Class BREDS:

**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main(){

configuration = **"parameters.cfg"***#sys.argv[1]*sentences\_file =**"sentencesVahab.txt"***#fSentences.read()# sys.argv[2]*seeds\_file =**"seeds\_positive.txt"** *#sys.argv[3]*negative\_seeds = **"seeds\_negative.txt"***#sys.argv[4]*similarity =0.7*#0.6# tebghe rahnemaei proje dar GitHub sys.argv[5]*confidence =0.7*#0.8#tebghe rahnemaei dar GitHub=https://github.com/davidsbatista/BREDS sys.argv[6]*

**class BREDS**(object):  
 **def** \_\_init\_\_(self, config\_file, seeds\_file, negative\_seeds, similarity,  
 confidence):  
 self.curr\_iteration = 0  
 self.patterns = list()  
 self.processed\_tuples = list()  
 self.candidate\_tuples = defaultdict(list)  
 self.config = **Config**(config\_file, seeds\_file, negative\_seeds,  
 similarity, confidence)

**class Config**(object):  
  
 **def** \_\_init\_\_(self, config\_file, positive\_seeds, negative\_seeds,  
 similarity, confidence):  
 print(**"I am in Config......."**)  
  
 *# http://www.ling.upenn.edu/courses/Fall\_2007/ling001/penn\_treebank\_pos.html  
 # select everything except stopwords, ADJ and ADV* self.**filter\_pos** = [**'JJ'**, **'JJR'**, **'JJS'**, **'RB'**, **'RBR'**, **'RBS'**, **'WRB'**]  
 self.regex\_clean\_simple = re.compile(**'</?[A-Z]+>'**, re.U)  
 self.regex\_clean\_linked = re.compile(**'</[A-Z]+>|<[A-Z]+ url=[^>]+>'**, re.U)  
 self.tags\_regex = re.compile(**'</?[A-Z]+>'**, re.U)  
 self.**e\_types** = {**'ORG'**: 3, **'LOC'**: 4, **'PER'**: 5}  
 self.positive\_seed\_tuples = **set**()  
 self.negative\_seed\_tuples = **set**()  
 self.vec\_dim = 0

self.e1\_type = **None**

self.e2\_type = **None** self.stopwords = stopwords.words(**'english'**)  
 self.lmtzr = **WordNetLemmatizer**()  
 self.threshold\_similarity = similarity  
 self.instance\_confidence = confidence  
 self.reverb = **Reverb**()  
 self.word2vec = **None** self.vec\_dim = **None** *# simple tags, e.g.: # <PER>Bill Gates</PER>* self.**regex\_simple** = re.compile(**'<[A-Z]+>[^<]+</[A-Z]+>'**, re.U)

**number\_iterations**=4

# minimum number of patterns that generated a tuple so that tuple can be used  
# in the clustering phase  
**min\_pattern\_support**=2

# parameters for the cosine similarity between the three  
# relationships vector contexts  
**alpha** = 0.0  
**beta** = 1.0  
**gamma** = 0.0

# **Word2Vec models** #  
word2vec\_path=**afp\_apw\_xin\_embeddings.bin**

breads = **BREDS**(configuration, seeds\_file, negative\_seeds, float(similarity), float(confidence))

**if** sentences\_file.endswith(**'.txt'**):  
 breads.**generate\_tuples**(**sentences\_file**)

**def generate\_tuples**(self, sentences\_file):  
*"""  
 Generate tuples instances from a text file with sentences where  
 named entities are already tagged*

*"""*

self.config.read\_word2vec()  
tagger = load(**'taggers/maxent\_treebank\_pos\_tagger/english.pickle'**)  
print(**"Tagger=\n "**, tagger)  
print(**"\nGenerating relationship instances from sentences"**)  
f\_sentences = codecs.open(sentences\_file, encoding=**'utf-8'**)  
count = 0  
**for** line **in** f\_sentences:

*# create a sentence object not text only***sentence** = **Sentence**(line.strip(), self.config.e1\_type, self.config.e2\_type,  
 self.config.max\_tokens\_away, self.config.min\_tokens\_away,  
 self.config.context\_window\_size,tagger,**self.config**)

**class Sentence**:  
  
 **def** \_\_init\_\_(self, sentence, e1\_type, e2\_type, max\_tokens, min\_tokens, window\_size, pos\_tagger=**None**, config=**None**):  
 self.relationships = list()  
 self.tagged\_text = **None** *# determine which type of regex to use according to  
 # how named-entities are tagged* entities\_regex = **None  
 if** config.tag\_type == **"simple"**:  
 entities\_regex = config.regex\_simple  
 ***# find named-entities*****entities** = []

**for** m **in** re.finditer(entities\_regex, sentence):  
 **entities.append(m)**

**if** len(entities) >= 2:  
 ***# clean tags from text*****sentence\_no\_tags** = **None  
 if** config.tag\_type == **"simple"**:  
 **sentence\_no\_tags** = re.sub**(**config.regex\_clean\_simple, **""**, sentence**)**

print(**"... vahab Sentence No tag="**, **sentence\_no\_tags**)

**text\_tokens** = **word\_tokenize**(**sentence\_no\_tags**)

**# extract information about the entity, create an Entity instance**

entities\_info = set()  
 **for** x **in** range(0, **len(entities)**):  
 **if** config.tag\_type == **"simple"**:  
 **entity = entities[x].group()????**  
 e\_string = re.findall(**'<[A-Z]+>([^<]+)</[A-Z]+>'**, entity)[0]  
 e\_type = re.findall(**'<([A-Z]+)'**, entity**)[0]**  
 **e\_parts**, locations = **find\_locations**(e\_string, text\_tokens)  
 e = **EntitySimple**(e\_string, **e\_parts**, e\_type, locations)

**entities\_info.add(e)**

***# create an hash table:*** *# - key is the starting index in the tokenized sentence of an entity # - value the corresponding Entity instance*locations = dict()  
**for** e **in** entities\_info:  
 **for** start **in** e.locations:  
 locations[start] = e

before = self.tagged\_text[:sorted\_keys[i]]  
before = before[-window\_size:]  
between = self.tagged\_text[sorted\_keys[i]+len(e1.parts): sorted\_keys[i+1]]   
after = self.tagged\_text[sorted\_keys[i+1]+len(e2.parts):]  
after = after[:window\_size]

**if** config.tag\_type == **"simple"**:  
 **r** = **Relationship**(sentence, before, between, after, e1.string,  
 e2.string, e1\_type, e2.type)

self.**relationships.append**(**r**)

*# now an object of “***class Sentence***” of Sentence is made*

**for** rel **in** sentence.relationships:

t = Tuple(rel.e1, rel.e2, rel.sentence, rel.before,  
 rel.between, rel.after, self.**config**)

print(**"Tuple = \n"**,t.aft\_words,t.aft\_vector)

self.**processed\_tuples.append(t)**

**class** Tuple(object):  
 *# http://www.ling.upenn.edu/courses/Fall\_2007/ling001/penn\_treebank\_pos.html* filter\_pos = [**'JJ'**, **'JJR'**, **'JJS'**, **'RB'**, **'RBR'**, **'RBS'**, **'WRB'**]  
  
 **def** \_\_init\_\_(self, \_e1, \_e2, \_sentence, \_before, \_between, \_after,  
 **config**):  
 self.e1 = \_e1  
 self.e2 = \_e2  
 self.sentence = \_sentence  
 self.confidence = 0  
 self.bef\_tags = \_before  
 self.bet\_tags = \_between  
 self.bet\_filtered = **None** self.aft\_tags = \_after  
 self.bef\_words = **" "**.join([x[0] **for** x **in** self.bef\_tags])  
 self.bet\_words = **" "**.join([x[0] **for** x **in** self.bet\_tags])  
 self.aft\_words = **" "**.join([x[0] **for** x **in** self.aft\_tags])  
 self.bef\_vector = **None** self.bet\_vector = **None** self.aft\_vector = **None** self.passive\_voice = **False  
 self.construct\_vectors(config)**

**def construct\_vectors**(self, **config**):  
 *# Check if BET context contains a* ***ReVerb pattern***

***#****if there was a reverb pattern, sets it as bet\_word O.W. bet\_w=bet\_txt* **reverb\_pattern** = config.reverb.extract\_reverb\_patterns\_tagged\_ptb(self.bet\_tags)  
 **if** len(reverb\_pattern) > 0:  
 *# test for passive voice presence* self.**passive\_voice** = config.reverb.detect\_passive\_voice(  
 **reverb\_pattern)**  
 **bet\_words** = reverb\_pattern  
 **else**:  
 self.passive\_voice = **False  
 bet\_words** = self.bet\_tags  
  
 self.bet\_filtered = [t[0] **for** t **in** bet\_words **if** t[0].lower()  
 **not in** config.stopwords **and** t[1]**not in** self.filter\_pos]  
  
 *# compute the vector over the filtered BET context* self.**bet\_vector** = self.**pattern2vector\_sum**(self.bet\_filtered, config)

*# compute the vector for words before the first entity,  
 # and for words after the second entity* bef\_no\_tags = [t[0] **for** t **in** self.bef\_tags]  
 aft\_no\_tags = [t[0] **for** t **in** self.aft\_tags]  
 self.bef\_vector = self.pattern2vector\_sum(bef\_no\_tags, config)  
 self.aft\_vector = self.pattern2vector\_sum(aft\_no\_tags, config)

**def pattern2vector\_sum**(tokens, config):  
 pattern\_vector = zeros(config.vec\_dim)  
 **if** len(tokens) > 1:  
 **for** t **in** tokens:   
 **try**:  
 **vector** = **config**.**word2vec**[t[0].strip()]  
 pattern\_vector += vector  
 **except** KeyError: **continue**  
 self.word2vecmodelpath, binary=**True**)

**elif** len(tokens) == 1:  
 **try**:  
 pattern\_vector = config.word2vec[tokens[0].strip()]  
 **except** KeyError:  
 **pass  
  
 return** pattern\_vector

“word2vec = **Word2Vec.load\_word2vec\_format**(**word2vecmodelpath**)”