Biologia Computacional 2019/2 - Lista de Exercícios 4 - Parte II

Aluno: Leonardo Oliveira Wellausen

**1) Código:**

```
lines = open('leukemia_big.csv', 'r').read().splitlines()
n_samples = len(lines[0].split(','))

labels = lines.pop(0).split(',')

samples = []

for i in range(n_samples):
    this_line = [labels[i]]
    for line in lines:
        l = line.split(',')
        this_line.append(float(l[i]))
    samples.append(this_line)

n_generations = 100
n_individuals = 50
alpha = 0.002
size_elite = 0.2 * n_individuals
size_middle = 0.3 * n_individuals + size_elite
size_rest = 0.5 * n_individuals + size_middle
n_genes = len(samples[0]) - 1
target_genes = n_genes // 2
```

Início do código. Parseamos o documento csv e inicializamos as variáveis de controle dos algoritmos.

```
gen = [[np.inf] + [random.choice([0, 1]) for i in range(n_genes)] for ind in range(n_individuals)]
errlist = []
genlist = []
sumlist = []
for g in range(n_generations):
    indi = 0
    print('Generation: ', g)
    pool = ThreadPool(4)
    results = pool.starmap(n2means, zip(itertools.repeat(samples), gen))

    for ind in range(n_individuals):
        count = gen[ind][1:].count(1)
        gen[ind][0] = results[ind] + count * alpha
    gen = sorted(gen, key=lambda t: t[0])
    next_gen = []
    for i in range(n_individuals):
        if i <= size_elite:
            next_gen.append(crossover_elite(gen, i))
        elif i <= size_middle:
            next_gen.append(crossover_middle(gen, size_middle))
        else:
            next_gen.append(crossover_rest(gen))
    count = gen[0][1:].count(1)
    print('Melhor indivíduo da geração' , g, '. Erros:', gen[0][0] - count*alpha, ' Com tantos genes: ', count)
    sumlist.append(gen[0][0])
    errlist.append(gen[0][0] - count*alpha)
    genlist.append(count)
    gen = deepcopy(next_gen)
```

Trecho principal do código. Inicialmente geramos uma geração de indivíduos aleatórios. Após, para o número de gerações, rodamos um k-means para cada indivíduo (utilizando multithreading); inserimos o fitness retornado para cada indivíduo como o primeiro elemento na lista; ordenamos a geração pelo melhor fitness; fazemos crossovers para gerar a próxima geração; salvamos e exibimos melhor resultado obtido para a atual geração; avançamos a geração.

O que é um indivíduo? Cada indivíduo é uma máscara binária do mesmo tamanho das samples originais. Casa posição na máscara representa um gene original, se o valor em uma posição é 1 nós utilizaremos este gene, se for 0 ele não será utilizado. Assim evitamos problemas de repetir genes no crossover, e também podemos reduzir o número de genes selecionados.

O que é o erro? O erro é o número de samples classificadas de forma errada somado com o número de genes selecionados na amostra que gera esse erro (este número é ponderado por um valor alpha).

```python
def crossover_elite(gen, i):
    return gen[i]


def crossover_middle(gen, middle):
    i_parent1, i_parent2 = random.choices(range(int(middle) + 1), k=2)
    parent1 = gen[i_parent1]
    parent2 = gen[i_parent2]

    newborn = [np.random.choice([parent1[i], parent2[i]], p=[0.5, 0.5]) for i in range(len(parent1))]
    return newborn


def crossover_rest(gen):
    return [np.inf] + [random.choice([0, 0, 0, 0, 0, 0, 0, 0, 0, 1]) for i in range(len(gen[0]) - 1)]
```

Funções de crossover: os indivíduos **elite** serão repassados para a geração seguinte; os indivíduos **classe média** serão eliminados mas deixarão herdeiros gerados por crossover entre indivíduos aleatórios de **classe média** e **elite**; o **resto** é completamente aniquilado e substituído por novos indivíduos aleatórios com bias para 0 (pegar menos genes!).

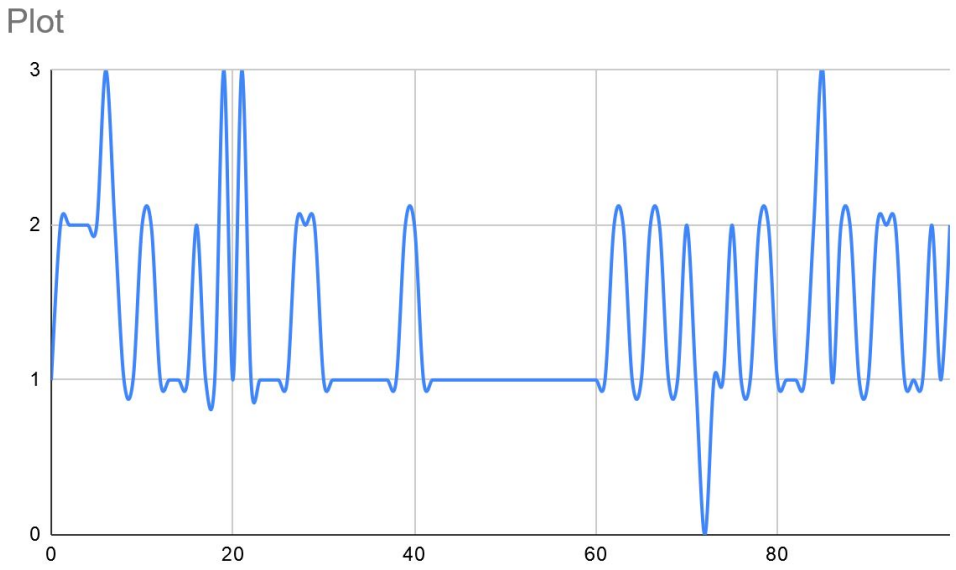
```python
def n2means(samples, mask):
    global indi
    n = 5
    print('Idividual: ', indi)
    indi += 1
    new_samples = []
    for sample in samples:
        new_sample = []
        for j in range(1, len(samples[0])):
            if mask[j] == 1:
                new_sample.append(sample[j])
        new_samples.append([sample[0], np.array(new_sample)])
```

A função de k-mean é a mesma da lista anterior, com poucas mudanças. Agora a função nkmeans será uma thread paralela dentro do programa; antes de chamarmos o k-means de fato com as samples recebidas, temos que filtrá-los com a máscara (cada indivíduo genética é uma máscara que será aplicada para todos os samples), assim os samples passados ao k-means terão número de genes reduzidos pela máscara. Outra alteração é o maior uso de arrays **numpy**, pois a versão anterior com listas python não tinha preocupação alguma com tempo de execução, porém ao executar o k-means 50 vezes para 100 gerações a falta de otimização de tornou gritante.

```
    best_score = np.inf
    best_model = None
    for i in range(n):
        groups = kmeans(new_samples, 2)

        e0, e1 = eval_group(groups)

        erro = 0

        if e0['ALL'] >= e0['AML']:
            erro += e0['AML']
            erro += e1['ALL']
        else:
            erro += e1['AML']
            erro += e0['ALL']

        if erro <= best_score:
            best_score = erro
            best_model = groups

    return best_score
```

Relembrando o erro: é  número de samples classificados no grupo errado, aqui o o erro será usado como fitness, então queremos minimizar o fit.

**Resultados)**




Erros obtidos para uma rodada do algoritmo com alpha (peso para número de genes) = 0.001. Dado o fator estocástico do k-means, às vezes vamos para valores piores do que os já obtidos. Inclusive passamos por uma geração onde havia um erro de 0!  Porém este mesmo indivíduo apresentou erro maior quando utilizado novamente em um k-means.

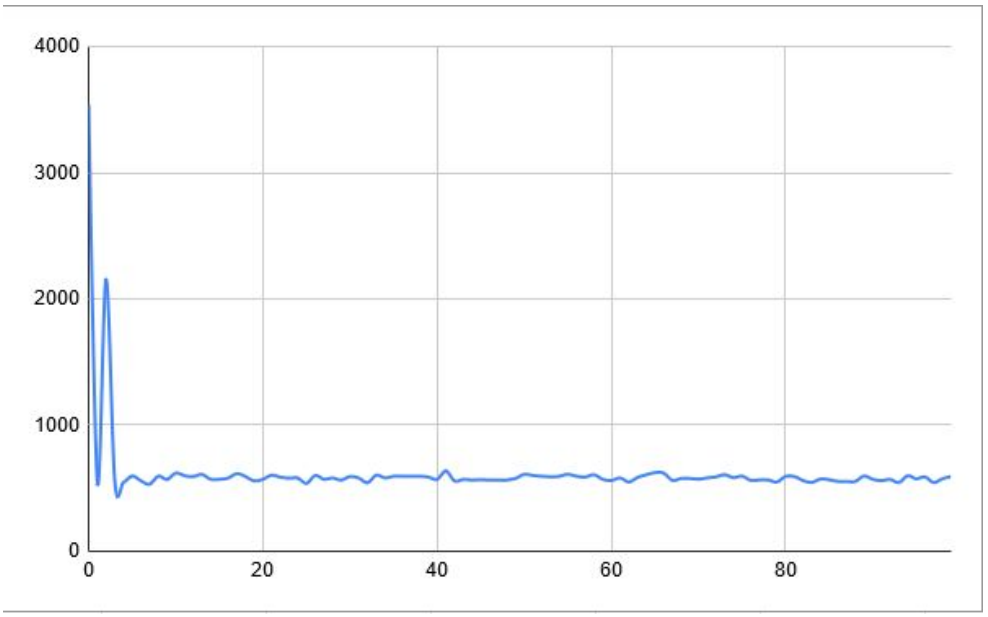



Gráfico para a mesma rodada anterior, porém mostrando o número de genes selecionados. O menor número de genes selecionado foi **531**.

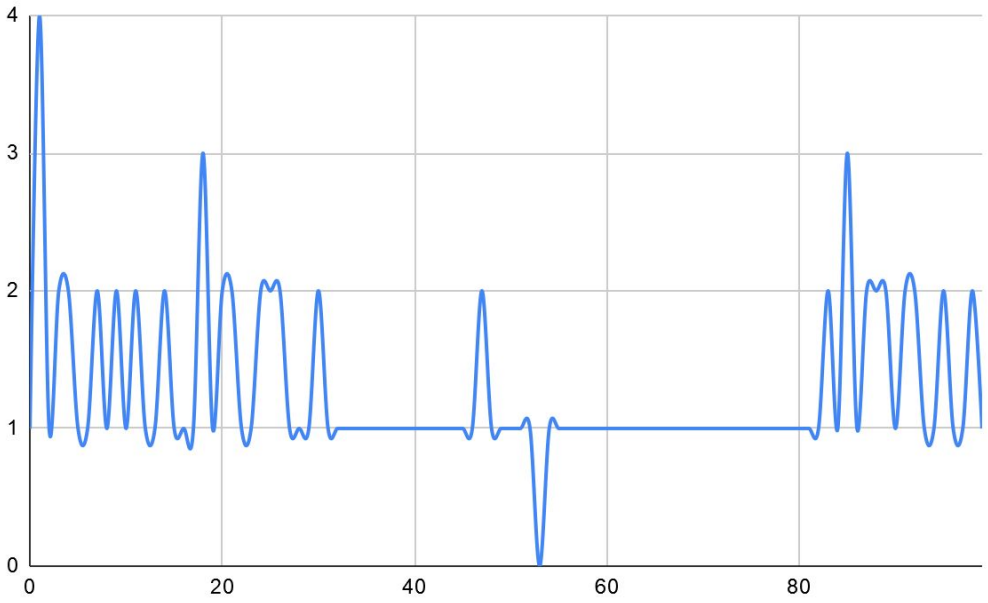
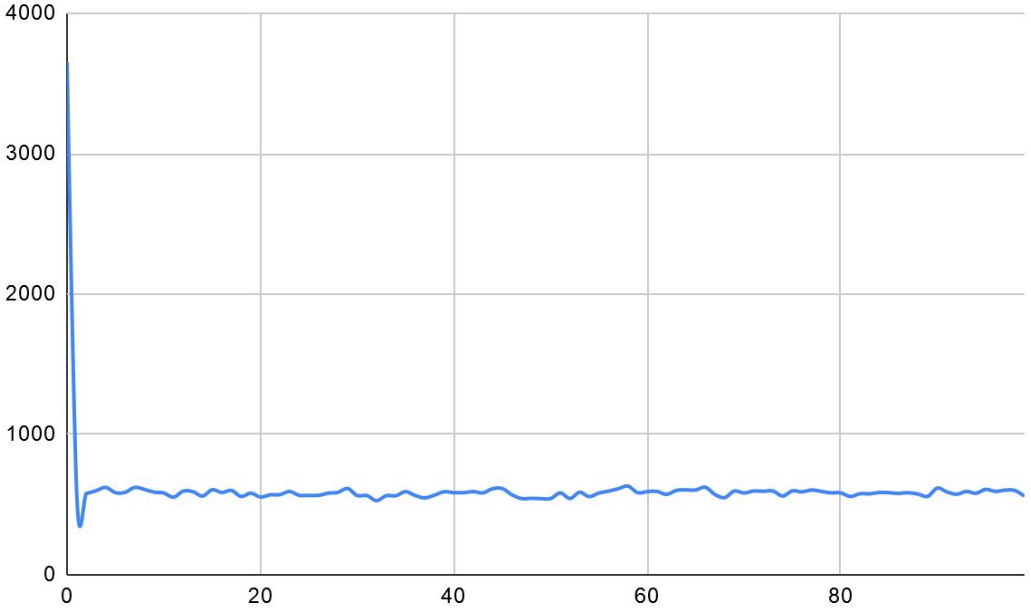
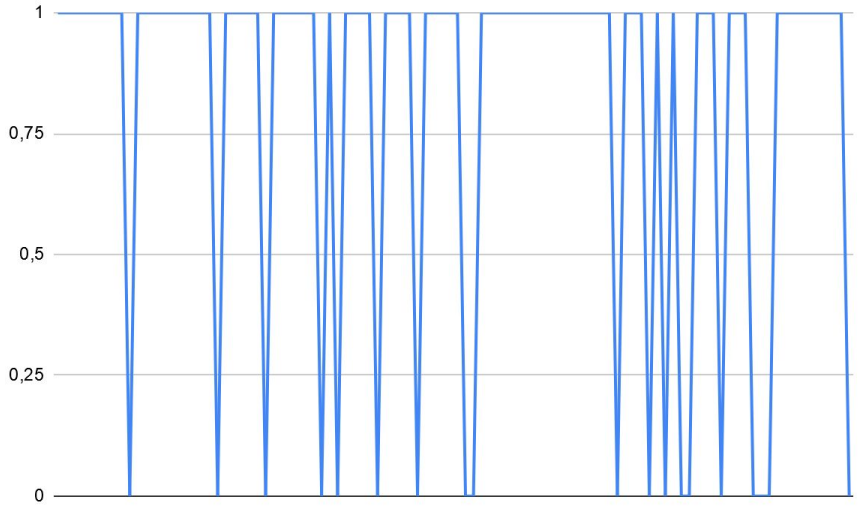



Gráfico rodando um genético com alpha=0.005. resultado muito semelhante ao primeiro, também encontrando uma solução perfeita em algum ponto porém a perdendo em seguida devido ao comportamento estocástico do k-means.

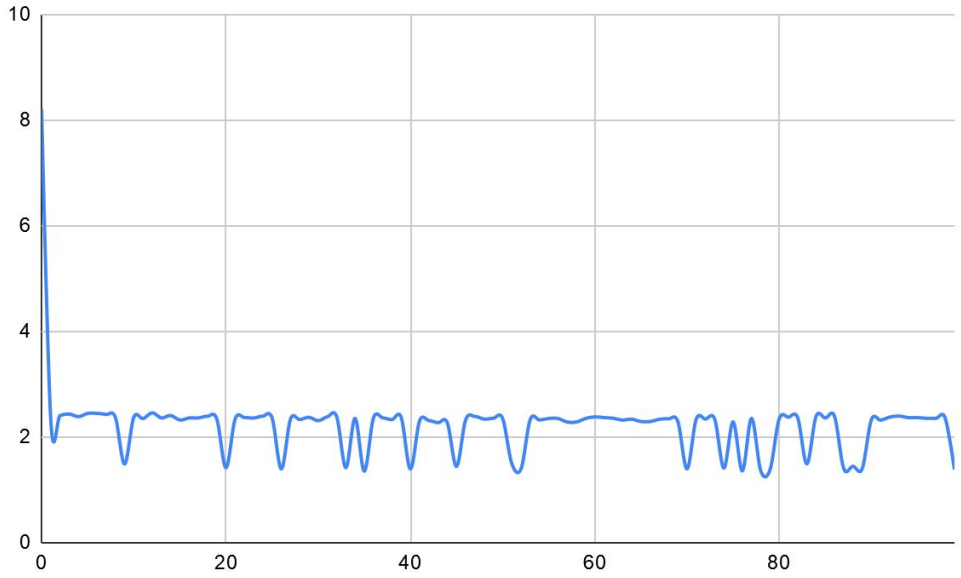



Também como anteriormente, número de genes selecionados.

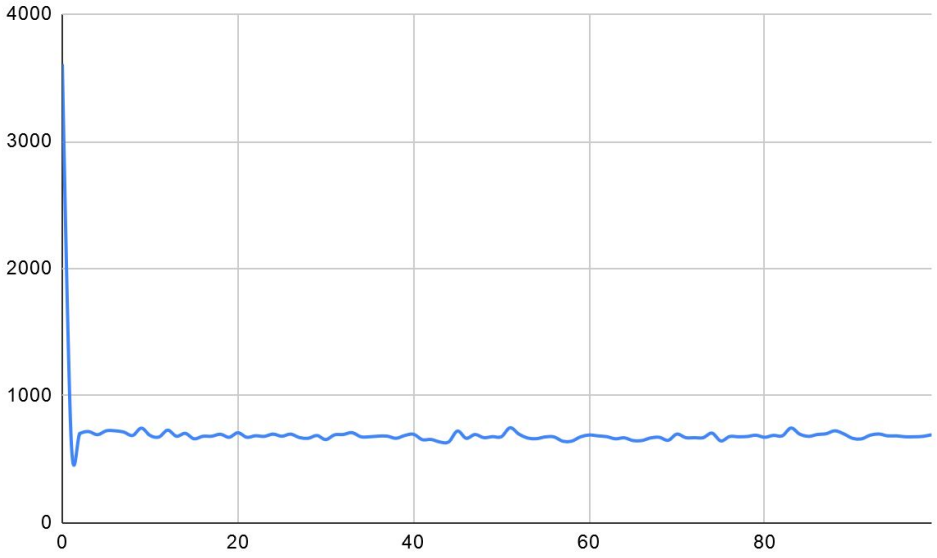
Estas execuções foram na ordem de alguns minutos. Não foi contabilizado, mas o tempo é algo em torno de 15 minutos.




Para tentar apaziguar o efeito estocástico do k-means, como na lista anterior, para cada indivíduo (máscara de genes)  foi executado n vezes k-means (5 aqui, nos resultados anteriores era uma única vez). Notamos que, realmente, o algoritmo ficou mais estável, porém continua variando. Há indivíduos que diversas vezes alcançaram erro 0, porém em execuções seguintes ele geram erro 1! Isto também é, provavelmente, devido ao fitness considerado também levar em conta o número de genes. Execução com 5 vezes k-means rodou na ordem de horas.




Para melhor visualizar o efeito mencionado, este é o gráfico do fitness (erro) total utilizado pelo algoritmo genético (combinação de erros de classificação e penalidade por número de genes). Notamos também o mesmo comportamento alternado.




Por último, o gráfico de número de genes para esta execução. Nota-se que rodando mais vezes o k-means ele não reduziu tanto o número de genes. Aqui o mínimo encontrado foi **660**.