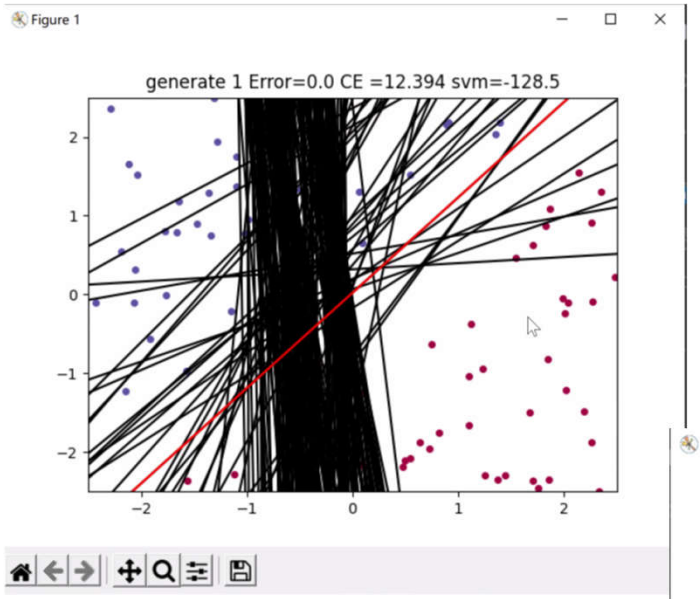
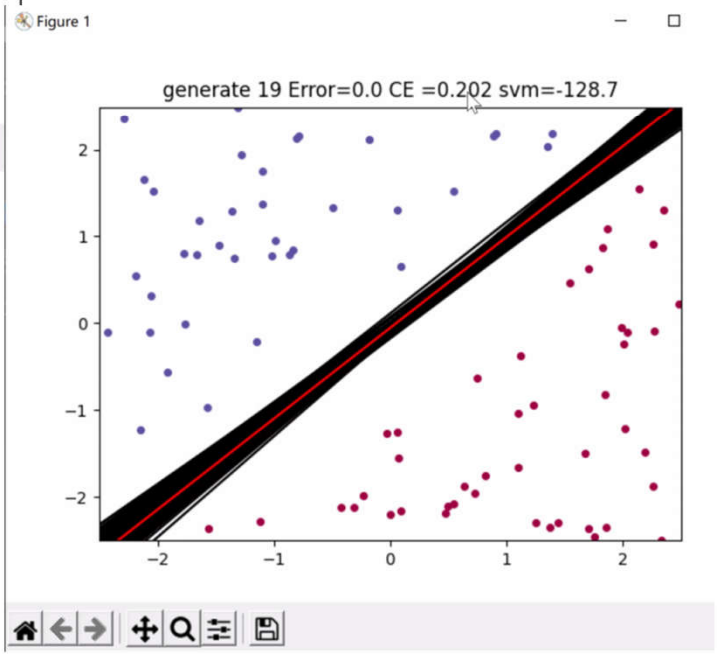


Python编程与人工智能实践

算法篇：遗传算法（Genetic Algorithm）



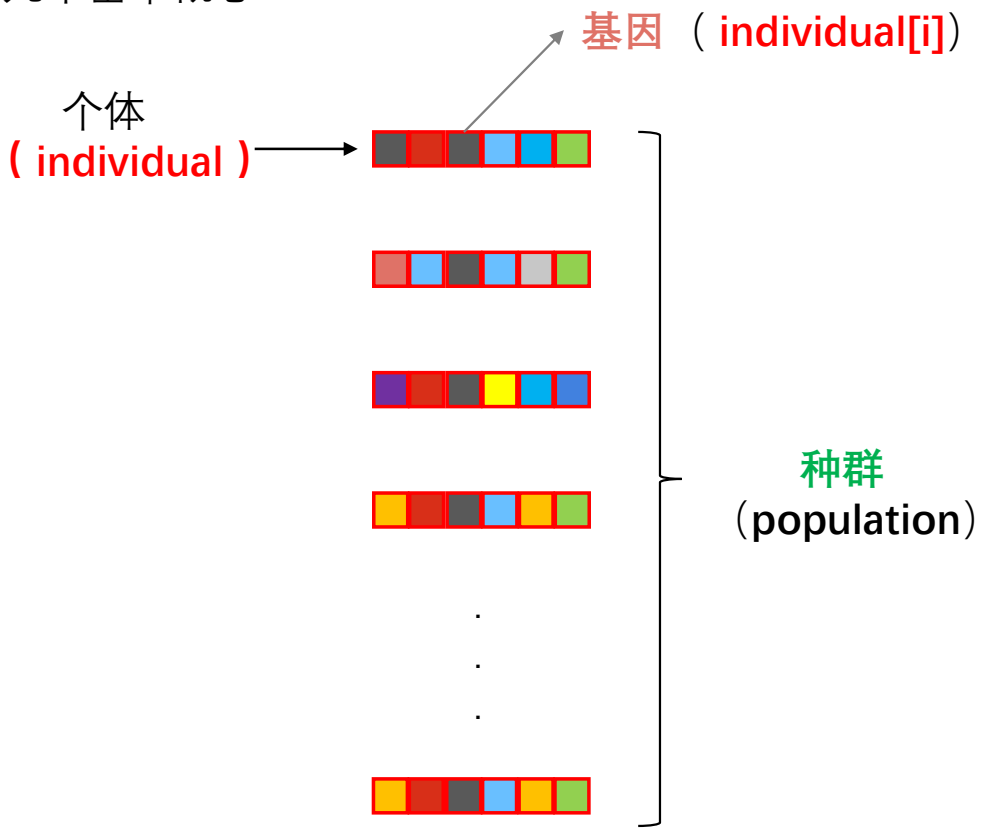
于泓
鲁东大学
信息与电气工程学院
2021.4.1

遗传算法（Genetic Algorithm）

遗传算法在计算机科学以及运筹学中有着广泛的应用，他是一种收到自然选择过程启发的启发式算法属于进化算法大类。遗传算法依赖与生物启发的算子，如变异、交叉和选择等，来生成高质量的优化和搜索问题的解决方案。

遗传算法属于一种**黑盒式**的算法，**分类问题**、**回归问题**、**决策问题**等，都可以利用遗传算法进行实现

几个基本概念:



个体 中存放着需要解决的问题

（1）根据任务需求设计个体的结构
(个体中每个基因的物理意义)

（2）随机生成一组 个体，构成种群

（3）根据种群中的每条个体对环境的适应度，进行种群繁衍

顺序不固定

(3.1) 选择 （selection）
挑选出对环境适应性强的个体

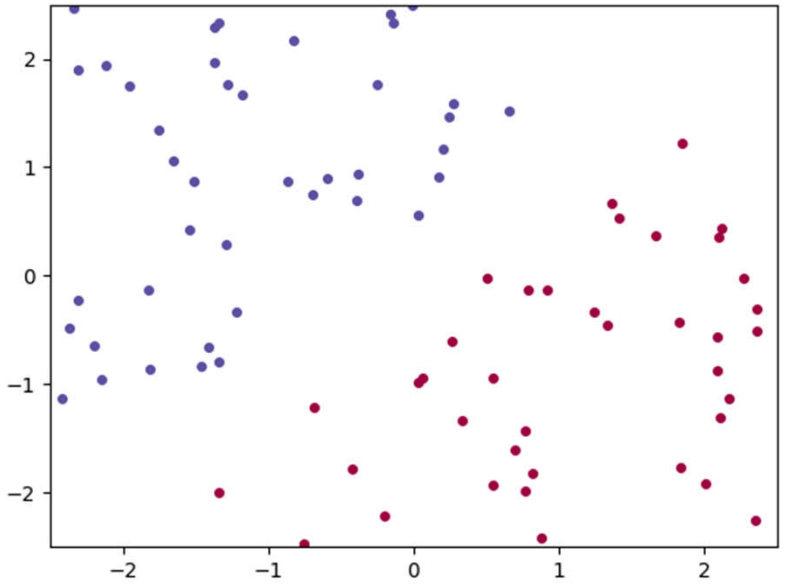
(3.2) 交叉/交配（crossover）
随机选取（父/母）个体，进行基因重组生成新的个体

(3.3) 突变（mutate）
对个体中随机点位的基因进行随机改变

（4）挑选出新种群中适应性最强的个体，并查看是否满足需求. 不满足。跳转步骤（3）

具体案例：简单二分类问题

已知一组二维的样本点，以及其相应的标签
通过遗传算法寻找一条分类直线。



（1）设计**个体**的结构

在本列中，学习的目标是分类直线的权重

基因　[**W0,W1,W2**] 是一个三维的矢量

W0*x+**W1***y+**W2** = 0 → **分类直线**

构造一个**个体** → 3

```
def create_individual(individual_size):
    """
    Create an individual.
    """
    return [random() for i in range(individual_size)]
```

（2）生成一组 **个体**，构成**种群**

```
def create_population(individual_size, population_size):
    """
    Create an initial population.
    """
    return [create_individual(individual_size) for i in range(population_size)]
```

（3）根据种群中的每个**个体**对环境的**适应度**，进行种群繁衍

准确率：$h(X;\mathbf{w})=\text{sigmoid}(x*w_0+y*w_1+w_2)=\text{sigmoid}(X\mathbf{w})$

交叉熵： $L(\mathbf{w})=-\sum_{i=1}^{N} Y_i \log\left(h(X_i;\mathbf{w})\right)+\left(1-Y_i\right)\log\left(1-h(X_i;\mathbf{w})\right)$

经过了维度扩展 [N,3]

```
def test_accuracy(datas,labs,w):

    if len(np.array(w).shape)!=2:
        w = np.array(w)
        w = np.expand_dims(w,axis=-1)

    N,D = np.shape(datas)
    z = np.dot(datas,w) # Nx1
    h = sigmoid(z)          # Nx1
    lab_det = (h>0.5).astype(np.float)
    error_rate=np.sum(np.abs(labs-lab_det))/N
    return error_rate
```

```
def CE(X,Y,W):
    if len(np.array(W).shape)!=2:
        W = np.array(W)
        W = np.expand_dims(W,axis=-1)
    z = np.dot(X,W)
    h = sigmoid(z)
    eps= 0.000001
    loss_ce = -np.sum(Y*np.log(h+eps)+(1-Y)*np.log(1-h+eps))
    return loss_ce
```

交叉熵： $$L(\mathbf{w})=-\sum_{i=1}^{N} Y_i \log\left(h(X_i;\mathbf{w})\right)+\left(1-Y_i\right)\log\left(1-h(X_i;\mathbf{w})\right)$$

```
def CE(X,Y,W):
    if len(np.array(W).shape)!=2:
        W = np.array(W)
        W = np.expand_dims(W,axis=-1)
    z = np.dot(X,W)
    h = sigmoid(z)
    eps= 0.000001
    loss_ce = -np.sum(Y*np.log(h+eps)+(1-Y)*np.log(1-h+eps))
    return loss_ce
```

svm距离：

(+1, -1)

$$\text{loss}_{\text{svm}}=-\left|\sum_{i=1}^{N}\frac{L_i(X_iW)}{|W|}\right|$$

```
def loss_SVM(datas,labs,w):
    if len(np.array(w).shape)!=2:
        w = np.array(w)
        w = np.expand_dims(w,axis=-1)
    labs_svm = labs.copy()
    labs_svm = labs_svm*2-1

    z = np.dot(datas,w) # Nx1
    loss_svm = np.abs(np.sum(z*labs_svm))/np.linalg.norm(w)
    return -loss_svm
```

(3) 根据种群中的每条**染色体**对环境的**适应度**，进行种群繁衍

计算一条染色体的适应度

```python
# 计算损失函数
# 评估一条基因样本（individual）的好坏
# datas labs 用来评估的数据
def get_fitness(individual, datas,labs):
    # 使用了三种评估手段

    # error 错误率
    error_rate = test_accuracy(datas,labs,individual)

    # ce_loss 交叉熵
    ce_loss = CE(datas,labs,individual)

    # svm_loss
    svm_loss = loss_SVM(datas,labs,individual)

    # 三种损失 都是越小越好
    return {'CE': ce_loss,
            'error': error_rate*100,
            'svm':svm_loss,
            'coeff': individual}
```

储存最好

计算适应度并**（3.1）选择**:

评估方法

存储最好结果

选择数目

```python
def evaluate_population(population,datas,labs,
                        method,
                        selection_size,
                        best_individuals_stash):

    # 计算每条样本的适应度
    fitness_list = [get_fitness(individual, datas,labs)
                    for individual in population]

    # 以method 作为标准对 对获取的individuals 按照
    # 适应度从好到坏进行排序
    error_list = sorted(fitness_list, key=lambda i: i[method])

    #  选取最好的selection_size条
    best_individuals = error_list[: selection_size]

    # 将其中最好的一条先储存起来
    best_individuals_stash.append(best_individuals[0]['coeff'])

    # 将种群中最好的那条样本的适应度打印出来
    print('Error: ', best_individuals[0]['error'],
          'CE: ', best_individuals[0]['CE'])

    # 返回最好的selection_size条样本
    return best_individuals
```

返回最好的 selection_size条

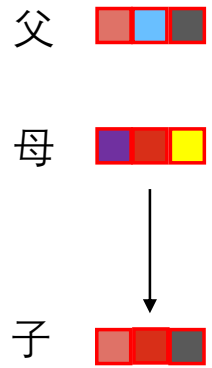
(3.2) 交叉/交配（crossover）

```
def crossover(parent_1, parent_2):

    individual_size = len(parent_1)

    loci = [i for i in range(0, individual_size)]
    # loci_1: 从parent_1 中选取的基因点位
    loci_1 = sample(loci, floor(0.5*(individual_size)))
    # loci_2 : 从parent_2 中选取的基因点位
    loci_2 = [i for i in loci if i not in loci_1]

    # 基因融合
    child = np.zeros(individual_size)
    child[loci_1] = np.array(parent_1['coeff'])[loci_1]
    child[loci_2] = np.array(parent_2['coeff'])[loci_2]
    return child.tolist()
```

父本提供若干**基因**

母本提供若干**基因**

基因融合，形成**新的个体**



(3.3) 突变（mutate）

对染色体中随机点位的基因进行随机改变

发生突变的基因点位的比例

```
# 基因突变
# 突变的概率 probability_of_gene_mutating
# 即，在一条染色体中 有 多少个基因会突变
def mutate(individual,probability_of_gene_mutating):
    """
    Mutate an individual.
    The gene transform decides whether we'll add or deduct a random value.
    """
    individual_size = len(individual)
    loci = [i for i in range(0, individual_size)]
    no_of_genes_mutated = floor(probability_of_gene_mutating*individual_size)
    # 随机选中突变的基因点
    loci_to_mutate = sample(loci, no_of_genes_mutated)

    for locus in loci_to_mutate:
        gene_transform = choice([-1, 1])
        change = gene_transform*random()
        # 对该基因点进行随机的加减
        individual[locus] = individual[locus] + change
    return individual
```

随机加减，进行突变

选择后较为优秀的一些个体

```
def get_new_generation(selected_individuals,population_size,
                       probability_of_individual_mutating,
                       probability_of_gene_mutating):

    # 在比较优秀的个体中中随机选择2个作为父母
    # 生成新的种群
    parent_pairs = [sample(selected_individuals, 2)
                    for i in range(population_size)]
    # 生成 population_size个 新的个体
    offspring = [crossover(pair[0], pair[1]) for pair in parent_pairs]

    # 再从中选取部分进行基因突变
    offspring_indices = [i for i in range(population_size)]
    offspring_to_mutate = sample(
        offspring_indices,
        floor(probability_of_individual_mutating*population_size)
    )

    mutated_offspring = [[i, mutate(offspring[i],probability_of_gene_mutating)]
                         for i in offspring_to_mutate]
    for child in mutated_offspring:
        offspring[child[0]] = child[1]
    return offspring
```

种群中发生突变的**个体**的比例

个体中发生突变的**基因**的比例

主程序部分

```
if __name__ == "__main__":
    # 数据生成
    datas_in,labs_in = data_generate()

    # 对输入数据增加一个维度
    N,D = np.shape(datas_in)

    # 增加一个维度
    datas = np.c_[np.ones([N,1]),datas_in]

    # 对lab 进行维度调整 变为Nx1
    labs= np.expand_dims(labs_in,axis=-1) # Nx1
```

```
from math import floor
def data_generate():
    Points = []
    N = 100
    for i in range(N):
        X = uniform(-2.5, 2.5)
        Y = uniform(-2.5, 2.5)
        Points.append((X, Y))
    class_1 = []
    class_2 = []

    for point in Points:

        if point[1]-point[0]-0.5>0:
            class_1.append(point)
        elif point[1]-point[0]+0.5<0:
            class_2.append(point)

    N1 = len(class_1)
    N2 = len(class_2)
    print(N1,N2)
    datas = class_1+class_2
    labs = [0]*N1+[1]*N2
    return np.array(datas),np.array(labs)
```

y= x +0.5

y= x -0.5

参数设置

```
# 遗传算法
# 获取每条基因的长度
individual_size = len(datas[0])

# 繁衍种群的数目
population_size = 1000

# 从选取最好的rate_select 进行下一轮的种群繁衍
rate_select = 0.2
selection_size = floor(rate_select*population_size)

# 繁衍的轮次
max_generations = 50

# 新生的种群中有 10% 的染色体会发生基因突变
probability_of_individual_mutating = 0.1

# 每个发生突变的染色体中有100%的基因会发生变化
probability_of_gene_mutating = 1

#best_possible = multiple_linear_regression(inputs, outputs)
# 用来存储每轮最好的基因
best_individuals_stash = [create_individual(individual_size)]
```

```
# 种群初始化
initial_population = create_population(individual_size, population_size)
current_population = initial_population
# termination = False
generation_count = 0
# 使用交叉熵作为 优质染色体选择的标准
method = 'svm'
for i in range(max_generations):
    plt.ion()
    current_best_individual = get_fitness(best_individuals_stash[-1], datas,labs)
    print('Generation: ', i)
    # 基因选择 从当前种群中选取最好的selection_size个染色体
    best_individuals = evaluate_population(current_population,
                                           datas,labs,method,
                                           selection_size,
                                           best_individuals_stash)
    # 使用交叉、突变的方法繁衍新的种群
    current_population = get_new_generation(best_individuals,
                                            population_size,
                                            probability_of_individual_mutating,
                                            probability_of_gene_mutating,
                                            )

    print(best_individuals_stash[-1])
    ws = [individual['coeff'] for individual in best_individuals]
    str_title="generate %d Error=%.1f CE =%.3f svm=%.1f"%(i,
                                                          best_individuals[0]['error'],
                                                          best_individuals[0]['CE'],
                                                          best_individuals[0]['svm'])
    my_draw_line(datas_in,labs_in,ws,n_cluster=2,str_title=str_title)
    plt.pause(0.5)
    plt.ioff()

my_draw_line(datas_in,labs_in,[best_individuals_stash[-1]],n_cluster=2)
plt.show()
```

打印当前种群中最好染色体的适应度

选择

交叉突变

结果显示

```python
def my_draw_line(datas,labs,ws,n_cluster=2,str_title=""):
    plt.cla()

    colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1,n_cluster)]

    # 画点
    for i,lab in enumerate(labs):
            plt.scatter(datas[i,0],datas[i,1],s=16.,color=colors[lab-1])

    # 画线
    # 画判决线
    min_x = np.min(datas[:,1])
    max_x = np.max(datas[:,1])

    x = np.arange(min_x,max_x,0.01)

    for i,w in enumerate(ws):
        y = -(x*w[0]+w[2])/w[1]
        plt.plot(x,y,color=(0,0,0,1.0))

    w = ws[0]
    y = -(x*w[0]+w[2])/w[1]
    plt.plot(x,y,color=(1.0,0,0,1.0))

    plt.xlim(-2.5,2.5)
    plt.ylim(-2.5,2.5)

    plt.title(str_title)

    plt.show()
```

画点