Lab 1 实验报告

PB20000296 郑滕飞

py 文件：

1、整体框架

由于采用梯度下降法并需要绘制损失曲线，需要计算损失函数及其梯度。而为了作出预测，需要计算 Sigmoid 函数。这三个函数直接根据 ppt 提供的公式实现即可：

[此处计算损失函数及其梯度是按照 ppt 的公式，并没有对样本数归一化，具体对比见第三部分]

```
def sigmoid(self, x):
    """The logistic sigmoid function"""
    return 1.0 / (1 + np.exp(-np.dot(self.w, x)) )
```

```
def count_loss(self, X, y):
    """count loss"""
    loss = 0
    for i in range(y.size):
        loss -= y[i] * np.dot(self.w, X[i])
        loss += np.log(1 + np.exp(np.dot(self.w, X[i])))
```

```
def count_grad_loss(self, X, y):
    """count gradient of loss"""
    grdloss = np.zeros(X[0].size)
    for i in range(y.size):
        grdloss -= (y[i] - 1 + 1.0/(1 + np.exp(np.dot(self.w, X[i]))))*X[i]
```

对于学习的部分，需要先考虑 fit 的写法。注意到，匹配截距相当于给 X 增加一列全为 1 的列，因此这个操作可直接于最开始进行，之后再根据当前 X 的列数建立 w 即可。如果匹配截距，w 的最后一个分量即为截距：

```
if self.fit_intercept == True:
    X = np.c_[X, np.ones(y.size)]
self.w = np.zeros(X[0].size)
```

梯度下降的过程则直接按照算法进行：

```
while np.sum(self.count_grad_loss(X, y) ** 2) > tol:
    if count == max_iter:
        print("reached max iter")
        break
    count += 1
    ls.append(self.count_loss(X, y))
    self.w = self.w - lr * self.count_grad_loss(X, y)
```

由于在 count_loss 中计算出了损失的值，直接将这些值储存成向量 ls 返回，即可绘制损失曲线。

lr 用于控制学习率，也即梯度下降的速度，而 tol 和 max_iter 从两个不同方向确定循环的界。在初值设定 tolerance 1e-4 与 max_iter 1e3 时，一般是由于到达最大迭代次数而返回。

预测部分，直接通过 Sigmoid 函数生成预测值：

```
if self.fit_intercept == True:
    X = np.c_[X, np.ones(X.shape[0])]
y_p = []
for i in range(X.shape[0]):
    if (self.sigmoid(X[i]) > 0.5):
        y_p.append(1)
    else:
        y_p.append(0)
```

2、正则化

刚才的过程中，并没有考虑 penalty 参数与 gamma 参数形成的正则化项。通过查阅资料发现，penalty 为 l1 相当于在损失中添加一项$\gamma||w||_1$，而为 l2 则相当于添加$\frac{1}{2}\gamma||w||_2^2$。前者的梯度为 gamma 倍的 sgn(w)，其中 sgn 为符号函数，而后者的梯度即为 gamma 倍的 w。于是，通过正则化参数可以得到最终的损失与损失梯度：

```
if self.penalty == "l1":
    return loss + self.gamma * np.sum(np.abs(self.w))
return loss + self.gamma * np.sum(self.w ** 2) / 2.0
```

```
if self.penalty == "l1":
    return self.gamma * np.sign(self.w) + grdloss
return self.gamma * self.w + grdloss
```

可以发现，正则化由于加入了 w 的模长作为最小化条件，控制了 w 的模不会过大，这点在迭代次数升高时尤为明显[见后方参数比较]。

ipynb 文件：

1、Data Cleaning & Encode

由于较多项目有 Null 项，对连续的属性采取平均值填充后再去掉 Null：

```
df["LoanAmount"].fillna(df["LoanAmount"].mean(), inplace=True)
df["Loan_Amount_Term"].fillna(df["Loan_Amount_Term"].mean(), inplace=True)
df.dropna(inplace=True)
```

而编码时，为了之后归一化方便，离散属性的编码均使用 0 到 1 间的实数：

```
df.Gender=df.Gender.map({'Male':1,'Female':0})
df.Education=df.Education.map({'Graduate':1,'Not Graduate':0})
df.Married=df.Married.map({'Yes':1,'No':0})
df.Dependents=df.Dependents.map({'0':0,'1':0.25,'2':0.5,'3+':1})
df.Self_Employed=df.Self_Employed.map({'Yes':1,'No':0})
df.Property_Area=df.Property_Area.map({'Urban':1,'Semiurban':0.5,'Rural':0})
df.Loan_Status=df.Loan_Status.map({'Y':1,'N':0})
```

对后代数量，此处假设 1、2、3+的比例是 1:2:4。

2、Data Process

这块分为四个部分：导入 numpy 向量/矩阵、归一化、随机打乱与按比例划分训练集/测试集。

导入部分先用行导入列，再进行转置，归一化则需要利用属性的最大值进行：

```
X[0] = np.array(df.Gender)
X[1] = np.array(df.Married)
X[2] = np.array(df.Dependents)
X[3] = np.array(df.Education)
X[4] = np.array(df.Self_Employed)
X[5] = np.array(df.ApplicantIncome)
X[6] = np.array(df.CoapplicantIncome)
X[7] = np.array(df.LoanAmount)
X[8] = np.array(df.Loan_Amount_Term)
X[9] = np.array(df.Credit_History)
X[10] = np.array(df.Property_Area)
X[5] = X[5] / np.max(X[5])
X[6] = X[6] / np.max(X[6])
X[7] = X[7] / np.max(X[7])
X[8] = X[8] / np.max(X[8])
X = X.T
```

由于数据集不大，且正例与反例比例接近 2:1，差别不大，不需要进行过采样/欠采样等。此外，原数据中分布就已经随机排列，事实上不打乱也可以直接学习。不过为了检验效果，仍随机打乱样本，并将前一部分划分为训练集[默认比例 70%]，后一部分为测试集：

```
index = np.random.permutation(y.size)
X = X[index]
y = y[index]
t = (int)(7 * y.size / 10)
X_train = X[0:(t-1),:]
X_test = X[t:(y.size-1),:]
y_train = y[0:(t-1)]
y_test = y[t:(y.size-1)]
print('size: ' + str(y.size))
print('train size: ' + str(t))
```

3、Train & Test

采用 py 文件中写好的类进行训练：

```
LR = LogisticRegression("l2", 0.5, True)
dr = LR.fit(X_train, y_train, 0.005, 1e-7, 1e3)
```

这里的 dr 保存了每次迭代损失所构成的向量，测试后直接用其进行画图：

```
y_p = LR.predict(X_test)
acc = 1 - np.sum(np.abs(y_p - y_test))/y_p.size
```

```
plt.plot(dr)
print("accuracy:")
print(acc)
```

展示与对比：

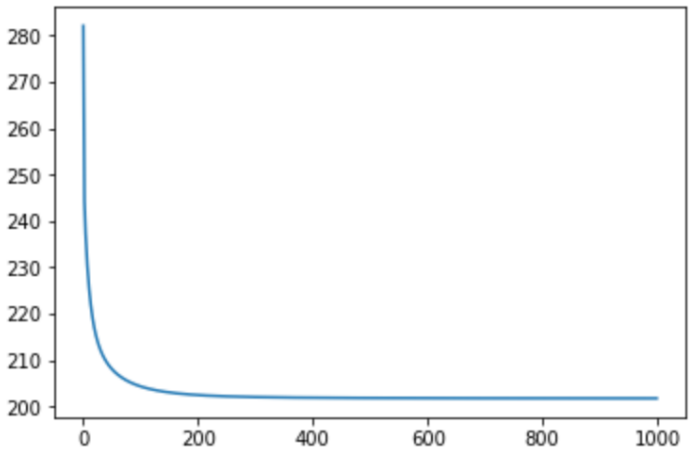
1、最佳准确率及其损失曲线

数据集分割：数据集大小 511 训练集大小 408(80%) 随机分割

参数:

```
penalty = ’l2’ gamma = 0 fit_intercept = False
lr = 0.005 tol = 1e-7 max_iter = 1e3
```

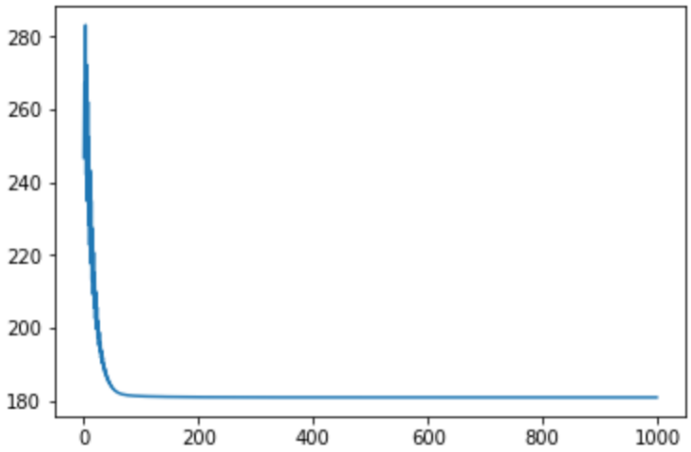
```
accuracy:
0.8823529411764706
```



数据集分割：数据集大小 511 训练集大小 357(70%) 随机分割
参数:

```
penalty = ’l2’ gamma = 0.45 fit_intercept = True
lr = 0.007 tol = 1e-7 max_iter = 1e3
```

```
accuracy:
0.8627450980392157
```



2、参数影响

默认参数为

```
penalty = ’l2’ gamma = 0 fit_intercept = True
lr = 0.01 tol = 1e-4 max_iter = 1e3
```

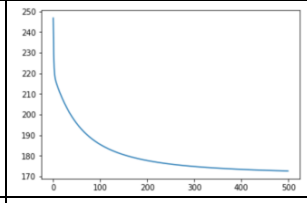
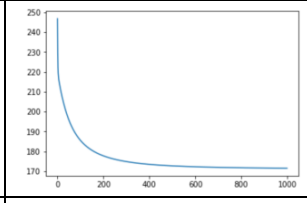
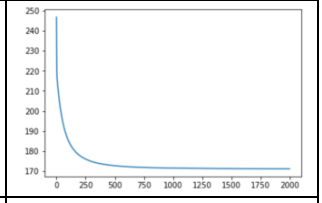
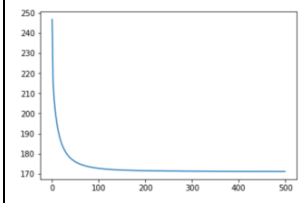
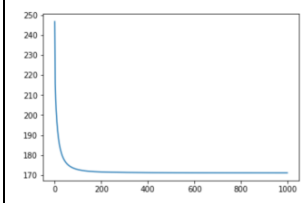
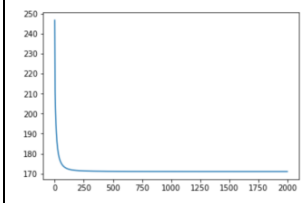
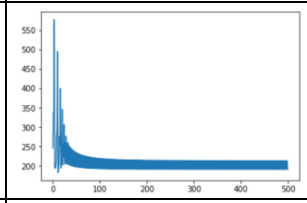
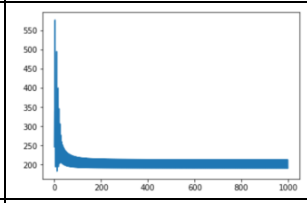
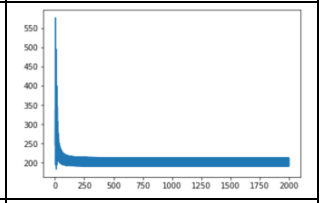
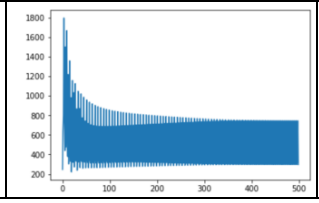
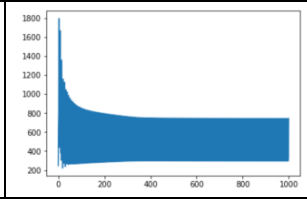
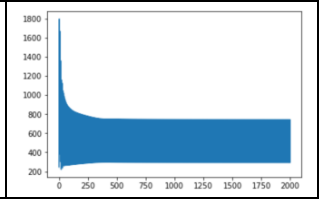
以下叙述参数影响时，未提及的参数均为默认参数。

首先，关于 loss 的归一化问题，根据计算过程不难发现，在计算过程中对 loss 进行归一化，相当于对学习率乘了样本量的倒数，因此，loss 归一化的影响可直接通过学习率进行判别。

其次，由于数据集本身是乱序分布，以下为了对比不同参数造成的影响，去掉了随机打乱的步骤，而是直接以前 70%作为训练集，后 30%作为测试集。

学习率与迭代次数的影响，从直觉上来说，学习率越低则越稳定，但收敛速度会有一定下降，具体的列表如下[表格最左代表 lr，最上代表 max_iter]：

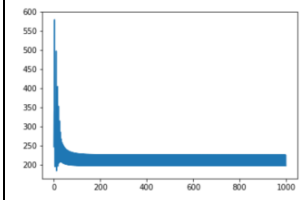
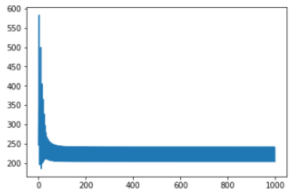
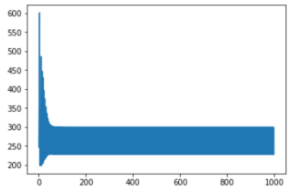
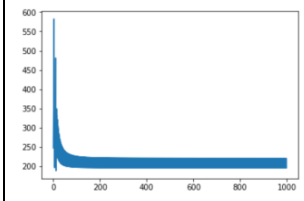
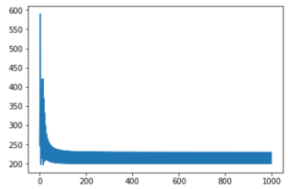
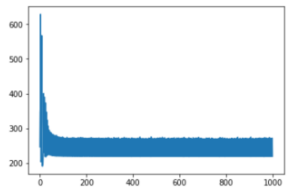
acc	5e2	1e3	2e3
1e-3	0.824	0.824	0.810
5e-3	0.804	0.797	0.797
1e-2	0.824	0.824	0.824
2e-2	0.418	0.817	0.424

loss	5e2	1e3	2e3
1e-3			
5e-3			
1e-2			
2e-2			

可以发现，lr 在 0.005 时曲线都较为稳定，而 0.01 开始已经出现了明显的震荡。当学习率进一步增大时，过拟合也变得更加明显(lr 为 0.02 时迭代次数升高后准确率大幅下降)。

下面观察正则化项的影响[作为参考，上方默认参数时准确率为 0.824]。

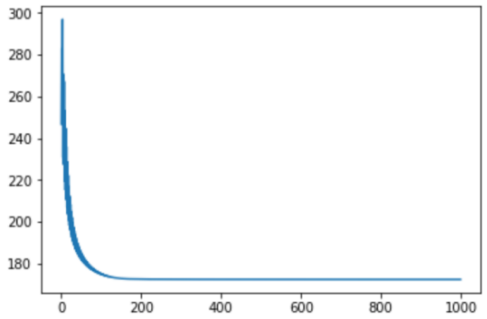
acc	0.2	0.5	2
l1	0.837	0.837	0.778
l2	0.837	0.837	0.837

loss	0.2	0.5	2
l1			
l2			

在正则化项增强时，损失曲线事实上变得更加波动了，这也符合其防止过拟合、保持一定损失的要求。大部分情况下，增加正则化项后结果可以变得更好。不过，当它过强时，由于造成的波动太大，拟合精度事实上降低了。

此外，l2 正则化的效果比 l1 更加明显，这是由于其梯度直接与 w 的值相关，而不是只与符号相关。

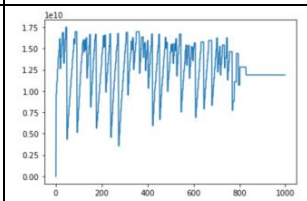
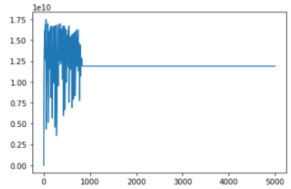
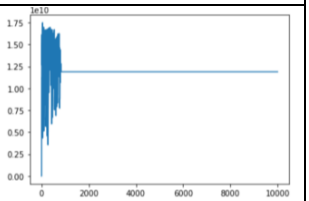
最后，不匹配截距时，收敛事实上变得更加稳定了(这很可能是因为归一化后截距参数造成的影响是很大的)，但准确率下降到了 0.804。



[对于 tol 参数，由于其事实上影响的是迭代次数，可以通过迭代次数的情况观察出它的可能影响。其在实际操作时可以作为一个终点，也有防止过拟合的作用。]

2、数据集操作影响

有关归一化：若不进行归一化，由于数字的量级不同，收敛速度会慢非常多，结果也不尽人意，在其他参数默认时不进行归一化的结果如下：

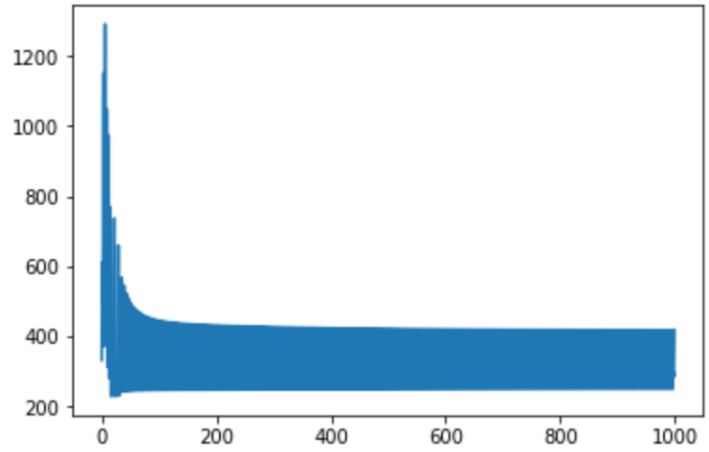
iter	1e3	5e3	1e4
loss			
acc	0.294	0.706	0.307

*由于 loss 计算过程出可能无现穷值，这时需要进行“去头”才能画出曲线：

```
for i in range(len(dr)):
    if dr[i] == float('inf'):
        dr[i] = dr[i-1]
```

正因如此，最终的“稳定”并不是真实的稳定值，而是由于无穷值被填充为了上一个值，此后的 loss 仍然非常大。

有关去掉 NA 值：如果把所有的 Null 全部去掉，样本个数会进一步减少。去掉离散值 Null 后剩余 511 个样本，而去掉全部后只剩下 480 个，学习效果也普遍更差[默认参数准确率 0.811，随机打乱后平均基本小于 80]：



4、算法评价

参数[默认参数加入一定正则化]：

penalty = ’l2’ gamma = 0.3 fit_intercept = True

lr = 0.01 tol = 1e-4 max_iter = 1e3

10 次随机 7:3 留出的结果：

.817 .791 .817 .778 .778 .817 .830 .797 .843 .837

*平均为 81.05%

总结：

虽然乍看实验文档啥也没看明白，探索、写出自己的学习器还是颇有趣的过程。

*调参真的会上瘾啊（恼

Lab 2 实验报告

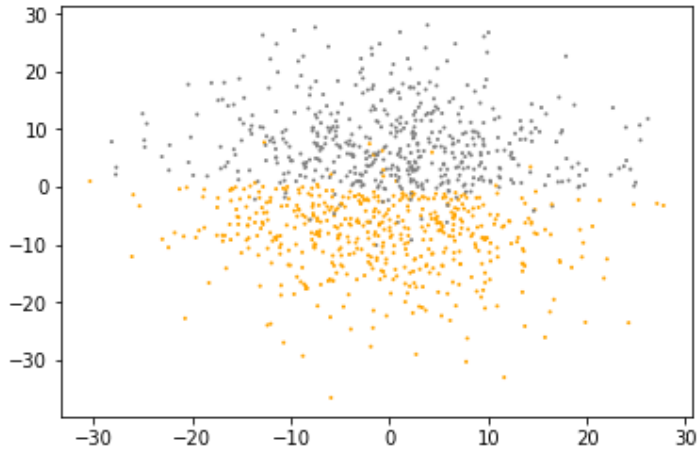
PB20000296 郑滕飞

框架部分：

1、数据生成部分

这部分已经由助教的代码实现了，此处解释一些性质，用于之后对结果的分析。

生成标错的方法是随机反转一部分标签，且 pred 的绝对值越低(也即越接近边界的)越容易被反转，因此，利用二维数据绘制散点图的结果类似于此：



如果想要消除标错，只需要注释掉 `label[i] *= -1` 一句，而提升标错只需要将函数中 `np.abs(pred_n[i])`的比重提升。

此外，由于点的随机生成是对每个维度取均值为 0 方差为 10 的高斯分布随机数，数据的量级是相近的，不需要对初始数据进行额外的归一化等处理。

2、SVM 类

由于两个 SVM 中有很多共同部分，这里先构造了 SVM 类，再将 SVM1 与 SVM2 作为它的子类构造。

SVM 类包含除了匹配之外的所有部分，即创建与预测。此外，为了减轻误差，我还设计了一个函数 drop_error。init 直接根据维度建立 w，并建立 b 即可，预测函数如下：

```
def predict(self, X):
    """
    Use the trained model to generate predicti
    collection of data points.
    """
    y_p = []
    for i in range(X.shape[0]):
        if (np.dot(X[i],self.w) + self.b > 0):
            y_p.append(1)
        else:
            y_p.append(-1)
    return np.array(y_p)
```

drop_error 函数的设计来源于一个构想：由于数据中一开始存在一些噪点，而 SVM 对误差的敏感程度较高，如果通过预处理能减轻噪音，就可以增加精度。

于是，我试图通过判断每个点到自己类与另一类的点的最小距离来确定噪点。如果一个点到自己类的最小距离小于到另一类的，即很有可能为噪点。由于样本规模问题，对每个样本随机选取若干个(实际操作时一般为 100)进行这样的比较，然后确定是否舍去：

```
def drop_error(self, X, y, q):
    num = y.size
    dim = X[0].size
    drop = []
    for i in range(num):
        max_same = 0
        max_diff = 0
        for j in np.random.choice(num, q):
            if j != i:
                temp = np.sum(np.abs(X[i] - X[j]))
                if y[j] == y[i]:
                    if temp > max_same:
                        max_same = temp
                else:
                    if temp > max_diff:
                        max_diff = temp
        if max_diff < max_same:
            drop.append(i)
    X = np.delete(X, drop, axis = 0)
    y = np.delete(y, drop)
    return X, y
```

(此预处理方式的实际效果会在后方结果展示时说明)

3、生成数据与训练

```
# generate data
# X_data, y_data, mislabel = generate_data()
dim, size = 20, 10000

X, y, mislabel = generate_data(dim, size)
y = y.ravel()
print("mislabel: " +str(mislabel))

# split data
index = np.random.permutation(y.size)
X = X[index]
y = y[index]
t = (int)(6 * size / 10)
X_train = X[0:t,:]
X_test = X[t:size,:]
y_train = y[0:t]
y_test = y[t:size]
```

生成数据与分割数据与上一个实验中的类似，为随机打乱后按比例分割。由于实际对比时训练样本至少为 1000，确定分类边界不需要过多样本，这里对数据与测试按照 6:4 进行留出。然后是训练部分：

```
m1 = SVM1(dim)
t1 = time.perf_counter()
loss1 = m1.fit(X_train, y_train, soft_method="hinge", iter=1e3, if_drop=False)
t2 = time.perf_counter()
print("model 1 time: " + str(t2 - t1))

m2 = SVM2(dim)
t1 = time.perf_counter()
loss2 = m2.fit(X_train, y_train, choose_method="random", iter=1e3, if_drop=False)
t2 = time.perf_counter()
print("model 2 time: " + str(t2 - t1))
```

这里利用 time module 进行计时，而 fit 函数的返回值为每次迭代后的损失，用来绘制曲线。

此部分最后会打印数据集的误标率与两种训练方式的时间。

4、预测与对比

```
# make prediction
# pred = model1.predict()
pred1 = m1.predict(X_test)
pred2 = m2.predict(X_test)

# compared with answer
acc1 = 1 - np.sum(np.abs(pred1 - y_test))/(2.0 * y_test.size)
acc2 = 1 - np.sum(np.abs(pred2 - y_test))/(2.0 * y_test.size)
sim = 1 - np.sum(np.abs(pred2 - pred1))/(2.0 * y_test.size)

# compare each methods
print("model 1 acc: " + str(acc1))
print("model 2 acc: " + str(acc2))
print("prediction similarity: " + str(sim))
plt.figure(figsize=(5, 8))
plt.sca(plt.subplot(2,1,1))
plt.plot(loss1)
plt.sca(plt.subplot(2,1,2))
plt.plot(loss2)
```

这里将打印三个数据：两个模型的效果与它们预测的相似度。此外，利用 matplotlib 绘制两个模型训练时的损失曲线。

两种 SVM 匹配算法：

1、方法选择

由于数据集中存在误标，为线性不可分数据集，想获得较好结果必须采用软间隔的 SVM，且根据实验提示，无需使用核方法。

由于软间隔的条件，硬间隔所对应的不等式约束最优化问题被转化为了无约束最优化(且为二次)，于是牛顿迭代、梯度下降等应对无约束最优化的方法都可以采用。这里采取较好实现的梯度下降，并且对比 Hinge、Exp、Log 三种误差函数下的特性。

第二种算法则选取为 SVM 量身定制的 SMO 算法，并且对比不同的选择策略的结果。

2、梯度下降

此方法的主要匹配函数实现较为简单：

```
def fit(self, X, y, soft=2, soft_method="hinge", lr=1e-5, iter=1e3, if_drop=False, if_draw=False):
    """
    Fit the coefficients via your methods
    """
    if soft_method not in ["hinge", "exp", "log"]:
        print("soft method error")
        return

    if (if_drop):
        X, y =  self.drop_error(X, y, 100)

    self.soft = soft

    loss = []
    for i in range(int(iter)):
        if soft_method == "hinge":
            if if_draw: loss.append(self.hinge_loss(X, y))
            gradw, gradb = self.hinge_grad_loss(X, y)
        elif soft_method == "exp":
            if if_draw: loss.append(self.exp_loss(X, y))
            gradw, gradb = self.exp_grad_loss(X, y)
        else:
            if if_draw: loss.append(self.log_loss(X, y))
            gradw, gradb = self.log_grad_loss(X, y)
        self.w -= lr * gradw
        self.b -= lr * gradb
    return loss
```

其中，soft 代表软间隔的强度，soft_method 则是选取的损失函数，按照 PPT，选择了 Hinge 损失、Exp 损失和 Log 损失三种，分别构造了对应的计算损失与损失梯度的函数，如 Hinge 损失的计算：

```
def z(self, x, y):
    return y * (np.dot(self.w, x) + self.b)

def hinge_loss(self, X, y):
    loss = np.dot(self.w, self.w) / 2.0
    for i in range(y.size):
        loss += self.soft * max(0, 1 - self.z(X[i], y[i]))
    return loss

def hinge_grad_loss(self, X, y):
    gradw = self.w
    gradb = 0
    for i in range(y.size):
        if (self.z(X[i], y[i]) < 1):
            gradw = gradw - self.soft * y[i] * X[i]
            gradb -= self.soft * y[i]
    return gradw, gradb
```

lr 和 iter 分别是学习率与迭代次数，这里的损失梯度并没有对样本数量归一化，因此

学习率的值需要设定至较低[Lab 1 实验报告中已解释归一化效果与学习率设定相同]。

3、SMO

SMO 算法的迭代分为三个部分：按照当前的向量λ_i更新解与预测、选取向量中两个要优化的分量、优化。根据理论推导，软间隔的 SMO 算法中，软间隔系数 C（与梯度下降方法中的 soft 类似)会成为λ_i的上界，而下界为 0。

初始化部分如下：

```
def fit(self, X, y, C=2.0, choose_method="turning", iter=1e3, if_drop=False):
    """
    Fit the coefficients via your methods
    """
    if choose_method not in ["turning", "random", "maxerr"]:
        print("choose method error")
        return

    if (if_drop):
        X, y =  self.drop_error(X, y, 100)
```

```
loss = []
num = y.size
lam = np.zeros(num)

pos = 0
neg = 0
for i in range(num):
    if y[i] == 1:
        pos += 1
    else:
        neg += 1

for i in range(num):
    if y[i] == 1:
        lam[i] = C / float(pos)
    else:
        lam[i] = C / float(neg)

a1 = 0

self.w = np.zeros(X[0].size)
for i in range(num):
    self.w = self.w + lam[i] * y[i] * X[i]
```

这里选取的初始化策略是全部赋值为非零值，这是由于 lambda 全为 0 的初始化策略会导致优化效率降低，之后会进行对比。参数 choose method 代表选取第一个分量的方法，分为最大偏差(老师 PPT 上)、轮流与随机三种方式。

此外，这里先计算出了 w 的初值，这样每次更新时 w 的改变只需要计算更新的分量即可。

更新预测与计算损失采用了单独的函数：

```python
def construct_predict(self, X, y, lam, C):
    num = y.size
    u = np.zeros(num)
    for i in range(num):
        u[i] = np.dot(self.w, X[i])

    count = 0
    self.b = 0
    for i in range(num):
        if lam[i] > 0 and lam[i] < C:
            self.b += y[i] - u[i]
            count += 1
    if count:
        self.b /= float(count)

    return u + self.b

def count_loss(self, u, y, lam):
    loss = np.dot(self.w, self.w)/2.0
    for i in range(y.size):
        loss += lam[i] * (1 - y[i] * u[i])
    return loss
```

这里，先更新 u[i]为不加上 b 的版本，再以此计算对应的 b，避免了多次进行点乘运算。最后，将 u[i]加上预测的 b 作为真实的 u[i]返回。而对于损失函数，直接利用公式计算即可。值得注意的是，此处的损失函数由于 SMO 解决的是对偶问题，所以损失函数需要进行最大化，损失函数作图也是上升的。

选取分量采用了三种方式：最多违反 KKT、轮流、随机。第一种方式根据 lam 向量的情况，按照 u*y-1 与正确情况的差距进行选取：

```python
if (choose_method == "maxerr"):
    m = 0
    a1 = -1
    for j in range(num):
        if lam[a1] > 0 and lam[a1] < C:
            temp = abs(u[j]*y[j] - 1)
        elif lam[a1] == 0:
            temp = 1 - u[j]*y[j]
        else:
            temp = u[j]*y[j] - 1
        if temp > m:
            m = temp
            a1 = j
    if a1 == -1:
        return loss
```

第二、三种都是每次改变选取对象，找到第一个违反的：

```
else:
    count = 0
    if (choose_method == "turning"):
            a1 = (a1 + 1) % num
    else:
        a1 = np.random.randint(num)
    while (lam[a1] > 0 and lam[a1] < C and abs(u[a1]*y[a1] - 1) < 1e-6)\
        or (lam[a1] == 0 and u[a1]*y[a1] >= 1)\
        or (lam[a1] == C and u[a1]*y[a1] <= 1):
        if (choose_method == "turning"):
            a1 = (a1 + 1) % num
        else:
            a1 = np.random.randint(num)
        count += 1
        if count == num:
            return loss
```

其中 turning 直接寻找了下一个，而 random 则进行随机选择。

选取完第一个后，第二个按照改变最多进行寻找：

```
e1 = u[a1] - y[a1]

m = 0
a2 = -1
for j in range(num):
    e2 = u[j] - y[j]
    de = e1 - e2
    if abs(de) > abs(m):
        a2 = j
        m = de
if (a2 == -1):
    return loss
```

无论何种选取策略，只要所有分量都满足要求，迭代即自动停下，也就是收敛。

最后，按照查找的算法对分量进行更新：

```
sum = lam[a1] * y[a1] + lam[a2] * y[a2]
lam2new = lam[a2] + y[a2] * m / np.dot(X[a1]-X[a2],X[a1]-X[a2])

if y[a1] == y[a2]:
    l = max(0, lam[a2] + lam[a1] - C)
    h = min(lam[a1] + lam[a2], C)
else:
    l = max(0, lam[a2] - lam[a1])
    h = min(C + lam[a2] - lam[a1], C)

self.w = self.w - lam[a2] * y[a2] * X[a2] - lam[a1] * y[a1] * X[a1]

lam[a2] = min(max(lam2new, l), h)
lam[a1] = (sum - lam[a2] * y[a2]) * y[a1]

self.w = self.w + lam[a2] * y[a2] * X[a2] + lam[a1] * y[a1] * X[a1]
```

此处，在更新 lam 向量的同时，也将 w 对应更新，减去之前的分量，加上更新后的分

量。这样比起每次重新计算可以节省大量的时间。

***SMO 算法原理**[主要参考 https://zhuanlan.zhihu.com/p/257866920]:

先将软间隔问题转化为其对偶问题:

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j x_i^T x_j - \sum_{i=1}^{m}\alpha_i$$

$$s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$0 \le \alpha_i \le C, \quad i = 1, 2, 3 \ldots, m$$

而由于原问题的特殊性，最优解时应满足 KKT 条件，即α_i为 0 到 C 时时对应的预测 u[i]*y[i]为 1，当其为 0 时 u[i]*y[i]大于等于 1，其为 C 时对应的 u[i]*y[i]小于等于 1。

由此，SMO 的思路为每次选取两个，看作单变量(由于两个之间存在约束)最优化问题进行更新。为使效果最好，原则上第一个采用违反最多的，第二个采用能导致更新量最大化的(更新量可以由二次函数极值得到具体公式，可发现与两个 u[i]-y[i]的差距密切相关)。选取完后，利用计算出的结果:

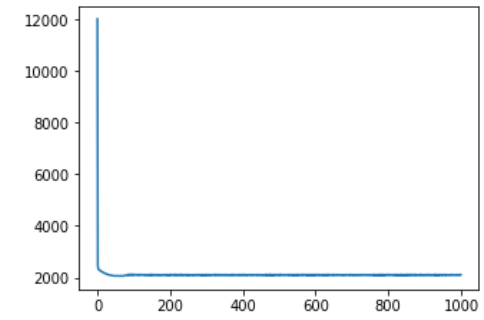
$$\alpha_{2new,unc} = \alpha_{2o} + \frac{y_2(E_1 - E_2)}{K_{11} + K_{22} - 2K_{12}}$$

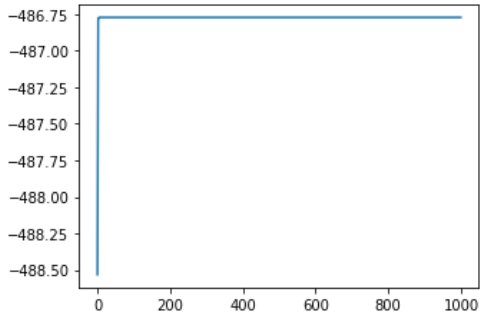
进行更新，但若越过边界需要用不等式条件加以截断，再进一步计算得到第一个分量的更新即可。

展示与对比：

1、标准结果

当维度为 20，数据集为 10000，按照 6:4 进行分割，默认参数的情况下结果如下:





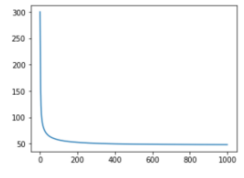
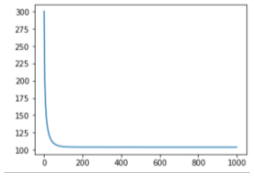
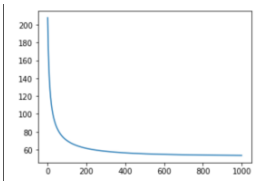
```
mislabel: 0.0385
model 1 time: 24.046289399999978
model 2 time: 15.842080399999986
```

```
model 1 acc: 0.94725
model 2 acc: 0.94925
prediction similarity: 0.9745
```

这里的时间性能并不准确，因为计算出损失函数绘制曲线也会十分耗时。在之后对比时会给出更明确的结果。
[为此，在函数中添加了 if_draw 参数，若为 False 则不计算损失]

2、梯度下降相关对比

通过损失曲线可以发现，迭代收敛的速度事实上比默认参数要快。改为 100 次后，发现仍能有类似的收敛结果。以下为迭代次数 100、学习率 1e-5 时各个方法的对比[由于数据有随机性，每个为三次取平均][此处计算时间时均未绘制损失曲线]：

软间隔参数	Hinge 损失	Exp 损失	Log 损失
0.05	95.2%	94.7%	95.1%
0.5	95.4%	95.0%	95.2%
1	95.6%	溢出	95.8%
2	94.9%	溢出	95.2%
5	92.0%	溢出	95.0%
时间	1.06s	4.39s	4.47s
损失曲线 [0.05，1000 次]			

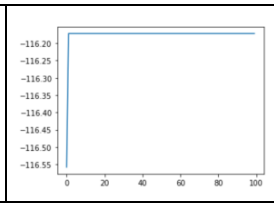
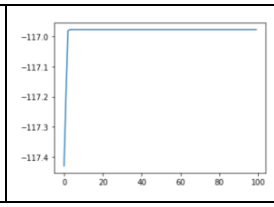
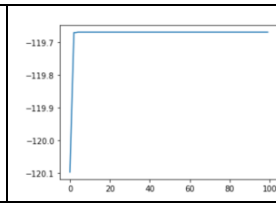
观察损失曲线可以看出， Exp 损失的收敛速度快于 Hinge，Hinge 又快于 Log。而 Log 的准确性一般高于剩下两个。不过，考量速度与参数容忍因素，总体来看，当数据量较大时，Hinge 损失的表现是更优的。

3、SMO 相关对比

SMO 自身的对比主要有两点：向量 lam 初值的影响与选取方式的影响。

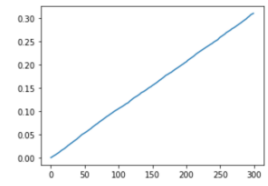
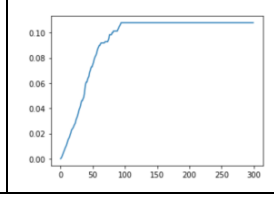
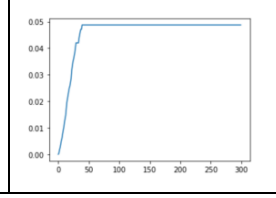
同样以迭代次数 100 构造表格[这里三种方法都是在开始变化很大，之后变化很小，以 100 为次数构造表格仍然是合理的] [此处计算时间时仍然均未绘制损失曲线]：

软间隔参数	最大选取	轮流选取	随机选取
0.05	94.5%	94.3%	93.7%
1	94.3%	94.7%	95.0%
5	94.9%	94.2%	93.8%
时间	2.05s	1.44s	1.41s

损失曲线 **[1，100 次]**			

随机选取造成的结果比起其他两种略差，另两种则未看出明显差别。由于最大选取需要比较所有，总体时间是劣于剩下两种的。

不过，以上的比较都是针对全部非零的初值构造。若初值全部为 0，情况会很不一样[下方参数为 C=1，迭代次数 300，此时轮流选取与随机选取基本收敛]：

选取方式	最大选取	轮流选取	随机选取
准确率	73.9%	84.2%	82.2%
时间	6.03s	4.13s	4.10s
损失曲线			

增加迭代次数会发现，这样的初值会导致最大选取的第一个参数每次为同一个，无法找到好的收敛结果。。

由此看来，第一个选取违反 KKT 条件最大的参数这一原则，在合适的初值选取下，可以被轮流优化替代，且效果并无明显差别。

4、两种算法对比

从刚才的结果来看，在调节到合适的参数时，梯度下降的时间与 SMO 基本相近，且结果的准确率更高。不过，这并不是全部的情况。

在之前，标错率基本在 0.03 到 0.04 之间。当修改函数让此值升高时，结果会有变化[下方表格参数 C 与 soft 均为 1，方法分别为 Hinge 与 Turning，迭代 100 次]：

标错率	**梯度下降准确度**	**SMO 准确度**
0	99.2%	97.7%
0.038	95.1%	94.4%
0.076	91.7%	91.2%
0.107	88.4%	88.2%
0.151	84.2%	84.3%

此外，减小训练集会导致两者的准确度同步下降，但梯度下降的结果仍然普遍优于 SMO。

另一个有趣的事实是超低迭代次数的情况：

迭代次数、数据集大小	**梯度下降准确度**	**SMO 准确度**
1、10000	94.5%	94.6%
10、10000	95.5%	95.0%
1、100000	95.7%	95.7%
10、100000	96.4%	95.8%

无论是梯度下降还是 SMO，似乎在低迭代次数时都表现出了不错的结果。

最后，回到一开始试图优化结果的 drop error 方法。当在数据集中去掉了部分样本后，无论是梯度下降还是 SMO 的准确率都并未受到什么影响。不过，这样选取后的时间能提升约 20%。

就目前的对比而言，SMO 算法的时间性能好于梯度下降，而梯度下降方法在选取合适的参数后可以做到比 SMO 更优。不过，SMO 对参数变化的稳定性要强不少。尤其是其不需要控制学习率进行调整。

总结：

在开始写这个实验之前，结果似乎是清楚的。然而写着写着，问题变得越来越多了——事情是从发现一次迭代也能有很好的结果开始不对劲起来的。总体来说，大概是因为生成的数据过于标准了，两种方法的收敛速度都非常快，也都能在合适情况下达到 95%左右的正确率(考虑到错标率，实际正确率可能有 98%)。此外，在和人交流的过程中，深刻感受到了每个人都有自己的 SMO，大家的方法细微处差别挺大，结果也有较大的差别。

Lab 3 实验报告

PB20000296 郑滕飞

目的/原理：

1、实验目的

以回归树为基学习器，通过 XGBoost 算法集成为最终的学习器，以对连续属性作出预测。

2、实验原理-XGBoost

*参考自 zhuanlan.zhihu.com/p/75217528

XGBoost 算法对每个学习器的结果是直接相加关系，而在训练每个学习器时都是优化最终误差+正则化惩罚项。与传统梯度提升[Gradient Boost]相比，XGBoost 作出了三项优化：添加了正则化惩罚项避免过拟合、展开到二阶项得出更精确信息、可以处理缺失值。

不过，由于我们的实验中使用的损失函数是平方损失，二阶导数为常数，实际上没有信息，且数据集中并没有缺失值，实际体现的优化方式只有正则化项部分。

一般的 XGBoost 算法分为两部分，找第一个与有前 t-1 个时训练第 t 个。前者为方便起见，直接以全零作为初始预测，而对后者，训练f_t时一般公式是优化

$$Obj^{(t)} = \sum_{i=1}^{n}\left(g_i f_t(x_i) + \frac{1}{2}h_i f_t(x_i)^2\right) + penalty(f_t)$$

其中g_i, h_i分别为泰勒展开中得到的损失函数的一阶、二阶导数。

3、实验原理-回归树

由于此为对连续值预测，基学习器采用了回归树。一棵回归树事实上可以看作一个数据到叶结点的映射与叶结点的结果，因此这里的f_t只与叶结点有关。假设树的结构确定，有 T 个叶结点，每个的结果[在 Boosting 的视角为权重]为w_t，则上方的公式变为

$$\mathrm{Obj}^{(t)} = \sum_{j=1}^{T}\left(G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2\right) + \gamma \mathrm{T}$$

其中G_j, H_j代表映射到第 j 个叶结点的数据的 g 与 h 求和。

由于这对w_j是二次函数，最优时有

$$w_j = -\frac{G_j}{H_j + \lambda}, Obj^{(t)} = -\sum_{j=1}^{t}\frac{G_j^2}{2(H_j + \lambda)} + \gamma T$$

而事实上，在实际设计算法时，就是根据新的树的 Obj 比起原树增加的程度[Gain]来决定在何处进行划分。根据 Obj 公式容易计算出，划分结点的收益为

$$Gain = \frac{G_L^2}{2(H_L + \lambda)} + \frac{G_R^2}{2(H_R + \lambda)} - \frac{G^2}{2(H + \lambda)} - \gamma$$

其中 L，R 分别代表划分后左、右的子集。

实验步骤：

1、整体框架

我的整体框架分为三个部分：学习器类定义[学习器的基本代码]、读取文件，并多次划分、训练、测试考察平均性能。通过

```
yp = boosttree.predict(X_test)
rmse = np.sqrt(np.sum((yp-y_test) ** 2) / y_test.size)
r2 = 1 - np.sum((yp-y_test)**2) / np.sum((yp-np.average(y_test))**2)
```

分别计算出每次的预测结果，对应的均方误差与R^2，并且作出损失曲线进行比较。

2、XGBoost

排除去构造树的部分，XGBoost 算法较为简单，也即重复执行：预测、构造当前预测对应的 g[由于 h 是常数，不需要构造]、由 g 训练基学习器。

```
self.trees.append(self._fit_node(X, 2 * gdiv2, 0))
gdiv2 = self.predict(X) - y
```

其中，gdiv2 是当前预测值与真实值的差距，对损失函数求导可以发现其也就是 g 值除以 2。

注意到，整个决策树的生成过程中，除了属性集外只用到了 g，并不会用到 y，因此只需要传入 g 即可。第三个参数 0 代表深度，方便在深度达到阈值时停止。

而关于结束策略，通过 gdiv2 容易算出均方误差，若训练新树导致的均方误差减小不到某个阈值[这里设定为 1e-7]，就直接停止。此外，设置了最大树数量的限制，避免不收敛时进入无限循环。不过，在实际测试中，从来没有到达过默认 20 的最大树数量。

3、决策树

生成决策树的部分 fit node 就较为复杂了。总体思路是，每次选取增益最大的分割点进行分割[注意到分割只涉及顺序，因此没有必要对属性作归一化]，直到达到停止条件：

```
node = BTree()
feature, split, max_gain = self._get_best_split(X, g)
if max_gain < self.gain_tol\
    or depth >= self.max_depth\
    or length(X) <= self.min_num:
    set node's property as a leaf
else:
    set node's property as an inner node
    split data to l, r
    node.lchild = _fit_node(x[l],g[l],depth+1)
    node.rchild _fit_node(x[r],g[r],depth+1)
```

这里的停止条件综合了增益、深度与结点中最小的数据量，关于停止条件的后续比较会在结果展示中进行。

而对于具体的获得划分的过程，根据

$$Gain = \frac{G_L^2}{2(H_L+\lambda)} + \frac{G_R^2}{2(H_R+\lambda)} - \frac{G^2}{2(H+\lambda)} - \gamma$$

可以计算。不过，注意到对于某个特定属性，只要进行排序，就有 L 每次增加一个，R 每次减少一个，因此**预排序可以大大简化计算增益的过程**。

具体的方式是，对属性进行遍历，在针对某一属性寻找最优划分时，可以先将这个属性连同 g 一起排序，并将初始的 GL 设置为 0，GR 设置为 G[也即所有结点都在左侧]，接着进

行划分:

```
for i = 0 to size-2:
    Gl += g[i], Gr -= g[i], Hl += 2, Hr -= 2
    if feature[i] == feature[i+1]:
        continue
    score = Gl**2 / (Hl + self.lam) + Gr**2 / (Hr + self.lam) - root_score
    if score > max_score:
        max_score = score
        split_best = (feature[i] + feature[i+1]) / 2
```

由于每个属性有大量重复，只需要在排序前一个与下一个有不同的地方进行划分比较，此外，这里计算的 score 并不是真实的 Gain 值，Gain 值为其除以 2 后减去 gamma。由于获取最优划分点只需要找到最大处，过程并不需要得到 Gain，只要返回时操作最大的 score 即可。

最后，通过每个属性的最优划分 Gain 进行比较，即得到了整体的最优划分，结合上方就完成了算法。

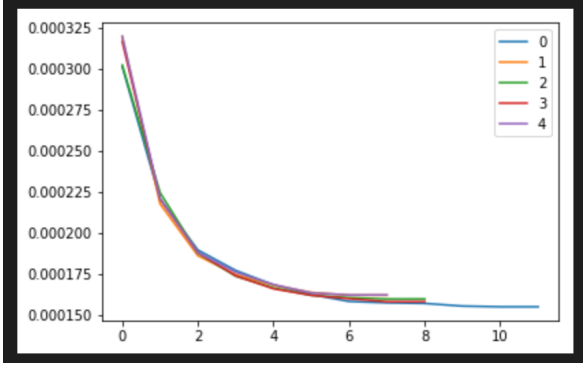
展示/分析：

*默认参数[经过寻找后比较好的参数效果]:
lam = 150, gamma = 1e-7, tol = 1e-7
gain_tol = 1e-7, max_depth = 10, min_num = 20
[默认 max_tree = 20，但以下的讨论中没有出现达到最大数量的情况]
划分比例 7:3

1、输出结果示例

```
time: 8.124472  RMSE: 0.000182  R^2: 0.773633
time: 6.114985  RMSE: 0.000184  R^2: 0.770567
time: 6.207348  RMSE: 0.000183  R^2: 0.777023
time: 6.570523  RMSE: 0.000185  R^2: 0.772318
time: 6.220283  RMSE: 0.000183  R^2: 0.780014

<matplotlib.legend.Legend at 0x24cce3ff0b8>
```



这是用默认参数进行五次训练、测试的结果，可以发现平均在 7 秒以内就已经达到了 0.00018 的均方误差与 0.77 的R^2，足见 XGBoost 结合回归树算法的效果之好。

损失曲线绘制的是每训练出一棵树后在训练集上的均方误差，通过横轴可以看出每次训练得到的回归树个数。

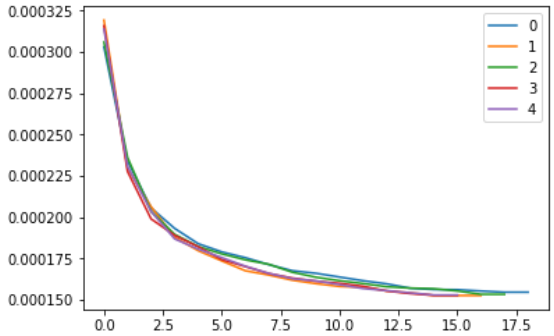
2、停止策略对比

具体的过程见 ipynb 文件，这里给出停止策略的结果[测试五次]：

策略	中位时间	中位 RMSE	中位R^2
增益小于 1e-6	3.24s	1.94e-4	0.729
增益小于 1e-7	7.31s	1.84e-4	0.762
增益小于 1e-8	9.61s	1.85e-4	0.771
深度高于 5	14.23s	1.83e-4	0.775
深度高于 7	14.19s	1.82e-4	0.784
深度高于 10	18.16s	1.86e-4	0.762
集合低于 200	18.89s	1.86e-4	0.771
集合低于 300	17.77s	1.84e-4	0.773
集合低于 500	21.79s	1.81e-4	0.786

*集合低于若干的测试额外添加了深度高于 10 的终止条件，否则会导致划分过多

从结果可以发现，总体来说深度结合集合低于值的策略性能最稳定，也相对最好。值得注意的是，单棵树结束划分快并不代表总时间短，因为可能会导致划分的树更多。表现最好的集合低于 500+深度高于 10 的终止策略，平均划分了 16 棵树，损失函数如下：



可以分析，这种情况下收敛慢意味着某种“**好而不同**”，因此在集成学习中最后表现出了更好的效果。不过，它的时间开销也显著大。

3、参数对比

下面在默认停止策略的情况下对比参数：

策略	中位时间	中位 RMSE	中位R^2
lambda = 50	5.64s	1.90e-4	0.755
lambda = 100	6.31s	1.84e-4	0.777
lambda = 200	7.94s	1.80e-4	0.774
默认参数	6.46s	1.90e-4	0.751
gamma = 1e-5	1.75s	2.32e-4	0.564
gamma = 1e-6	3.75s	1.94e-4	0.738
gamma = 0	10.13s	1.85e-4	0.766

同样可以发现，时间开销较大的时候意味着倾向于训练出的树更多才收敛，结果一般更好。此外，由于总树基本不会超过 20，过拟合可能性不大，不加入正则化项或许效果更好。

总结：

这次实验的任务难度主要在于两部拆分的构造，尤其是回归树部分如何优化。此外，在梯度提升每次优化误差的思想改进下，XGBoost 能得到更加快且精确的结果。根据最后的对比部分，个体学习器少进行一些划分、增加学习器的数量会对集成产生较明显的正面效果。

Lab 4 实验报告

PB20000296 郑滕飞

目的/原理：

1、实验目的

用 DPC 算法实现聚类，达到结合 k-means 与 DBSCAN 优点且计算复杂度较低的聚类效果。本文中，之后的复杂度分析都假设总点数为 n，聚类簇数为 m。

2、实验原理

此聚类算法基于一个假设：簇中心的邻域内的点的局部密度低于中心，且中心到比其局部密度更大的点的距离会显著更远。由此，确定中心只需要寻找局部密度较大且到局部密度更大的点显著远的点[定义比某点局部密度更大的点中与其最近的点为“高密度近邻”，定义此距离为“高密度距离”](若没有比其局部密度更大的点，将它与其他点的最远距离作为高密度距离)。

[值得注意的是,此假设确定了簇中心的性质,但并不代表符合假设的点全部为簇中心,之后实验步骤处会进一步解释。]

实验中采用 dc 为半径的邻域内点的个数作为局部密度的定义，并通过设置局部密度阈值与高密度距离阈值来控制怎样的点是可以选作簇中心的。此外，若一个点的高密度距离很高，但局部密度不大，则这个点有较大可能是孤立的，或者看作一点成为一簇。

在确定簇中心后,进一步决定每簇中的点时,要求每个点与其高密度近邻所在的簇一致。由于每次取高密度近邻若不结束，一定可以找到密度最大的点，而其高密度距离是与其他点的最远距离，必然显著大，因此必然最后能找到某个簇中心。

实验步骤：

1、整体框架

这次的框架分为四个部分：绘制、评估、聚类与测试，前三个部分都是定义了一个函数。由于这个聚类算法并不存在内部状态,而是在输入数据后直接判断,我选择用函数直接实现,不需要利用类。

绘制部分,输入每点第一个坐标、第二个坐标与标签组成的数组,绘制出对应的散点图。评估部分，由于我自己的 sklearn 包并没有包含 DBI 计算，更新出现了一些问题，自己构建了计算 DBI 的函数。聚类部分即为输入数据后计算聚类并画图、计算 DBI。最后的测试部分，输入数据并进行测试。

2、绘制散点图与 DBI 计算

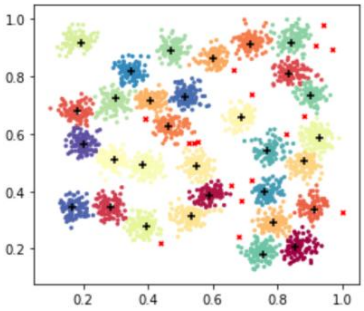
在我的算法中，簇的标签是簇中心元素的下标，也即标签等于下标的点就是簇中心，而孤立点的标签统一为-1。

画图时，将这三类点分别画图：

```
now.set_title(name)
now.scatter(x1[normal],x2[normal],c=y[normal],cmap=plt.cm.Spectral, s=5)
now.scatter(x1[centers], x2[centers], c="black", marker="+")
now.scatter(x1[out], x2[out], c="red", marker="x", s=12)
```

normal 代表不是中心也不是孤立点的点，根据 matplotlib 提供的 Spectral 颜色映

射绘制；centers 代表中心，绘制为黑色的加号；out 代表孤立点，绘制为红色的叉号，示例效果如下：



DBI 直接按照公式计算，其中的范数为二范数。步骤为：区分中心、构建每类分开的二维数组 X、计算 S、计算 M、计算 R。分类过程如下：

```
a = []
for i in centers:
    a.append([i])
for i in range(y.size):
    if y[i] >= 0 and y[i] != i:
        for t in range(classes):
            if y[i] == centers[t]:
                a[t].append(i)
```

由于 a 每行的第一个位置就是对应类的中心，后续找中心时直接找 a[i][0]即可。

*DBI 计算忽略了被选为孤立点的点，当选取孤立点过多时，小 DBI 未必优。

3、聚类算法

首先，由于数据尺度的差别，先对数据进行归一化。又因为后面大量涉及距离的比较，先计算出任意两点间的距离矩阵 d[i][j]，方便之后直接使用。

```
# 预计算距离方阵
d = np.zeros([N, N])
for i in range(N):
    for j in range(N):
        if (dis == "l2"):
            d[i][j] = (X[i][0] - X[j][0])**2 + (X[i][1]-X[j][1])**2
        elif (dis == "l1"):
            d[i][j] = abs(X[i][0] - X[j][0]) + abs(X[i][1] - X[j][1])
        else:
            d[i][j] = max(abs(X[i][0] - X[j][0]), abs(X[i][1] - X[j][1]))
```

这里采取了不同的距离函数，对应不同的区域形状(由于只涉及大小比较，l2 范数没有必要取根号)，在之后的参数比较中会进行比较。

之后，按照算法要求，计算 rho 与 delta。为了之后寻找高密度近邻方便，这里额外记录了 delta 取到时的 i(mr 为 rho 的最大值，已预先计算出)，保存在 argd[i]中：

```
if (rho[i] == mr):
    delta[i] = np.max(d[i])
else:
    for j in range(N):
        if (rho[j] > rho[i]):
            if (d[i][j] < delta[i]):
                delta[i] = d[i][j]
                argd[i] = j
```

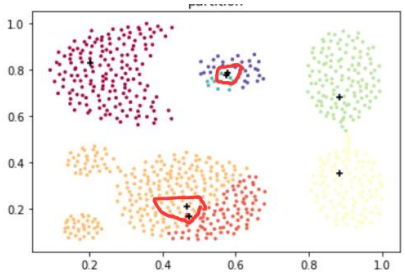
argd 的初值均为-1，因此 rho 为最大值得点的高密度近邻记为-1.

在分析中心时，严格按照算法的中心选取是这样的(这里也同时分出了孤立点)：

```
if delta[i] > dth:
    if rho[i] > rth:
        centers.append(i)
        y[i] = i
    else:
        y[i] = -1
```

其中 dth 与 rth 为提前选定的 r 与 d 的阈值。

但是，直接以其作为中心可能会出现如下的情况：



红圈部分出现了过于密集的中心。根据之前的算法过程，这是由于“高密度近邻”需要的是严格大于而非大于等于，于是可能有若干密度相等且接近的点都被选为了中心。事实上，这些“备选中心”中选择任何一个都可以。

于是，在上方的中心算法之后加了一段：

```
for t in centers:
    for i in range(N):
        if d[i][t] < dc:
            y[i] = y[t]
```

在加上这个处理之后，如果先后选出的中心很接近，后被选出的中心所在的簇会被先选出的覆盖掉，从而不会被视为两个簇。不仅如此，将中心的近邻与中心分为同簇还可以减少之后确定所在的簇的搜索次数。

这样从备选中心得到最终中心后，就可以确定簇了，方法如下：

```
for i in range(N):
    if y[i] == -3:
        temp = []
        j = i
        while y[j] == -3:
            temp.append(j)
            j = argd[j]
        for t in temp:
            y[t] = y[j]
```

y[i]的初值是-3，也以此表示还没有确定的点。在找高密度近邻的搜索过程中，会出现一条链 temp，将链上的所有元素进行赋值即得到了整个结果。

算法复杂度为：预处理、计算 rho 与 delta 均为 n*n，寻找中心 n，中心周围处理 m*n，确定簇 n(每个点最多被访问三次)，因此总复杂度是n^2量级。

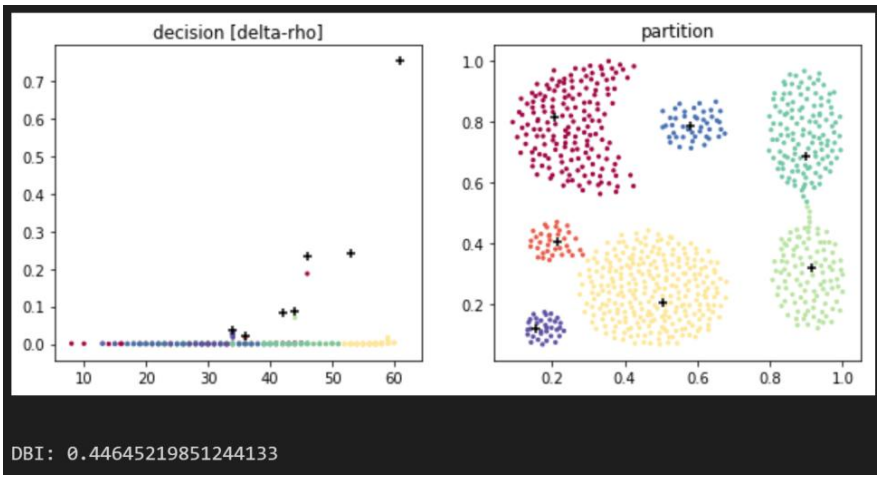
由于过程中需要计算的距离包括小的与大的，哪怕进行预排序，需要提前计算的距离也不能减少，因此复杂度量级改进的可能性不大。最后，绘制两个图，并返回计算的 DBI：

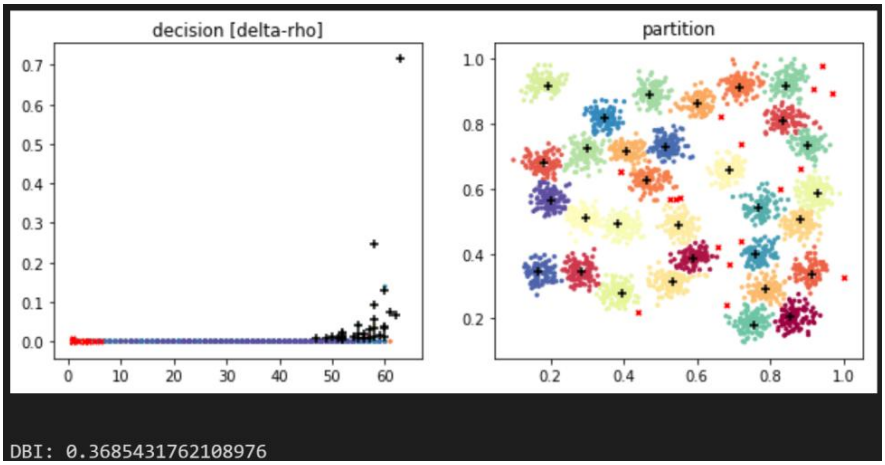
```
fig = plt.figure(figsize=(10,4))
draw_result(rho, delta, y, "decision [delta-rho]", fig, 121)
draw_result(X.T[0], X.T[1], y, "partition", fig, 122)
plt.show()
return DBI(X, y)
```

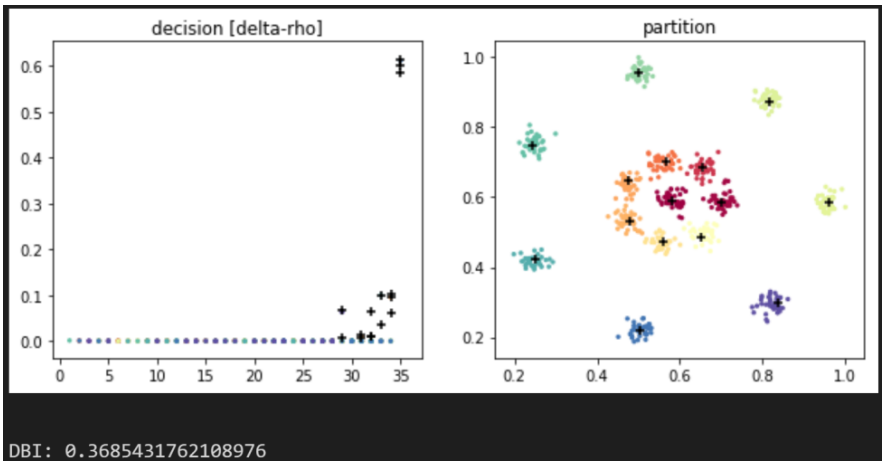
展示/分析：

1、测试样例输出

以下为三个样例在选取到合适参数后的输出结果：



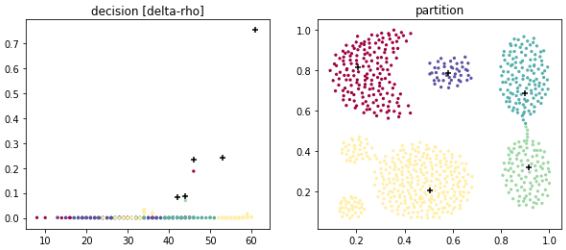


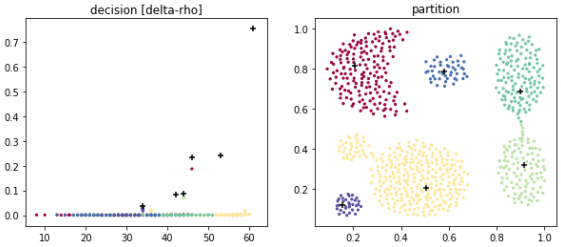


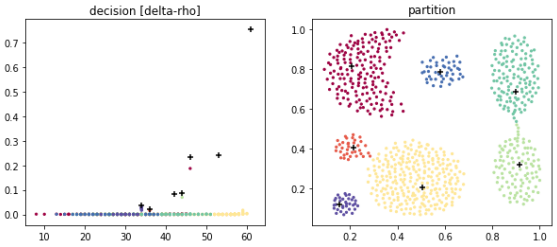
第一张图中，很明显发现有 rho 相同的点最后只选取了一个为中心，这也是对中心进行进一步处理的效果。

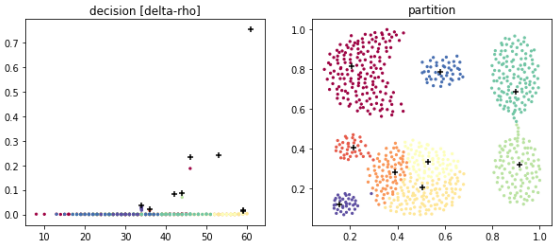
2、阈值相关对比

delta 的阈值对结果的影响是明显的，因为这直接关系到分出的簇的数量，以下四张图为 delta 逐渐减少时的结果：

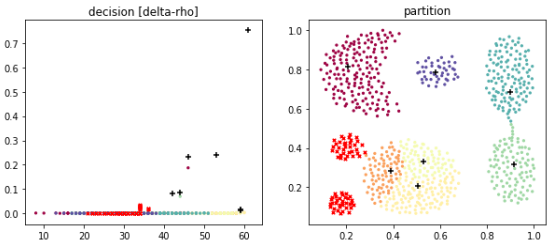








rho 的阈值则是主要影响将多大的簇视为孤立，例如上一张图中将 rho 的阈值调大后，会出现如下情况：



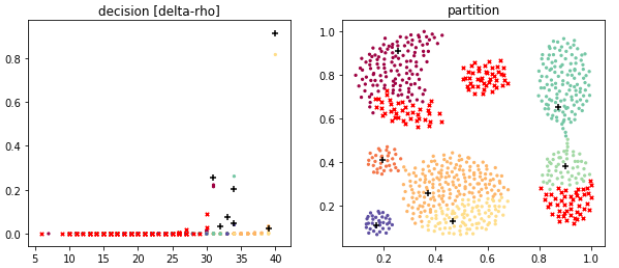
左边的两个小簇现在不再被视为簇，而是看作孤立点，这是由于两个对应的中心被舍弃了。

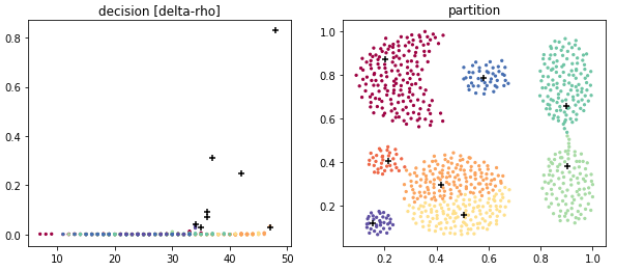
在上方的例子中，第三张图，也即聚类与目测较为一致时，对应的 DBI 最小。

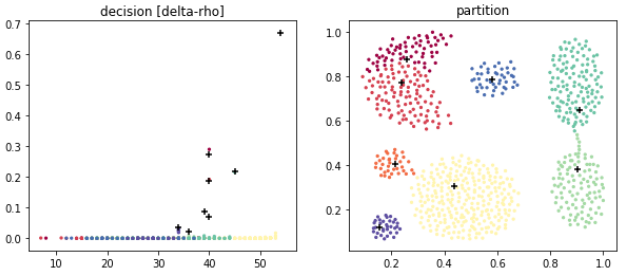
另一个改变 d 阈值的例子见代码文件后方的“参数比较”部分。

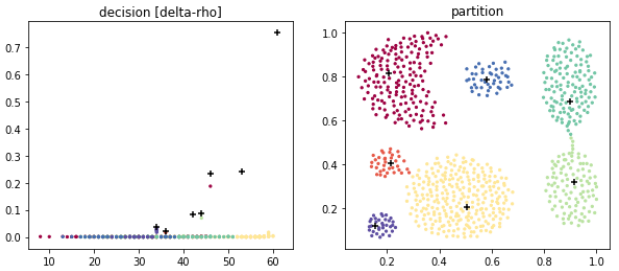
3、距离相关对比

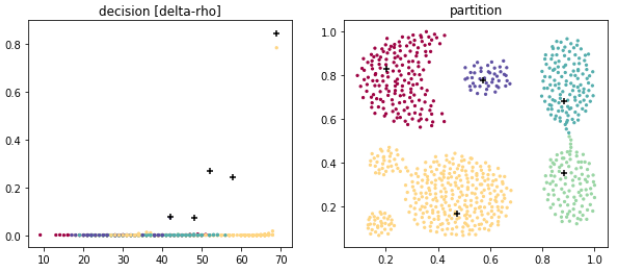
距离 dc 的影响比较微妙。假设两个阈值不变，在 dc 越小时划分得越精细(这是由于 dc 的意义事实上是容许的簇半径)[第一张图中由于 rho 的阈值设置，有些簇被视为孤立]:





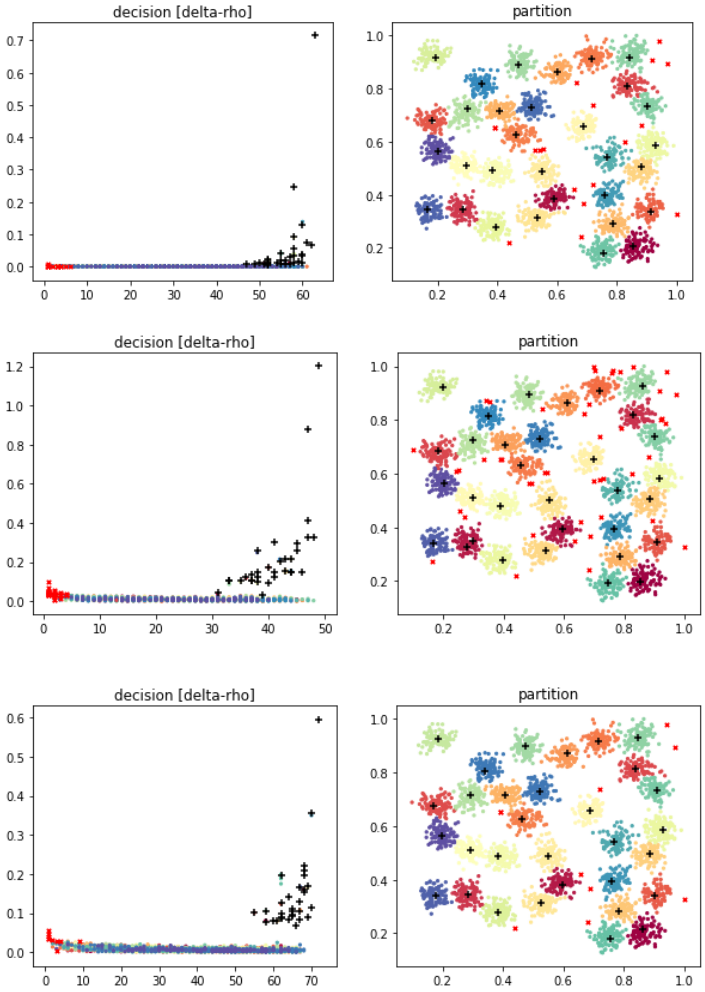






而范数的选取则是关乎区域的形状。值得注意的是，为了对比不同的参数，在 1 范数与

无穷范数时需要给阈值和 dc 作平方根，这是由于之前计算二范数为了节省时间没有进行平方，对比结果如下：



从上到下依次为二范数、一范数、无穷范数的结果，DBI 最小的事实上是无穷范数。

不同范数的选取会导致区域形状的差别，二范数为圆，一范数为竖着的正方形，而无穷范数是横着的正方形。当实际区域形状与对应的情况越接近时，实际效果越好，而之外的点会被视为孤立点。

更多对比见代码文件后方的“参数比较”部分。

总结：

这次实验的任务难度主要在于，不能尽信论文。在自己进行了一些处理之后，可以获得比直接按论文操作更好的结果。除了文中涉及的部分外，局部密度、高密度近邻的选取算法也都可以进行一些处理(如将直接设置阈值变为加权软阈值)，或许能获得更好的结果。

Lab 5 实验报告

PB20000296 郑滕飞

目的/原理：

1、实验目的

用不同机器学习算法对指定数据集进行四分类预测，并对比各种预处理方式与机器学习算法的效果。

2、实验原理

本次实验涉及的预处理方式包括对空数据与异常数据的处理、对单个特征基于一定评判标准(基尼系数、信息熵、方差)的筛选、主成分分析。而具体对比的学习器有普通逻辑斯蒂回归、决策树、神经网络、XGBoost 与 SVM 五种。除了神经网络之外，其余学习器都是在之前实验中实现过的(其中决策树与 XGBoost 共同形成回归树，由于与本次多分类任务不同，采用调包实现)，而调用的自己实验的代码包括 Logistic.py[内含逻辑斯蒂回归二分类器]与 SVM.py[内含 SMO 算法实现的 SVM]。

3、文件目录

main.ipynb 主要文件，包含对比测试与最终结果输出
main_drop.ipynb 另一种处理方式(未被选取)，也包含对比测试与最终结果
test.py 包含测试用的部分函数
test_pre.ipynb 对比各种预处理方式
test_choice.ipynb 对比五种学习器的参数选择
Logistic.py 自己实现的逻辑斯蒂回归[来自以往实验]
SVM.py 自己实现的支持向量机[来自以往实验]
test_label.csv 最终预测的标签
report.pdf 实验报告

预处理[test_pre.ipynb]：

1、空值与异常处理

读取文件后，为了保证能正确存入 numpy 的向量中，需要去除空值。考虑到越界值的存在，这里定义了直接去掉空值所在行的读取方式 readfile_drop 与用本列中位数填充空值的读取方式 readfile。

此外，直接观察数据可发现，数据中有明显的异常值，因此构造了函数 drop_err，将绝对值在均值绝对值三百倍以上的点删除。

记读取并填充空值的数据集为 X，删除空值的为 Xd，填充空值后删除异常为 Xr，删除空值与异常的为 Xrd，也将它们的 y 对应命名，之后的对比将在这些之中进行。

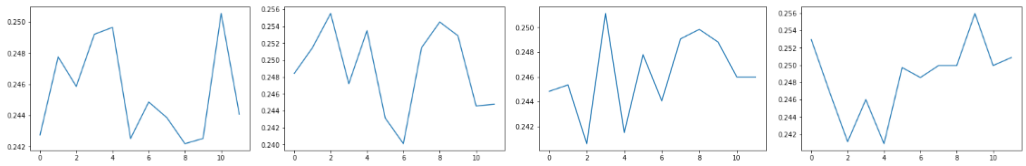
2、学习器选择

为了对比预处理的效果，需要构造一个统一的学习器。由于数据中有异常数据与空值，直接将所有数据的所有特征进行学习的学习器应无法达到测试效果，于是回归与神经网络等模型可以排除。但是，决策树与基于决策树的 XGBoost 又以关心标签的顺序关系为主，无法刻画具体大小幅度造成的影响，因此，最终选择利用 sklearn 中默认参数的 SVC 作为用

于比较的标准学习器[这里为了计算速度，调包实现了，最后比较结果时利用的是自己实现的]。以下计算的所有准确率都是三次随机 7:3 划分数据-测试的平均准确率。

2、特征选取-信息熵

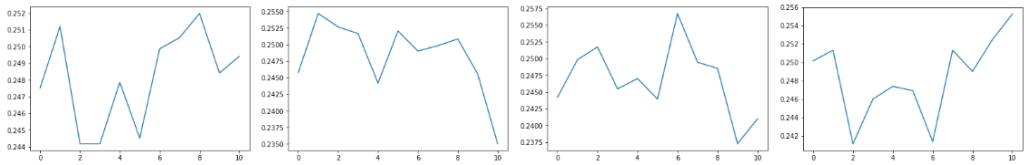
根据一些预先的测试，不同特征之间的差别很大，且由于异常数据无法完全剔除，利用方差进行预选取是完全无效的。因此，预处理的第一部分中，实现了基于比较信息熵的特征选取(即删掉信息熵相对最低的若干特征)，并在 X、Xd、Xr 与 Xrd 上测试，根据删除特征的数量为横轴，准确率为纵轴进行绘图，依次为[详细图片见 ipynb 文件中]:



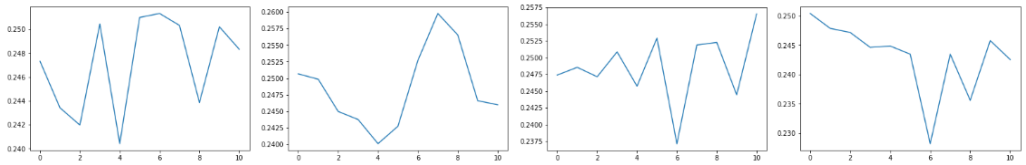
信息熵初筛一般筛选掉 20-30 个特征，对应横轴 2-3 的位置，而在这个段落内，可以发现单特征选择的表现几乎都不理想(只有 Xd 超过了 0.25，且根据进一步测试可知较为偶然)，于是基本排除了单特征的选取方式。对基尼系数选取，结果是类似的。

3、特征选取-RFE

单特征既然选取失败，就对综合多个特征的选取方式，这里使用了代表性的递归特征删除法。递归特征删除法的原理是，先用学习器进行学习，再将学习器学习中占到较低重要性的特征删除，重复直到剩下的特征数量符合要求。这里首先尝试了与单特征较为接近的决策树为基学习器的选取:



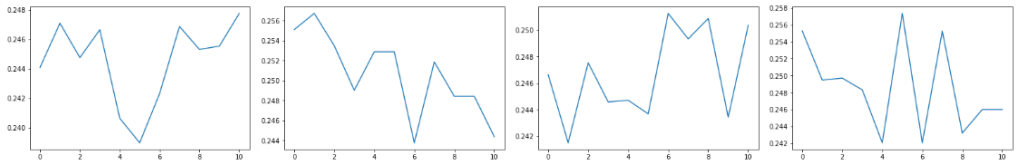
总体来看，它的效果要比信息熵进行特征选取要好一些，而且在删除特征不是过多时相对稳定。之后又实验了以逻辑斯蒂回归为基学习器的选取:



直接回归的结果并不如决策树进行特征选取，这是由于异常数据未经处理，容易影响回归结果。根据之后的进一步实验，先用决策树进行 RFE，再用回归方法进行 RFE 可以在压缩时损失信息较少。

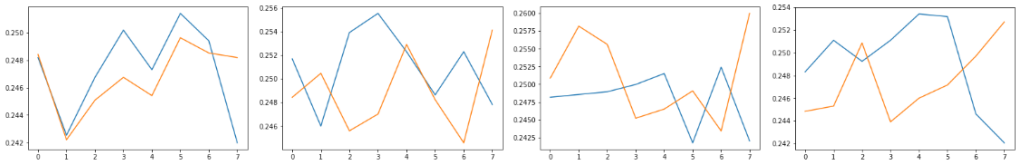
3、主成分分析

虽然事先已有主成分分析会导致无效特征信息过多的猜测，但还是进行了测试:

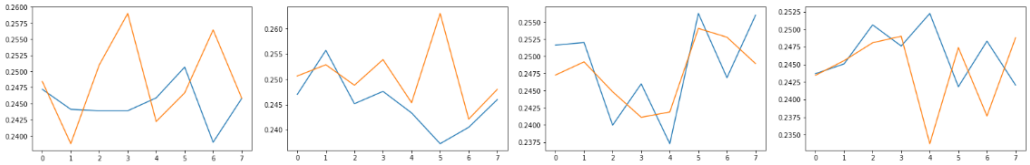


从结果来看，提取出的主成分越少，效果反而越差(X 和 Xd 始终保持在 0.25 以下，而 Xr 和 Xrd 总体下降)。

后来还测试了在有 RFE 预处理时的 PCA[蓝线为保留较多，黄线为保留较少，下同]:



和有回归预处理时的 PCA:



总体来看，都没有对结果起到多少作用。因此，最终确定的选择方式是先通过决策树作 RFE，将特征压缩到 85 个，再通过回归，进一步压缩到 60 个。

参数选择[test_choice.ipynb]:

1、读文件与预处理

最终选择的预处理如上文。根据进一步的实验与分析，将某特征在均值 300 倍以上的数据认为异常可能会损失有效数据，因此最终通过丢弃比例的确定将数字变为了均值的 500 倍。

此外，在对比参数时，为了做到充分学习、对比，采用了填充空特征的方式，也就是说与上文 Xd 的生成基本相同。

2、学习器的集成方式

由于之前实现的 LR 与 SVM 都是二分类学习器，这里使用了 ECOC 的方式将其进行集成，以 LR 为例，代码如下:

```
def lr_mul(X, y, p, X_t, it, lr):
    r = LogisticRegression(gamma = 1e-4)
    y1 = y
    for j in range(y1.size):
        if y1[j] in p: y1[j] = 1
        else: y1[j] = 0
    r.fit(X, y1, max_iter=it, lr=lr)
    return r.predict(X_t)

def test_lr(X, y, X_t, it, lr):
    yp1 = lr_mul(X, y, [0,1], X_t, it, lr)
    yp2 = lr_mul(X, y, [0,2], X_t, it, lr)
    yp3 = lr_mul(X, y, [0,3], X_t, it, lr)
    yp4 = lr_mul(X, y, [0], X_t, it, lr)
    yp5 = lr_mul(X, y, [1], X_t, it, lr)
```

```
    yp6 = lr_mul(X, y, [2], X_t, it, lr)
    yp7 = lr_mul(X, y, [3], X_t, it, lr)
    yp = np.zeros(yp1.size, dtype="int")
    x = np.zeros(4, dtype="int")
    for i in range(yp.size):
        now = np.array([yp1[i],yp2[i],yp3[i],yp4[i],yp5[i],yp6[i],yp7[i]])
        x[0] = np.sum(np.array([1,1,1,1,0,0,0]) == now)
        x[1] = np.sum(np.array([1,0,0,0,1,0,0]) == now)
        x[2] = np.sum(np.array([0,1,0,0,0,1,0]) == now)
        x[3] = np.sum(np.array([0,0,1,0,0,0,1]) == now)
        a = np.concatenate(np.argwhere(x == np.max(x)))
        yp[i] = np.random.choice(a)
    return yp
```

由于是二分类变为四分类，总共有 7 种可能的有效分割方式，于是训练 7 个相同参数的学习器，并根据每个学习器的结果进行集成，选择海明距离最小的作为最终结果。若有相同最小值，则在其中随机选择一个(这里比较了相同的元素个数，因此取最大值)。

3、核心参数的选取

为了方便对比与减少时间复杂度，对于五种学习器，通过一些实验与调查找到了各自的两个本实验中的核心参数，并对它们进行比较。

对逻辑斯蒂回归，由于无需考虑稀疏性，采用 L2 范数。此外，简单对比后选取了 1e-4 的正则化项，而其核心参数在于迭代次数与学习率。

对决策树，最重要的是最高层数与每个分支最小的数据量。

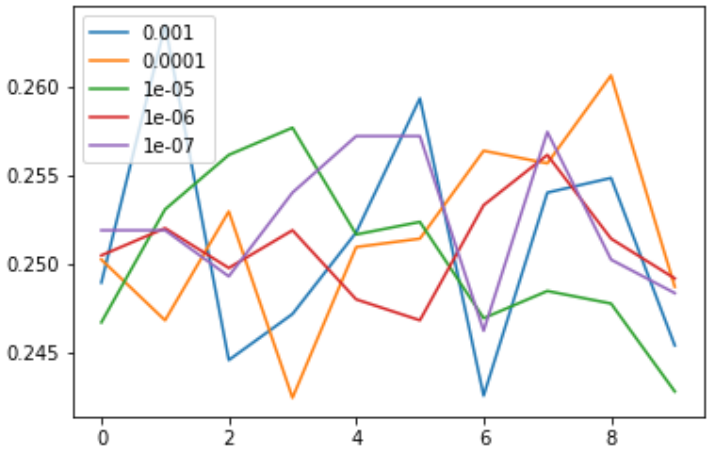
对神经网络，最重要的是神经元层数与迭代次数。

对 XGBoost，最重要的是基学习器个数与作为基学习器的决策树的最高层数。

对 SVM，效率起见，这里 SMO 算法中优化的目标是在可优化中随机选择的，最重要的参数是迭代次数与权重 C。

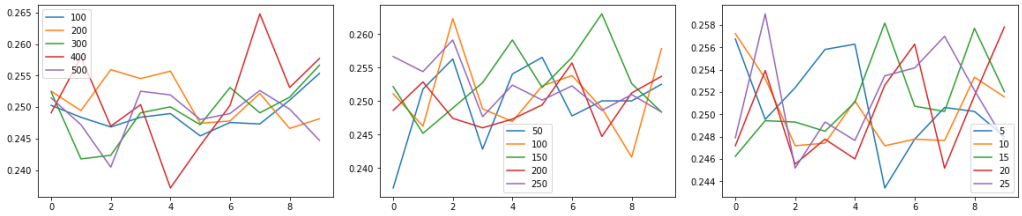
4、参数选取结果

逻辑斯蒂回归的结果如下，图例为学习率，横轴与迭代次数相关：



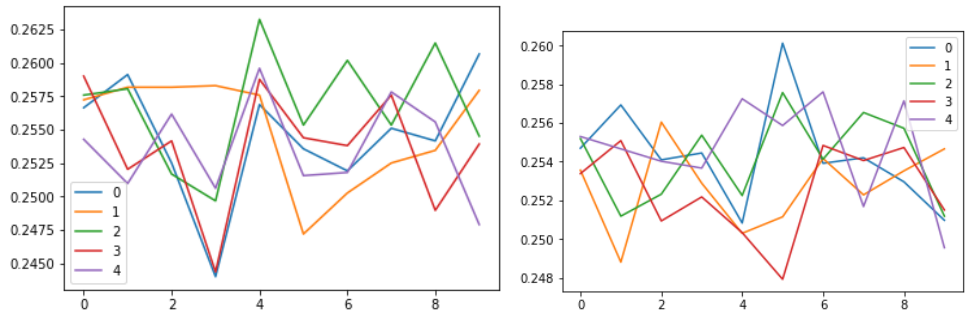
这里学习率 1e-7 时保证了稳定良好的结果，迭代次数取为 50，对应横坐标 4。

类似逻辑斯蒂回归，决策树、神经网络、XGBoost 参数结果如下：



决策树每个分支最小 200 时相对稳定，取最高层数 16；神经网络迭代 150 次时稳定较好，取层数 80；XGB 最大深度 15 时相对较好，取基学习器个数 60。

最后是自己实现的 SVM：



这里两张图对应颜色的线对应横坐标的参数是完全相同的，只是第一张图每个位置三次随机测试，第二张图 10 次，这是因为在下一部分中初步确定 SVM 最优后进行了进一步的参数调整。根据第一张图初步调整的结果，选择了 1.5 作为 C 的取值(即绿线)，并进行 50 次迭代。第二张图进行进一步判别后，仍取值 1.5，但迭代 60 次。

对比与输出[main.ipynb/main_drop.ipynb]：

1、test.py 构建

此 py 文件包含对每个学习器进行测试的函数，输入训练集的 X、y 与测试集的 X，返回预测的 y。

之前展示的代码就是对自己集成的学习器的测试，而对调包的则类似(这里的参数已经在上一部分确定)：

```
def test_xgb(X, y, X_t):
    c = XGBClassifier(n_estimators=60, max_depth=15)
    c.fit(X, y)
    return c.predict(X_t)
```

2、main 与 main_drop

这里的特征选择与上一部分中过程相同。根据之前预处理的结果，将空值直接删去是比填充总体效果更好的。但是，考虑到最终的 test_feature 中也有空值需要填充，无法确定究竟那种更好，因此两种情况下都做了对比，相当于之前的 Xr 与 Xrd。

3、学习器对比

此处对比与画图的代码如下：

```
def test(X0, y0, n):
```

```
    acc = np.zeros([5, n])
    dif = np.zeros([10, n])
    for j in range(n):
        X, X_t, y, y_t = partition(X0, y0)
        res = np.zeros([5, y_t.size], dtype="int")
        res[1] = tt.test_tree(X, y, X_t)
        res[2] = tt.test_mlp(X, y, X_t)
        res[3] = tt.test_xgb(X, y, X_t)
        res[4] = tt.test_svm(X, y, X_t)
        res[0] = tt.test_lr(X, y, X_t)
        for i in range(5):
            acc[i][j] = score(res[i], y_t)
        count = 0
        for i in range(4):
            for k in range(i+1, 5):
                dif[count][j] = 1 - score(res[i], res[k])
                count += 1
    return acc, dif

def draw_result(X0, y0, n):
    acc, dif = test(X0, y0, n)
    plt.plot(acc[0], label="lr")
    plt.plot(acc[1], label="tree")
    plt.plot(acc[2], label="mlp")
    plt.plot(acc[3], label="xgb")
    plt.plot(acc[4], label="svm")
    plt.legend()
    print("average acc:")
    print(np.average(acc, axis=1))
    dif_a = np.average(dif, axis=1)
    print("lr's dif with other 4:")
    print(dif_a[0:4])
    print("tree's dif with other 3:")
    print(dif_a[4:7])
    print("mlp's dif with other 2:")
    print(dif_a[7:9])
    print("xgb's dif with svm:")
    print(dif_a[9:10])

draw_result(X0, y0, 50)
```

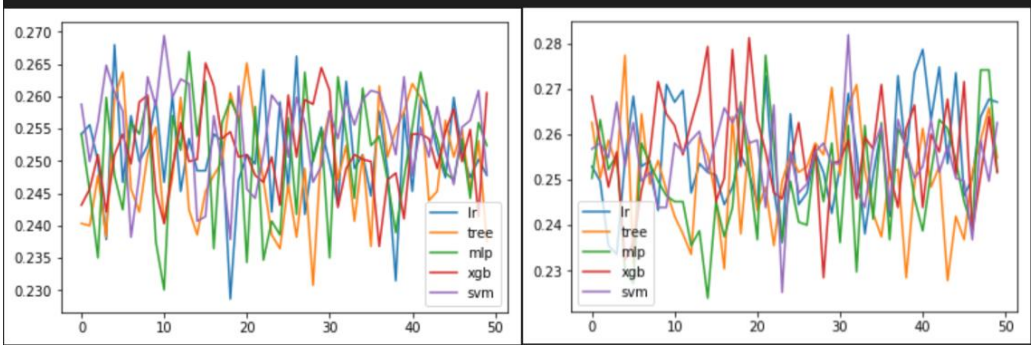
这里 X0 与 y0 是之前得到的训练集与测试集。在确定的参数下，进行 n 次测试，并计算每次每个学习器的准确率与不同学习器预测的结果相差，最终将准确率画图并输出平均准确率与差距。

3、各自结果

左侧为 main 中的结果，右侧为 main_drop 中的结果，排列顺序为回归、决策树、神经网络、XGBoost、SVM:

```
average acc:
[0.25110245 0.24899681 0.25048564 0.25142857 0.25453385]
lr's dif with other 4:
[0.739447   0.7467352  0.74238922 0.68472173]
tree's dif with other 3:
[0.74100674 0.70826657 0.73291032]
mlp's dif with other 2:
[0.73451258 0.73495923]
xgb's dif with svm:
[0.73929103]
```

```
average acc:
[0.25546976 0.25072072 0.24934363 0.25633205 0.25487773]
lr's dif with other 4:
[0.72671815 0.74487773 0.74217503 0.6748906 ]
tree's dif with other 3:
[0.73921493 0.69723295 0.73745174]
mlp's dif with other 2:
[0.72788932 0.74660232]
xgb's dif with svm:
[0.73662806]
```



从准确率上来看，填充空值时 SVM 最优，且比其他显著好；删除空值时 XGBoost、回归、SVM 排前三，且都高于了填充时的最优情况。这样的结果是正常的，因为删去空值也删去了对应的 y，对有效数据的预测会更准确。

从接近程度上，回归与 SVM 始终是最相似的，而回归与神经网络则一直能在相差最大榜上排前二(在删除空值时，神经网络与 SVM 的相差是最大的)。

无论哪种情况，普通决策树和神经网络都是效果最差的。前者未经集成时本身较弱，而后者过于不可控，并不适合这种相对简单的样例。

4、最终选择

将 SVM 进行进一步调参后，我们在 main 与 main_drop 中分别针对 SVM 与 XGBoost 进行最后测试：将各自的训练集随机划分 100 次，计算预测的平均准确率与最大准确率，用最大准确率的一次预测测试集，作为预测结果。两者表现出的平均准确率与最大准确率如下:

```
SVM:
max acc: 0.27437079049982277
average: 0.25501240694789096

XGBoost:
max acc: 0.2812097812097812
average: 0.25332689832689825
```

这个结果里，虽然 XGBoost 表现出的最大准确率更高，但平均准确率甚至还不如填充空值的 SVM。于是，最终选择在填充空值的情况下取 SVM 对测试集的最大准确率时预测结果为最终结果，因为虽然最大值看似比起来更低，但它经过了更多数据的测试，也更加稳定。

最后将其输出到文件:

```
with open("test_label.csv", "w") as fi:
```

```
    print("label", file=fi)
    for i in predict: print(i, file=fi)
```

作为最终结果。

总结：

说实话，最后做出来的结果也还是挺差，主要来源于，并没有得到任何真正显著高于 25% 正确率(大概需要至少 30%)的学习器，因此也无法以更好的信息进行优化。整个实验中预处理与参数的选择虽然用了很多时间(ipynb 文件只展现了部分)，但大部分还是和盲测无区别。

不过没想到的是，这种情况下，线性 SVM 竟然表现出了相对良好与稳定的性能，也算是有了一个基本的思考方向(但离真正有效的结果还是很远)。

[总之 ML 实验完结，是时候新年快乐了]

后续：

虽然并未得到官方确认，但根据同学找到原始数据集并分析后发现，数据集打乱过程中出现了标签和样本分别被打乱的情况，导致结果相当于随机。