

global requirements\_df

global domain\_df

global dataelements\_df

global config

global BRD\_config

global Path

Location = *""*.join((Path,requirements\_file\_name))

#get\_file(requirements\_file\_name,Location)

excel = pd.ExcelFile(Location)

requirements\_df = excel.parse(requirements\_sheet\_name)

Location = *""*.join((Path,domain\_file\_name))

#get\_file(domain\_file\_name,Location)

excel = pd.ExcelFile(Location)

domain\_df = excel.parse(domain\_sheet\_name)

Location = *""*.join((Path,dataelements\_file\_name))

#get\_file(dataelements\_file\_name,Location)

excel = pd.ExcelFile(Location)

dataelements\_df = excel.parse(dataelements\_sheet\_name)

#rule\_text = open(Path+configFileName)

#config = rule\_text.read()

BRD\_rule\_text = open(Path+BRD\_configFileName)

BRD\_config = BRD\_rule\_text.read()

def **prepare\_artifact\_dataframes**():

*""" Prepare artifact dataframes by creating necessary output columns*

*"""*

global requirements\_df

global domain\_df

global dataelements\_df

req\_cols\_len = len(requirements\_df.columns)

dom\_cols\_len = len(domain\_df.columns)

dat\_cols\_len = len(dataelements\_df.columns)

requirements\_df.insert(req\_cols\_len, *"ClassifiedText"*,*""*)

requirements\_df.insert(req\_cols\_len+1, *"Keywords"*,*""*)

requirements\_df.insert(req\_cols\_len+2, *"DomainMatchScore"*,*""*)

domain\_df.insert(dom\_cols\_len, *"ClassifiedText"*,*""*)

domain\_df.insert(dom\_cols\_len+1, *"Keywords"*,*""*)

domain\_df.insert(dom\_cols\_len+2, *"DataElementsMatchScore"*,*""*)

dataelements\_df.insert(dat\_cols\_len, *"ClassifiedText"*,*""*)

dataelements\_df.insert(dat\_cols\_len+1, *"Keywords"*,*""*)

dataelements\_df.insert(dat\_cols\_len+2, *"RequirementsMatchScore"*,*""*)

def **mod\_req\_text\_classifier\_output**(artifact\_df, BRD\_config, output\_column\_name,DOC\_TYPE):

*""" Add text classifier output to the artifact dataframe based on rule defined in config*

*"""*

for index, row in artifact\_df.iterrows():

summary = row[*"I want to <perform some task>"*]

modID = row[*"ID"*]

# print(modID)

modID = modID.replace(*"R"*,*"UC"*)

user = row[*"As a <type of user>"*]

user = *""*.join((user,*" want to "*))

summary = *""*.join((user,summary))

#print("--------------")

#print(summary)

classifier\_journey\_output = classify\_BRD\_text(summary, BRD\_config,DOC\_TYPE)

#print("classifieer ourney out",classifier\_journey\_output)

artifact\_df.at[index, output\_column\_name]= classifier\_journey\_output

return artifact\_df

def **add\_text\_classifier\_output**(artifact\_df, config, output\_column\_name, DOC\_TYPE):

*""" Add text classifier output to the artifact dataframe based on rule defined in config*

*"""*

for index, row in artifact\_df.iterrows():

summary = row[*"Description"*]

#print("--------------")

#print(summary)

classifier\_journey\_output = classify\_BRD\_text(summary, BRD\_config,DOC\_TYPE)

#print(classifier\_journey\_output)

artifact\_df.at[index, output\_column\_name]= classifier\_journey\_output

return artifact\_df

def **add\_keywords\_entities**(artifact\_df, classify\_text\_column\_name, output\_column\_name):

*""" Add keywords and entities to the artifact dataframe"""*

for index, artifact in artifact\_df.iterrows():

keywords\_array = []

for row in artifact[classify\_text\_column\_name][*'Keywords'*]:

#print("add key word entities: classifiedtext[keywords] ",row)

if not row[*'User'*] in keywords\_array and row[*'User'*]!=*""*:

#print("add key word entities: classifiedtext[entities] ",keywords\_array)

keywords\_array.append(row[*'User'*])

for entities in artifact[classify\_text\_column\_name][*'Entities'*]:

if not entities[*'text'*] in keywords\_array and entities[*'text'*]!=*""*:

keywords\_array.append(entities[*'text'*])

artifact\_df.at[index, output\_column\_name]= keywords\_array

#print(keywords\_array)

return artifact\_df

#requirements\_df, domain\_df, keywords\_column\_name, output\_column\_name)

def **populate\_text\_similarity\_score**(artifact\_df1, artifact\_df2, keywords\_column\_name, output\_column\_name):

*""" Populate text similarity score to the artifact dataframes*

*"""*

heading1 = *"Description"*

heading2 = *"Description"*

try:

artifact\_df1[heading1]

except:

heading1 = *"I want to <perform some task>"*

try:

artifact\_df2[heading2]

except:

heading2 = *"I want to <perform some task>"*

for index1, artifact1 in artifact\_df1.iterrows():

matches = []

top\_matches = []

#for tag in artifact1['Keywords']:

if(True): # replace with above for statement to run the code with each tag instead of whole keywords

#tags=[]

#tags.extend(eval('["'+tag+'"]'))

for index2, artifact2 in artifact\_df2.iterrows():

matches.append({*'ID'*: artifact2[*'ID'*],

*'cosine\_score'*: 0,

*'SubjectID'*:artifact1[*'ID'*]})

cosine\_score = compute\_text\_similarity(artifact1[heading1],artifact2[heading2],artifact1[*'Keywords'*],artifact2[*'Keywords'*])

matches[index2][*"cosine\_score"*] = cosine\_score

sorted\_obj = sorted(matches, key=lambda x : x[*'cosine\_score'*], reverse=True)

# This is where the lower cosine value to be truncated is set and needs to be adjusted based on output

for obj in sorted\_obj:

if obj[*'cosine\_score'*] > 0.55:

top\_matches.append(obj)

artifact\_df1.at[index1, output\_column\_name]= top\_matches

return artifact\_df1

# ## 6.3 Process flow

# \*\* Prepare data \*\*

# \* Load artifacts from object storage and create pandas dataframes

# \* Prepare the pandas dataframes. Add additional columns required for further processing.

# In[19]:

load\_artifacts()

prepare\_artifact\_dataframes()

# \*\* Run Text Classification on data \*\*

# \* Add the text classification output to the artifact dataframes

# In[20]:

DOC\_TYPE = [*'REQ\_ACTION'*,*'DOMAIN\_ACTION'*,*'DE\_ACTION'*]

output\_column\_name = *"ClassifiedText"*

requirements\_df = mod\_req\_text\_classifier\_output(requirements\_df, BRD\_config, output\_column\_name,DOC\_TYPE[0])

domain\_df = add\_text\_classifier\_output(domain\_df,BRD\_config, output\_column\_name,DOC\_TYPE[1])

dataelements\_df = add\_text\_classifier\_output(dataelements\_df,BRD\_config, output\_column\_name,DOC\_TYPE[2])

classify\_text\_column\_name = *"ClassifiedText"*

output\_column\_name = *"Keywords"*

requirements\_df = add\_keywords\_entities(requirements\_df, classify\_text\_column\_name, output\_column\_name)

domain\_df = add\_keywords\_entities(domain\_df, classify\_text\_column\_name, output\_column\_name)

dataelements\_df = add\_keywords\_entities(dataelements\_df, classify\_text\_column\_name, output\_column\_name)

*'''*

*writer = pd.ExcelWriter(Path+'../output/intermediate\_before.xlsx')*

*requirements\_df.to\_excel(writer,sheet\_name='Requirements')*

*domain\_df.to\_excel(writer,sheet\_name='domain')*

*dataelements\_df.to\_excel(writer,sheet\_name='dataelements')*

*writer.save()'''*

temp\_df = pd.DataFrame(data=None, columns=requirements\_df.columns, index=None)

for index,row in requirements\_df.iterrows():

#print("row in requirents after keywords: ",row.loc['Keywords'])

action\_keywords = row.loc[*'Keywords'*]

#classified\_text = row.loc['ClassifiedText']

if len(action\_keywords)>0:

for action in action\_keywords:

line = row

line[*'Keywords'*]=eval(*'["'*+action+*'"]'*)

temp\_df = temp\_df.append(line,ignore\_index=True)

requirements\_df = temp\_df

#to check intermediate output

*'''writer = pd.ExcelWriter(Path+'../output/intermediate.xlsx')*

*requirements\_df.to\_excel(writer,sheet\_name='Requirements')*

*domain\_df.to\_excel(writer,sheet\_name='domain')*

*dataelements\_df.to\_excel(writer,sheet\_name='dataelements')*

*writer.save()'''*

#a = time.time()

keywords\_column\_name = *"Keywords"*

output\_column\_name = *"DomainMatchScore"*

requirements\_df = populate\_text\_similarity\_score(requirements\_df, domain\_df, keywords\_column\_name, output\_column\_name)

output\_column\_name = *"DataElementsMatchScore"*

domain\_df = populate\_text\_similarity\_score(domain\_df, dataelements\_df, keywords\_column\_name, output\_column\_name)

output\_column\_name = *"RequirementsMatchScore"*

dataelements\_df = populate\_text\_similarity\_score(dataelements\_df, requirements\_df, keywords\_column\_name, output\_column\_name)

#print(time.time()-a)

# # This section will be used to create the Output in excell format

# In[59]:

def **extract\_action**(summary):

for entities in summary:

return entities

def **lookup\_use\_case**(temp,artifact3\_df,column\_name):

#print(artifact3\_df.(0,'ID'))

val = *""*

rowNum = len(artifact3\_df.index)

#print(rowNum)

for j in range(0,rowNum):

if temp == artifact3\_df.at[j,*'ID'*]:

val = artifact3\_df.at[j,column\_name]

#print(val)

return val

def **extract\_match**(summary,no\_of\_matches,artifact3\_df,column\_name):

match\_array\_description = []

match\_array\_id = []

for index in range(0,no\_of\_matches):

try:

temp = summary[index][*"ID"*]

except:

break

temp = summary[index][*"ID"*]

#print(temp)

use\_case = lookup\_use\_case(temp,artifact3\_df,column\_name)

match\_array\_id.append(temp)

#match\_array\_description.append(use\_case + "(" + str(round(summary[index]["cosine\_score"], 2)) +")")

match\_array\_description.append(use\_case)

#print(use\_case)

return (match\_array\_description,match\_array\_id)

def **extract\_action\_requirements\_df**(artifact1\_df, artifact2\_df):

*""" Add text classifier output to the artifact dataframe based on rule defined in config*

*"""*

for index, row in artifact2\_df.iterrows():

summary = row.loc[*"Keywords"*]

#print("summary ",summary)

classifier\_journey\_output = extract\_action(summary)

#print("classifer out",classifier\_journey\_output)

artifact1\_df.at[index, *'Use Case'*]= classifier\_journey\_output

return artifact1\_df

def **extract\_bestmatch**(artifact1\_df, artifact2\_df, artifact3\_df, artifact4\_df):

*""" Extract best Match*

*"""*

No\_of\_matches\_user\_function = 2

No\_of\_matches\_data\_elements = 5

best\_match\_output\_domain\_function = []

best\_match\_output\_dataelement\_function = []

print(*"-------------------------"*)

for index, row in artifact2\_df.iterrows():

temp1=*""*

summary = row[*"DomainMatchScore"*]

#print(summary)

(best\_match\_output\_domain\_function,best\_match\_output\_domain\_id) = extract\_match(summary, No\_of\_matches\_user\_function, artifact3\_df,*"User Function"*)

#print(best\_match\_output\_domain\_id)

artifact1\_df.at[index, *'Functionality'*]= best\_match\_output\_domain\_function

#print("\*\*\*\*\*\*\*\*\*\*\*\*")

for index2 in best\_match\_output\_domain\_id:

#print(index2)

row\_domain = len(artifact3\_df.index)

for p in range(0,row\_domain):

if index2 == artifact3\_df.at[p,*'ID'*]:

dataelement\_summary = artifact3\_df.at[p,*'DataElementsMatchScore'*]

#print(dataelement\_summary)

#print("------")

(best\_match\_output\_dataelement\_function,best\_match\_output\_dataelement\_id) = extract\_match(dataelement\_summary, No\_of\_matches\_data\_elements, artifact4\_df, *"Short"*)

if best\_match\_output\_dataelement\_function:

best\_match\_output\_dataelement\_function = set(best\_match\_output\_dataelement\_function)

temp1 = temp1 + str(best\_match\_output\_dataelement\_function)

#print("best elemenets ",temp1)

artifact1\_df.at[index, *'Attributes'*]= temp1

return artifact1\_df

# In[60]:

import pandas as pd

no\_of\_rows\_brd = len(requirements\_df.index)

index = range(0,no\_of\_rows\_brd)

columns = [*'ID'*,*'User'*,*'Use Case'*, *'Functionality'*, *'Attributes'*]

SimMean = pd.DataFrame(index=requirements\_df.index, columns=columns)# change to requirements index for getting index from intermediate output

SimMean.loc[0:no\_of\_rows\_brd,*'ID'*] = requirements\_df.loc[0:no\_of\_rows\_brd,*'ID'*].values

SimMean.loc[0:no\_of\_rows\_brd,*'User'*] = requirements\_df.loc[0:no\_of\_rows\_brd,*'As a <type of user>'*].values

SimMean = extract\_action\_requirements\_df(SimMean,requirements\_df)

SimMean = extract\_bestmatch(SimMean,requirements\_df,domain\_df,dataelements\_df)

#SimMean.loc[SimMean['Attributes']].astype(str).replace('[','{').replace(']','}')

*'''writer = pd.ExcelWriter(Path+'../data/intemediate.xlsx')*

*SimMean.to\_excel(writer, sheet\_name='Sheet1',index=False)*

*writer.save()'''*

def **create\_connection**(db\_file):

*""" create a database connection to the SQLite database*

*specified by db\_file*

**:param** *db\_file: database file*

**:return:** *Connection object or None*

*"""*

try:

conn = sqlite3.connect(db\_file)

return conn

except ValueError as e:

print(e)

return None

conn=create\_connection(*"../db/brd.db"*)

cur=conn.cursor()

with open(*"../output/code\_skeleton.txt"*,*"w"*) as fp:

for index,rows in SimMean.iterrows():

fct\_names = rows[*"Functionality"*]

dataelements = str(rows[*"Attributes"*]).replace(*'}{'*,*','*)

class\_name = rows[*"Use Case"*]

unwanted\_chars = [*'{'*,*'}'*,*'\''*,*'['*,*']'*]

for ch in unwanted\_chars:

dataelements=dataelements.replace(ch,*""*)

fp.write(*"Class "*+str(class\_name).strip().replace(*' '*, *'\_'*)+*"{\n"*)

for element in dataelements.split(*','*):

fp.write(*" private "*+element+*"=0;\n"*)

ele= [*""*,*""*]# to create string array of size 2

i=0

for element\_split in str(rows[*"Attributes"*]).split(*'}{'*):

for ch in unwanted\_chars:

element\_split=element\_split.replace(ch,*""*)

ele[i] = element\_split

i=1

i=0

for fct in fct\_names:

fct=fct.replace(*' '*,*'\_'*)

fp.write(*"\n void "*+str(fct).strip().replace(*' '*, *'\_'*)+*'{'*)

query\_string = *"create table if not EXISTS "*+str(fct)+*'('*

for element in ele[i].split(*','*):

fp.write(*" \n"*+element+*"=0;\n"*)

query\_string=query\_string+element+*" text,"*

query\_string=query\_string[:-1]+*')'*

print(query\_string)

if conn is not None:

cur.execute(query\_string)

conn.commit()

fp.write(*"\n \n \n --------- code section-----"*+*"\n"*+*"\n"*+*" }"*+*"\n \n"*)

i=1

fp.write(*"}\n"*)

conn.close()

temp\_df=SimMean[SimMean[*"ID"*].duplicated(keep=False)]

merged\_df=temp\_df.astype(str).groupby(temp\_df.ID, as\_index=False).agg(*','*.join)

temp\_df=SimMean.drop(temp\_df.index,axis=0).append(merged\_df,ignore\_index=True).sort\_values(*"ID"*).drop(columns=*"ID"*)

SimMean = temp\_df

SimMean[*'User'*]=temp\_df[*'User'*].str.split(*','*).apply(set).str.join(*""*)

writer = pd.ExcelWriter(Path+*'../data/output.xlsx'*)

SimMean.to\_excel(writer, sheet\_name=*'Sheet1'*,index=False)

writer.save()

**Config File:**

{

"configuration": {

"classification": {

"stages": {

"steps": [

{

"type": "chunking",

"chunk": [

{

"tag": "REQ\_ACTION",

"pattern": "ACTION:{<VB.?>+(<TO>|<DT>|<RB.?>|<PRP.?>|<IN>|<JJ.?>)\*<NN.?|NNPS>+ (<PRP.?>|<VB.?>|<TO> |<CC>|JJ.?)\* <NN.?|NNPS>\*}"

},

{

"tag": "DOMAIN\_ACTION",

"pattern": "ACTION:{(<VB.?>+(<TO>|<DT>|<RB.?>|<PRP.?>|<IN>|<JJ.?>)\*<NN.?|NNPS>+)|(<VB.?>|<NN.?>|<NNPS>)}"

},

{

"tag": "DE\_ACTION",

"pattern": "ACTION:{<VB.?>|<NN.?>|<NNPS>}"

}

]

}

]

}

}

}

}