**1) Splitting spaceless words (like hashtags)**

**File**: utilities/wordSeparator.py

2 important functions in this file:

**parseTagSingleWord**(term): **input** is a single word, eg #helloworld or helloworld. Note it can have or not have a hashtag at the starting. Output is a string with space separated words ‘hello world’

**parseTag**(term): term can be a list of strings or a single string, eg: [‘#helloworld’, ‘holycow’], or ‘holycow’. So basically its functionality is a superset of parseTagSingleWord(). Output is a list of space separated strings [‘hello world’, ‘holy cow’]

**Note**: if input is #Hello output is hello, that is ‘caps’ is converted to lower case. Except for cases like IBMRocks, which splits to ‘IBM rocks’. That is group of caps are preserved as they are probably a single entity.

The function works with or without hashtag at the beginning of the words.

**2) Spell checking and clean up**

**File:** utilities/spellChecker.py

2 important functions in this file:

**correct**(word): corrects a single word. Suggested not to use this because it has to compute the variable NWORDS everytime you call this function.

**spellCorrectTokenizedTweets**(TokenizedTweets, ignoreTags, threshold):

**Inputs**: TokenizedTweets: list of tokenized tweets as got from khanh’s twokenizer. Eg [[(), ()], [(), (), ()]], each if the tuples in the example are things like (‘cow’, ‘N’, 0.95)

ignoreTags: lets you specify tags where you don’t want spell check to happen (eg ‘#’, ‘E’, etc). Sugested tags to ignore are: ['E', ',', 'U', '&', '^', '!', '#']

threshold: perform spellcheck only when confidence is above threshold.

Returns in same format as TokenizedTweets, except with spellings corrected.

**3) Normalizing tweets**

**File:** normalizeTweets.py

Main functions: wordReplace and normTweet1

**wordReplace** (tokenizedTweets, ignoreTags, threshold): Works exactly the same way as **spellCorrectTokenizedTweets.** It’s a simple spelling corrector. For input output description see spellCorrectTokenizedTweets.

**normTweet1**(tweets, ops = [0,1], retain = 1, separateTokens = ['#', 'E'])

**input**: tweets: list of strings that are tweets eg [‘hello guys’, ‘zombie attack’]

ops: suggest you pass [0,1]. Op[0] is wordReplace function and op[1] is spellCorrectTokenizedTweets functions. Basically lets you choose what kind of spell check you want to apply.

separateTokens and retain: more details below

**description**:

Input is say: [‘#helloBye there’, ‘an :) #spelingMistake’, ‘Wrld’], retain = 0

First each tweet is twokenized using Khanh’s function tokenizeCMUPython. So we get the following output: tokenizedTweets = [[('#helloBye', '#', 0.9), ('there', 'D', 0.9)], [('an', 'D', 0.9), (':)', 'E', 0.9), ('#spelingMistake', '#', 0.9)], [('Wrld', 'N', 0.9)]]

Now getSpecialTokens is called which gives us:  
separatedByTokenType = [[[('hello bye', '#', 0.765)], [('spe ling mistake', '#', 0.8214)], []], [[], [(':)', 'E', 0.9852)], []], [[('there', 'R', 0.5307)], [('an', 'D', 0.7642)], [('Wrld', 'N', 0.3949)]]]  
separatedByTokenType is of length len(separateTokens)+1. The first list (in green) contains only ‘#’ from each tweet (for ‘#’ word separation is performed too). The second in blue contains only ‘E’ and the third contains the rest of the text. Note retain =0, there fore in ‘rest of the text’ ‘#’ and ‘E’ are not retained. If retain = 1, then ‘#’ and ‘E’ would have been retained unprocessed (without performing word separation on ‘#’). So separatedByTokenType[i] is of length 3 (as three tweets were input). Note separatedByTokenType[0][2] is empty as the 3rd tweet had no ‘#’ in it. Note that spell check is done on the last ‘the rest’ entry of separatedByTokenType, that is separatedByTokenType[-1]

Finally this is returned: return [separatedByTokenType, joinBackTokens(separatedByTokenType[-1])]

joinBackTokens simply joins back the separated out (tokenized) tweets into space separated list of texts.

So the output is a list of length 2. The first element is separatedByTokenType, the second is joinBackTokens(separatedByTokenType[-1]).

**Note**: If you only want clean up and do not want to separate out ‘#’ or ‘E’, just pass separateTokens = [].

I think the most useful way to use this function is with these arguments: (tweets, ops = [0,1], retain = 0, separateTokens = ['#', 'E'])

**Demo**: demoNormalizeTweet.py

**4) Rhyming words**

**File**: utilities/rhymingWords.py

Important functions:

How to use:

To compare 2 words:

entries = **getWordSyllableDict**(); **longestSyllablesMatch**(entries.get('bang', []), entries.get('clang', []))

To compare for pair wise rhymes for a whole sentences:

entries = **getWordSyllableDict**();tweet = ['clang bang here there fsfdfd', 'hello world'];  
**getRhymingScoreForText**(tweet, entries)

#returns this: #[[0.42857142857142855, 0, 0, 0, 0, 0, 0, 0.26666666666666666, 0, 0], [0.125]]

**5) reading Data**

**File**: utilities/readTweet.py

Important functions:

**readCSV(filename, categoryFilter = None, numFilter = None): reads the csv (not the actual tweet data)**

Some samples

allControl = [readCSV(csvFileLoc, {'condition':'control', 'fold':[i]}) for i in range(0,10) ] #read in every fold separately for control group

allSch = [readCSV(csvFileLoc, {'condition':'schizophrenia', 'fold':[i]}) for i in range(0,10) ] #read in every fold separately for sch group

x = readCSV(csvFileLoc, {'condition':'control', 'gender':'M', 'fold':[1,2]}, {'age':[(20,22), (23,25)], 'num\_tweets':[(1,1000),(2000,5000)]}) #Males in control group from fold 1 and 2, who are in age 20 to 22 or 23 to 25 and who have tweets <= 1000 or tweets between 2000 to 5000

y = readCSV(csvFileLoc, {'condition':'control', 'gender':'F', 'fold':[1,2]}, {'age':[(20,22), (23,25)], 'num\_tweets':[(1,3000)]}) #females in control group from folds 1 and 2, between age 20-22 or 23-25, who have number of tweets between 1 to 3000

So **readCSV** returns a list of dictionaries (I am calling this a ‘group’). Each dictionary represents a row in the csv and has 6 keys (6 columns in the csv). eg:

[{'anonymized\_name':’ggjdabd’, 'condition':’control’, 'age':22.6, 'gender':’M’, 'num\_tweets':2566, 'fold':5}, {'anonymized\_name':’sfgfjasdbkjak’, 'condition':’control’, 'age':45.3, 'gender':’F’, 'num\_tweets':1658, 'fold':6} … ]

**setOpsGroups(groupList, setFunc): #performs: A setfunc B setfunc C...**

eg: z = setOpsGroups([x,y], set.union) #we can also have set.intersection

Input is something from readCSV (a ‘group’), output is in similar format (‘group’).

**getNamesFromGroup**(group): input is a group (something out of readCSV, output is a list of the ‘anonymized\_name’s entries. Output list is same length as input list obviously.

**readTweetsFromFile**(inpFile, tweetFilterFunc = lambda x: True, fields = None): #can read both gz and extracted tweets. tweetFilterFunc should be a function that takes in a tweet (dict) and return true/false

First the input tweet file is read (json.loads), then tweetFilterFunc is applied to check if this tweet satisfies some condition. If it does, then return a dictionary with only those key value pairs that are present in ‘fields’. If fields == None, all key value pairs are returned.

Note tweetFilterFunc can be used to say filter out tweets, which have < 3 characters in their text.

Fields argument can be used to get only specific information, say fields = [‘text’, ‘retweeted’] will return [{‘text’:’hello world’, ‘retweeted’:False}, {‘text’:’hello world again’, ‘retweeted’:True},]. If Fields == None every thing is returned.

Output is a list of dictionaries. The length of the list is the number of tweets in that file (unless anything is filtered out by tweetFilterFunc). The keys of the dictionary depend on ‘fields’

**getTweetsForGroup**(grp, controlFolder, schizoFolder, tweetFilterFunc = lambda x: True, fields = None)

controlFolder, schizoFolder are folders conaining the zipped files.

Grp is a group (something out of readCSV)

Adds a new entry to each dictionary of the group, with key name ‘tweets’

The value of the ‘tweets’ key is a dictionary, with key-val pairs being generated from **readTweetsFromFile.** So the value of ‘tweets’ is a list of dictionaries, each dictionary represents a tweet for that user.

This function takes a long time to run. You can consider pickling and dumping its output for faster future runs. (don’t commit the pickle files though)

**getFieldFromGroup**(grp, field = 'text')

Once **getTweetsForGroup** has been used to read in the files, we can use this function on a group to get a particular field. Say field is ‘text’. It returns a list of strings which contain all the tweet text from all the users in the group. (So which tweet came from which user information is lost)

**getFieldFromGroupPerUser** (grp, field = 'text')

Returns a list of length = len(grp). That is if there were 10 users in that group a list of length 10 is formed. In each of the sublists, the data for that field is stored.

So if field = ‘text’ you get: [[‘user 1’s tweet’, ‘another of his tweets’…], [‘this is user 2’s tweet’, ‘more from user 2’…]…]

An example of how to use these reading utilities (say for doing unigram logit)

start **=** timeit**.**default\_timer**()**

allControl **=** **[**readCSV**(**csvFileLoc**,** **{**'condition'**:**'control'**,** 'fold'**:[**i**]})** **for** i **in** range**(**0**,**10**)** **]** #read in every fold separately

allSch **=** **[**readCSV**(**csvFileLoc**,** **{**'condition'**:**'schizophrenia'**,** 'fold'**:[**i**]})** **for** i **in** range**(**0**,**10**)** **]**

**print** 'read csv'**,** timeit**.**default\_timer**()** **-** start #0.16s

#get tweets by read files.

start **=** timeit**.**default\_timer**()**

allControlTweets **=** **[**getTweetsForGroup**(**user**,** control\_folder\_path**,** sch\_folder\_path**,** fields **=** **[**'text'**])** **for** user **in** allControl**]** #read the files and get the tweet (only the text as specified in 'field')

allSchTweets **=** **[**getTweetsForGroup**(**user**,** control\_folder\_path**,** sch\_folder\_path**,** fields **=** **[**'text'**])** **for** user **in** allSch**]**

**print** 'got tweets'**,** timeit**.**default\_timer**()** **-** start #267s

**with** open**(**'dumpdata1\_unigramLogit.pickle'**,** 'w'**)** **as** f**:**

pickle**.**dump**([**allControlTweets**,** allSchTweets**],** f**)**

#get a particular field from above data structure in list form (so the next lines return all tweet texts in a group (fold) as a list

start **=** timeit**.**default\_timer**()**

#allControlTweets is a list of groups of length 10 (since there are 10 folds)

#so for each element of allControlTweets (which is a group in a particular fold), we iterate over each element of the group (ie each user) and get their tweet text.

# note its getFieldFromGroup([user], 'text') and not getFieldFromGroup(user, 'text'), since getFieldFromGroup is a group, and [user] makes it a group with a single user.

allControlTextList **=** **[**getFieldFromGroup**(**foldGroup**,** 'text'**)** **for** foldGroup **in** allControlTweets**]** #so allControlTextList is a list of size 10 (denoting 10 folds) each element of this 10 length list is a list (say x). x is a list of lists (say y). y contains all the tweets text of a particular user

allSchTextList **=** **[**getFieldFromGroup**(**foldGroup**,** 'text'**)** **for** foldGroup **in** allSchTweets**]**

#so allControlTextList[2][4][1] is the 2nd fold's 4th user's 1st tweet

#join all strings in a fold

start **=** timeit**.**default\_timer**()**

allControlTextList **=** **[[(**' '**).**join**(**user**)** **for** user **in** fold**]** **for** fold **in** allControlTextList**]**

allSchTextList **=** **[[(**' '**).**join**(**user**)** **for** user **in** fold**]** **for** fold **in** allSchTextList**]**

**for testfoldID in range(0,10):**

**[X\_train, Y\_train, X\_test, Y\_test] = splitData(allControlTextList, allSchTextList, testfoldID, randomize =1) #implement splitData**

**#Do training stuff**