Lista 4 Parte 2

Análise de dados de microarray,
K-Means para clusterização,
e algoritmo genético para seleção de features

Pedro Foletto Pimenta - 00229778
INF05018 - Biologia Computacional - Turma: U (2019/2)
Prof. Márcio Dorn

K-Means (implementado na parte 1)

- Método de clustering
  - Particiona um conjunto de vetores em K grupos/classes
- **Entradas:**
  - Vetores a serem classificados
  - K (número de classes)
- **Saídas:**
  - Posição final dos K centróides
  - Classe atribuída a cada um dos vetores de entrada

Algoritmo Genético

- Método de otimização inspirado na seleção natural
- Pseudocódigo:
    a. Cria população inicial de soluções
    b. Enquanto não atingir o número máximo de gerações (ou outra condição de parada):
        i. Avaliar população
        ii. Criar nova geração da população com base nos scores
    c. Retorna melhor solução da população da última geração

Geração da população

- **População:**

  array de
  (lista de 50 soluções)

```
[  31   91  287  312  365  376  604  665  710  737  789  810 1040 1056 1141
 1269 1318 1362 1444 1525 1583 1615 1703 1833 1839 1860 1876 1904 1917 1978
 2051 2173 2175 2178 2213 2361 2471 2518 2543 2583 2670 2705 2754 2835 2859
 2878 3055 3137 3141 3167 3177 3209 3258 3300 3320 3336 3476 3586 3800 3924
 4090 4097 4118 4147 4151 4203 4229 4307 4308 4334 4372 4375 4460 4493 4649
 4840 4893 4895 5117 5322 5329 5355 5359 5789 5817 5877 5891 5898 6027 6036
 6061 6144 6152 6568 6633 6791 6795 6818 6848 7067]
```

Exemplo de uma solução

- **População inicial:**

  seleção aleatória de índices de 0 a 7128

- **Geração da próxima geração:**

  a. Os primeiros 10% são copiados dos 10% melhores da geração passada
  b. Os próximos 30% são cruzas de boas soluções da população passada + chance de mutação
  c. Os próximos 30% são cruzas de uma boa solução da população passada e uma solução aleatória da população passada + chance de mutação
  d. Os próximos 30% são cruzas de duas soluções aleatórias da população passada + 100% de chance de mutação

Avaliação de uma população

- Para cada solução dentro de uma população
  a. Se filtram dos dados brutos de microarray somente as feature definidas pela solução
  b. É executado o K-Means com K=2 nesses dados filtrados
  c. Compara-se os dois clusters com os rótulos originais (AML x ALL)
  d. Quanto melhor separados estão os elementos de classe AML e ALL, maior o score
  e. O score de uma solução é a média de 30 avaliações (para compensar pela variação do resultado de uma avaliação)

Implementação

- **Python 3**
- Bibliotecas utilizadas:
  - **random** (para gerar os centróides iniciais)
  - **pandas** (para carregar os dados do arquivo csv)
  - **numpy** (para manipular os vetores e matrizes)
  - **time** (para medir o tempo de execução)

Implementação: parâmetros

```
# parametros fixos do alg genetico
NEW_NUM_DIM = 75 # num de genes (features) escolhidos do total
# 5 -> 0.95
# 10 -> 0.97
# 50 -> 1.0
# 150 -> 1.0
# 500 -> 0.97
# 1500 -> 0.95
# 3500 -> 0.93

POPULATION_SIZE = 50 # tem q ser 50
NUM_GENERATIONS = 100 # tem q ser 100

# parametros 'variaveis' do alg genetico
PROB_MUTACAO = 0.2 # chance de ocorrer uma mutacao em um novo individuo
NUM_MUTACOES = 10 # numero de genes mutados no evento de uma mutacao
NUM_SCORE_MEAN = 50 # numero de vezes a validar uma solucao pra fazer o score
```

Implementação

```
# retorna um conjunto reduzido de genes
# que separe o melhor possivel o conjunto de amostras
# de acordo com os rotulos (ALL e AML)
def alg_genetico(points, labels):…

# gera nova populacao de solucoes com base na performance da ultima
def evaluate_population(population, points, labels):…

# gera nova populacao de solucoes com base na performance da ultima
def generate_new_population(population, scores, num_dim):…

# retorna a solucao sem repeticoes
def ajusta_solucao(solucao, num_dim):…

# retorna uma nova solucao gerada pela combinacao
# das solucoes a e b
def crossover(solucao_a, solucao_b):…

# aplica uma mutacao em uma solucao
def mutacao(solucao, num_dim):…

# ordena as solucoes de uma populacao com base nos seus scores
def sort_population(population, scores):…

# retorna o score de um agrupamento de pontos/vetores
# OBS: so funciona com clustering em dois grupos (k=2 : ALL e AML)
def get_clustering_score(classes, labels):…
```

Funções relativas ao algoritmo genético

```
# retorna True com probabilidade oneProb
# e False com probabilidade (1 - oneProb)
def choseWithProb( oneProb ):…

# retorna um numpy array com vetores normalizados de 0 a 1
def normalize_points(points):…

# retorna um numpy array contendo a posicao dos k centroides
def update_centroids(points, classes, k):…

# retorna um numpy array onde cada posiçao i contem a classe atribuida ao ponto i
def classify_points(points, centroids, num_points, k):…

# retorna as classes de cada ponto e os centroides de cada classe
# (dois numpy arrays)
def k_means(k, points):…
```

Funções relativas ao K-Means (parte 1)

Implementação

```
# retorna um conjunto reduzido de genes
# que separe o melhor possivel o conjunto de amostras
# de acordo com os rotulos (ALL e AML)
def alg_genetico(points, labels):

    num_points, num_dim = np.shape(points)

    # populacao aleatoria inicial
    population = np.array([random.sample(range(num_dim), NEW_NUM_DIM) for i in range(POPULATION_SIZE)])

    for generation in range(NUM_GENERATIONS):
        # avalia populacao
        scores = evaluate_population(population, points, labels)
        print("Generation "+str(generation)+" scores: "+str(np.shape(scores))+"  mean score: % .4f  max
        # gera nova populacao
        population = generate_new_population(population, scores, num_dim)

    # sort population to get the best solution
    population, scores = sort_population(population, scores)
    best_solution = population[0].astype(int)
    classes, _ = k_means(2, points[:, best_solution])
    best_solution_score = scores[0]

    return best_solution, best_solution_score
```

Implementação

```python
# retorna uma nova solucao gerada pela combinacao
# das solucoes a e b
def crossover(solucao_a, solucao_b):
    assert(len(solucao_a) == len(solucao_b))
    size = len(solucao_a)
    resultado = np.empty((size)).astype(int)

    #separacao = int(size/2) # crossover simples
    separacao = random.randint(1, size-2) # crossover 2.0 ihaaaa

    resultado[:separacao] = solucao_a[:separacao]
    resultado[separacao:] = solucao_a[separacao:]

    return resultado
```

```python
# aplica uma mutacao em uma solucao
def mutacao(solucao, num_dim):
    # muda NUM_MUTACOES valores aleatoriamente
    for i in range(NUM_MUTACOES):
        random_index = random.randint(0, len(solucao)-1)
        solucao[random_index] = random.randint(0, num_dim-1)
    return solucao
```

Implementação: "main"

```
## main

# carregar dados do arquivo csv
print("carregando dados...")
df = pandas.read_csv('leukemia_big.csv', header=None)

# get data from dataframe
labels = np.array(df.iloc[0].values) # get labels (first row)
df = df.drop(0) # remove labels from dataframe
df = df.T # transpose data
points = (df.values).astype(np.float) # convert strings to floats and put it in a numpy array
points = normalize_points(points) # normalize to [0,1] interval

k=2

print("rodando algoritmo genetico...")
startTime = time.time() # medir o tempo de execucao a partir daqui

# get melhor combinaçao de 3572 genes
best_solution, best_solution_score = alg_genetico(points, labels)
print("\n\n...Melhor selecao de genes encontrada:\n" + str(best_solution))
print("...score: " + str(best_solution_score))
endTime = time.time()
totalTime = endTime - startTime
print("...tempo de execucao: %.3f segundos"%(totalTime))
```

Resultado

- Variando o tamanho do vetor solução obtemos diferentes scores finais:
  - tamanho 5: score de 0.95
  - tamanho 10: score de 0.97
  - **tamanho 15: score de 1.0**
  - **tamanho 50: score de 1.0**
  - **tamanho 150: score de 1.0**
  - tamanho 500: score de 0.97
  - tamanho 1500: score de 0.95
  - tamanho 3572: score de 0.93

Resultado (numero de features = 75)

```
pfpimenta@s-72-202-01:~/biocomp2019/lista4parte2$ python e4-2.py
carregando dados...
rodando algoritmo genetico...
Generation 0 scores: (50,)  mean score:  0.6756  max score:  0.8750  min score:  0.5694
Generation 1 scores: (50,)  mean score:  0.6478  max score:  0.9722  min score:  0.5000
Generation 2 scores: (50,)  mean score:  0.6858  max score:  0.9583  min score:  0.5000
Generation 3 scores: (50,)  mean score:  0.6994  max score:  0.9028  min score:  0.5278
Generation 4 scores: (50,)  mean score:  0.7464  max score:  0.9306  min score:  0.5417
Generation 5 scores: (50,)  mean score:  0.7408  max score:  0.9167  min score:  0.5278
Generation 6 scores: (50,)  mean score:  0.7536  max score:  0.9167  min score:  0.5000
Generation 7 scores: (50,)  mean score:  0.8039  max score:  0.9444  min score:  0.5556
Generation 8 scores: (50,)  mean score:  0.8272  max score:  0.9583  min score:  0.6528
```

⋮

Resultado (numero de features = 75)

⋮

```
Generation 89 scores: (50,)  mean score:  0.9708  max score:  1.0000  min score:  0.5556
Generation 90 scores: (50,)  mean score:  0.9422  max score:  1.0000  min score:  0.5556
Generation 91 scores: (50,)  mean score:  0.9425  max score:  1.0000  min score:  0.5417
Generation 92 scores: (50,)  mean score:  0.9564  max score:  1.0000  min score:  0.5139
Generation 93 scores: (50,)  mean score:  0.9722  max score:  1.0000  min score:  0.6944
Generation 94 scores: (50,)  mean score:  0.9608  max score:  1.0000  min score:  0.6111
Generation 95 scores: (50,)  mean score:  0.9622  max score:  1.0000  min score:  0.5139
Generation 96 scores: (50,)  mean score:  0.9250  max score:  1.0000  min score:  0.5000
Generation 97 scores: (50,)  mean score:  0.9675  max score:  1.0000  min score:  0.5278
Generation 98 scores: (50,)  mean score:  0.9461  max score:  1.0000  min score:  0.5139
Generation 99 scores: (50,)  mean score:  0.9314  max score:  1.0000  min score:  0.5139


...Melhor selecao de genes encontrada:
[ 167  330  531  608  757  828  839 1104 1106 1553 1672 1769 1798 1833 1878
 1906 2094 2127 2212 2297 2486 2555 2621 2695 2710 2816 2874 2919 3008 3063
 3091 3153 3177 3201 3287 3486 3548 3550 3587 3617 3648 3714 3733 4211 4212
 4266 4290 4524 4602 4642 4886 4893 4943 4990 5021 5044 5313 5468 5603 5847
 5898 6049 6164 6214 6334 6592 6608 6698 6729 6750 6861 6943 7081 7088 7091]
...score: 1.0
...tempo de execucao: 81.420 segundos
```

Resultado (conclusões)

- Isolando um subconjunto features (de ~15 a ~150) se consegue separar as amostras de modo a amplificar as diferenças entre amostras AML e ALL

- A solução é prejudicada com features desnecessárias

Lista 4 Parte 2

Análise de dados de microarray,
K-Means para clusterização,
e algoritmo genético para seleção de features

Pedro Foletto Pimenta - 00229778
INF05018 - Biologia Computacional - Turma: U (2019/2)
Prof. Márcio Dorn