Sentimental Analysis?

Sentimental Analysis is a task to analyze given text to measure it's polarity. Polarity is the net attitude expressed by an individual regarding a certain text.

"Since the beginning of times, short-term traders can only rely on just 2 factors to predict returns: price and volume", says Ernest Chan, Managing Director of QTS Capital.
- Addition of sentiment as another short-term factor is a very effective approach.

Why Sentimental Analysis?

According to Google Trends, a good daily search estimator, the word "sentiment analysis" has been gaining steady traction over the past 5 years.
As humans, we can easily interpret the attitude of an article. However, the problem lies when this number reaches to an impossible value. For a trader with portfolio in 1000s of companies, with many of them having frequent online mentions, keeping track of sentiments corresponding to each and news article is just impossible. This existence of valuable massive text data creates the need for sentimental analysis.

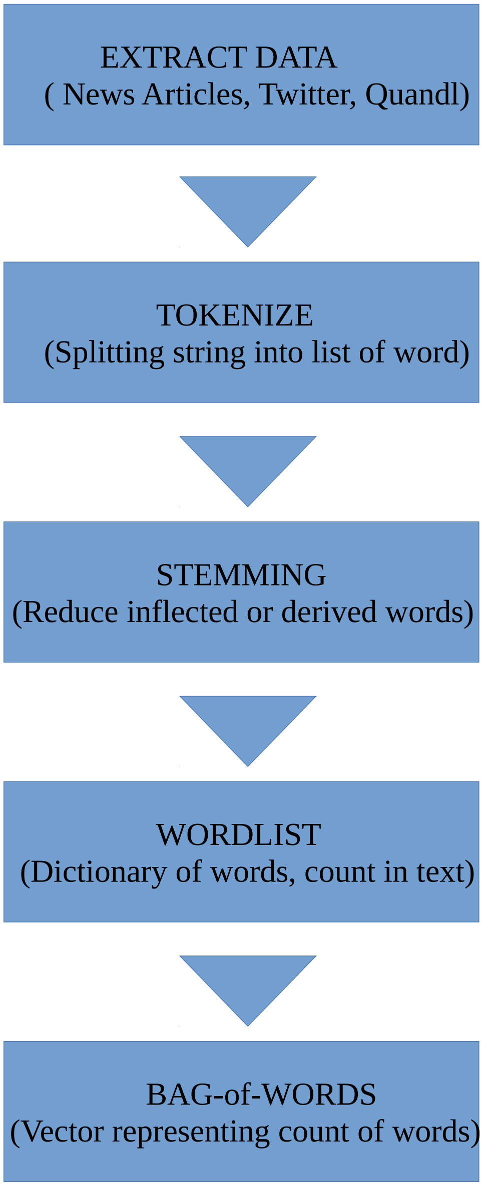
Bag-of-Words

Bag-of-words is one of the most conventional approaches to sentiment analysis. It is a method to keep track of number of occurrences different words in a text. This individual occurrences can be treated as features or columns for text classifiers. The resulting vector, representing the occurrences, is multiplied by the polarity corresponding to each word. For the basics, we take values from { -1 , 0 , 1}. The sum of this multiplication is the net polarity corresponding to the text.

Note : In place unigrams (taking vector of individual words), combination of unigram and bigram (vector of pair of words) will be more efficient.

Steps in Bag-of-Words :

I) Extracting Data
ii) Tokenizing words
iii) Stemming
iv) CreateWordlist
iv) Generate Bag-of-Words


EXTRACT DATA
(News Articles, Twitter, Quandl)
TOKENIZE
(Splitting string into list of word)
STEMMING
(Reduce inflected or derived words)
WORDLIST
(Dictionary of words, count in text)
BAG-of-WORDS
(Vector representing count of words)

Getting Data

quandl :

```
url = "https://www.quandl.com/api/v3/datasets/{database code}/{dataset
        code}/data.csv?api_key={YOUR API KEY}"
import urllib2
response = urllib2.urlopen(url)
data = pd.read_csv(response)
```

twitter :

```
import tweepy
from tweepy import OAuthHandler

consumer_key = 'XXXXXXXXXXXXXXXXXXXXXXXXX'
consumer_secret = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
access_token = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
access_token_secret = 'XXXXXXXXXXXXXXXXXXXXXXXXXX'

self.auth = OAuthHandler( consumer_key, consumer_secret )
self.auth.set_access_token( access_token, access_token_secret)
self.api = tweepy.API(self.auth)

fetched_tweets = self.api.search( q = query, c = count)
```

moneycontrol :

```
def extract_article(script):
     pattern = re.compile('(\"[a-zA-Z0-9@]+\")(\:)(\s)*(\"[^\"]+\")')
     for match in re.finditer(pattern, script.text):
          script_extract = match.groups()
          if script_extract[0] == '"articleBody"': return script_extract[3]
```

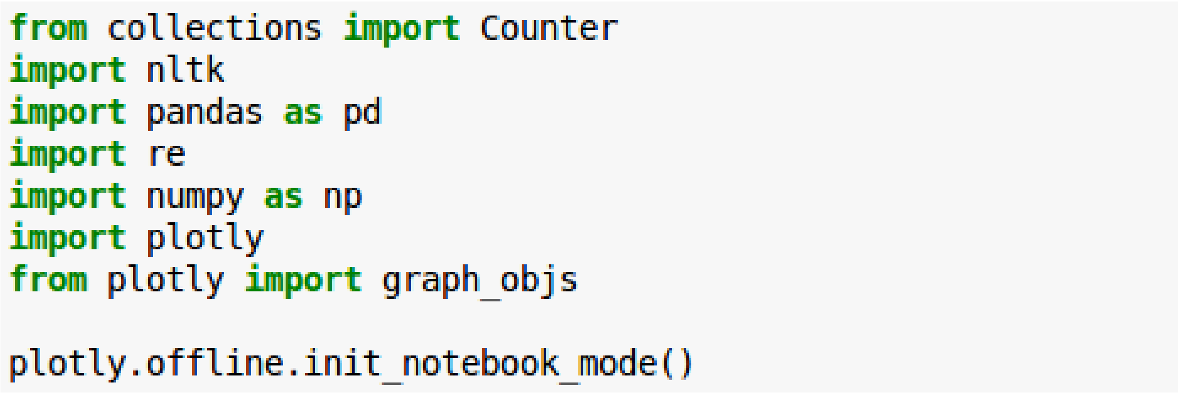
```
url = 'http://www.moneycontrol.com/company-article/{ Company
       Name }/news/{ Company Code }#{ Company Code }'
html = requests.get(url_list)
soup = BeautifulSoup(html.text, 'html.parser')
link = soup.find('a', class_='arial11_summ')
lxml = requests.get(link)
soup = BeautifulSoup(lxml.text, 'lxml')
scripts = soup.findAll('script', type='application/ld+json')

article = [extract_article(script).replace('\r\n', ' ') for script in scripts if
                     extract_article(script) != None]
```

We'll use the extracted twitter data.

Process

1. Initialize - This step defines the attributes of data according to our needs and initialize them.
2. Cleansing - This step cleans the twitter data by removing URLs and unwanted characters from text.
3. Tokenize - For this step we'll use nltk.word_tokenize() method.
4. Stemming -The step uses stem() method from nltk.PorterStemmer().
5. Wordlist - This step creates a dictionary of words with their numer of occurrences in the text data.
6. Bag-of-Words- Transform the processed data into bag-of-words representation.

IMPORTS



```
from collections import Counter
import nltk
import pandas as pd
import re
import numpy as np
import plotly
from plotly import graph_objs

plotly.offline.init_notebook_mode()
```

numpy - It is library built for arrays/matrices with different mathematical operations.

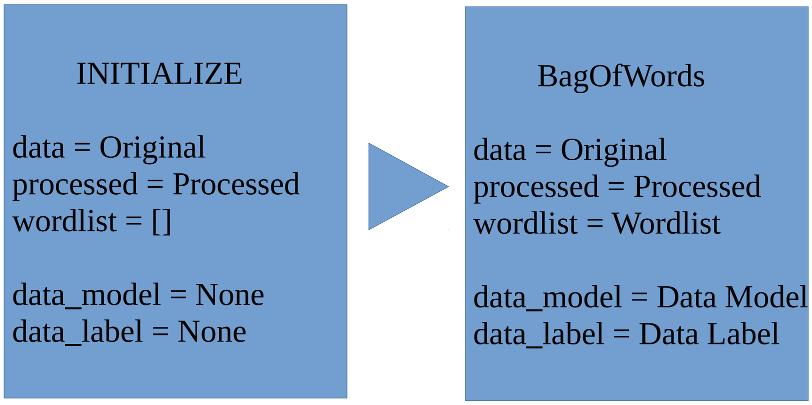
pandas - It is a dataframe library, with methods to operate dataframes.

Counter - It is a dict subclass of Collections for counting hashable objects.

re - It is a module to provide regular expression matching operations.

nltk - It is a suite of libraries and programs for symbolic and statistical natural language processing for English.

plotly - It is a package consisting data analytics and visualization tools.



INITIALIZE

data = Original
processed = Processed
wordlist = []

data_model = None
data_label = None

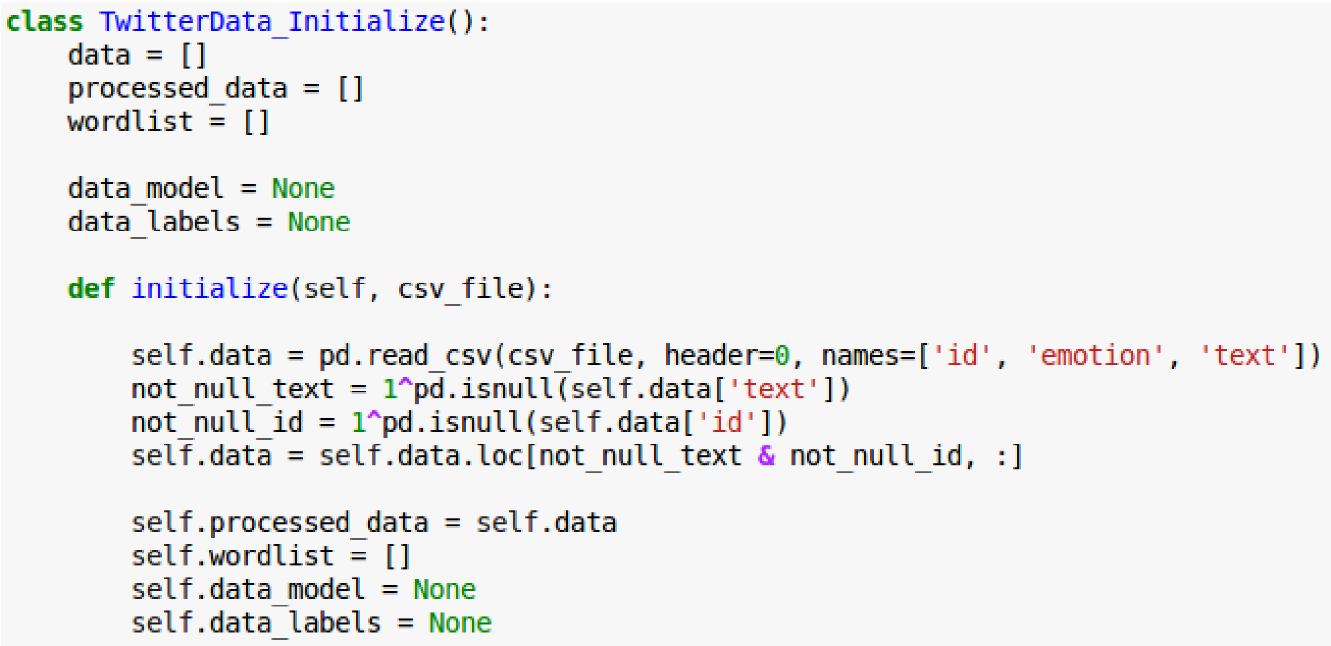
BagOfWords

data = Original
processed = Processed
wordlist = Wordlist

data_model = Data Model
data_label = Data Label

Initialize

This is a class to define attributes that should be present in the data.
At any line of code, we may need original data, processed data, wordlist, data-model, data-labels. **[We'll talk about wordlist, data-model and data-labels in future]**



```
class TwitterData_Initialize():
    data = []
    processed_data = []
    wordlist = []

    data_model = None
    data_labels = None

    def initialize(self, csv_file):

        self.data = pd.read_csv(csv_file, header=0, names=['id', 'emotion', 'text'])
        not_null_text = 1^pd.isnull(self.data['text'])
        not_null_id = 1^pd.isnull(self.data['id'])
        self.data = self.data.loc[not_null_text & not_null_id, :]

        self.processed_data = self.data
        self.wordlist = []
        self.data_model = None
        self.data_labels = None
```

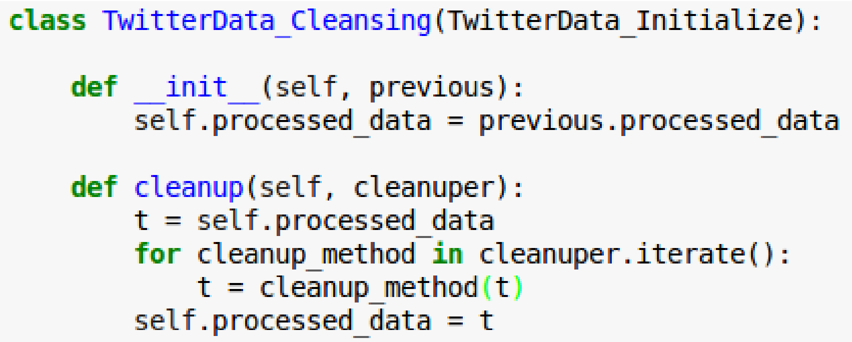
pd.isnull outputs the series of True/False for the column

^ : XOR operator; 1^0 = 1 ; 1^1 = 0; 0^0 = 1

not_null_id & not_null_text will remove the indices with null value for either id or text

Cleansing

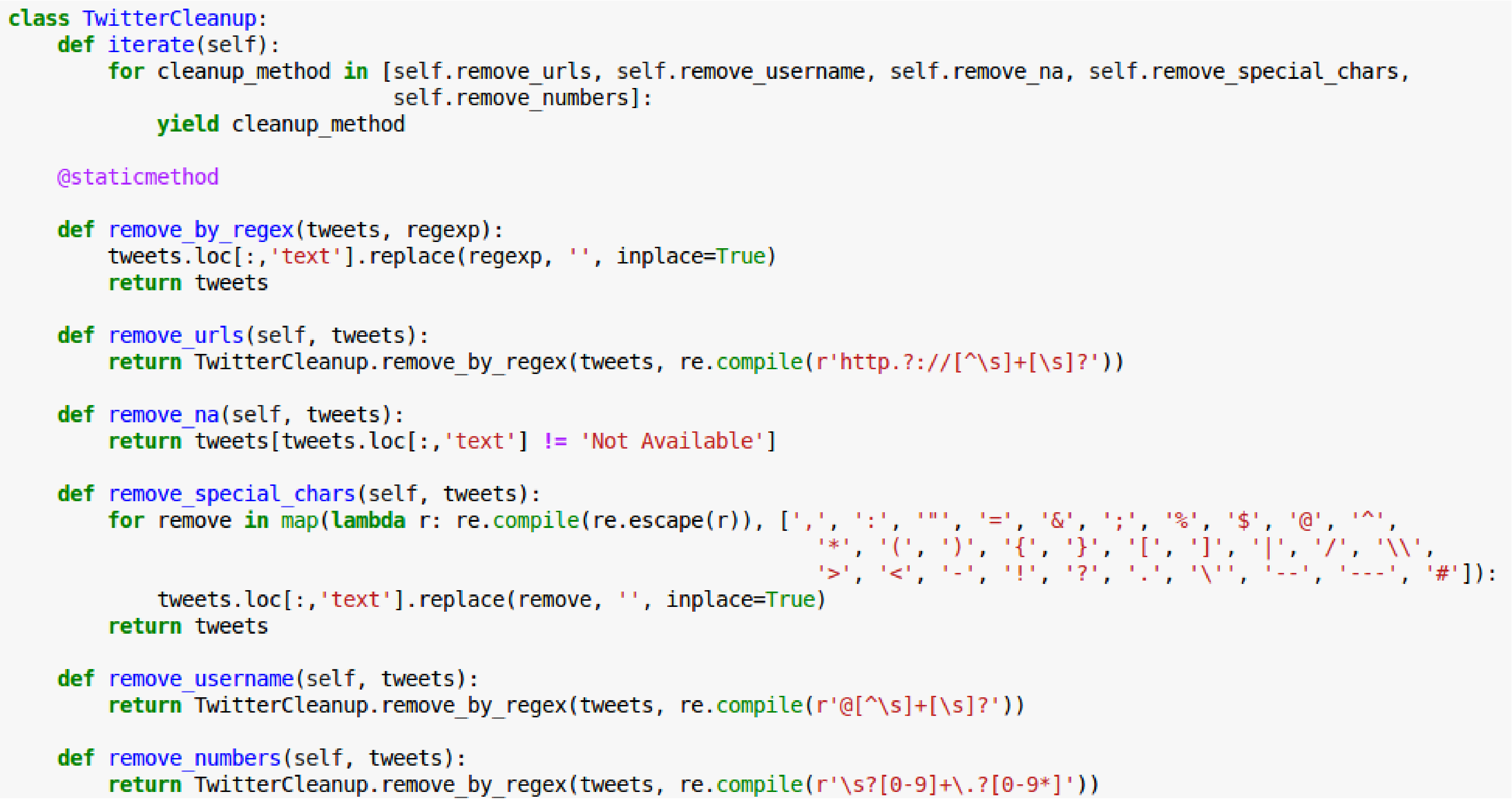
This is a subclass to **TwitterData_Initialize**. It will clean the data by removing unwanted texts : URLs, usernames, numbers, special characters and NA values.



```python
class TwitterData_Cleansing(TwitterData_Initialize):

    def __init__(self, previous):
        self.processed_data = previous.processed_data

    def cleanup(self, cleanuper):
        t = self.processed_data
        for cleanup_method in cleanuper.iterate():
            t = cleanup_method(t)
        self.processed_data = t
```

cleanuper is an object of class **TwitterCleanup**.

Class TwitterCleanup contains the iterate, remove_urls, remove_na, remove_na, remove_special_chars, remove_username and remove_numbers class methods and remove_by regex static method.



```python
class TwitterCleanup:
    def iterate(self):
        for cleanup_method in [self.remove_urls, self.remove_username, self.remove_na, self.remove_special_chars,
                               self.remove_numbers]:
            yield cleanup_method

    @staticmethod

    def remove_by_regex(tweets, regexp):
        tweets.loc[:,'text'].replace(regexp, '', inplace=True)
        return tweets

    def remove_urls(self, tweets):
        return TwitterCleanup.remove_by_regex(tweets, re.compile(r'http.?://[^\s]+[\s]?'))

    def remove_na(self, tweets):
        return tweets[tweets.loc[:,'text'] != 'Not Available']

    def remove_special_chars(self, tweets):
        for remove in map(lambda r: re.compile(re.escape(r)), [',', ':', '"', '=', '&', ';', '%', '$', '@', '^',
                                                               '*', '(', ')', '{', '}', '[', ']', '|', '/', '\\',
                                                               '>', '<', '-', '!', '?', '.', '\'', '--', '---', '#']):
            tweets.loc[:,'text'].replace(remove, '', inplace=True)
        return tweets

    def remove_username(self, tweets):
        return TwitterCleanup.remove_by_regex(tweets, re.compile(r'@[^\s]+[\s]?'))

    def remove_numbers(self, tweets):
        return TwitterCleanup.remove_by_regex(tweets, re.compile(r'\s?[0-9]+\.?[0-9*]'))
```

A static method is specific to class instead of class object. Hence, it dœsn't contain self argument.

itertate() creates a generator function containing iterator of different cleaning methods.

yield is used when defining a generator function, it is used like return except the function here return a generator.

Tokenize and Stemming

Tokenization is the act of breaking up a sequence of strings into pieces such as words, keywords, phrases, symbols and other elements called tokens.

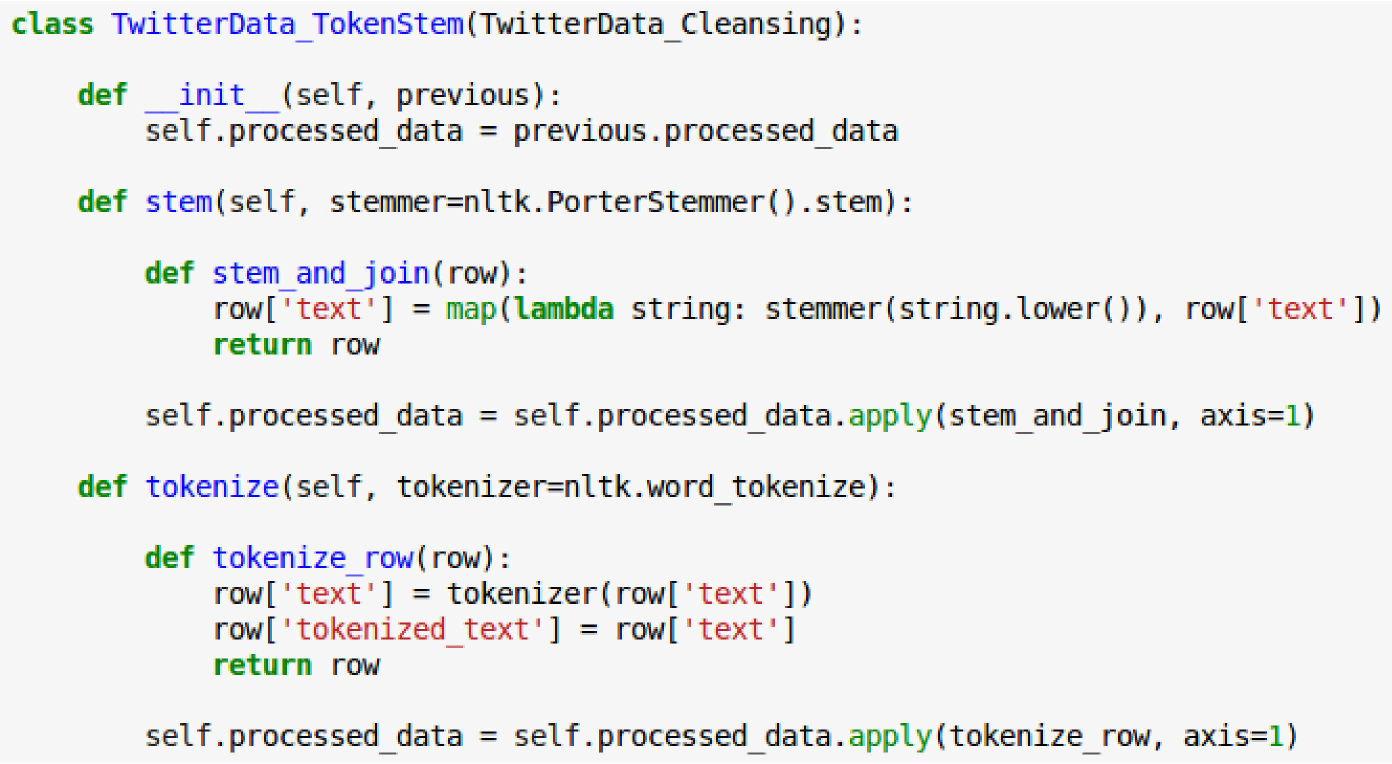
Here, we'll break the text into words.

tokenize('he is running on the street') → ['he', 'is', 'running', 'on', 'the', 'street']

Stemming is the act of reducing inflected or derived words.

stem('runs') → 'run'

TwitterData_TokenStem is a subclass to TwitterData_Cleansing. It contains the two methods tokenize(convert to tokens) and stem(word stemming).



```
class TwitterData_TokenStem(TwitterData_Cleansing):

    def __init__(self, previous):
        self.processed_data = previous.processed_data

    def stem(self, stemmer=nltk.PorterStemmer().stem):

        def stem_and_join(row):
            row['text'] = map(lambda string: stemmer(string.lower()), row['text'])
            return row

        self.processed_data = self.processed_data.apply(stem_and_join, axis=1)

    def tokenize(self, tokenizer=nltk.word_tokenize):

        def tokenize_row(row):
            row['text'] = tokenizer(row['text'])
            row['tokenized_text'] = row['text']
            return row

        self.processed_data = self.processed_data.apply(tokenize_row, axis=1)
```

map(function, iterator) returns the list of outputs of function taking one by one elements of iterator as an argument.

apply(function, axis=1) applies the function on each row.

Here, for stemming we have used nltk.PorterStemmer.stem().

For tokenize we have used nltk.word_tokenize.

Both the methods are applied row-wise thus apply() is used.

row['text'] represents a list of words. map takes each word(string) convert it to lower case using lower() and stems it using stemmer()

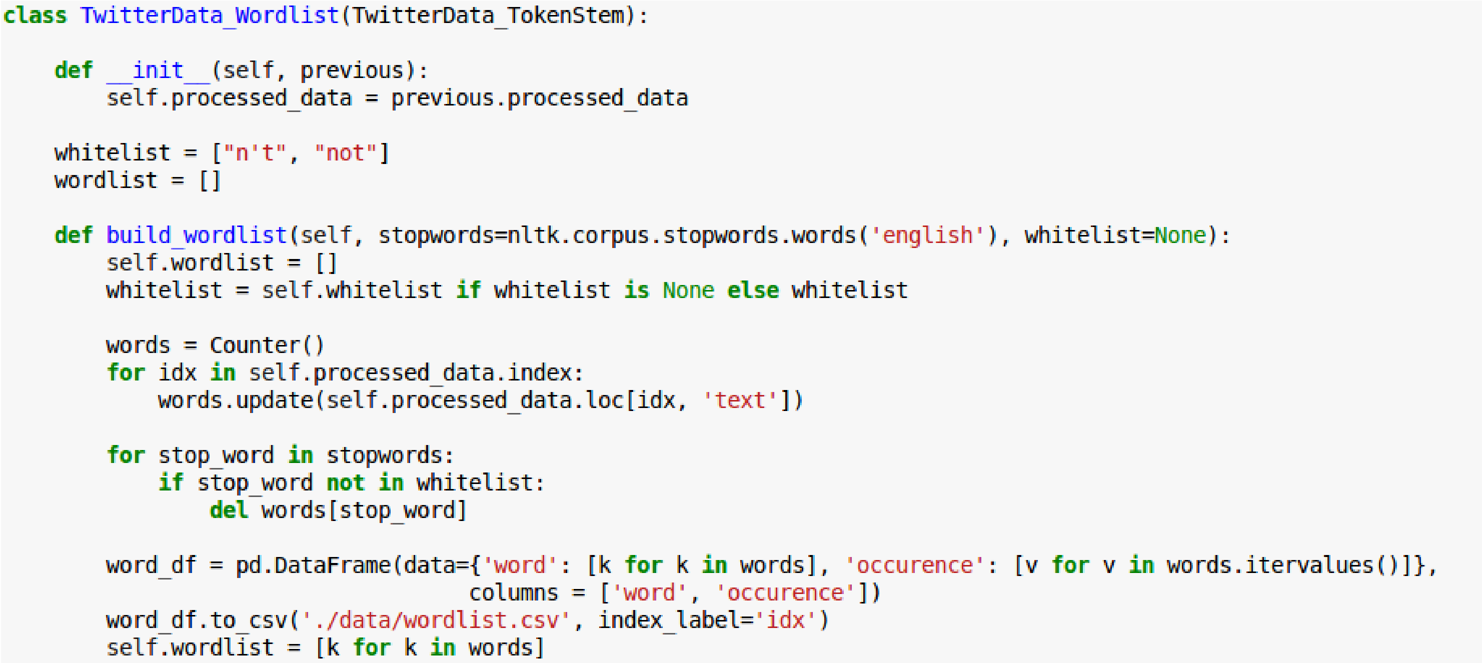
Wordlist

Wordlist is a dictionary of words in the text dataset with (word, no. of occurrences) as (key, value) pair of dictionary.

Whitelist is the list of words with high polarity like not, n't, these words plays major role in text analytics, but can also have high occurrences in the text and thus they can be a part of stop words too. Thus, while filtering stopwords we'll check for the existence of these words and skip them.

Stopswords are the most common words in general english text which does not lead to any information and thus are filtered out before processing natural language text data.

TwitterData_Wordlist is a subclass of TwitterData_TokenStem.



```
class TwitterData_Wordlist(TwitterData_TokenStem):

    def __init__(self, previous):
        self.processed_data = previous.processed_data

    whitelist = ["n't", "not"]
    wordlist = []

    def build_wordlist(self, stopwords=nltk.corpus.stopwords.words('english'), whitelist=None):
        self.wordlist = []
        whitelist = self.whitelist if whitelist is None else whitelist

        words = Counter()
        for idx in self.processed_data.index:
            words.update(self.processed_data.loc[idx, 'text'])

        for stop_word in stopwords:
            if stop_word not in whitelist:
                del words[stop_word]

        word_df = pd.DataFrame(data={'word': [k for k in words], 'occurence': [v for v in words.itervalues()]},
                               columns = ['word', 'occurence'])
        word_df.to_csv('./data/wordlist.csv', index_label='idx')
        self.wordlist = [k for k in words]
```

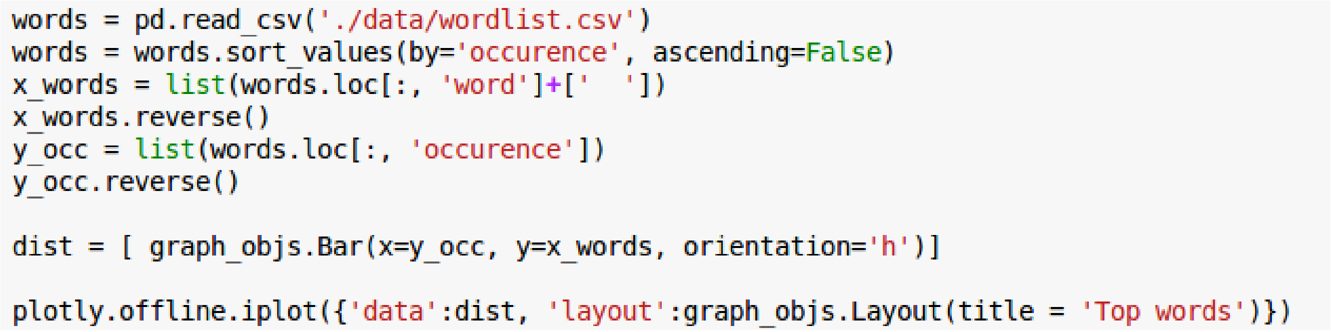
To create the dictionary for wordlist, we used Counter which is an alternative to Python's general purpose container dict.

For stopwords, we used nltk.stopwords.words('english').

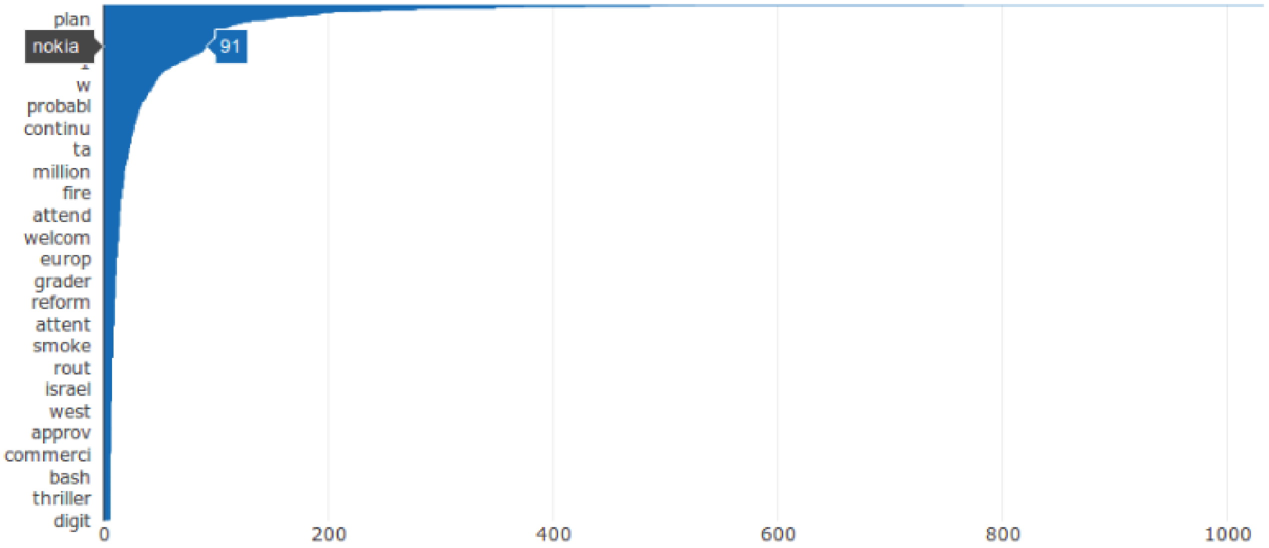
Since the argument wordlist has value None, we have created a wordlist as a list of "not", "n't". We can also use file to update whitelist.

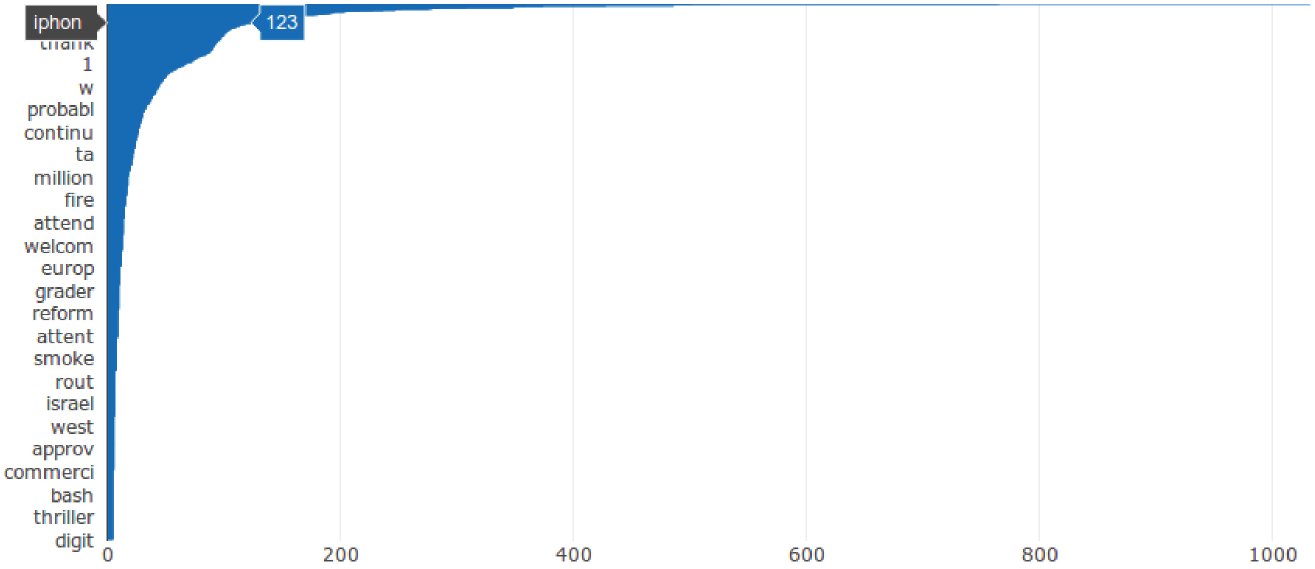
[k for k in words] will iterate the keys and create a list of keys in dictionary 'words'

[v for v in words.itervalues()] will create iterate the values and create a list of values corresponding to the keys in dictionary 'words'



```
words = pd.read_csv('./data/wordlist.csv')
words = words.sort_values(by='occurence', ascending=False)
x_words = list(words.loc[:, 'word']+['   '])
x_words.reverse()
y_occ = list(words.loc[:, 'occurence'])
y_occ.reverse()

dist = [ graph_objs.Bar(x=y_occ, y=x_words, orientation='h')]

plotly.offline.iplot({'data':dist, 'layout':graph_objs.Layout(title = 'Top words')})
```

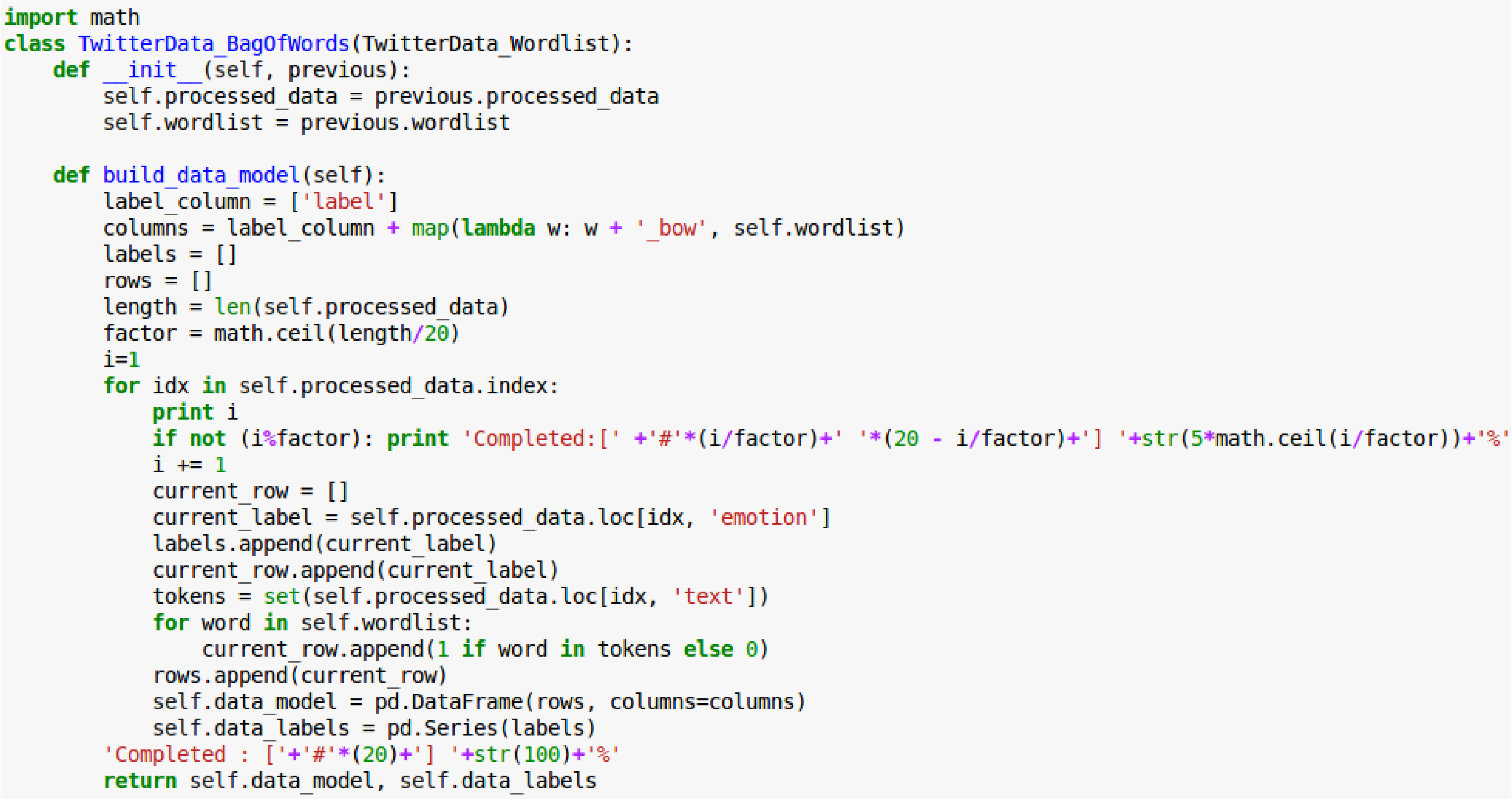




Bag-of-words

This model focuses completely on the words, or sometimes a string of words, but usually pays no attention to the "context" so-to-speak. It consists of a large list, sort of "dictionary," which are considered to be words that carry sentiment and have their own "value" when found in text. The value can presence(**'1' or count**) and absence(**'0'**) of respective words . The values are typically all added up and the result is a sentiment valuation. The equation to add can vary, but this model mainly focuses on the words, and makes no attempt to actually understand language fundamentals.

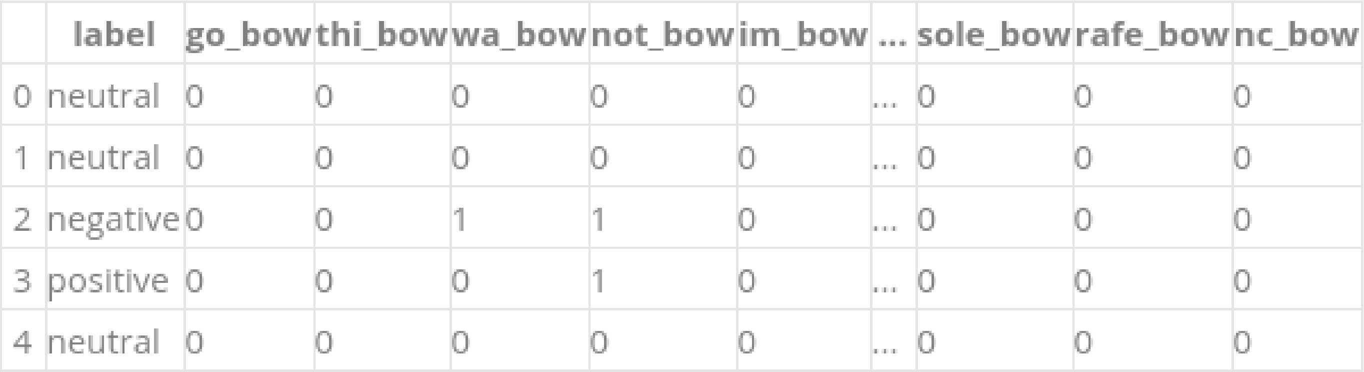
TwitterData_BagOfWords is a subclass of TwitterData_Wordlist.



```
import math
class TwitterData_BagOfWords(TwitterData_Wordlist):
    def __init__(self, previous):
        self.processed_data = previous.processed_data
        self.wordlist = previous.wordlist

    def build_data_model(self):
        label_column = ['label']
        columns = label_column + map(lambda w: w + '_bow', self.wordlist)
        labels = []
        rows = []
        length = len(self.processed_data)
        factor = math.ceil(length/20)
        i=1
        for idx in self.processed_data.index:
            print i
            if not (i%factor): print 'Completed:[' +'#'*(i/factor)+' '*(20 - i/factor)+'] '+str(5*math.ceil(i/factor))+'%'
            i += 1
            current_row = []
            current_label = self.processed_data.loc[idx, 'emotion']
            labels.append(current_label)
            current_row.append(current_label)
            tokens = set(self.processed_data.loc[idx, 'text'])
            for word in self.wordlist:
                current_row.append(1 if word in tokens else 0)
            rows.append(current_row)
            self.data_model = pd.DataFrame(rows, columns=columns)
            self.data_labels = pd.Series(labels)
        'Completed : ['+'#'*(20)+'] '+str(100)+'%'
        return self.data_model, self.data_labels
```

If the words is in tokens then the row of the dataframe will store it's value as 1, else 0.

Here, we have stored the bag-of-words in the data_models and the corresponding labels in the data_labels



	label	go_bow	thi_bow	wa_bow	not_bow	im_bow	...	sole_bow	rafe_bow	nc_bow
0	neutral	0	0	0	0	0	...	0	0	0
1	neutral	0	0	0	0	0	...	0	0	0
2	negative	0	0	1	1	0	...	0	0	0
3	positive	0	0	0	1	0	...	0	0	0
4	neutral	0	0	0	0	0	...	0	0	0

go_bow, thi_bow, etc are the columns representing the bag-of-words and label is the column for the net sentiment towards the corresponding row.

Don’t get confuse about how we get this labels, it was already present in the dataset, as we’ll use them to create a classifier.