基于布谷鸟搜索算法的 SVM 调参优化

1 引言

据美国疾病控制预防中心的数据，现在美国 1/7 的成年人患有糖尿病。到 2050 年，这个比例将会快速增长至高达 1/3。我们在 UCL 机器学习数据库里一个糖尿病数据集，通过这一数据集，利用机器学习来帮助我们预测糖尿病。SVM 算法能够很好的解决二分类问题，利用 SVM 来对糖尿病数据集进行分类预测能达到一个比较好的效果，但是对于 SVM 的参数选择是一个值得探究的问题，使用 PSO、GA 算法无法保证全局收敛，CS 算法在寻优过程中使用 Levy 飞行机制进行解的更新，使得算法的全局搜索能力较强 ，算法又在每次获得新解的时候进行择优保留，迫使算法逐步向最优解逼近 。

2 布谷鸟搜索算法

2.1 基本概念

在 2009 年，剑桥大学的 Xinshe Yang 和拉曼工程大学的 DEB Suash 对布谷鸟的寻窝产卵行为进行模拟，提出了一个新的启发式优化算法——布谷鸟搜索算法（Cuckoo Search）。

这种算法源于模拟布谷鸟巢寄生的行为和莱维飞行习性两个方面。

布谷鸟是一种巢寄生的鸟类，它从不自己筑巢，而是在宿主鸟开始孵卵之前，乘宿主鸟离巢外出时快速寄生产卵放入宿主鸟的巢中，并且移走宿主鸟的一个蛋或者所有的蛋来提高自己蛋孵化的可能性。如果宿主鸟回到巢后发现自己的鸟蛋被移走，就会将布谷鸟蛋扔出自己鸟巢之外或者自己在别的地方再建一个新的鸟巢来繁殖后代。一旦布谷鸟蛋有雏鸡孵出，就会把巢中宿主鸟的蛋推出巢外，并模仿宿主鸟的声音，独享宿主鸟的孵育。

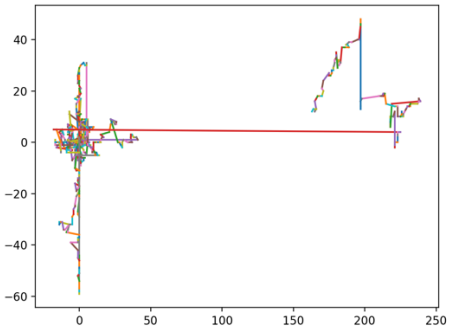
ＣＳ算法在寻优过程中使用 Ｌｅｖｙ飞行机制进行解的更新，使得算法的全局搜索能力较强，算法又在每次获得新解的时候进行择优保留，迫使算法逐步向最优解逼近，另外，算法还采用随机淘汰机制，又能有效避免陷入局部最优解 。

2.2 莱维飞行

莱维飞行是由数学家莱维发现的一种自相似分形系统，这里就直接用鲨鱼捕食的例子来描述自相似：鲨鱼们首先会在比较小的范围内随意游动，不过，

在一段时间后就会忽然跃迁比较长的距离到达另一处地方，并重复随意游动的过程，如果观察鲨鱼游弋的路径，可以发现，大范围的跃迁路径，和局部小范围的游动路径，形态非常相似。各种研究表明，许多动物和昆虫的飞行行为表现出了具有幂律规律的 Lévy 飞行的典型特征，Levy flight 随机游走模式包括大量短距离的运动和少量长距离的运动。很多动物在寻觅食物的时候遵循的就是这种行为模式。

莱维飞行属于随机游走的一种，行走的步长满足一个重尾的稳定分布，在这种形式的行走中，短距离的探索和偶尔长距离的行走相见。在智能算法中采用莱维飞行，能够扩大搜索范围，增加种群多样性，更容易跳出局部最优点，YangXS 和 DebS 认为布谷鸟的寻窝过程是按照莱维飞行搜索路径进行的。



2.3 布谷鸟搜索算法流程

三个假设：为了模拟布谷鸟的寻窝方式，需要假设三个理想化规则

（1）每只布谷鸟每次只会产一个蛋，并把蛋放到随机选择的一个鸟巢中来孵化；

（2）在一组鸟巢中，最好的鸟巢所拥有的鸟蛋将会保留到下一代；

（3）宿主鸟巢的数量 n 是确定的，布谷鸟蛋可能被宿主鸟发现的概率是 Pa (0,1)。

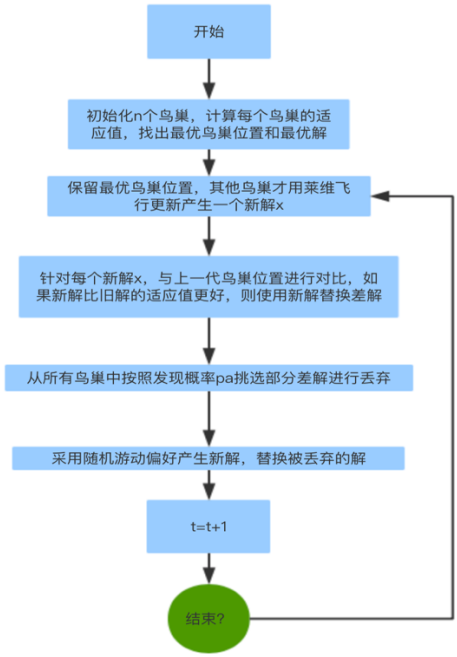
假设(1)说明在每一个巢中只有一个布谷鸟蛋，而不会出现一个鸟巢有多个布谷鸟蛋的情况，每个布谷鸟蛋所在的鸟巢代表一个解。假设(2)说明保留了每一代最好的鸟巢位置，其他的鸟巢位置则要进行位置更新，如果新的位置好则替换旧位置。假设(3)说明如果布谷鸟蛋被宿主鸟发现的概率为 Pa，那么一个鸟巢的鸟蛋被宿主鸟发现的概率大于 Pa，宿主鸟就很有可能发现这个鸟巢不是自己的鸟蛋，宿主鸟就会推走布谷鸟蛋或者放弃鸟巢并建立一个新的鸟巢。 布谷鸟在进行全局寻窝的过程是伴随有 Lév y 飞行的。

Step1：初始化。随机产生 n 个鸟巢的初始位置,计算每个鸟巢的目标函数值，并找出当前最优的鸟巢位置最优解 fmin。

Step2：保留上代最优的鸟巢位置，其他的鸟巢位置按照莱维飞行进行更新，并计算这组鸟巢位置的目标函数值，与上一代鸟巢位置进行对比，目标函数值较好的鸟巢位置替代目标函数值较差的鸟巢位置，从而得到一组当前较优鸟巢位置。

Step3：用随机数 r ∈[0,1]与每个鸟巢被发现的概率 Pa 进行比较，如果r<Pa，则保留鸟巢的位置；如果 r>Pa，则更新鸟巢位置，得到一组新的鸟巢位置，比较更新前后鸟巢位置的目标函数值，目标函数值较好的鸟巢位置替代目标函数值较差的鸟巢位置，从而得到一组当前较优的鸟巢位置。

Step4：选出最优的鸟巢位置，并计算它的目标函数值 fmin 是否满足终止条件。如果满足终止条件则输出最优位置和最优值。否则，返回 Step2。 布谷鸟搜索算法的流程图如下图所示。



2.4 基于布谷鸟搜索算法的 SVM 调参优化

SVM 在 sklearn 库中主要三个参数有 kernel（核函数 linear、RBF），C 是惩罚系数，即对误差的宽容度，c 越高，说明越不能容忍出现误差,容易过拟合。C 越小，容易欠拟合，C 过大或过小，泛化能力变差），gamma 是选择 RBF 函数作为 kernel 后，该函数自带的一个参数。隐含地决定了数据映射到新的特征空间后的分布，gamma 越大，支持向量越少，gamma 值越小，支持向量越多。支持向量的个数影响训练与预测的速度。）

现在我们选择布谷鸟算法对 SVM 中的 C 惩罚系数和 gamma 值进行调优。

3 实验

3.1 实验环境

系统： Mac OS10.13.5；
管理平台：Anaconda3 ，Python3.6
IDE：Pycharm CE

3.2 数据集及预处理

3.2.1 数据集描述

本实验采用的数据来自于 UCI,下载地址如下：
http://archive.ics.uci.edu/ml/datasets/Adult

pima-indians-diabetes.csv
23.2 KiB

文件内容

0	1	2	3	4	5	6	7	8
6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288	33	1
5	116	74	0	0	25.6	0.201	30	0
3	78	50	32	88	31	0.248	26	1

该数据集涵盖了皮马人的医疗记录，以及过去 5 年内是否有糖尿病，所有的数据都以数字的形式呈现。通过分类算法模型，我们可以判断所选对象是否有糖尿病（是为 1 否为 0），该数据集中有 8 个属性及 1 个类别，属于二分类问题。该数据集中有 8 个属性及 1 个类别，表示如下：

属性名	类型
Pregnancies	怀孕次数 --- Number of times pregnant
Glucose	2 小时口服葡萄糖耐量试验中的血浆葡萄糖浓度
BloodPressure	舒张压（毫米汞柱）--- Diastolic blood pressure (mm Hg)
SkinThickness	三头肌皮褶厚度 (毫米) --- Triceps skin fold thickness (mm)
Insulin	2 小时血清胰岛素（mu U/ml) --- 2-Hour serum insulin (mu U/ml)
BMI	体重指数（BMI）--- Body mass index
DiabetesPedigreeFunction	糖尿病血系功能 --- Diabetes pedigree function
Age	年龄（年）--- Age (years)
Outcome	类别：过去 5 年内是否有糖尿病 --- Class variable (0 or 1)

3.2.2 数据处理

- 缺失值处理

通过过对数据进行简单观察，所给的属性数据中有多处标记为“0”的数据缺失，对于这部分缺失数据的处理采用均值替代。

```
# 填充连续型变量缺失值，以均值替代
data[['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin','BMI']]=data[['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin','BMI']].replace(0,np.NaN)# 将 0 替代为空值
data.fillna(data.mean(),inplace=True)# 将空值填充为均值
```

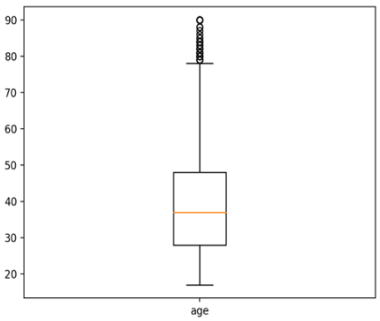
- 去重处理

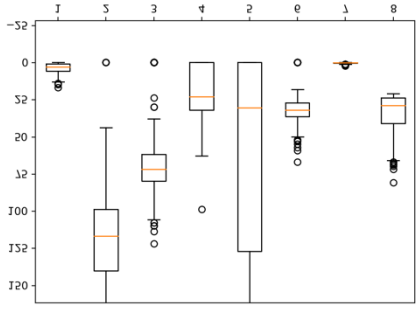
查找出数据重复行，进行删除

```
#查找出数据重复行
dIndex = data.duplicated()
print(dIndex)
#去除重复
data = data.drop_duplicates()
```

- 异常值处理

异常值处理是对连续型属性数据进行处理，主要依靠箱型图的方式来观察异常值。





通过箱形图观察，在 Insulin318 以上的点为异常点，而 SkinThickness 在 61 以上的判定为异常点，在 BMI 大于 53 以上的点判定为异常点，将异常点按照均值赋值。

```
# 重新赋值，根据筛选出的值进行赋值
data['Insulin'] = data['Insulin'].replace([846,744,680],58)    #将 846，744，680 赋值为 58（58 是均值）
data['SkinThickness'] = data['SkinThickness'].replace(67.1,23) #将 67.1 赋值为 23（均值）
```

```
data['BMI'] = data['BMI'].replace(99,32) #将 99 赋值为 32（均值）
```

- 划分测试集和训练集

按照 train_test_split 随机划分测试集，设置样本比例为 30%

```
X=np.array(data[['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
 'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age']])
Y=np.array(data['Outcome'])
#随机划分,导入交叉验证库
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, ran
dom_state=42)
```

- 数据标准化

要求所有的特征要在相似的度量范围内变化。我们需要重新调整各特征值尺度使其基本上在同一量表上

```
from sklearn.svm import SVC
from sklearn.preprocessing import MinMaxScaler
scaler =MinMaxScaler()
X_train_scaled =scaler.fit_transform(X_train)
X_test_scaled =scaler.fit_transform(X_test)
```

3.3 分类器构造

3.3.1KNN 算法

```
# KNN 算法
from sklearn.neighbors import KNeighborsClassifier
training_accuracy =[]
test_accuracy =[]
# try n_neighbors from 1-10
neighbors_settings=range(1,20)
for n_neighbors in neighbors_settings:
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X_train,Y_train)
    training_accuracy.append(knn.score(X_train,Y_train))
    test_accuracy.append(knn.score(X_test,Y_test))
```

确定 n 的个数

```
# 确定 n 的个数
```

```
plt.plot(neighbors_settings,training_accuracy,label="training_accuracy")
plt.plot(neighbors_settings,test_accuracy,label="test_accuracy")
plt.xlabel("n")
plt.ylabel("Accuracy")
plt.legend()
plt.savefig('knn_compare_model')
plt.show()
```

通过观察，n=9 时效果最好，开始使用 knn 进行聚类

```
# 选取 n=9
knn =KNeighborsClassifier(n_neighbors=9)
knn.fit(X_train,Y_train)
print('Accuracy of KNN classifier on train set:{:.3f}'.format(knn.score(X_train,Y_train)))
print('Accuracy of KNN classifier on test set:{:.3f}'.format(knn.score(X_test,Y_test)))
from sklearn.metrics import classification_report
print(knn.predict(X_test))
target_names = ['losing', 'active']
print(classification_report(Y_test,knn.predict(X_test), target_names=target_names))
```

n=9 Accuracy of K-NN classifier on test set:0.70

3.3.2 决策树算法

```
# 决策树
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train,Y_train)
print('Accuracy of tree classifier on train set:{:.3f}'.format(tree.score(X_train,Y_train)))
print('Accuracy of tree classifier on test set:{:.3f}'.format(tree.score(X_test,Y_test)))
```

Accuracy of tree classifier on test set:0.714

3.3.3 支持向量机

```
# 支持向量机
from sklearn.svm import SVC
from sklearn.preprocessing import MinMaxScaler
scaler =MinMaxScaler()
```

```
svc=SVC ()
svc.fit(X_train,Y_train)
print('Accuracy of SVM classifier on train set:{:.3f}'.format(svc.score(X_train,Y_train)))
print('Accuracy of SVM classifier on test set:{:.3f}'.format(svc.score(X_test,Y_test)))
```

Accuracy of SVM classifier on test set:0.73

3.3.4 分类器评估选择

分类器	准确率（测试集）
KNN	70%
决策树	71.4%
SVM	73%

通过三种分类器的对比，SVM 的效果要优于其他两种，因此我们选择以 SVM 来进行分类，继而用优化算法进行调优。

3.4 参数设置

CS 算法有 4 个重要的参数：鸟巢数目 n，发现概率 pa，步长 a 以及莱维飞行的参数λ，其中后面三个参数控制着算法进行全局以及局部的搜索平衡在布谷鸟算法中，发现概率 pa 和步长 a 在初始化时设置为固定的值，一般 pa 设置为 25%，因此这两个值在以后的每一次迭代中都不改变。参数λ和鸟巢数目对算法的影响比较小，可以忽略，通常将设置的λ为 1.5，n=20-30，在本次实验中，我们设置迭代次数为 2000，巢穴数量 n=20，pa=0.25，设置 SVM 的两个参数的上下界范围均为（0.00001，10000）

```
time=2000  #迭代次数
n=20     #n 为巢穴数量
dim=2  #需要寻优的参数个数
pa=0.25 #发现概率
Lb=np.array([0.00001,0.00001]) #参数下界
Ub=np.array([10000,10000])  #参数上界
```

3.5 优化模型设计

- 目标函数（适应值）

以 SVM 对测试集的预测错误率作为目标函数，即需要找到适应值的最小值

```
#目标函数求最优函数
from sklearn import svm
def fobj(bestnest,X_train,Y_train,X_test,Y_test):
    model=svm.SVC(C=bestnest[0],gamma=bestnest[1])
    model.fit(X_train,Y_train)
    r=model.score(X_test,Y_test)
    fitness=1-r    #以预测错误率作为函数优化目标
    return fitness
```

- 初始化

初始化鸟巢位置和目标函数，随机产生 n 个鸟巢的初始位置 ，对目标函数值初始化赋值为 1，因为错误率最高为 1 即 100%分错的情况，其余情况均优于这种情况。

```
#随机初始化巢穴
nest=np.zeros((n,dim))
for i in range(n):
nest[i,:]=Lb+(Ub-Lb)*np.random.rand(1,len(Lb))  #对每个巢穴初始化参数
#目标函数值初始化 （设置为 1）
fitness=np.ones([n,1])
 #开始迭代
```

- 开始迭代

迭代过程一共包含四个函数部分

第一部分随机走动函数，寻找鸟窝位置 nest(更新鸟窝公式)

```
def get_cuckoos(nest,best,Lb,Ub):
    n=nest.shape[0]    #鸟巢个数，矩阵行数
    beta=3/2
    sigma=(math.gamma(1+beta)*math.sin(math.pi*beta/2)/(math.gamma((1+beta)/
2)*beta*2**((beta-1)/2)))**(1/beta)
    for j in range(n):
        s=nest[j,:]    #提取当前鸟巢的参数
        w=s.shape
        u=np.random.randn(w[0])*sigma    #生成服从 N(0,sigma^2)的随机数 u，
        v=np.random.randn(w[0])           #生成服从 N(0,1)的随机数 v 向量
        step=u/abs(v)**(1/beta)          #计算步长
        stepsize=0.01*step*(s-best)     #巢穴位置变化量
        s=s+stepsize*np.random.randn(w[0])        #步长调整，更新另外一组鸟巢
        b=simplebounds(s,Lb,Ub)
```

```
        nest[j,0]=simplebounds(s,Lb,Ub)[0]
        nest[j,0]=simplebounds(s,Lb,Ub)[1]
    return nest
```

第二部分找到当前最优鸟巢

```
def get_best_nest(nest,new_nest,fitness,X_train,Y_train,X_test,Y_test):
    w=nest.shape[0]
    for j in range(w):
        fnew=fobj(new_nest[j,:],X_train,Y_train,X_test,Y_test)      #对每个新巢穴，计算目标函数值
        if fnew <=fitness[j]:    #如果新巢穴的目标函数值优于对应旧目标的函数值
            fitness[j]=fnew       #更新当前巢穴目标函数值
            nest[j,:]=new_nest[j,:]   #更新对应的目标函数
    fmin=np.min(fitness)        #找到当前鸟巢最优函数值
    K=np.where(fitness==np.min(fitness))[0]     #找到最优函数值位置（k 代表第几行，当前 k 个鸟巢为最优鸟巢）
    best=new_nest[K,:]    #找到最优鸟巢位置，k 代表第 K 行的鸟巢，即前 k 个最优
    return fmin,best,nest,fitness,fnew,K
```

第三部分构建新鸟巢来代替，发现并更新劣质巢穴

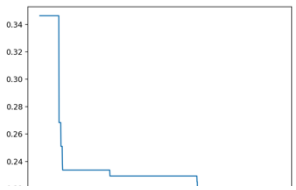
```
def empty_nest(nest,Lb,Ub,pa):
    n=nest.shape[0]    #鸟巢个数
    K=(np.random.rand(nest.shape[0],nest.shape[1])>pa)+0 #鸟巢是否会被发现
    nest1= nest[np.random.permutation(n),:]   # 更新随机改变鸟巢位置
    nest2 = nest[np.random.permutation(n), :]
    stepsize=np.random.rand()*(nest1-nest2)       #计算调整步长
    new_nest=nest+stepsize*K       #新巢穴
    p=new_nest.shape[0]
    for j in range(p):       #遍历每个巢穴
        s=new_nest[j,:]      #提取当前巢穴的参数
        new_nest[j,0] = simplebounds(s, Lb, Ub)[0]
        new_nest[j,1] = simplebounds(s, Lb, Ub)[1]
    return new_nest
```

第四部分边界拉回，使落在边界以外的点使用 simplebounds 落在定义域内

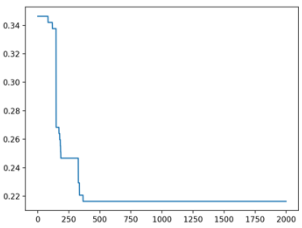
```
def simplebounds(s,Lb,Ub):
    ns_tem=s   #复制临时变量(参数）
    ns_tem=ns_tem.reshape(-1,1)
    Lb=Lb.reshape(-1,1)
    if ns_tem[0]<Lb[0]:   #判断参数是否小于下临界值
```

```
            ns_tem[0]=Lb[0]
    if ns_tem[1]<Lb[1]:   #判断参数是否小于下临界值
            ns_tem[1]=Lb[1]
    if ns_tem[0]>Ub[0]:   #判断参数是否大于下临界值
            ns_tem[0]=Ub[0]
    if ns_tem[1] > Ub[1]:  # 判断参数是否大于下临界值
            ns_tem[1] = Ub[1]
    s=ns_tem       #更新参数
    return s
```

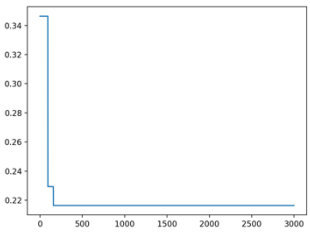
3.6 实验结果



迭代 1000 次，在 time=657 左右开始收敛
fmin= 0.21645021645021645
C=0.293、gamma=0.29
达到 79%的预测率



迭代 2000 次，在 time=357 左右开始收敛
fmin= 0.21645021645021645
C=0.22、gamma=0.22
达到 79%的预测率



迭代 3000 次，在 time=180 左右开始收敛
fmin= 0.21645021645021645
C=0.24、gamma=0.214
达到 79%的预测率

经过多次试验，可以发现 fmin 的最优值为 0.21645021645021645，即对应的预测正确率为 79%左右，此时对应的 C、gamma 值有多组，大致范围在 0.2 左右，本次试验结果即将 SVM 的准确率从 73%提升到了 79%，没有办法再进一步提高。