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]*

breads = BREDS(configuration, seeds_file, negative_seeds, float(similarity), float(confidence))

```
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
```

```
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)
```

```
class Tuple(object):
        #
http://www.ling.upenn.edu/courses/Fall_2007/ling001/penn_treebank_pos.ht
ml
        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
```

```
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
```

"word2vec = **Word2Vec.load_word2vec_format(word2vecmodelpath)**"

```
# 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)
```

```
print("Tuple = \n",t.aft_words,t.aft_vector)

self.processed_tuples.append(t)
```