Lab5

May 12, 2021

1 Metrics in text

1.1 Paweł Kruczkiewicz

04.05.2021 r. Zadanie dotyczy różnych metryk w przestrzeni napisów.

1. Zaimplementuj przynajmniej 3 metryki spośród wymienionych: cosinusowa, LCS, DICE, euklidesowa.
2. Zaimplementuj przynajmniej 2 sposoby oceny jakości klasteryzacji (np. indeks Daviesa-Bouldina).
3. Stwórz stoplistę najczęściej występujących słów.
4. Wykonaj klasteryzację zawartości załączonego pliku (lines.txt) przy użyciu przynajmniej ~~2 algorytmów~~ 1 algorytmu oraz metryk zaimplementowanych w pkt. 1. i metryki Levenshteina. Każda linia to adres pocztowy firmy, różne sposoby zapisu tego samego adresu powinny się znaleźć w jednym klastrze.
5. Porównaj jakość wyników sposobami zaimplementowanymi w pkt. 2.
6. Czy masz jakiś pomysł na poprawę jakości klasteryzacji w tym zadaniu?

Sprawozdanie powinno zawierać porównanie wyników wszystkich metryk z użyciem stoplisty i bez. Można jako wzorcową klasteryzację użyć pliku clusters.txt.

[1]:
```
from sklearn.cluster import DBSCAN
from math import sqrt, inf

FILE_PATH = "lines.txt"
```

1.2 Zad 1 - Metryki

1.2.1 LCS

[2]:
```
def lcs(text_a, text_b):
    m, n = len(text_a), len(text_b)
    similarities = [[0 for _ in range(n+1)] for _ in range(m+1)]

    for i in range(1, m+1):
        for j in range(1, n+1):
            if text_a[i-1] == text_b[j-1]:
                similarities[i][j] = similarities[i-1][j-1] + 1
            else:
                similarities[i][j] = similarities[i - 1][j - 1]
```

```
    f = max([max(row) for row in similarities])
    return 1 - f/max(m, n)
```

1.2.2 DICE

`k_dice` zwraca funkcję `dice` z ustalonym paramtrem `k`. Pozwala to zautomatyzowac testy.

[3]:
```
def dice(text_a, text_b, k):
    def make_n_grams_set(text):
        return {text[i:i + k] for i in range(len(text) - k + 1)}

    set_a = make_n_grams_set(text_a)
    set_b = make_n_grams_set(text_b)
    common_digrams = len(set_a.intersection(set_b))
    len_a, len_b = len(set_a), len(set_b)

    return 1 - 2 * common_digrams / (len_a + len_b)


def k_dice(k):
    def dice_metric(text_a, text_b):
        return dice(text_a, text_b, k)
    return dice_metric
```

1.2.3 Cosinus

`k_cosine` dziala analogicznie do `k_dice`

[4]:
```
def cosine(text_a, text_b, k):
    def make_n_gram_dict(text):
        result_dict = dict()
        for i in range(len(text) - k + 1):
            n_gram = text[i: i + k]
            result_dict.setdefault(n_gram, 0)
            result_dict[n_gram] += 1
        return result_dict

    def vec_sum(dict_vec):
        return sqrt(sum(map(lambda x: x**2, dict_vec.values())))

    dict_a, dict_b = make_n_gram_dict(text_a), make_n_gram_dict(text_b)
    len_a, len_b = vec_sum(dict_a), vec_sum(dict_b)

    scalar_value = 0
    for key_a, val_a in dict_a.items():
        if key_a in dict_b:
            val_b = dict_b[key_a]
            scalar_value += val_a * val_b
```

```
    return 1 - scalar_value/(len_a*len_b)

def k_cosine(k):
    def cosine_metric(text_a, text_b):
        return cosine(text_a, text_b, k)
    return cosine_metric
```

1.2.4 Odległość edycyjna

[5]:
```
def levenshtein_distance(text_a, text_b):
    delta = lambda x, y: 0 if x == y else 1

    len_a = len(text_a)
    len_b = len(text_b)

    dist_table = [[0 for _ in range(len_b + 1)] for _ in range(len_a + 1)]

    for i in range(1, len_a + 1):
        dist_table[i][0] = i
    for j in range(1, len_b + 1):
        dist_table[0][j] = j

    for i in range(1, len_a + 1):
        for j in range(1, len_b + 1):
            x, y = text_a[i - 1], text_b[j - 1]  # current letters

            up_cost, left_cost, diag_cost = dist_table[i - 1][j] + 1,␣
 ↪dist_table[i][j - 1] + 1, \
                                            dist_table[i - 1][j - 1] + delta(x,␣
 ↪y)
            dist_table[i][j] = min(up_cost, left_cost, diag_cost)

    return dist_table[len_a][len_b]/max(len_a, len_b)
```

1.3 Zad 2 - ocena miary klasteryzacji

Funkcje pomocnicze

[6]:
```
def dist_between_elems(cluster, dist):
    return [[dist(elem_a, elem_b) for elem_a in cluster] for elem_b in cluster]


def sigma(cluster, dist):
    distances = dist_between_elems(cluster, dist)
    return sum([sum(distances[i]) for i in range(len(distances))])/
 ↪len(cluster)**2
```

```
def centroid(cluster):
    return cluster[0]


def dist_between_clusters(clust_a, clust_b, dist):
    centr_a = centroid(clust_a)
    centr_b = centroid(clust_b)
    return dist(centr_a, centr_b)


def size_of_cluster(cluster, dist):
    distances = dist_between_elems(cluster, dist)
    return max([max(distances[i]) for i in range(len(distances))])
```

Davies-Bouldin

[7]:
```
def davies_bouldin(clusters, dist):
    n = len(clusters)
    sigmas = [sigma(cluster, dist) for cluster in clusters]
    rs = [[(sigmas[i] + sigmas[j])/dist_between_clusters(clusters[i],␣
 ↪clusters[j], dist)
           if i != j else 0
          for i in range(n)]
          for j in range(n)]
    d = max(rs)
    return sum(d)/n
```

Dunn

[8]:
```
def dunn(clusters, dist):
    distances_between_clusters = [[dist_between_clusters(clusters[i],␣
 ↪clusters[j], dist)
                                   if i != j else inf
                                   for j in range(i, len(clusters))]
                                  for i in range(len(clusters))]
    min_dist = min([min(one_clust_dist) for one_clust_dist in␣
 ↪distances_between_clusters])
    clust_sizes = [size_of_cluster(cluster, dist) for cluster in clusters]
    max_size = max(clust_sizes)
    return min_dist/max_size
```

1.4 Zad 3 - stoplista

Zliczanie słów

[9]:
```
def count_words(text):
    count_dict = {}
    no_words = 0
    for line in text:
        words = line.split()
        for word in words:
            count_dict.setdefault(word, 0)
            count_dict[word] += 1
            no_words += 1
    freq_dict = {key: val/no_words for key, val in count_dict.items()}
    return freq_dict
```

Tworzenie stoplisty

[10]:
```
def make_stop_list(text, threshold):
    freq_dict = count_words(text)
    return set(map(lambda x: x[0], filter(lambda x: x[1] >sa threshold,␣
 ↪list(freq_dict.items()))))
```

Usuwanie wyrazów ze stop listy z oryginalnego tekstu

[11]:
```
def del_elems(text, stop_list):
    new_text = []
    for line in text:
        words = line.split()
        valid_words = []
        for word in words:
            if word not in stop_list:
                valid_words.append(word)
        new_text.append(" ".join(valid_words))
    return new_text
```

1.5 Zad 4 - klasteryzacja

Wczytanie pliku

[12]:
```
def open_file(path):
    with open(FILE_PATH, "r") as file:
        lines = file.readlines()
        lines = [line.strip() for line in lines]
    return lines
```

Tworzenie macierzy odległości

[13]:
```
def matrix_of_similarities(dist, text):
    return [[dist(line_x, line_y) for line_x in text] for line_y in text]
```

Klasteryzacja i stworzenie zbioru zbiorów znajdujących sie w jednym klastrze

[14]:
```
def cluster(text, measure, eps=1, min_sample=1):
    sim_matrix = matrix_of_similarities(measure, text)
    clusters = DBSCAN(eps=eps, min_samples=min_sample).fit_predict(sim_matrix)
    temp_dict = {cluster: [] for cluster in clusters}
    for i in range(len(text)):
        temp_dict[clusters[i]].append(text[i])

    return list(temp_dict.values())
```

1.6 Zad 5 - Testy

Algorytm klasteryzacji: DBSCAN (z parametrami eps=1 oraz min_sample=1)

Cosinus oraz DICE liczone są dla n-gramów długosci 5

[15]:
```
from time import time

def single_test(text, metric, stop_list_threshold=0.01):
    working_text = text if stop_list_threshold is None \
            else del_elems(text, make_stop_list(text, stop_list_threshold))

    clusters = cluster(working_text, metric, eps=1, min_sample=1)
    db_eval = davies_bouldin(clusters, metric)
    dunn_eval = dunn(clusters, metric)
    print(f'{metric.__name__:20s}\tStoplist threshold:␣
 ↪{stop_list_threshold}\nDav-boul: {db_eval:5f}\tDunn: {dunn_eval:5f}')


text = open_file(FILE_PATH)
text = text[:125] # ograniczenie liczby linii

k = 5
metrics = [lcs, k_dice(k), k_cosine(k), levenshtein_distance]
eval_metrics = [davies_bouldin, dunn]
stoplist_thresholds = [None, 0.01]
for stoplist_threshold in stoplist_thresholds:
    for metric in metrics:
        a = time()
        single_test(text, metric, stoplist_threshold)
        b = time()
        print(f'Time of test: {round(b-a, 5)} [s]\n')
```

```
lcs                  Stoplist threshold: None
Dav-boul: 0.556470      Dunn: 0.691141
Time of test: 200.25987 [s]

dice_metric          Stoplist threshold: None
Dav-boul: 0.418169      Dunn: 0.880364
Time of test: 1.59586 [s]
```

```
cosine_metric           Stoplist threshold: None
Dav-boul: 0.425149      Dunn: 0.849912
Time of test: 6.75103 [s]

levenshtein_distance    Stoplist threshold: None
Dav-boul: 0.499485      Dunn: 0.830428
Time of test: 373.85223 [s]

lcs                     Stoplist threshold: 0.01
Dav-boul: 0.630522      Dunn: 0.655169
Time of test: 150.42608 [s]

dice_metric             Stoplist threshold: 0.01
Dav-boul: 0.401262      Dunn: 0.852130
Time of test: 1.98703 [s]

cosine_metric           Stoplist threshold: 0.01
Dav-boul: 0.408359      Dunn: 0.803537
Time of test: 7.54199 [s]

levenshtein_distance    Stoplist threshold: 0.01
Dav-boul: 0.389439      Dunn: 0.913655
Time of test: 290.46506 [s]
```

1.7 Zad 6

Mam kilka pomysłów, dzięki którym być może można poprawic jakość klasteryzacji:

1. **Lepsze dopasowanie parametrów** - w przestawionych powyżej testach uzyto jednego parametru do dopasowania klastrów. Można jednak użyć algorytmów ML do znalezienia najlepszego dopasowania, np. epsilona czy progu częstotliwości w stopliście.
2. **Unifikacja zbioru wejściowego** - linie analizowanego tekstu posiadają wiele niespójności - ten sam adres czasem pisany jest z wielkiej litery, czasem w całości małymi albo wielkiemi symbolami. Oddala to znacząco teksty w przedstawionych wyżej metrykach (np. LCS). Zmniejszenie wszystkich liter, usunięcie podwójnych spacji, być może również znaków interpunkcji - mogłoby pozytywnie wpłynąć na jakość klasyfikacji. Minusem jest oczywiscie lekka zmiana danych wejściowych, jednak moim zdaniem nie jest ona drastyczna i nie zmniejsza znacząco przejrzystości informacji zawartych w danych.
3. **Postaranie się o lepsze dane** - wykorzystany wyżej DBSCAN wykorzystuje parametr `min_sample`, czyli najmniejsza liczba elementów w pojedynczym klastrze. W sytuacji, w której w naszym zbiorze wejściowym znajdowałyby się jedynie takie linie, co do których mamy pewność, że znajdują się im odpowiadające `k` linii, algorytm mógłby (przy równoczesnym ustaleniu większego epsilonu) uniknąć false_negative'ów.

[ ]: