```
from __future__ import division
from __future__ import print_function
import random
import gensim
from random import randint
from translate import Translator
import numpy as np
from transliterate import translit
import argparse, sys
import matplotlib.pyplot  as plt
from scipy.spatial import distance
import seaborn as sns
import pandas as pd
from nltk.corpus import wordnet
from tabulate import tabulate
def checkStoredWords(kwords, word):
    """

         This function updates a list of known words with a new word.
If the spell type and
    language exists in the list the value is append by 1 otherwise,
it is appended to
    the end of the list with a value of 1.

    :param kwords: List of spell types and language with associated
frequencies.
    :param word: One being the spell type and the other being the
origin language.
    :type kwords: [[[str, str], int]...]
    :type word: str
    :return: the updated list of known words.
    """

    found = False
    for kword in kwords:
        if kword[0] == word:
            kword[1] += 1
            found = True
    if found == False:
        kwords.append([word, int(1)])
    return kwords




def count_instances(fname):
    """

         Reads supplied file, where it splits it up. Then it appends
each word to the data
    set building a list of words and frequencies using
checkStoredWords(kwords, word).

    :param fname: This is the name of the CSV file in which the spell
data is stored.
    :type fname: str
    :return: returns a list of languages and the probabilities for
each one.

```

```
    """

    file = open(fname, 'r')
    data = []

    for line in file:
        temp = line.rstrip()
        temp = temp.split(",")
        data = checkStoredWords(data, temp)
    file.close()
    data = calcProb(data)
    return data



def totalSpells(data):
    """

         Counts the number of spells in the dataset.

    :param data: List of spell types and origin language with
frequency.
    :type data: [[[str,str], int]...]
    :return: an integer value of total number of spells.
    """

    total = 0
    for d in data:
        total += d[1]
    return total


def calcProb(data):
    """

         Calculates the probabilities for spells of each type.

    :param data: List of spell types and origin language with
frequency.
    :type data: [[[str, str], int]...]
    :return: A list of type of spells and their associated
probabilities.
    """

    total = totalSpells(data)
    prob = 0.0
    for d in data:
             prob = d[1] / total
             d.append(prob)
    return data


def generateScale(data):
    """

         This stacks the probabilities of spells so that each spell
has a boundary in which
    it a spell can be selected over another.

```

```
95
96      :param data: list of spell names and their associated frequencies
    and probabilities.
97      :type data: [[[str,str],int,float]...]
98      :return: a list of spells and the value between 0-1 in which that
    name will be selected.
99      """
100
101     value = 0
102     index = -1
103     scale = []
104     for d in data:
105         value += d[2]
106         index += 1
107         scale.append((value, d[0]))
108     return scale
109
110
111 def getSpellType(scale, rndNum):
112     """
113
114          Selects a spell according to the random number passed.
115
116     :param scale: A list of tuples which contains the probability
    associated with each spell and type.
117     :param rndNum: The random number used to select a spell type.
118     :type scale: [(str,str,float)..]
119     :type rndNum: float
120     :return: A string which is the spell type.
121     """
122
123     for i in range(-1, len(scale) - 1):
124         if i == -1:
125             temp2 = scale[i + 1]
126             if rndNum >= 0:
127                 if rndNum < temp2[0]:
128                     return temp2[1]
129         else:
130             temp = scale[i]
131             temp2 = scale[i + 1]
132             if rndNum >= temp[0]:
133                 if rndNum < temp2[0]:
134                     return temp2[1]
135
136     temp2 = scale[0]
137     return temp2[1]
138
139
140 def is_valid(string):
141     """
    check to see whether a word consists of alpha characters.

    :param string: The string to be checked.
    :type string: str
    :return: Boolean value.
    """
    if string.isalpha():
        return False
```

```
142     return True
143
144
145
146 def langCode(language): #this now works with python 2.7 i believe.
147     """
148
149     Converts a language name into a language code for the translator.
150
151     :param language: Full name of the language, for example latin.
152     :type language: tr
153     :return: The string code for the language.
154     """
155     return {
156         'Latin': 'la',
157         'Greek': 'el',
158         'Portuguese': 'pt',
159         'West African Sidiki': 'it',  # CANT BE TRANSLATED. - Returns
160 italian
161         'Aramaic': 'el',  # CANT BE TRANSLATED - RETURNS GREEK
162         'Pig Latin': 'PL',  # implement a seperate function to
163 convert to pig latin.
164         'English': 'en',
165         'French': 'fr',
166         'Spanish': 'es',
167         'Italian': 'it',
168     }.get(language, 'la')  # returns latin as default - if language
169 is not found.
170
171
172 def translate2(word, lang):
173     """
174          Translates a word to a target language.
175
176     :param word: The word you want to convert.
177     :param lang: the lang code of the language you want to convert
178 to.
179     :type word: str
180     :type lang: str
181     :return: a string containing the translated word in the latin
182 alphabet.
183     """
184     translator = Translator(to_lang=lang)
185     try:
186         out = translator.translate(word)
187         if lang == 'el':
188             return translit(word, lang, reversed=True)
189         return out
190     except:
191         log("Error Cannot translate: " + word)
192
193 def log(text):
194     logfile = open("log.txt", "a")
195     logfile.write(text.encode("utf-8") + "\n")
196     logfile.close()
197
198 def sentenceToWord(sentence, model, oword):
```

```
    """

    Takes a string and converts it into a vector. Then from that it
picks a similar word that doesn't contain an underscore.

    :param sentence: A string which contains a sentence to be
converted into one word.
    :type sentence: str
    :return: A string containing a similar word.
    """

    sentence = sentence.split()
    output = []
    top_val = 20
    selected = []
    bogus_words = 0
    for word in sentence:
        try:
            output.append(model[word])
        except KeyError:
            log("key error in vector file" + word)

    output = np.array(output)
    vector_sum = output.sum(axis=0)
    output = model.most_similar(positive=[vector_sum], topn=top_val)
    final_output = output[randint(0, (top_val - 1))]
    while is_valid(final_output[0]):
        num = randint(0, top_val - 1)
        final_output = output[num]
        if num in selected:
            if len(selected) == top_val:
                top_val = top_val * 2
                output = model.most_similar(positive=[vector_sum],
topn=top_val)
        else:
            selected.append(num)

        bogus_words+=1
    return final_output, bogus_words


def pigLatin(source):
    """

         Takes a source string and converts it from english to pig
latin.

    :param source: Takes string of english words and changes it into
pig latin.
    :type source: str
    :return: a string containing pig latin words.

    """

    letters = ['sh', 'gl', 'ch', 'ph', 'tr', 'br', 'fr', 'bl', 'gr',
'st', 'sl', 'cl', 'pl', 'fl']
    source = source.split()
    for k in range(len(source)):

```

```
        i = source[k]
        if i[0] in ['a', 'e', 'i', 'o', 'u']:
            source[k] = i + 'ay'
        elif f(i) in letters:
            source[k] = i[2:] + i[:2] + 'ay'
        elif i.isalpha() == False:
            source[k] = i
        else:
            source[k] = i[1:] + i[0] + 'ay'
    return ' '.join(source)


def f(str):
    """
    Returns the first two chacters from the string.

    :param str: A word that is passed.
    :type str: str
    :return: a string that only contains the first two letters.

    """
    if len(str) ==1:
        return str[0]
    return str[0] + str[1]


def generateSpell(sentence, model, oword):
    """
    Generates a Spell from a sentence.

    :param sentence: string which is the definition of the spell you
want to create.
    :type sentence: str
    :return: list containing the spell and the spell type.
    :param model: loaded vector orepresentation of words.
         :type model: data file loaded.
    """

    spell = []
    vector,temp_bogus = sentenceToWord(sentence, model, oword)
    vector = vector[0]
    scale = generateScale(count_instances('spell_prob.csv'))
    selection = random.random()
    spell_meta = getSpellType(scale, selection)

    try:
        target_lang = langCode(spell_meta[1])
    except:
        log("langCode function didn't work. Using default latin.")
        target_lang = "la"

    if target_lang == "PL":
        spell.append(pigLatin(vector))
    else:
        spell.append(translate2(vector, target_lang))
    spell.append(spell_meta[0])
    spell.append(vector) #The original word before translation is
also added onto the end for evaluation purposes.
```

```
283     return spell, temp_bogus
284
285
286 def load_vectors(path, is_binary):
287     """
288     This loads the vectors supplied by the path.
289
290     :param path: The path to the vector file
291     :type path: str
292     :param is_binary: states whether file is a binary file.
293     :type is_binary: boolean
294     :return: The loaded model.
295     """
296     print("Loading: ", path)
297     model = gensim.models.Word2Vec.load_word2vec_format(path,
298 binary=is_binary)
299     model.init_sims(replace=True)
300     print("Loaded: ", path)
301     return model
302
303
304
305 def is_synonym(n_word, o_word):
306     """
307     This function uses a combination of NLTK's wordnet to
308     list all synonyms for a word and to check if a new word is a
309 synonym.
310
311     :param n_word: The new word generated.
312     :type n_word: str
313     :param o_word: The original word in the definition.
314     :type o_word: str
315     :return: Returns a boolean indicating whether n_word is a synonym
316 of o_word.
317     """
318     synonyms=[]
319     synsets = wordnet.synsets(o_word)
320     for synset in synsets:
321         synonyms = synonyms+ synset.lemma_names()
322
323     return n_word in synonyms
324
325
326 def run_experiment(model, num_experiments):
327     """
328         This function runs the experiments with the paramters set.
329         It then returns all the necessary data for processing and
330 output.
331
332         :param model: The vectors loaded.
333         :type model: The loaded vector object
334         :param num_experiments: The number of experiments to run.
335         :type num_experiments: int
336         :return: A list of averages scores, one entry per experiment.
337         :return: A list of the average number of synonyms produced,
338 one  entry per experiment.
339
```

```
330         :return: The average score across the experiments.
331         :return: A list of average cosine similarity scores, one
entry per experiment.
332         :return: The number of experiments.
333         :return: A list containing the number of bogus words
produced, one entry per expeirment.
334         :return: A list containing lists with each sublist containing
the scores produced for that definition length.
335         :return: A list containing list with each sublist containing
number of bogus words produced for that definition length.
336     """
337     average = 0.0
338     iterationCount = 0
339     scores = []
340     cos_dists = []
341     avg_cos_dists = []
342     syn_experiments = []
343     bword_counts = []
344     scores_per_spell=[[] for x in range(10)] #size of definition
length.
345     table1 = []
346     table2 = []
347     bwords_spell= [[] for x in range(10)] #size of definition length.
348     for i in range(0, num_experiments):
349         table1 = []
350         table2 = []
351         print("---------------", i, "---------------")
352         log("---------------"+str(i) +  "---------------")
353         bogus_words = 0
354         spellFile = open("spells.csv")
355         entry = []
356         score = 0
357         count = 0
358         syn_counts = 0
359         for line in spellFile:
360             count+=1
361             line = line.strip("\n")
362             entry = line.split(",")
363             spell, temp_bogus = generateSpell(entry[1],
model,entry[3] )
364             bwords_spell[len(entry[1].split(" "))].append(temp_bogus)
365             bogus_words+= temp_bogus
366 
367             if args.verbose:
368                 print("Your new spell is: ", spell[0])
369 
370             if spell[2].lower() not in entry[1].split():
371                 score +=1
372                 scores_per_spell[len(entry[1].split(" "))].append(1)
373             else:
374                 scores_per_spell[len(entry[1].split(" "))].append(0)
375 
376             table1.append([spell[0]])
            table2.append([spell[2]])
            #calculate the cosine similarity.
            og_wd = model[entry[-1].strip()]
            nw_wd = model[spell[-1]]
            cos_dists.append(distance.cosine(og_wd, nw_wd))
```

```
            if is_synonym(spell[2].lower(), entry[-1]):
                syn_counts +=1

        print("Experiment Results")
        print("Num of spells that don't feature in definition: ",
score)
        print("Percentage: ", ((float(score)/count) * 100),"%")
        print("Average Cosine-simalarity:", float(sum(cos_dists) /
len(cos_dists)))
        print("Num of spells which are synonyms: ", syn_counts)
        print("Num of words selected that are not real words: ",
bogus_words)
        scores.append((float(score)/count) * 100)
        syn_experiments.append(syn_counts)
        bword_counts.append(bogus_words)
        spellFile.close()
        iterationCount +=1
        average += (float(score)/count)*100
        avg_cos_dists.append(float(sum(cos_dists) / len(cos_dists)))
    return scores, syn_experiments,average, avg_cos_dists,
iterationCount, bword_counts, scores_per_spell, bwords_spell


#
================================================================================
# Main part of the program.
#
================================================================================
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
            'Use Word2Vec or GloVe datasets to generate Harry Potter
Spells')
    parser.add_argument('--glove', action='store_const', const =
'glove',
            help='Use the GloVe dataset instead of the default
Word2Vec.')
    parser.add_argument('--exp',
    help="Specifies the number of experiments on this run. Default is
20.",
            action='store', type=int)
    parser.add_argument('--verbose', action='store_const', const =
'verbose',
            help='Prints out the spell names')
    parser.add_argument('--comp', action= 'store_const',
const='comp',
            help = "Runs the word2vec vectors, and the GloVe
vectors")
    args = parser.parse_args()

    logFile = open("log.txt", 'w' )
    logFile.close()
    num_experiments = 20

    if args.exp != None:
        num_experiments = args.exp
```

```
424
425     if args.comp: # comparison mode.
426         print("Compare Mode")
427         log("----------------------Compare
    Mode-----------------------")
428         print("Vectors used: Word2Vec")
429         log("---------------"+ "Vectors used: Word2Vec"+
    "---------------")
430         model = load_vectors("../../vectors/GoogleNews-vectors-
    negative300.bin", True)
431
432         #Run word2vec experiments and then stores data in dataframe.
433         w_scores, w_syn_experiments, w_average, w_avg_cos_dists,
    iterationCount, w_bword_counts,  w_spells_per, w_bwords_per=
    run_experiment(model, num_experiments)
434         w_vec=["word2vec" for x in w_scores]
435         del model
436         print("Vectors used: GloVe")
437         log("---------------" +  "Vectors used: GloVe"+
    "---------------")
438         model = load_vectors("../../vectors/glove.txt.vw", False)
439
440         # run experiments and move results into data frame.
441         g_scores, g_syn_experiments, g_average, g_avg_cos_dists,
    iterationCount, g_bword_counts,  g_spells_per, g_bwords_per=
    run_experiment(model, num_experiments)
442         g_vec = ["glove" for x in g_scores]
443
444         scores=w_scores + g_scores
445         syn_experiments = w_syn_experiments + g_syn_experiments
446         avg_cos_dists = w_avg_cos_dists + g_avg_cos_dists
447         bword_counts = w_bword_counts + g_bword_counts
448         vectors = w_vec + g_vec
449
450         ##for the ts plots
451         g_vec = ["GloVe" for x in g_spells_per]
452         w_vec = ["Word2Vec" for x in w_spells_per]
453         bwords_per = w_bwords_per + g_bwords_per
454         spells_per = w_spells_per + g_spells_per
455         vec = w_vec + g_vec
456
457         ##adds values for empty rows.#might want to remove empty rows
    later.
458         for row in spells_per:
459             if len(row) == 0:
460                 row.append(0)
461         for row in bwords_per:
462             if len(row) == 0:
463                 row.append(0)
464
465         spells_per_avg = [float(sum(l)/len(l)) for l in spells_per]
466         length= [x for x in range(1, len(w_spells_per)+1)] + [x for x
    in range(1, len(g_spells_per)+1)]
467         bwords_per_avg = [float(sum(l)/len(l)) for l in bwords_per]
468         len_results =
    pd.DataFrame({"originality":spells_per_avg,"length":length,"bwords":b
    words_per_avg, "vectors":vec})
469
470
```

```
        box_len = []
        box_score= []
        box_vec=[]


        for i in range(0,len(w_spells_per)):
            for row in w_spells_per[i]:
                box_len.append(i+1)
                box_score.append(row)
                box_vec.append("word2vec")

            for row2 in g_spells_per[i]:
                box_len.append(i+1)
                box_score.append(row2)
                box_vec.append("GloVe")

        box_data = pd.DataFrame({"length":box_len,
"originality":box_score, "vectors":box_vec})

        #originality vs size plots.
        ax = sns.tsplot(time="length", value="originality",
unit="vectors",condition="vectors",data=len_results  )
        # sns.plt.xticks([0,1,2,3,4,5,6,7,8,9,10])
        sns.plt.show()

        ax = sns.distplot(box_score)
        sns.plt.show()
        #box plot
        ax = sns.boxplot(x="length", y = "originality",
hue="vectors", data=box_data)
        sns.plt.show()

        box_len = []
        box_score= []
        box_vec=[]

        for i in range(0,len(w_bwords_per)):
            for row in w_bwords_per[i]:
                box_len.append(i+1)
                box_score.append(row)
                box_vec.append("word2vec")
            for row2 in g_bwords_per[i]:
                box_len.append(i+1)
                box_score.append(row2)
                box_vec.append("GloVe")

        box_data = pd.DataFrame({"length":box_len,
"bwords":box_score, "vectors":box_vec})
        #histogram
        ax = sns.distplot(box_score)
        sns.plt.show()
        #box plot
        ax = sns.boxplot(x="length", y = "bwords", hue="vectors",
data=box_data)
        sns.plt.show()
        ##output results.

        print("----------------word2vec Experiment
```

```
Results------------------")
        print("The mean average percentage over ", iterationCount ,
"tests: ",
                (w_average/iterationCount), "%")
        print("The mean cosine simalarity over ", iterationCount,
"tests: ",
                float(sum(w_avg_cos_dists)/ len(w_avg_cos_dists)))
        print("The mean amount of synonyms", (sum(w_syn_experiments)/
iterationCount))
        print("Average number of words that are not fit for
translation: ",float(sum(w_bword_counts)/iterationCount))


        print("----------------GloVe Experiment
Results------------------")
        print("The mean average percentage over ", iterationCount ,
"tests: ",
                (g_average/iterationCount), "%")
        print("The mean cosine simalarity over ", iterationCount,
"tests: ",
                float(sum(g_avg_cos_dists)/ len(g_avg_cos_dists)))
        print("The mean amount of synonyms", (sum(g_syn_experiments)/
iterationCount))
        print("Average number of words that are not fit for
translation: ",float(sum(g_bword_counts)/iterationCount))


        results = pd.DataFrame({"scores":scores,
"similarity":avg_cos_dists, "synonyms":syn_experiments,
"vectors":vectors, "bwords":bword_counts})


        sim = sns.violinplot(x="vectors", y="similarity",
data=results)
        sns.plt.title("Comparison of Similarity over "+str(
iterationCount)+ " experiments")
        sns.plt.show()
        sc = sns.violinplot(x="vectors", y="scores", data=results)
        sns.plt.title("Comparison of accuracy scores over
"+str(iterationCount)+ " experiments")
        sns.plt.show()
        bw = sns.violinplot(x="vectors", y="bwords", data=results)
        sns.plt.title("Comparison of invalid words over "+
str(iterationCount)+ " experiments")
        sns.plt.show()
        sns.plt.title("Comparison of synonyms over " +str(
iterationCount) +" experiments")
        syn = sns.violinplot(x="vectors", y="synonyms", data=results)
        sns.plt.show()

    else: # test an individual mode.
        if args.glove:
            print("Vectors used: GloVe")
            log("---------------" +  "Vectors used: GloVe"+
"---------------")
            model = load_vectors("../../vectors/glove.txt.vw", False)
        else:
            print("Vectors used: Word2Vec")
```

```
            log("---------------"+ "Vectors used: Word2Vec"+
"---------------")
            model = load_vectors("../../vectors/GoogleNews-vectors-
negative300.bin", True)


        scores, syn_experiments, average, avg_cos_dists,
iterationCount, bword_counts,  spells_per, bwords_per=
run_experiment(model, num_experiments)
        print("-----------------Experiment Results------------------")
        print("The mean average percentage over ", iterationCount ,
"tests: ",
                (average/iterationCount), "%")
        print("The mean cosine simalarity over ", iterationCount,
"tests: ",
                float(sum(avg_cos_dists)/ len(avg_cos_dists)))
        print("The mean amount of synonyms", (sum(syn_experiments)/
iterationCount))
        print("Average number of words that are not fit for
translation: ",float(sum(bword_counts)/iterationCount))
        results = pd.DataFrame({'scores': scores, 'similarity':
avg_cos_dists})

        #loop through and add an entry to any empty fields.
        for row in spells_per:
            if len(row) == 0:
                row.append(0)

        spells_per_avg = [float(sum(l)/len(l)) for l in spells_per]
        length= [x for x in range(0, len(spells_per_avg))]

        vec = ["vector" for x in spells_per_avg]
        sns.plt.show()
        ax2 = sns.violinplot(x=results["similarity"])
        sns.plt.show()
        ax = sns.violinplot(x="scores", y="similarity", data=results)
        sns.plt.show()
```

```
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
```