pa1_solution

April 8, 2020

[ ]:
```
import re # preprocessing
import math # PMI calculation
import numpy as np # 2d arrays for clustering functions
import codecs # file handle
import argparse # parse command line arguments
import Clustering_functions as cl # import clustering functions

#fancy preprocessing
import nltk.data, nltk.tag
from collections import Counter
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
from nltk.tag.perceptron import PerceptronTagger


# Class for calculating cosine similarity between words
class SimTester():

    # Initialize with the needed variables
    def __init__(self):
        self.sum_all = 0 # used in PMI formula
        self.path_b = '' # path to the file B.txt
        self.path_t = '' # path to the file T.txt
        self.path_corpus = '' # path to the corpus file
        self.words = [] # list of all words from the raw text
        self.lemmatized_words = [] # lemmatized words (for fancy preprocessing)
        self.b = [] # list of B words
        self.t = [] # list of T words
        self.matrix = {} # feature matrix (raw frequencies)
        self.pmi_matrix = {} # feature matrix (with PMI weights)
        self.cos_sim_matrix = {} # 2d array cosine similarity
        self.cos_dist_matrix = {} # 2d array distance
        self.cos_sim = np.empty(shape=(1, 1)) # numpy 2d array for cosine␣
↪similarity
        self.cos_dist = np.empty(shape=(1, 1)) # numpy 2d array for distance

    # Lowercase and delete all punctuation
```

```
    def preprocess(self, line):
        new_line = re.sub(u'[^\w ]+', '', line, flags=re.UNICODE)   # remove␣
↪punctuation
        new_line = new_line.lower() # lowercase
        self.words += new_line.split() # separate words by white spaces

    # Open file with a corpus, preprocess it
    def read_corpus(self):
        file = codecs.open(self.path_corpus, 'r', 'utf-8')
        for line in file:
            self.preprocess(line)

    # Read B and T set from text files
    def create_set(self, path_set, list_set):
        file = codecs.open(path_set, 'r', 'utf-8')
        for line in file:
            list_set.append(line.strip())

    # Initialize matrix
    def init_matrix(self, matrix, arr_length, init_value):
        for target in self.t:
            matrix[target] = self.init_vector(arr_length, init_value)

    # Initialize vector
    def init_vector(self, arr_length, init_value):
        vector = []
        i = 0
        while i <= arr_length-1:
            vector.append(init_value)
            i += 1
        return vector

    # Context windows size 5
    def find_windows(self, fancy_preprocessing):
        if not fancy_preprocessing:
            word_array = self.words
        else:
            self.improve()
            word_array = self.lemmatized_words

        for target in self.t:
            for i in range(len(word_array)):
                if target == word_array[i]:
                    # First word
                    if i == 0:
                        context = [word_array[i+1], word_array[i+2]]
                    # Second word
```

```
                elif i == 1:
                    context = [word_array[i-1], word_array[i+1],␣
↪word_array[i+2]]
                # Penultimate word
                elif i == len(word_array)-2:
                    context = [word_array[i-2], word_array[i-1],␣
↪word_array[i+1]]
                # Last word
                elif i == len(word_array)-1:
                    context = [word_array[i-2], word_array[i-1]]
                # Normal case
                else:
                    context = [word_array[i-2], word_array[i-1],␣
↪word_array[i+1], word_array[i+2]]
                self.raw_freq(target, context)
        self.sum_values()

    # Raw frequencies
    def raw_freq(self, target, context):
        b_indices = []
        for word in context:
            if word in self.b:
                b_index = self.b.index(word)
                b_indices.append(b_index)

        for key in self.matrix:
            if key == target:
                for b_index in b_indices:
                    self.matrix[key][b_index] += 1

    # Sum all values in the matrix
    def sum_values(self):
        for key in self.matrix:
            for value in self.matrix[key]:
                self.sum_all += value

    # Count PMI
    def count_pmi(self):
        self.init_matrix(self.pmi_matrix, len(self.b), 0)
        for c in range(len(self.b)):

            # divisor: 2nd part
            p_j = 0
            for key in self.matrix:
                p_j += self.matrix[key][c]
            p_j = p_j / float(self.sum_all)
```

```
            for w in self.t:

                # dividend
                f_ij = self.matrix[w][c]
                p_ij = f_ij / float(self.sum_all)

                # divisor: 1st part
                p_i = 0
                for freq in self.matrix[w]:
                    p_i += freq
                p_i = p_i / float(self.sum_all)

                pmi_value = self.pmi(p_ij, f_ij, p_i, p_j)
                self.pmi_matrix[w][c] = round(pmi_value, 2)

    # PMI formula
    def pmi(self, p_ij, f_ij, p_i, p_j):
        if f_ij != 0 and p_i != 0 and p_j != 0:
            result = math.log(float(p_ij) / (p_i * p_j), 2)
            if result < 0:
                return 0
            else:
                return result
        else:
            return 0

    # Cosine similarity and distance
    def count_cos(self):
        self.init_matrix(self.cos_sim_matrix, len(self.t), 0)
        self.init_matrix(self.cos_dist_matrix, len(self.t), 0)
        for i in range(len(self.t)):
            for j in range(len(self.t)):
                vec_1 = self.pmi_matrix[self.t[i]]
                vec_2 = self.pmi_matrix[self.t[j]]
                dot_product = 0
                sqr_1 = 0
                sqr_2 = 0
                for val in range(len(vec_1)):
                    dot_product += vec_1[val]*vec_2[val]
                    sqr_1 += vec_1[val]*vec_1[val]
                    sqr_2 += vec_2[val]*vec_2[val]
                if sqr_1 != 0 and sqr_2 != 0:
                    cos = float(dot_product) / (math.sqrt(sqr_1) * math.
↪sqrt(sqr_2))
                    self.cos_sim_matrix[self.t[i]][j] = round(cos, 2)
                    if cos != 0:
```

```
                        self.cos_dist_matrix[self.t[i]][j] = round(1/
↪float(cos), 2)
                    else:
                        self.cos_dist_matrix[self.t[i]][j] = 0

    # Transform to numpy 2d arrays
    def np_array_transform(self):
        self.cos_sim = np.empty(shape=(len(self.t), len(self.t)))
        self.cos_dist = np.empty(shape=(len(self.t), len(self.t)))
        for i in range(len(self.t)):
            self.cos_sim[i] = self.cos_sim_matrix[self.t[i]]
            self.cos_dist[i] = self.cos_dist_matrix[self.t[i]]

        print('SIMILARITY MATRIX: ')
        print(self.cos_sim)

        print('\nDISTANCE MATRIX: ')
        print(self.cos_dist)

    # Improve: lemmatize, exclude determiners, conjunctions, prepositions
    def improve(self):
        wnl = WordNetLemmatizer()
        tagger = PerceptronTagger()
        tagset = None
        for word in self.words:
            pos = nltk.tag._pos_tag([word], tagset, tagger, lang='eng')
            stop_pos = ['DT', 'IN', 'CC', 'PRP']
            if pos[0][1] not in stop_pos:
                self.lemmatized_words.append(
                    wnl.lemmatize(word, self.get_pos(word)))

    # For improvement: get the top POS from the list of POS
    def get_pos(self, word):
        w_synsets = wordnet.synsets(word)
        pos_counts = Counter()
        # noun, verb, adj, adv
        pos_counts["n"] = len([item for item in w_synsets if item.pos() == "n"])
        pos_counts["v"] = len([item for item in w_synsets if item.pos() == "v"])
        pos_counts["a"] = len([item for item in w_synsets if item.pos() == "a"])
        pos_counts["r"] = len([item for item in w_synsets if item.pos() == "r"])
        most_common_pos_list = pos_counts.most_common(3)
        return most_common_pos_list[0][0]

if __name__ == "__main__":
    # Example how to run: python pa1_solution.py -b B.txt -t T.txt -f text.txt
    # Example with additional preprocessing: python pa1_solution.py -b B.txt -t␣
↪T.txt -f text.txt -p
```

```
    # Command line arguments description
    parser = argparse.ArgumentParser(description="Similarity")
    parser.add_argument("-f", "--file", dest="path_corpus",
                        help="file path to the corpus to search for words'␣
↪similarity")
    parser.add_argument("-b", dest="path_b",
                        help="file path to the set B")
    parser.add_argument("-t", dest="path_t",
                        help="file path to the set T")
    parser.add_argument("-p", "--preprocess",
                        action="store_true", dest="fancy_preprocessing",
                        default=False,
                        help="perform extra preprocessing: lemmatization")
    args = parser.parse_args()

    # Initialize
    st = SimTester()
    st.path_corpus = args.path_corpus
    st.path_b = args.path_b
    st.path_t = args.path_t

    # Read corpus
    st.read_corpus()

    # Create sets B and T from files B.txt and T.txt
    st.create_set(st.path_b, st.b)
    st.create_set(st.path_t, st.t)

    # Initialize matrix with Add-2 smoothing
    laplace = 2
    st.init_matrix(st.matrix, len(st.b), laplace)

    # Find windows and create a feature matrix (raw frequency)
    st.find_windows(args.fancy_preprocessing)

    # Uncomment to see raw frequencies
    # print('\nRAW FREQUENCIES: ')
    # print(st.b)
    # for target in st.t:
    #     print(target, st.matrix[target])

    # Calculate PMI values and create a feature matrix with weights
    st.count_pmi()

    # Uncomment to see PMI values
    # print('\nPMI VALUES: ')
```

```
# print(st.b)
# for target in st.t:
#     print(target, st.pmi_matrix[target])

# Calculate cosine similarity and distance, print them
st.count_cos()
st.np_array_transform()

# Output plots and clusters
if not args.fancy_preprocessing:
    output_1 = 'plot_1.png'
    output_2 = 'plot_2.png'
else:
    output_1 = 'plot_improved_1.png'
    output_2 = 'plot_improved_2.png'

# Run hiersrchical clustering
cl.hierarchical_clusters_print(st.cos_sim, st.t)
cl.hierarchical_clusters_draw(st.cos_sim, st.t, output_filename=output_1)

# Run k-means clustering
cl.kmeans_clusters_print(st.cos_sim, st.t)
cl.pca_plot(st.cos_sim, st.t, output_filename=output_2)
```