

```

1 from __future__ import division
2 from __future__ import print_function
3 import random
4 import gensim
5 from random import randint
6 from translate import Translator
7 import numpy as np
8 from transliterate import translit
9 import argparse, sys
10 import matplotlib.pyplot as plt
11 from scipy.spatial import distance
12 import seaborn as sns
13 import pandas as pd
14 from nltk.corpus import wordnet
15 from tabulate import tabulate
16 def checkStoredWords(kwords, word):
17     """
18     This function updates a list of known words with a new word.
19     If the spell type and language exists in the list the value is append
20     by 1 otherwise, it is appended to the end of the list with a value of
21     1.
22     :param kwords: List of spell types and language with associated
23     frequencies.
24     :param word: One being the spell type and the other being the
25     origin language.
26     :type kwords: [[str, str], int]...
27     :type word: str
28     :return: the updated list of known words.
29     """
30     found = False
31     for kword in kwords:
32         if kword[0] == word:
33             kword[1] += 1
34             found = True
35     if found == False:
36         kwords.append([word, int(1)])
37     return kwords
38
39 def count_instances(fname):
40     """
41     Reads supplied file, where it splits it up. Then it appends
42     each word to the data set building a list of words and frequencies
43     using checkStoredWords(kwords, word).
44     :param fname: This is the name of the CSV file in which the spell
45     data is stored.
46     :type fname: str
47     :return: returns a list of languages and the probabilities for
48     each one.
49     """

```

```

48     file = open(fname, 'r')
49     data = []
50
51     for line in file:
52         temp = line.rstrip()
53         temp = temp.split(",")
54         data = checkStoredWords(data, temp)
55     file.close()
56     data = calcProb(data)
57     return data
58
59 def totalSpells(data):
60     """
61         Counts the number of spells in the dataset.
62
63         :param data: List of spell types and origin language with
64         frequency.
65         :type data: [[[str,str], int]...]
66         :return: an integer value of total number of spells.
67     """
68     total = 0
69     for d in data:
70         total += d[1]
71     return total
72
73 def calcProb(data):
74     """
75         Calculates the probabilities for spells of each type.
76
77         :param data: List of spell types and origin language with
78         frequency.
79         :type data: [[[str, str], int]...]
80         :return: A list of type of spells and their associated
81         probabilities.
82     """
83     total = totalSpells(data)
84     prob = 0.0
85     for d in data:
86         prob = d[1] / total
87         d.append(prob)
88     return data
89
90 def generateScale(data):
91     """
92         This stacks the probabilities of spells so that each spell
93         has a boundary in which it a spell can be selected over another.
94
95         :param data: list of spell names and their associated frequencies
96         and probabilities.

```

```

95     :type data: [[[str,str],int,float]...]
96     :return: a list of spells and the value between 0-1 in which that
97     name will be selected.
98     """
99     value = 0
100    index = -1
101    scale = []
102    for d in data:
103        value += d[2]
104        index += 1
105        scale.append((value, d[0]))
106    return scale
107
108 def getSpellType(scale, rndNum):
109     """
110         Selects a spell according to the random number passed.
111
112         :param scale: A list of tuples which contains the probability
113         associated with each spell and type.
114         :param rndNum: The random number used to select a spell type.
115         :type scale: [(str,str,float)..]
116         :type rndNum: float
117         :return: A string which is the spell type.
118         """
119     for i in range(-1, len(scale) - 1):
120         if i == -1:
121             temp2 = scale[i + 1]
122             if rndNum >= 0:
123                 if rndNum < temp2[0]:
124                     return temp2[1]
125             else:
126                 temp = scale[i]
127                 temp2 = scale[i + 1]
128                 if rndNum >= temp[0]:
129                     if rndNum < temp2[0]:
130                         return temp2[1]
131             temp2 = scale[0]
132         return temp2[1]
133
134 def is_valid(string):
135     """
136         check to see whether a word consists of alpha characters.
137
138         :param string: The string to be checked.
139         :type string: str
140         :return: Boolean value.
141         """
142     if string.isalpha():
143         return False
144     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
190 def log(text):
191     logfile = open("log.txt", "a")
192     logfile.write(text.encode("utf-8") + "\n")
193     logfile.close()
194
195 def sentenceToWord(sentence, model, oword):
196     """
197     Takes a string and converts it into a vector. Then from that it

```

```

189 picks a similar word that doesn't contain an underscore.
190
191 :param sentence: A string which contains a sentence to be
192 converted into one word.
193 :type sentence: str
194 :return: A string containing a similar word.
195 """
196
197 sentence = sentence.split()
198 output = []
199 top_val = 20
200 selected = []
201 bogus_words = 0
202 for word in sentence:
203     try:
204         output.append(model[word])
205     except KeyError:
206         log("key error in vector file" + word)
207
208 output = np.array(output)
209 vector_sum = output.sum(axis=0)
210 output = model.most_similar(positive=[vector_sum], topn=top_val)
211 final_output = output[randint(0, (top_val - 1))]
212 while is_valid(final_output[0]):
213     num = randint(0, top_val - 1)
214     final_output = output[num]
215     if num in selected:
216         if len(selected) == top_val:
217             top_val = top_val * 2
218             output = model.most_similar(positive=[vector_sum],
219 topn=top_val)
220         else:
221             selected.append(num)
222
223     bogus_words+=1
224 return final_output, bogus_words
225
226 def pigLatin(source):
227     """
228
229     Takes a source string and converts it from english to pig
230 latin.
231
232     :param source: Takes string of english words and changes it into
233 pig latin.
234     :type source: str
235     :return: a string containing pig latin words.
236     """
237
238     letters = ['sh', 'gl', 'ch', 'ph', 'tr', 'br', 'fr', 'bl', 'gr',
239 'st', 'sl', 'cl', 'pl', 'fl']
240     source = source.split()
241     for k in range(len(source)):
242         i = source[k]
243         if i[0] in ['a', 'e', 'i', 'o', 'u']:
244             source[k] = i + 'ay'

```

```

236         elif f(i) in letters:
237             source[k] = i[2:] + i[:2] + 'ay'
238         elif i.isalpha() == False:
239             source[k] = i
240         else:
241             source[k] = i[1:] + i[0] + 'ay'
242     return ' '.join(source)
243
244 def f(str):
245     """
246     Returns the first two chacters from the string.
247
248     :param str: A word that is passed.
249     :type str: str
250     :return: a string that only contains the first two letters.
251
252     """
253     if len(str) == 1:
254         return str[0]
255     return str[0] + str[1]
256
257 def generateSpell(sentence, model, oword):
258     """
259     Generates a Spell from a sentence.
260
261     :param sentence: string which is the definition of the spell you
262     want to create.
263     :type sentence: str
264     :return: list containing the spell and the spell type.
265     :param model: loaded vector orepresentation of words.
266     :type model: data file loaded.
267
268     """
269     spell = []
270     vector, temp_bogus = sentenceToWord(sentence, model, oword)
271     vector = vector[0]
272     scale = generateScale(count_instances('spell_prob.csv'))
273     selection = random.random()
274     spell_meta = getSpellType(scale, selection)
275
276     try:
277         target_lang = langCode(spell_meta[1])
278     except:
279         log("langCode function didn't work. Using default latin.")
280         target_lang = "la"
281
282     if target_lang == "PL":
283         spell.append(pigLatin(vector))
284     else:
285         spell.append(translate2(vector, target_lang))
286     spell.append(spell_meta[0])
287     spell.append(vector) #The original word before translation is
288     also added onto the end for evaluation purposes.
289     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 def is_synonym(n_word, o_word):
302     """
303     This function uses a combination of NLTK's wordnet to
304     list all synonyms for a word and to check if a new word is a
305     synonym.
306
307     :param n_word: The new word generated.
308     :type n_word: str
309     :param o_word: The original word in the definition.
310     :type o_word: str
311     :return: Returns a boolean indicating whether n_word is a synonym
312     of o_word.
313     """
314     synonyms=[]
315     synsets = wordnet.synsets(o_word)
316     for synset in synsets:
317         synonyms = synonyms+ synset.lemma_names()
318
319     return n_word in synonyms
320
321 def run_experiment(model, num_experiments):
322     """
323     This function runs the experiments with the paramters set.
324     It then returns all the necessary data for processing and
325     output.
326
327     :param model: The vectors loaded.
328     :type model: The loaded vector object
329     :param num_experiments: The number of experiments to run.
330     :type num_experiments: int
331     :return: A list of averages scores, one entry per experiment.
332     :return: A list of the average number of synonyms produced,
333     one entry per experiment.
334     :return: The average score across the experiments.
335     :return: A list of average cosine similarity scores, one
336     entry per experiment.

```

```

330         :return: The number of experiments.
331         :return: A list containing the number of bogus words
332 produced, one entry per experiment.
333         :return: A list containing lists with each sublist containing
334 the scores produced for that definition length.
335         :return: A list containing list with each sublist containing
336 number of bogus words produced for that definition length.
337 """
338     average = 0.0
339     iterationCount = 0
340     scores = []
341     cos_dists = []
342     avg_cos_dists = []
343     syn_experiments = []
344     bword_counts = []
345     scores_per_spell=[[] for x in range(10)] #size of definition
length.
346     table1 = []
347     table2 = []
348     bwords_spell= [[] for x in range(10)] #size of definition length.
349     for i in range(0, num_experiments):
350         table1 = []
351         table2 = []
352         print("-----", i, "-----")
353         log("-----"+str(i) + "-----")
354         bogus_words = 0
355         spellFile = open("spells.csv")
356         entry = []
357         score = 0
358         count = 0
359         syn_counts = 0
360         for line in spellFile:
361             count+=1
362             line = line.strip("\n")
363             entry = line.split(",")
364             spell, temp_bogus = generateSpell(entry[1],
365 model,entry[3] )
366             bwords_spell[len(entry[1].split(" "))].append(temp_bogus)
367             bogus_words+= temp_bogus
368
369             if args.verbose:
370                 print("Your new spell is: ", spell[0])
371
372             if spell[2].lower() not in entry[1].split():
373                 score +=1
374                 scores_per_spell[len(entry[1].split(" "))].append(1)
375             else:
376                 scores_per_spell[len(entry[1].split(" "))].append(0)
377
378             table1.append([spell[0]])
379             table2.append([spell[2]])
380             #calculate the cosine similarity.
381             og_wd = model[entry[-1].strip()]
382             nw_wd = model[spell[-1]]
383             cos_dists.append(distance.cosine(og_wd, nw_wd))
384
385             if is_synonym(spell[2].lower(), entry[-1]):
386                 syn_counts +=1

```



```

377
378     print("Experiment Results")
379     print("Num of spells that don't feature in definition: ",
380 score)
381     print("Percentage: ", ((float(score)/count) * 100), "%")
382     print("Average Cosine-similarity:", float(sum(cos_dists) /
383 len(cos_dists)))
384     print("Num of spells which are synonyms: ", syn_counts)
385     print("Num of words selected that are not real words: ",
386 bogus_words)
387     scores.append((float(score)/count) * 100)
388     syn_experiments.append(syn_counts)
389     bword_counts.append(bogus_words)
390     spellFile.close()
391     iterationCount +=1
392     average += (float(score)/count)*100
393     avg_cos_dists.append(float(sum(cos_dists) / len(cos_dists)))
394     return scores, syn_experiments, average, avg_cos_dists,
395 iterationCount, bword_counts, scores_per_spell, bwords_spell
396
397 #
398 =====
399 # Main part of the program.
400 #
401 =====
402 if __name__ == '__main__':
403     parser = argparse.ArgumentParser(
404         'Use Word2Vec or GloVe datasets to generate Harry Potter
405 Spells')
406     parser.add_argument('--glove', action='store_const', const =
407 'glove',
408         help='Use the GloVe dataset instead of the default
409 Word2Vec.')
410     parser.add_argument('--exp',
411         help="Specifies the number of experiments on this run. Default is
412 20.",
413         action='store', type=int)
414     parser.add_argument('--verbose', action='store_const', const =
415 'verbose',
416         help='Prints out the spell names')
417     parser.add_argument('--comp', action= 'store_const',
418 const='comp',
419         help = "Runs the word2vec vectors, and the GloVe
420 vectors")
421     args = parser.parse_args()
422
423     logFile = open("log.txt", 'w' )
424     logFile.close()
425     num_experiments = 20
426
427     if args.exp != None:
428         num_experiments = args.exp
429
430     if args.comp: # comparison mode.
431         print("Compare Mode")

```

```

424         log("-----Compare
425 Mode-----")
426         print("Vectors used: Word2Vec")
427         log("-----" + "Vectors used: Word2Vec"+
428 "-----")
429         model = load_vectors("../vectors/GoogleNews-vectors-
430 negative300.bin", True)
431         #Run word2vec experiments and then stores data in dataframe.
432         w_scores, w_syn_experiments, w_average, w_avg_cos_dists,
433 iterationCount, w_bword_counts, w_spells_per, w_bwords_per=
434 run_experiment(model, num_experiments)
435         w_vec=["word2vec" for x in w_scores]
436         del model
437         print("Vectors used: GloVe")
438         log("-----" + "Vectors used: GloVe"+
439 "-----")
440         model = load_vectors("../vectors/glove.txt.vw", False)
441         # run experiments and move results into data frame.
442         g_scores, g_syn_experiments, g_average, g_avg_cos_dists,
443 iterationCount, g_bword_counts, g_spells_per, g_bwords_per=
444 run_experiment(model, num_experiments)
445         g_vec = ["glove" for x in g_scores]
446
447         scores=w_scores + g_scores
448         syn_experiments = w_syn_experiments + g_syn_experiments
449         avg_cos_dists = w_avg_cos_dists + g_avg_cos_dists
450         bword_counts = w_bword_counts + g_bword_counts
451         vectors = w_vec + g_vec
452
453         ##for the ts plots
454         g_vec = ["GloVe" for x in g_spells_per]
455         w_vec = ["Word2Vec" for x in w_spells_per]
456         bwords_per = w_bwords_per + g_bwords_per
457         spells_per = w_spells_per + g_spells_per
458         vec = w_vec + g_vec
459
460         ##adds values for empty rows.#might want to remove empty rows
461 later.
462         for row in spells_per:
463             if len(row) == 0:
464                 row.append(0)
465         for row in bwords_per:
466             if len(row) == 0:
467                 row.append(0)
468
469         spells_per_avg = [float(sum(l)/len(l)) for l in spells_per]
470         length= [x for x in range(1, len(w_spells_per)+1)] + [x for x
471 in range(1, len(g_spells_per)+1)]
472         bwords_per_avg = [float(sum(l)/len(l)) for l in bwords_per]
473         len_results =
474 pd.DataFrame({"originality":spells_per_avg,"length":length,"bwords":b
475 words_per_avg, "vectors":vec})
476
477         box_len = []
478         box_score= []
479         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
510         for row2 in g_bwords_per[i]:
511             box_len.append(i+1)
512             box_score.append(row2)
513             box_vec.append("GloVe")
514
515     box_data = pd.DataFrame({"length":box_len,
516 "bwords":box_score, "vectors":box_vec})
517
518     #histogram
519     ax = sns.distplot(box_score)
520     sns.plt.show()
521     #box plot
522     ax = sns.boxplot(x="length", y = "bwords", hue="vectors",
523 data=box_data)
524     sns.plt.show()
525     ##output results.
526
527     print("-----word2vec Experiment
528 Results-----")
529     print("The mean average percentage over ", iterationCount ,
530 "tests: ",

```

```

518         (w_average/iterationCount), "%")
519     print("The mean cosine similarity over ", iterationCount,
520 "tests: ",
521         float(sum(w_avg_cos_dists)/ len(w_avg_cos_dists)))
522     print("The mean amount of synonyms", (sum(w_syn_experiments)/
523 iterationCount))
524     print("Average number of words that are not fit for
525 translation: ",float(sum(w_bword_counts)/iterationCount))
526
527     print("-----GloVe Experiment
528 Results-----")
529     print("The mean average percentage over ", iterationCount ,
530 "tests: ",
531         (g_average/iterationCount), "%")
532     print("The mean cosine similarity over ", iterationCount,
533 "tests: ",
534         float(sum(g_avg_cos_dists)/ len(g_avg_cos_dists)))
535     print("The mean amount of synonyms", (sum(g_syn_experiments)/
536 iterationCount))
537     print("Average number of words that are not fit for
538 translation: ",float(sum(g_bword_counts)/iterationCount))
539
540
541     results = pd.DataFrame({"scores":scores,
542 "similarity":avg_cos_dists, "synonyms":syn_experiments,
543 "vectors":vectors, "bwords":bword_counts})
544
545     sim = sns.violinplot(x="vectors", y="similarity",
546 data=results)
547     sns.plt.title("Comparison of Similarity over "+str(
548 iterationCount)+ " experiments")
549     sns.plt.show()
550     sc = sns.violinplot(x="vectors", y="scores", data=results)
551     sns.plt.title("Comparison of accuracy scores over
552 "+str(iterationCount)+ " experiments")
553     sns.plt.show()
554     bw = sns.violinplot(x="vectors", y="bwords", data=results)
555     sns.plt.title("Comparison of invalid words over "+
556 str(iterationCount)+ " experiments")
557     sns.plt.show()
558     syn = sns.violinplot(x="vectors", y="synonyms", data=results)
559     sns.plt.show()
560
561     else: # test an individual mode.
562         if args.glove:
563             print("Vectors used: GloVe")
564             log("-----" + "Vectors used: GloVe"+
565 "-----")
566             model = load_vectors(".././vectors/glove.txt.vw", False)
567         else:
568             print("Vectors used: Word2Vec")
569             log("-----"+ "Vectors used: Word2Vec"+
570 "-----")
571             model = load_vectors(".././vectors/GoogleNews-vectors-

```

```

565 negative300.bin", True)
566
567     scores, syn_experiments, average, avg_cos_dists,
568     iterationCount, bword_counts, spells_per, bwords_per=
569     run_experiment(model, num_experiments)
570     print("-----Experiment Results-----")
571     print("The mean average percentage over ", iterationCount ,
572 "tests: ",
573         (average/iterationCount), "%")
574     print("The mean cosine simalarity over ", iterationCount,
575 "tests: ",
576         float(sum(avg_cos_dists)/ len(avg_cos_dists)))
577     print("The mean amount of synonyms", (sum(syn_experiments)/
578 iterationCount))
579     print("Average number of words that are not fit for
580 translation: ", float(sum(bword_counts)/iterationCount))
581     results = pd.DataFrame({'scores': scores, 'similarity':
582 avg_cos_dists})
583
584     #loop through and add an entry to any empty fields.
585     for row in spells_per:
586         if len(row) == 0:
587             row.append(0)
588
589     spells_per_avg = [float(sum(l)/len(l)) for l in spells_per]
590     length= [x for x in range(0, len(spells_per_avg))]
591
592     vec = ["vector" for x in spells_per_avg]
593     sns.plt.show()
594     ax2 = sns.violinplot(x=results["similarity"])
595     sns.plt.show()
596     ax = sns.violinplot(x="scores", y="similarity", data=results)
597     sns.plt.show()
598
599
600
601
602
603
604
605
606
607
608
609
610
611

```

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