```
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     Converts a language name into a language code for the translator.
147
148     :param language: Full name of the language, for example latin.
149     :type language: tr
150     :return: The string code for the language.
151     """
152     return {
153         'Latin': 'la',
154         'Greek': 'el',
155         'Portuguese': 'pt',
156         'West African Sidiki': 'it',  # CANT BE TRANSLATED. - Returns
157 italian
158         'Aramaic': 'el',  # CANT BE TRANSLATED - RETURNS GREEK
159         'Pig Latin': 'PL',  # implement a seperate function to
160 convert to pig latin.
161         'English': 'en',
162         'French': 'fr',
163         'Spanish': 'es',
164         'Italian': 'it',
165     }.get(language, 'la')  # returns latin as default - if language
166 is not found.
167
168
169 def translate2(word, lang):
170     """
171          Translates a word to a target language.
172
173     :param word: The word you want to convert.
174     :param lang: the lang code of the language you want to convert
175 to.
176     :type word: str
177     :type lang: str
178     :return: a string containing the translated word in the latin
179 alphabet.
180     """
181     translator = Translator(to_lang=lang)
182     try:
183         out = translator.translate(word)
184         if lang == 'el':
185             return translit(word, lang, reversed=True)
186         return out
187     except:
188         log("Error Cannot translate: " + word)

189 def log(text):
190     """
191
192     Writes a message to a log file.
193
194     :param text: The text you want to write to log file.
195     :type text: str
196     """
197     logfile = open("log.txt", "a")
198
```

```
189     logfile.write(text.encode("utf-8") + "\n")
190     logfile.close()
191
192 def sentenceToWord(sentence, model, oword):
193     """
194
195     Takes a string and converts it into a vector. Then from that it
    picks a similar word that doesn't contain an underscore.
196
197     :param sentence: A string which contains a sentence to be
    converted into one word.
198     :type sentence: str
199     :return: A string containing a similar word.
200     """
201
202     sentence = sentence.split()
203     output = []
204     top_val = 20
205     selected = []
206     bogus_words = 0
207     for word in sentence:
208         try:
209             output.append(model[word])
210         except KeyError:
211             log("key error in vector file" + word)
212
213     output = np.array(output)
214     vector_sum = output.sum(axis=0)
215     output = model.most_similar(positive=[vector_sum], topn=top_val)
216     final_output = output[randint(0, (top_val - 1))]
217     while is_valid(final_output[0]):
218         num = randint(0, top_val - 1)
219         final_output = output[num]
220         if num in selected:
221             if len(selected) == top_val:
222                 top_val = top_val * 2
223                 output = model.most_similar(positive=[vector_sum],
224                         topn=top_val)
225         else:
226             selected.append(num)
227
228         bogus_words+=1
229     return final_output, bogus_words
230
231
232 def pigLatin(source):
233     """
234
235          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:
```

```
283         spell.append(translate2(vector, target_lang))
284     spell.append(spell_meta[0])
285     spell.append(vector) #The original word before translation is
    also added.
286     return spell, temp_bogus
287 
288 
289 def load_vectors(path, is_binary):
290     """
291     This loads the vectors supplied by the path.
292 
293     :param path: The path to the vector file
294     :type path: str
295     :param is_binary: states whether file is a binary file.
296     :type is_binary: boolean
297     :return: The loaded model.
298     """
299     print("Loading: ", path)
300     model = gensim.models.Word2Vec.load_word2vec_format(path,
301             binary=is_binary)
302     model.init_sims(replace=True)
303     print("Loaded: ", path)
304     return model
305 
306 
307 
308 def is_synonym(n_word, o_word):
309     """
310     This function uses a combination of NLTK's wordnet to
311     list all synonyms for a word and to check if a new word is a
    synonym.
312 
313     :param n_word: The new word generated.
314     :type n_word: str
315     :param o_word: The original word in the definition.
316     :type o_word: str
317     :return: Returns a boolean indicating whether n_word is a synonym
    of o_word.
318     """
319     synonyms=[]
320     synsets = wordnet.synsets(o_word)
321     for synset in synsets:
322         synonyms = synonyms+ synset.lemma_names()
323 
324     return n_word in synonyms
325 
326 
327 def run_experiment(model, num_experiments):
328     """
329         This function runs the experiments with the paramters set.
330         It then returns all the necessary data for processing and
    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
        :return: A list of averages scores, one entry per experiment.
        :return: A list of the average number of synonyms produced,
one  entry per experiment.
        :return: The average score across the experiments.
        :return: A list of average cosine similarity scores, one
entry per experiment.
        :return: The number of experiments.
        :return: A list containing the number of bogus words
produced, one entry per expeirment.
        :return: A list containing lists with each sublist containing
the scores produced for that definition length.
        :return: A list containing list with each sublist containing
number of bogus words produced for that definition length.
    """
    average = 0.0
    iterationCount = 0
    scores = []
    cos_dists = []
    avg_cos_dists = []
    syn_experiments = []
    bword_counts = []
    scores_per_spell=[[] for x in range(10)] #size of definition
length.
    table1 = []
    table2 = []
    bwords_spell= [[] for x in range(10)] #size of definition length.
    for i in range(0, num_experiments):
        table1 = []
        table2 = []
        print("---------------", i, "---------------")
        log("---------------"+str(i) +  "---------------")
        bogus_words = 0
        spellFile = open("spells.csv")
        entry = []
        score = 0
        count = 0
        syn_counts = 0
        for line in spellFile:
            count+=1
            line = line.strip("\n")
            entry = line.split(",")
            spell, temp_bogus = generateSpell(entry[1],
                    model,entry[3])
            bwords_spell[len(entry[1].split(" "))].append(temp_bogus)
            bogus_words+= temp_bogus

            if args.verbose:
                print("Your new spell is: ", spell[0])

            if spell[2].lower() not in entry[1].split():
                score +=1
                scores_per_spell[len(entry[1].split(" "))].append(1)
            else:
                scores_per_spell[len(entry[1].split(" "))].append(0)

            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

    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.
        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":bwords_per_avg, "vectors":vec})

        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.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()
```