Raport z laboratorium 5

Filip Nikolow

14 maja 2021

1 Cel laboratorium

Celem laboratorium była klasteryzacja tekstu za pomocą różnych metryk.

2 Realizacja poszczególnych poleceń

Kod do każdego z poleceń załączam na końcu sprawozdania.

2.1 Polecenie 1

Zaimplementowałem metryki: cosinusową, DICE, LCS oraz levenshteina. Metryki ngramowe domyślnie korzystają z ngramów długości 3 i taka też długość jest używana w klasteryzacji.

2.2 Polecenie 2

Zaimplementowałem dwa algorytmy oceny klasteryzacji: indeks Daviesa-Bouldina oraz indeks Dunna. Oba indeksy obliczają centroidy jako element klastra mający minimalną sumę odległości od pozostałych elementów klastra. Indeks DB potrzebuje centroidów bezpośrednio we wzorze, natomiast w indeksie Dunna używam centroidów jako reprezentantów klastrów do liczenia odległości międzyklastrowej.

2.3 Polecenie 3

Stoplistę utworzyłem najpierw zastępując w tekście znaki ['.', ',', ':', '/', ';', ""] spacjami, a następnie obliczyłem 50 najczęściej występujących słów (słowa rozdzielając białymi znakami).

```
['LTD', 'CO', 'TEL', 'ROAD', 'O', 'CHINA', 'LOGISTICS', 'POLAND', 'OF', 'UL', '58', 'LIMITED',
 'INTERNATIONAL', 'FAX', 'RUSSIA', 'AND', 'SP', 'GDYNIA', 'Z', 'OY', '+48', 'FINLAND', 'SHANGHAI',
 'OOO', 'AS', 'INDUSTRIAL', 'LLC', 'ORDER', 'OFFICE', 'ROOM', 'FORWARDING', 'GLOBAL', 'TO', '-',
  'NINGBO', 'BRANCH', '22', 'AGENT', '+7', 'SHENZHEN', 'COMPANY', 'KONG', 'SHIPPING']
```

2.4 Polecenia 4,5

Dokonałem klasteryzacji z użyciem każdej z 4 metryk algorytmem DBSCAN z sklearn.cluster. Algorytm uruchamiałem z eps=0.42, min_samples=2, metric='precomputed' a resztą parametrów domyślną. Do liczenia klasteryzacji z metrykami LCS oraz levenshteina użyłem bibliotecznych funkcji, aby ten kod miał szanse się policzyć. Dodatkowo też macierze odległości wyliczam wielowątkowo. Poniżej załączam statystyki klasteryzacji oraz wyniki oceny klasteryzacji. Ocena klasteryzacji została dokonana z pominięciem linii niezaklasyfikowanych do żadnej z klas (klasa -1 zwracana przez DBSCAN).
Preprocessing korzystający ze stoplisty polegał na zastąpieniu znaków z podpkt. 2.3 spacjami, następnie zastąpieniu elementów stoplisty spacjami, a następnie usunięciu wielokrotnych spacji.

```
Without stoplist
####################################

DICE metric
Outliers: 2619
Clusters: 1043
Davies-Bouldin index: 0.9140761450225531
Dunn index: 0.6970776881331544

Cosine metric
Outliers: 2309
Clusters: 1012
Davies-Bouldin index: 1.1688145054931731
Dunn index: 0.5044048744544906

Levenshtein metric
Outliers: 2605
Clusters: 1012
Davies-Bouldin index: 1.1544761769279235
Dunn index: 0.6009763876958781

LCS metric
Outliers: 5115
Clusters: 623
Davies-Bouldin index: 0.9172216859878096
Dunn index: 0.770157526611526


With stoplist
####################################

DICE metric
Outliers: 2550
Clusters: 1074
Davies-Bouldin index: 0.8617682589812278
Dunn index: 0.7146632102244141

Cosine metric
Outliers: 2295
Clusters: 1054
Davies-Bouldin index: 1.0171914682721106
Dunn index: 0.616151350523938

Levenshtein metric
Outliers: 2661
Clusters: 1032
Davies-Bouldin index: 1.0480100764096638
Dunn index: 0.6907797609418034

LCS metric
Outliers: 4119
Clusters: 872
```

```
Davies-Bouldin index: 0.810183747206242
Dunn index: 0.7803394943731271
```

Wyniki te są dość mylące, szczególnie gdy popatrzymy na ilość outlierów, czyli elementów niezaklasyfikowanych do żadnego klastra - np. metryka LCS daje ich ponad 5 tysięcy a jej indeksy wychodzą lepsze od na przykład metryki cosinusowej (indeks DB - mniejszy = lepszy, indeks Dunna - większy = lepszy). Nie jest to szczególnie zaskakujące, gdyż te indeksy nie uwzględniają outlierów w żaden sposób - w efekcie metody odrzucające prawie wszystkie elementy mogą mieć bardzo dobre wyniki. Generalnie na podstawie indeksów należałoby stwierdzić, że najlepszą klasteryzację daje LCS, z algorytmem DICE na drugim miejscu. Tymczasem po wizualnej inspekcji to metryka cosinusowa wydaje się działać najlepiej - ma najmniej outlierów i nie widać zbyt dużo wymieszanych klas.

Użycie stoplisty całkiem znacznie zmniejszyło ilość outlierów w klasteryzacji metryką LCS i trochę mniej spektakularnie w pozostałych przypadkach. - natomiast efekt końcowy klasteryzacji generalnie wydaje się być lepszy i tak też stwierdzają indeksy. Wszystkie klasteryzacje załączam w zipie razem ze sprawozdaniem (na początku plików najczęściej znajduje się klasa -1, czyli elementy nieprzydzielone do żadnego klastra - w szczególności można każdą linię tej klasy traktować jako jednoelementowy klaster, jeśli jest taka potrzeba).

2.5 Polecenie 6

Klasteryzacje możnaby polepszyć dostosowując hiperparametry algorytmu klasteryzującego jak epsilon, czy długość ngramów w metrykach cosinusowej i DICE - w szczególności mniejszy epsilon mogą skutkować lepszymi wartościami indeksów, gdyż algorytm odrzuci jako szum więcej elementów, ale klasy które zostaną, będą lepiej dopasowane (jak już wspomniałem, indeksy nie uwzględniają ilości niedopasowanych elementów). Oprócz tego można próbować użyć innej metryki, np. cosinusowej ze słowami zamiast ngramów - oczywiście słowa należałoby dostosować, czyli na przykład dokonać stemmingu. Dodatkowo pomocnym mogłoby być użycie innych metod oceny jakości klasteryzacji, jak metoda silhouette.

3 Kod

```python
from collections import defaultdict
from math import sqrt
from numba import njit
from Levenshtein import distance as levenshteinC
from pylcs import lcs2


@njit()
def lcs_metric(t1, t2):
    n = len(t1)
    m = len(t2)
    lcs = [[0 for i in range(m + 1)] for j in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if t1[i - 1] == t2[j - 1]:
                lcs[i][j] = lcs[i - 1][j - 1] + 1
    return 1 - (max([max(i) for i in lcs]) / max(n, m))

```

```
19
20  # Using a C library so the code will finish in a reasonable time
21  def lcs_metric2(t1, t2):
22      return 1 - (lcs2(t1, t2) / max(len(t1), len(t2)))
23
24
25  #@njit()
26  def dice_metric(t1, t2, token_size=3):
27      n = len(t1)
28      m = len(t2)
29      token_size = min([n, m, token_size])
30      s1 = set([t1[i - token_size:i] for i in range(token_size, n + 1)])
31      s2 = set([t2[i - token_size:i] for i in range(token_size, m + 1)])
32      product = s1.intersection(s2)
33      return 1 - (2 * len(product)) / (len(s1) + len(s2))
34
35
36  #@lru_cache(10000)
37  def ngram_stat(token, token_size):
38      d = defaultdict(int)
39      for i in range(token_size, len(token) + 1):
40          d[token[i - token_size:i]] += 1
41      return d
42
43
44  def scalar_on_dicts(d1, d2):
45      res = 0
46      for k, v in d1.items():
47          res += v * d2[k]
48      return res
49
50
51  #@njit()
52  def cosine_metric(t1, t2, token_size=3):
53      if min(len(t1), len(t2)) < token_size:
54          return 1
55      d1 = ngram_stat(t1, token_size)
56      d2 = ngram_stat(t2, token_size)
57      return 1 - (scalar_on_dicts(d1, d2) /
58                  (sqrt(scalar_on_dicts(d1, d1)) * sqrt(scalar_on_dicts(d2, d2))))
59
60
61  @njit()
62  def levenshtein(s1, s2):
63      n = len(s1)
64      m = len(s2)
65      distance = [[0 for i in range(m + 1)] for j in range(n + 1)]
66      for i in range(n + 1):
67          distance[i][0] = i
68      for i in range(m + 1):
69          distance[0][i] = i
70      for i in range(1, n + 1):
71          for j in range(1, m + 1):
```

```
            if s1[i - 1] == s2[j - 1]:
                distance[i][j] = distance[i - 1][j - 1]
            else:
                distance[i][j] = distance[i - 1][j]
                if distance[i][j] > distance[i][j - 1]:
                    distance[i][j] = distance[i][j - 1]
                if distance[i][j] > distance[i - 1][j - 1]:
                    distance[i][j] = distance[i - 1][j - 1]
                distance[i][j] += 1
    return distance[n][m] / max(n, m)


# Using a C library so the code will finish in a reasonable time
def levenshteinC_wrapper(s1, s2):
    return levenshteinC(s1, s2) / max(len(s1), len(s2))


if __name__ == '__main__':
    t1 = "abcdef"
    t2 = "bcdf"
    print(lcs_metric(t1, t2))
    print(dice_metric(t1, t2))
    print(cosine_metric(t1, t2))
```

```
from metrics import cosine_metric, dice_metric, levenshteinC_wrapper, lcs_metric2
from sklearn.cluster import DBSCAN
import numpy as np
from multiprocessing import Pool
from time import time, sleep
from collections import defaultdict
from collections import Counter

MAX_THREADS = 24


def calc_single_distance(a):
    t1, t2, metric = a
    # abs because numerical errors may go slighlty below 0
    return abs(metric(t1, t2))


def distances_parallel(tokens, metric):
    n = len(tokens)
    res = [(tokens[i], tokens[j], metric) for j in range(n) for i in range(n)]
    with Pool(MAX_THREADS) as p:
        res = p.map(calc_single_distance, res)
    res = np.array(res, dtype=np.float64)
    return res.reshape((n, n))


def distances(tokens, metric):
    n = len(tokens)
```

```
    res = np.ndarray((n, n), dtype=np.float64)
    for i in range(n):
        for j in range(i, n):
            res[i][j] = res[j][i] = abs(metric(tokens[i], tokens[j]))
    return res


def clusterize(lines, metric, original=None, timeit=False):
    t1 = time()
    X = distances_parallel(lines, metric)
    if timeit: print("Calculating distance matrix time:", time() - t1)
    t1 = time()
    clustering = DBSCAN(eps=0.42, min_samples=2, metric='precomputed').fit(X)
    if timeit: print("Calculating clusterization time:", time() - t1)
    d = defaultdict(list)
    for line, label in zip(lines, clustering.labels_):
        d[label].append(line)
    print("Outliers:", len(d[-1]))
    print("Clusters:", len(d))
    t1 = time()
    q = davies_bouldin_index(d, metric)
    if timeit: print("Calculating Davies-Bouldin index time:", time() - t1)
    print("Davies-Bouldin index:", q)
    t1 = time()
    q = dunn_index(d, metric)
    if timeit: print("Calculating Dunn index time:", time() - t1)
    print("Dunn index:", q)
    if original:
        d = defaultdict(list)
        for line, label in zip(original, clustering.labels_):
            d[label].append(line)
    return d


def print_classes(d, f=None):
    s = []
    for k, cluster in d.items():
        s.append(
            f"\n###################################\nclass {k}\n###################################\n"
        )
        for line in cluster:
            s.append(line)
            s.append('\n')
    s = ''.join(s)
    if f:
        with open(f, "w") as fi:
            fi.write(s)
    else:
        print(s)


# Centroid is chosen as a cluster element that
# has the lowest distance to all other cluster elements
```

```
82  def get_centroid_and_avg(cluster, metric):
83      n = len(cluster)
84      distances = []
85      avg = 0
86      with Pool(MAX_THREADS) as p:
87          for i in range(n):
88              dist = p.map(calc_single_distance,
89                           [(cluster[i], cluster[j], metric) for j in range(n)])
90              s = np.sum(dist)
91              distances.append(s)
92              avg += s
93      centroid = cluster[np.argmin(distances)]
94      avg /= n * (n - 1)
95      return (centroid, avg)
96
97
98  def davies_bouldin_index(d, metric):
99      dcp = d.copy()
100     dcp.pop(-1, None)
101     n = len(dcp)
102     res = 0
103     ca = [get_centroid_and_avg(v, metric) for v in dcp.values()]
104     for i in range(n):
105         lis = [0]
106         for j in range(n):
107             if i != j:
108                 (c1, a1) = ca[i]
109                 (c2, a2) = ca[j]
110                 lis.append((a1 + a2) / metric(c1, c2))
111         res += np.max(lis)
112     return res / len(dcp)
113
114
115 def dunn_index(d, metric):
116     dcp = d.copy()
117     dcp.pop(-1, None)
118     tmp = [get_centroid_and_avg(cluster, metric) for cluster in dcp.values()]
119     centroids = [x[0] for x in tmp]
120     avgs = [x[1] for x in tmp]
121     dist = distances_parallel(centroids, metric)
122     for i in range(dist.shape[0]):
123         dist[i][i] = float("inf")
124     dist = np.min(dist)
125     size = np.max(avgs)
126     return dist / size
127
128
129 def remove_chars(lines, to_replace=None):
130     if not to_replace:
131         to_replace = ['.', ',', ':', '/', ';', '"']
132     else:
133         to_replace = [' ' + rep + ' ' for rep in to_replace]
134     res = []
```

```
135     for line in lines:
136         res.append(line)
137         for a in to_replace:
138             res[-1] = res[-1].replace(a, " ")
139     return res
140 
141 
142 def stoplist(lines, n=50):
143     return [
144         b[0] for b in Counter([t for u in [line.split() for line in lines]
145                                for t in u]).most_common(n)
146     ]
147 
148 
149 def remove_stoplist(lines, n=50):
150     # Removing dots, commas etc.
151     lines = remove_chars(lines)
152     # Removing 50 most common words
153     res = remove_chars(lines, stoplist(lines))
154     # Removing multiple spaces
155     for i in range(10):
156         res = remove_chars(res, to_replace=[""])
157     return res
158 
159 
160 def test_all():
161     with open('../sources/lines.txt') as f:
162         lines = f.read().splitlines()
163     print("Without stoplist\n####################################")
164     print("\nDICE metric")
165     print_classes(clusterize(lines, dice_metric), "clusters_dice.txt")
166     print("\nCosine metric")
167     print_classes(clusterize(lines, cosine_metric), "clusters_cosine.txt")
168     print("\nLevenshtein metric")
169     print_classes(clusterize(lines, levenshteinC_wrapper), "clusters_levenshtein.txt")
170     print("\nLCS metric")
171     print_classes(clusterize(lines, lcs_metric2), "clusters_lcs.txt")
172     sleep(5)
173     print("\n\nWith stoplist\n####################################")
174     print("\nDICE metric")
175     print_classes(clusterize(remove_stoplist(lines), dice_metric, lines),
176                   "clusters_dice_stoplist.txt")
177     print("\nCosine metric")
178     print_classes(clusterize(remove_stoplist(lines), cosine_metric, lines),
179                   "clusters_cosine_stoplist.txt")
180     print("\nLevenshtein metric")
181     print_classes(clusterize(remove_stoplist(lines), levenshteinC_wrapper, lines),
182                   "clusters_levenshtein_stoplist.txt")
183     print("\nLCS metric")
184     print_classes(clusterize(remove_stoplist(lines), lcs_metric2, lines),
185                   "clusters_lcs_stoplist.txt")
186 
187 
```

```
188 def print_lines(lines):
189     for line in lines:
190         print(line)
191 
192 
193 if __name__ == '__main__':
194     with open('../sources/lines.txt') as f:
195         lines = f.read().splitlines()
196     # print(stoplist(remove_chars(stoplist(lines))))
197     # print_lines(stoplist(remove_chars(lines)))
198     # print_lines(remove_stoplist(lines))
199     test_all()
```