

```

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
20     language exists in the list the value is append by 1 otherwise,
21     it is appended to
22     the end of the list with a value of 1.
23     :param kwords: List of spell types and language with associated
24     frequencies.
25     :param word: One being the spell type and the other being the
26     origin language.
27     :type kwords: [[str, str], int]...
28     :type word: str
29     :return: the updated list of known words.
30     """
31     found = False
32     for kword in kwords:
33         if kword[0] == word:
34             kword[1] += 1
35             found = True
36     if found == False:
37         kwords.append([word, int(1)])
38     return kwords
39
40 def count_instances(fname):
41     """
42     Reads supplied file, where it splits it up. Then it appends
43     each word to the data
44     set building a list of words and frequencies using
45     checkStoredWords(kwords, word).
46     :param fname: This is the name of the CSV file in which the spell
47     data is stored.
48     :type fname: str
49     :return: returns a list of languages and the probabilities for
50     each one.

```

```

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

```

```

95
96     :param data: list of spell names and their associated frequencies
97 and probabilities.
98     :type data: [[str,str],int,float]...
99     :return: a list of spells and the value between 0-1 in which that
100 name will be selected.
101     """
102
103     value = 0
104     index = -1
105     scale = []
106     for d in data:
107         value += d[2]
108         index += 1
109         scale.append((value, d[0]))
110     return scale
111
112 def getSpellType(scale, rndNum):
113     """
114     Selects a spell according to the random number passed.
115
116     :param scale: A list of tuples which contains the probability
117 associated with each spell and type.
118     :param rndNum: The random number used to select a spell type.
119     :type scale: [(str,str,float)..]
120     :type rndNum: float
121     :return: A string which is the spell type.
122     """
123
124     for i in range(-1, len(scale) - 1):
125         if i == -1:
126             temp2 = scale[i + 1]
127             if rndNum >= 0:
128                 if rndNum < temp2[0]:
129                     return temp2[1]
130             else:
131                 temp = scale[i]
132                 temp2 = scale[i + 1]
133                 if rndNum >= temp[0]:
134                     if rndNum < temp2[0]:
135                         return temp2[1]
136
137     temp2 = scale[0]
138     return temp2[1]
139
140 def is_valid(string):
141     """
142     check to see whether a word consists of alpha characters.
143
144     :param string: The string to be checked.
145     :type string: str
146     :return: Boolean value.
147     """
148     if string.isalpha():
149         return False

```

```

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

```

```

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

```

```

236         i = source[k]
237         if i[0] in ['a', 'e', 'i', 'o', 'u']:
238             source[k] = i + 'ay'
239         elif f(i) in letters:
240             source[k] = i[2:] + i[:2] + 'ay'
241         elif i.isalpha() == False:
242             source[k] = i
243         else:
244             source[k] = i[1:] + i[0] + 'ay'
245     return ' '.join(source)
246
247 def f(str):
248     """
249     Returns the first two chacters from the string.
250
251     :param str: A word that is passed.
252     :type str: str
253     :return: a string that only contains the first two letters.
254
255     """
256     if len(str) == 1:
257         return str[0]
258     return str[0] + str[1]
259
260 def generateSpell(sentence, model, oword):
261     """
262     Generates a Spell from a sentence.
263
264     :param sentence: string which is the definition of the spell you
265     want to create.
266     :type sentence: str
267     :return: list containing the spell and the spell type.
268     :param model: loaded vector orepresentation of words.
269     :type model: data file loaded.
270
271     """
272     spell = []
273     vector, temp_bogus = sentenceToWord(sentence, model, oword)
274     vector = vector[0]
275     scale = generateScale(count_instances('spell_prob.csv'))
276     selection = random.random()
277     spell_meta = getSpellType(scale, selection)
278
279     try:
280         target_lang = langCode(spell_meta[1])
281     except:
282         log("langCode function didn't work. Using default latin.")
283         target_lang = "la"
284
285     if target_lang == "PL":
286         spell.append(pigLatin(vector))
287     else:
288         spell.append(translate2(vector, target_lang))
289     spell.append(spell_meta[0])
290     spell.append(vector) #The original word before translation is
291     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 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
331     :param model: The vectors loaded.
332     :type model: The loaded vector object
333     :param num_experiments: The number of experiments to run.
334     :type num_experiments: int
335     :return: A list of averages scores, one entry per experiment.
336     :return: A list of the average number of synonyms produced,
337     one entry per experiment.

```

```

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

```



```

377
378         if is_synonym(spell[2].lower(), entry[-1]):
379             syn_counts +=1
380
381         print("Experiment Results")
382         print("Num of spells that don't feature in definition: ",
383 score)
384         print("Percentage: ", ((float(score)/count) * 100),"%")
385         print("Average Cosine-similarity:", float(sum(cos_dists) /
386 len(cos_dists)))
387         print("Num of spells which are synonyms: ", syn_counts)
388         print("Num of words selected that are not real words: ",
389 bogus_words)
390         scores.append((float(score)/count) * 100)
391         syn_experiments.append(syn_counts)
392         bword_counts.append(bogus_words)
393         spellFile.close()
394         iterationCount +=1
395         average += (float(score)/count)*100
396         avg_cos_dists.append(float(sum(cos_dists) / len(cos_dists)))
397     return scores, syn_experiments, average, avg_cos_dists,
398 iterationCount, bword_counts, scores_per_spell, bwords_spell
399
400 #
401 =====
402 =====
403 # Main part of the program.
404 #
405 =====
406 =====
407 if __name__ == '__main__':
408     parser = argparse.ArgumentParser(
409         'Use Word2Vec or GloVe datasets to generate Harry Potter
410 Spells')
411     parser.add_argument('--glove', action='store_const', const =
412 'glove',
413         help='Use the GloVe dataset instead of the default
414 Word2Vec.')
415     parser.add_argument('--exp',
416         help="Specifies the number of experiments on this run. Default is
417 20.",
418         action='store', type=int)
419     parser.add_argument('--verbose', action='store_const', const =
420 'verbose',
421         help='Prints out the spell names')
422     parser.add_argument('--comp', action= 'store_const',
423 const='comp',
424         help = "Runs the word2vec vectors, and the GloVe
425 vectors")
426     args = parser.parse_args()
427
428     logFile = open("log.txt", 'w' )
429     logFile.close()
430     num_experiments = 20
431
432     if args.exp != None:
433         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-
431 negative300.bin", True)
432
433         #Run word2vec experiments and then stores data in dataframe.
434         w_scores, w_syn_experiments, w_average, w_avg_cos_dists,
iterationCount, w_bword_counts, w_spells_per, w_bwords_per=
435 run_experiment(model, num_experiments)
436         w_vec=["word2vec" for x in w_scores]
437         del model
438         print("Vectors used: GloVe")
439         log("-----" + "Vectors used: GloVe"+
"-----")
440         model = load_vectors(".././vectors/glove.txt.vw", False)
441
442         # run experiments and move results into data frame.
443         g_scores, g_syn_experiments, g_average, g_avg_cos_dists,
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
in range(1, len(g_spells_per)+1)]
471         bwords_per_avg = [float(sum(l)/len(l)) for l in bwords_per]
472         len_results =
pd.DataFrame({"originality":spells_per_avg,"length":length,"bwords":b
words_per_avg, "vectors":vec})

```

```

471         box_len = []
472         box_score= []
473         box_vec=[]
474
475         for i in range(0, len(w_spells_per)):
476             for row in w_spells_per[i]:
477                 box_len.append(i+1)
478                 box_score.append(row)
479                 box_vec.append("word2vec")
480
481             for row2 in g_spells_per[i]:
482                 box_len.append(i+1)
483                 box_score.append(row2)
484                 box_vec.append("GloVe")
485
486         box_data = pd.DataFrame({"length":box_len,
487 "originality":box_score, "vectors":box_vec})
488
489         #originality vs size plots.
490         ax = sns.tsplot(time="length", value="originality",
491 unit="vectors", condition="vectors", data=len_results )
492         # sns.plt.xticks([0,1,2,3,4,5,6,7,8,9,10])
493         sns.plt.show()
494
495         ax = sns.distplot(box_score)
496         sns.plt.show()
497         #box plot
498         ax = sns.boxplot(x="length", y = "originality",
499 hue="vectors", data=box_data)
500         sns.plt.show()
501
502         box_len = []
503         box_score= []
504         box_vec=[]
505
506         for i in range(0, len(w_bwords_per)):
507             for row in w_bwords_per[i]:
508                 box_len.append(i+1)
509                 box_score.append(row)
510                 box_vec.append("word2vec")
511
512             for row2 in g_bwords_per[i]:
513                 box_len.append(i+1)
514                 box_score.append(row2)
515                 box_vec.append("GloVe")
516
517         box_data = pd.DataFrame({"length":box_len,
518 "bwords":box_score, "vectors":box_vec})
519         #histogram
520         ax = sns.distplot(box_score)
521         sns.plt.show()
522         #box plot
523         ax = sns.boxplot(x="length", y = "bwords", hue="vectors",
524 data=box_data)
525         sns.plt.show()
526         ##output results.
527
528         print("-----word2vec Experiment

```

```

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

```

```

565         log("-----"+ "Vectors used: Word2Vec"+
566 "-----")
567         model = load_vectors("../../vectors/GoogleNews-vectors-
568 negative300.bin", True)
569
570         scores, syn_experiments, average, avg_cos_dists,
571 iterationCount, bword_counts, spells_per, bwords_per=
572 run_experiment(model, num_experiments)
573         print("-----Experiment Results-----")
574         print("The mean average percentage over ", iterationCount ,
575 "tests: ",
576             (average/iterationCount), "%")
577         print("The mean cosine similarity over ", iterationCount,
578 "tests: ",
579             float(sum(avg_cos_dists)/ len(avg_cos_dists)))
580         print("The mean amount of synonyms", (sum(syn_experiments)/
581 iterationCount))
582         print("Average number of words that are not fit for
583 translation: ", float(sum(bword_counts)/iterationCount))
584         results = pd.DataFrame({'scores': scores, 'similarity':
585 avg_cos_dists})
586
587         #loop through and add an entry to any empty fields.
588         for row in spells_per:
589             if len(row) == 0:
590                 row.append(0)
591
592         spells_per_avg = [float(sum(l)/len(l)) for l in spells_per]
593         length= [x for x in range(0, len(spells_per_avg))]
594
595         vec = ["vector" for x in spells_per_avg]
596         sns.plt.show()
597         ax2 = sns.violinplot(x=results["similarity"])
598         sns.plt.show()
599         ax = sns.violinplot(x="scores", y="similarity", data=results)
600         sns.plt.show()
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
632
633
634
635