```
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.

    :param data: list of spell names and their associated frequencies
and probabilities.

```

```
    :type data: [[str,str],int,float]...]
    :return: a list of spells and the value between 0-1 in which that
name will be selected.
    """

    value = 0
    index = -1
    scale = []
    for d in data:
        value += d[2]
        index += 1
        scale.append((value, d[0]))
    return scale


def getSpellType(scale, rndNum):
    """

         Selects a spell according to the random number passed.

    :param scale: A list of tuples which contains the probability
associated with each spell and type.
    :param rndNum: The random number used to select a spell type.
    :type scale: [(str,str,float)..]
    :type rndNum: float
    :return: A string which is the spell type.
    """

    for i in range(-1, len(scale) - 1):
        if i == -1:
            temp2 = scale[i + 1]
            if rndNum >= 0:
                if rndNum < temp2[0]:
                    return temp2[1]
        else:
            temp = scale[i]
            temp2 = scale[i + 1]
            if rndNum >= temp[0]:
                if rndNum < temp2[0]:
                    return temp2[1]

    temp2 = scale[0]
    return temp2[1]


def is_valid(string):
    """
    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
    return True
```

```
142
143
144 def langCode(language): #this now works with python 2.7 i believe.
145     """
146
147     Converts a language name into a language code for the translator.
148
149     :param language: Full name of the language, for example latin.
150     :type language: tr
151     :return: The string code for the language.
152     """
153     return {
154         'Latin': 'la',
155         'Greek': 'el',
156         'Portuguese': 'pt',
157         'West African Sidiki': 'it',  # CANT BE TRANSLATED. - Returns
158 italian
159         'Aramaic': 'el',  # CANT BE TRANSLATED - RETURNS GREEK
160         'Pig Latin': 'PL',  # implement a seperate function to
161 convert to pig latin.
162         'English': 'en',
163         'French': 'fr',
164         'Spanish': 'es',
165         'Italian': 'it',
166     }.get(language, 'la')  # returns latin as default - if language
167 is not found.
168
169
170 def translate2(word, lang):
171     """
172          Translates a word to a target language.
173
174     :param word: The word you want to convert.
175     :param lang: the lang code of the language you want to convert
176 to.
177     :type word: str
178     :type lang: str
179     :return: a string containing the translated word in the latin
180 alphabet.
181     """
182     translator = Translator(to_lang=lang)
183     try:
184         out = translator.translate(word)
185         if lang == 'el':
186             return translit(word, lang, reversed=True)
187         return out
188     except:
189         log("Error Cannot translate: " + word)
190
191 def log(text):
192     logfile = open("log.txt", "a")
193     logfile.write(text.encode("utf-8") + "\n")
194     logfile.close()
195
196 def sentenceToWord(sentence, model, oword):
197     """
198
199     Takes a string and converts it into a vector. Then from that it
200
```

```
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.
    return spell, temp_bogus


```

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

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

```
377
378         print("Experiment Results")
379         print("Num of spells that don't feature in definition: ",
score)
380         print("Percentage: ", ((float(score)/count) * 100),"%")
381         print("Average Cosine-simalarity:", float(sum(cos_dists) /
382 len(cos_dists)))
383         print("Num of spells which are synonyms: ", syn_counts)
384         print("Num of words selected that are not real words: ",
bogus_words)
385         scores.append((float(score)/count) * 100)
386         syn_experiments.append(syn_counts)
387         bword_counts.append(bogus_words)
388         spellFile.close()
389         iterationCount +=1
390         average += (float(score)/count)*100
391         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
392
393
394 #
395 ==========================================================================
396 =============
397 # Main part of the program.
398 #
399 ==========================================================================
400 =============
401 if __name__ == '__main__':
402     parser = argparse.ArgumentParser(
403             'Use Word2Vec or GloVe datasets to generate Harry Potter
404 Spells')
405     parser.add_argument('--glove', action='store_const', const =
406 'glove',
407             help='Use the GloVe dataset instead of the default
408 Word2Vec.')
409     parser.add_argument('--exp',
410     help="Specifies the number of experiments on this run. Default is
411 20.",
412             action='store', type=int)
413     parser.add_argument('--verbose', action='store_const', const =
414 'verbose',
415             help='Prints out the spell names')
416     parser.add_argument('--comp', action= 'store_const',
417 const='comp',
418             help = "Runs the word2vec vectors, and the GloVe
419 vectors")
420     args = parser.parse_args()
421
422     logFile = open("log.txt", 'w' )
423     logFile.close()
    num_experiments = 20

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

    if args.comp: # comparison mode.
        print("Compare Mode")
```

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

        #Run word2vec experiments and then stores data in dataframe.
        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)
        w_vec=["word2vec" for x in w_scores]
        del model
        print("Vectors used: GloVe")
        log("---------------" +  "Vectors used: GloVe"+
"---------------")
        model = load_vectors("../../vectors/glove.txt.vw", False)

        # run experiments and move results into data frame.
        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)
        g_vec = ["glove" for x in g_scores]

        scores=w_scores + g_scores
        syn_experiments = w_syn_experiments + g_syn_experiments
        avg_cos_dists = w_avg_cos_dists + g_avg_cos_dists
        bword_counts = w_bword_counts + g_bword_counts
        vectors = w_vec + g_vec

        ##for the ts plots
        g_vec = ["GloVe" for x in g_spells_per]
        w_vec = ["Word2Vec" for x in w_spells_per]
        bwords_per = w_bwords_per + g_bwords_per
        spells_per = w_spells_per + g_spells_per
        vec = w_vec + g_vec

        ##adds values for empty rows.#might want to remove empty rows
later.
        for row in spells_per:
            if len(row) == 0:
                row.append(0)
        for row in bwords_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(1, len(w_spells_per)+1)] + [x for x
in range(1, len(g_spells_per)+1)]
        bwords_per_avg = [float(sum(l)/len(l)) for l in bwords_per]
        len_results =
pd.DataFrame({"originality":spells_per_avg,"length":length,"bwords":b
words_per_avg, "vectors":vec})

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

```
471
472
473             for i in range(0,len(w_spells_per)):
474                 for row in w_spells_per[i]:
475                     box_len.append(i+1)
476                     box_score.append(row)
477                     box_vec.append("word2vec")
478
479                 for row2 in g_spells_per[i]:
480                     box_len.append(i+1)
481                     box_score.append(row2)
482                     box_vec.append("GloVe")
483
484             box_data = pd.DataFrame({"length":box_len,
485 "originality":box_score, "vectors":box_vec})
486
487             #originality vs size plots.
488             ax = sns.tsplot(time="length", value="originality",
489 unit="vectors",condition="vectors",data=len_results  )
490             # sns.plt.xticks([0,1,2,3,4,5,6,7,8,9,10])
491             sns.plt.show()
492
493             ax = sns.distplot(box_score)
494             sns.plt.show()
495             #box plot
496             ax = sns.boxplot(x="length", y = "originality",
497 hue="vectors", data=box_data)
498             sns.plt.show()
499
500             box_len = []
501             box_score= []
502             box_vec=[]
503
504             for i in range(0,len(w_bwords_per)):
505                 for row in w_bwords_per[i]:
506                     box_len.append(i+1)
507                     box_score.append(row)
508                     box_vec.append("word2vec")
509                 for row2 in g_bwords_per[i]:
510                     box_len.append(i+1)
511                     box_score.append(row2)
512                     box_vec.append("GloVe")
513
514             box_data = pd.DataFrame({"length":box_len,
515 "bwords":box_score, "vectors":box_vec})
516             #histogram
517             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
```