人工智能基础 Lab2 实验报告

- 因为个人觉得自己对 SVM 和 Bayes 的原理一知半解，所以手动实现了这两个分类器。而对于 MLP 用得学得比较多，就直接掉包了。
- 因为本次实验不要求调参（数据集划分、特征舍取等），所以主要的时间花在了理解和实现分类器上，没有花太多的时间在分析结果上。本报告只在以下两处进行了结果分析
  - SVM_valabel
  - Bayes_id
- 代码解释
  - data_process.py 是数据划分代码
  - Bayes.py，SVM.py 是自己实现贝叶斯、SVM分类器
  - Bayes_id/emotion.py， SVM_valabel.py， MLP_id/emotion/valabel.py， DT_id 是具体调用分类器模块进行训练和评估的模块
  - 评估结果存储在每个数据集对应的文件夹下，比如 Bayes_id.py 在 DEAP 数据集上的预测，存储在 data/DEAP/NB/目录下

本实验报告较长，请配合 PDF 或 Markdown 左侧大纲栏食用!

实验环境

- python 3.6.5
- numpy 1.14.3
- keras 2.2.0 (tensorflow 1.8.0)
- pydotplus，用于可视化决策树
- 因为代码 (classfier_id / emotion / valabel.py) 中使用了相对路径，所以请确保运行分类器的运行路径为该代码所在的目录

数据划分: data_process.py

大致策略

先把所有数据读取出来，然后整理分析数据，存储每个用户分别有多少数据。

核心代码

```
#整理数据
#testers_data[testers_id][record_index]
[video_id,feature,valabels,emotion_category]
for i in range(data_num):
    testers_id = subjects[i]-1
```

```
        testers_data_num[testers_id] += 1
        testers_data[testers_id].append([])
        index = len(testers_data[testers_id]) - 1
        testers_data[testers_id][index].append(videos[i])
        testers_data[testers_id][index].append(features[i])
        testers_data[testers_id][index].append(va_labels[i])
        testers_data[testers_id][index].append(emotions[i])

    #因为HCI数据有缺失，所以要确定每个用户每组有多少数据
    testers_average_num = np.array(testers_data_num/splits_num,dtype=int)
    for i in range(splits_num-1):
        for j in range(testers_num):
            splitted_data_num[i][j] = testers_average_num[j]
    i = splits_num-1
    for j in range(testers_num):
        splitted_data_num[i][j] = testers_data_num[j] -
    testers_average_num[j]*i

    #随机化
    for i in range(testers_num):
        random.shuffle(testers_data[i])
```

数据预处理

- 主要使用了归一化预处理，其他都是很简单的拼接和划分
- 拼接部分，除了标签，其他的数据维度都用上了，比如预测 id 时用到了 {subject_id, video_id, EEG_features, emotion_category}
- 在使用 MLP 的时候，需要将标签预处理成 One_Hot 独热码形式，这里使用 keras 后端函数

从文件获取数据

```
    #从文件获取数据
            subjects =
    np.loadtxt(root_dir+'subject/subject_video_'+str(i)+'.txt',dtype=int)
            features =
    np.loadtxt(root_dir+'feature/EEG_feature_'+str(i)+'.txt')
            va_labels =
    np.loadtxt(root_dir+'valabel/valence_arousal_label_'+str(i)+'.txt',dtype=i
    nt)
            if 'HCI' in root_dir:
                emotions =
    np.loadtxt(root_dir+'emotion/EEG_emotion_category_'+str(i)+'.txt',dtype=in
    t)
```

数据拼接

```
x_temp = []
        y1_temp = []
        y2_temp = []
        length = len(subjects)
        for j in range(length):
            x_temp.append([])
            x_temp[j].extend(subjects[j])
            if 'HCI' in root_dir:
                x_temp[j].append(emotions[j])
            x_temp[j].extend(features[j])
            y1_temp.append(va_labels[j][0])
            y2_temp.append(va_labels[j][1])
        x.append(x_temp)
        y1.append(y1_temp)
        y2.append(y2_temp)
```

归一化

```
    #归一化
    subjects = subjects /
np.array([np.max(subjects[:,0]),np.max(subjects[:,1])])
    features /= np.max(features)
    if 'HCI' in root_dir:
        emotions /= np.max(emotions)
    # va_labels = va_labels /
np.array([np.max(va_labels[:,0]),np.max(va_labels[:,1])])
```

交叉验证的数据划分

```
for i in range(splits_num):
        x_train = []
        y1_train = []
        y2_train = []
        x_test = []
        y1_test = []
        y2_test = []
        for j in range(splits_num):
            if j==i:
                x_test.extend(x[j])
                y1_test.extend(y1[j])
                y2_test.extend(y2[j])
            else:
                x_train.extend(x[j])
                y1_train.extend(y1[j])
                y2_train.extend(y2[j])
```

标签独热码化

```
from keras.utils import to_categorical
......
#具体是在 MLP_id 的 line 48
y_temp = to_categorical(y_temp)
```

SVM

1. 我对SMO算法的理解过程

1.1 软边界 SVM 的 KKT 条件

在 SVM 软间隔基本型的优化问题中，要满足的 KKT 条件为：

$$\begin{cases} \alpha_i \geq 0, \quad \mu_i \geq 0 \\ y_i f(x_i) - 1 + \xi_i \geq 0 \\ \alpha_i (y_i f(x_i) - 1 + \xi_i) = 0 \\ \xi_i \geq 0, u_i \xi_i = 0 \end{cases}$$

再结合求解出的约束 $0 \leq \alpha_i \leq C$ 以及 $C = \alpha_i + \mu_i$。

1.2 更新公式

具体推导过程参考 对SVM的推导和编码实践（二）SMO算法的推导

1.2.1 alpha 的更新

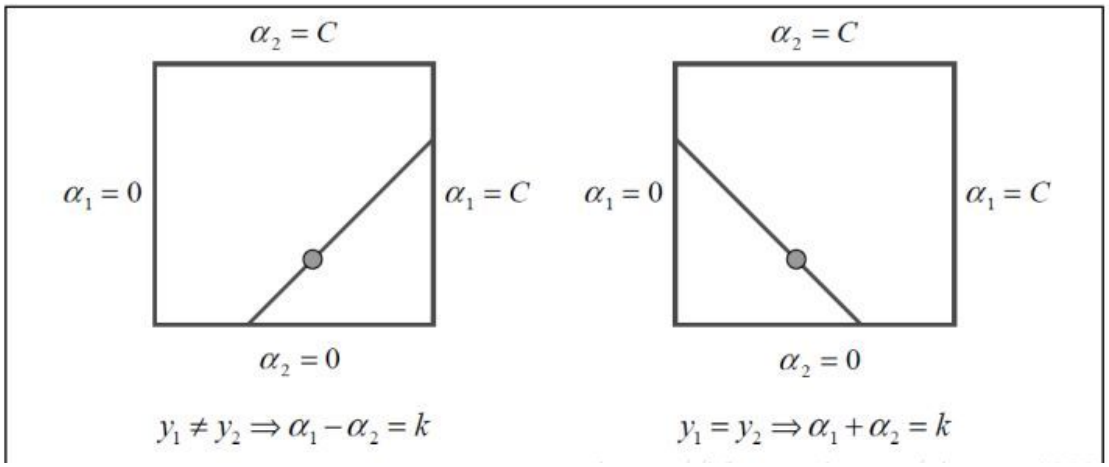
令$\eta = K_{11} - 2K_{12} + K_{22}$，则有：

$$\alpha_2^{new,ucp} = \frac{y_2(E_1 - E_2)}{\eta} + \alpha_2^{old}$$

$$\begin{aligned} \alpha_1^{new} &= \frac{\alpha_1^{old} y_1 + \alpha_2^{old} y_2 - \alpha_2^{new} y_2}{y_1} \\ &= \alpha_1^{old} - y_1 y_2 (\alpha_2^{new} - \alpha_2^{old}) \end{aligned}$$

1.2.2 alpha 的裁剪

SMO作者论文中给出了这样一张图：



基于上下界限定，我们需要对$\alpha_2^{new,ucp}$进行修剪：

$$\alpha_2^{new} = \begin{cases} L & if: \alpha_2^{new,ucp} < L \\ \alpha_2^{new,ucp} & if: L \leq \alpha_2^{new,ucp} \geq H \\ H & if: \alpha_2^{new,ucp} > H \end{cases}$$

1.2.3 b 的更新

综上：

$$b^{new} = \begin{cases} -E_1 + y_1 K_{11}(\alpha_1^{old} - \alpha_1^{new}) + y_2 K_{21}(\alpha_2^{old} - \alpha_2^{new}) + b^{old} & if: x_1, x_2 \text{ are both support vector} \\ \frac{b_1^{new} + b_2^{new}}{2} & others \end{cases}$$

2. 通用分类器实现: SVM.py

2.1 分类器接口介绍

2.1.1 类变量

```
class SVM_dual:
    def __init__(self,max_iter,ktup,C=1,toler=1e-3):
        self.C = C                          #Penalty parameter C of the error
term
        self.max_iter = max_iter            #max_iteration_times
        self.toler = toler                  #tolerance

        self.sample_num = 0                 #样本总数量
        self.dim = 0                        #输入维数
        self.alphas = None
        self.labels = None                  #标签集
        self.label_index = dict()
        self.label_back = dict()
        # 考虑到label可能缺失，如HCI数据集无8,9,10号emotion_category
        # 则label_index[11] = 7, label_back[7] = 11
```

```
        self.b = 0
        self.omega = None
        self.data = None                #训练数据
        self.label_array = None         #训练数据标签

        self.K = None                   #核函数转换矩阵
        self.ktup = ktup                #kernel type
        self.eCache = None              #误差缓存

        self.predict_y = None           #预测结果
        self.accuracy = 0               #准确率
```

2.1.2 fit 训练接口

```
 def fit(self,x,y):
```

2.1.3 score 评估接口

```
def score(self,x,y):
```

2.2 具体实现

训练过程完全遵循 PlattSMO 算法，所以注释比较简略

2.2.1 外循环

```
       #使用 PlattSMO 算法进行 SVM 分类器的训练
        while (iter < self.max_iter) and ((alphaPairsChanged > 0) or
(entireSet)):
            alphaPairsChanged = 0
            #先遍历所有样本
            if entireSet:   #go over all
                for i in range(self.sample_num):
                    alphaPairsChanged += self.inner_loop(i)
                iter += 1
            #再遍历非支持向量
            else:
                nonBoundIs = np.nonzero((self.alphas > 0) * (self.alphas <
self.C))[0]
                for i in nonBoundIs:
                    alphaPairsChanged += self.inner_loop(i)
                iter += 1
            if entireSet:
                entireSet = False #toggle entire set loop
            elif (alphaPairsChanged == 0):
```

```
                entireSet = True
            # print ("iteration number: %d" % iter )

        self.omega = np.dot(self.data.T,(self.alphas*self.label_array).T)
```

2.2.2 内循环

```
def inner_loop(self,i):
    Ei = self.calc_Ek(i)
    if ((self.label_array[i]*Ei < -self.toler) and (self.alphas[i] <
self.C)) \
        or ((self.label_array[i]*Ei > self.toler) and (self.alphas[i] >
0)):

        #启发式选择会让参数更新最大化的第二个 alpha
        j,Ej = self.selectJ(i,Ei)

        #准备更新
        alpha_i_old = self.alphas[i].copy()
        alpha_j_old = self.alphas[j].copy()

        #**********公式2.1.2*********
        #获取裁剪上下限
        if (self.label_array[i] != self.label_array[j]):
            L = max(0, self.alphas[j] - self.alphas[i])
            H = min(self.C, self.C + self.alphas[j] - self.alphas[i])
        else:
            L = max(0, self.alphas[j] + self.alphas[i] - self.C)
            H = min(self.C, self.alphas[j] + self.alphas[i])
        if L==H:
            return 0

        #**********公式2.1.1*********
        #即公示的的分母 η
        eta = 2.0 * self.K[i,j] - self.K[i,i] - self.K[j,j] #changed for
kernel
        if eta >= 0:
            return 0
        #更新 alpha_j
        self.alphas[j] -= self.label_array[j]*(Ei - Ej)/eta

        #**********公式2.1.2*********
        #裁剪，确保 0<alpha_j<C，因为要确保符合 KKT 条件
        self.clip_alpha(j,H,L)
        self.update_Ek(j) #added this for the Ecache
        if (abs(self.alphas[j] - alpha_j_old) < 0.00001):
            return 0
```

```
        #*********公式2.1.1*********
        #更新 alpha_i
        self.alphas[i] += self.label_array[j]*self.label_array[i]*
(alpha_j_old - self.alphas[j])
        self.update_Ek(i)

        #*********公式2.1.3*********
        #更新 b
        b1 = self.b - Ei- self.label_array[i] * (self.alphas[i] -
alpha_i_old) * self.K[i,i] \
                - self.label_array[j] * (self.alphas[j] - alpha_j_old) *
self.K[i,j]
        b2 = self.b - Ej- self.label_array[i] * (self.alphas[i] -
alpha_i_old) * self.K[i,j] \
                - self.label_array[j] * (self.alphas[j] - alpha_j_old) *
self.K[j,j]
        if (0 < self.alphas[i]) and (self.C > self.alphas[i]):
            self.b = b1
        elif (0 < self.alphas[j]) and (self.C > self.alphas[j]):
            self.b = b2
        else:
            self.b = (b1 + b2)/2.0
        return 1
    else:
        return 0
```

2.2.3 启发式选择第 2 个 alpha

```
#启发式选择第 2 个 alpha
def selectJ(self,i,Ei):
    maxK = -1
    maxDeltaE = 0
    Ej = 0
    self.eCache[i] = [1,Ei]
    validEcacheList = np.nonzero(self.eCache[:,0])[0]
    if (len(validEcacheList)) > 1:
        #遍历有效的误差缓存表，找到能让 |Ei-Ej| 最大化的 alpha_j
        for k in validEcacheList:
            if k == i:
                continue
            Ek = self.calc_Ek(k)
            deltaE = abs(Ei - Ek)
            if (deltaE > maxDeltaE):
                maxK = k
                maxDeltaE = deltaE
                Ej = Ek
```

```
            return maxK, Ej
        #无可选 alpha
        else:
            j = self.selectJ_rand(i)
            Ej = self.calc_Ek(j)
            return j, Ej
```

3. 具体分类任务

调用代码类似 **Naive Bayes** 模块的 ***2.1 代表代码：id 作为标签*** 小节，不再重复粘贴

3.1 分类结果

- 实验发现自己的实现的 SVM 训练过程十分缓慢，主要时间花费在 **核函数转换矩阵的初始化** 和 **内外循环的迭代**

3.1.1 SVM_valabel

3.1.1.1 归一化之前

- DEAP 数据集：
  - 5-fold：

    0.5781 0.3594 0.5508 0.4102 0.5820 0.5664 0.5469 0.3789 0.5104 0.4427
  - average：0.5536 0.4315
- MAHNOB-HCI 数据集
  - 5-fold：

    0.5189 0.4434 0.5849 0.4528 0.5566 0.5566 0.4717 0.4340 0.4495 0.4037
  - average：0.5163 0.4581

3.1.1.2 归一化之后

- DEAP 数据集：
  - 5-fold：

    0.7539 0.6719

    0.7852 0.6211

    0.7383 0.5898

    0.7930 0.6289

    0.7656 0.5729
  - average： 0.7672 0.6169
- MAHNOB-HCI 数据集：
  - 5-fold：

    0.8302 0.6509 0.8962 0.6887 0.9151 0.6604 0.8962 0.6698 0.8991 0.6147
  - average： 0.8874 0.6569

3.1.1.3 结果分析

归一化之后，我自己实现的 SVM 分类器的分类准确有了显著提高。

- 这应该是由于我自己实现的 SVM 没有内置的归一化操作，所以不归一化无法达到调包的准确率。
- 而 “不归一化无法达到调包的准确率”，是因为太大的输入值会让 C(误判代价) 和 toler(容错率) 太难调节（因为每次更新 alpha 每次更新之后都要根据 C 裁剪）

Naive Bayes

1. 通用分类器实现: Bayes.py

- 其中 Gaussian_NB 类要求输入的样本所有维数都是连续值，Mix_NB 类实现了 **"允许同时有离散值和连续值"**
- 实测运行速度和 sklearn 的 GaussianNB 分类器没有区别
- 在 id 上的测试准确率和 sklearn 的 GaussianNB 分类器完全相同

1.1 分类器接口介绍

1.1.1 类变量

```
class Mix_NB:
    def __init__(self):
        self.attri_option_nums = None
        self.dim = 0                    #输入维数
        self.label_num = 0              #标签数量
        self.sample_num = 0             #样本总数量
        self.num_per_label = None       #每个标签有多少样本
        self.labels = None              #标签集
        self.label_index = dict()
        self.label_back = dict()
        # 考虑到label可能缺失，如HCI数据集无8,9,10号emotion_category
        # 则label_index[11] = 7, label_back[7] = 11

        self.discr_num = 0              #离散属性数
        self.conti_num = 0              #连续属性数
        self.conti_start = 0

        self.wrong_predict = 0
        self.right_preidct = 0
        self.accuracy = 0
        self.predict_result = None


        #x_mus[label][i] = average(x_i | label), x_i代表x的第i维
        self.mus = None             #一阶矩mu
        self.sigmas = None          #二阶矩sigma
        self.priori = None          #先验概率
        self.miss_prob = None       #缺失属性值的类条件概率 = 1/(|D| + Ni)
```

```
        self.cprob = []
        #离散属性的类条件概率
        #       cprob[label][discrete_dim][option] :
        #           第discrete_dim维离散属性取 (option值 & label) 标签的概率
```

1.1.2 fit 训练接口

```
#合法的数据形式是numpy array
#attri_option_nums 是一个一维列表，从某个索引开始后面的值全为 0
#attri_option_nums[i] = k，代表 x_i(样本的第 i 维)有 k 个取值可能，0代表连续值
def fit(self,x,y,attri_option_nums):
```

1.1.3 score 评估接口

```
def score(self,x,y):
#返回预测准确率
```

1.1.4 predict_log_proba 预测接口

```
def predict_log_proba(self,x):
#返回每个样本属于每个 label 的概率（已取对数）
```

1.2 具体实现

因为代码注释得比较详细，没有再加其他的注释，并且这里只选取了核心代码 `update` 更新参数部分

1.2.1 update 训练更新参数

```
def update(self,x,y):
    for i in range(self.sample_num):
        label = self.label_index[y[i]]
        self.num_per_label[label] += 1

        for j in range(self.conti_start):
            if x[i][j] in self.cprob[label][j]:
                self.cprob[label][j][x[i][j]] += 1
            else:
                self.cprob[label][j][x[i][j]] = 0

        self.mus[label] += x[i][self.conti_start:]

    #update priori，带有拉普拉斯修正
    self.priori = (self.num_per_label + 1) / (self.sample_num +
self.label_num)
```

```
    #update mus
    for i in range(self.label_num):
        self.mus[i] /= self.num_per_label[i]

    #update sigmas
    for i in range(self.sample_num):
        label = self.label_index[y[i]]
        temp = (x[i][self.conti_start:]-self.mus[label])**2
        self.sigmas[label] += temp

    for i in range(self.label_num):
        self.sigmas[i] /= self.num_per_label[i]
    self.sigmas = np.sqrt(self.sigmas)

    #update condition_prob for discrete attribute
    for i in range(self.label_num):
        for j in range(self.conti_start):
            for key in self.cprob[i][j]:
                #拉普拉斯修正
                self.cprob[i][j][key] += 1
                self.cprob[i][j][key] /= (self.sample_num
                            + self.attri_option_nums[j])
```

2. 具体分类任务

没什么好写，数据预处理和分类器的实现都写过了，下面只举一个调用手动实现的模块来分类的代码

2.1 代表代码：id 作为标签

```
#交叉验证
for i in range(splits_num):
    ......
    model = Bayes.Mix_NB()
    model.fit(x_train,y_train,attri_option_nums)
    score = model.score(x_test,y_test)
    # temp = model.predict_log_proba(x_test)
    print("     %s: %.2f%%" % ('acc', score*100))
    cvscores.append(score * 100)

average_score = sum(cvscores)/len(cvscores)
write_score(cvscores,average_score,root_dir+'NB/subject_id_cvscores.txt')
    return average_score
```

2.2 测试结果

2.2.1 Bayes_id

2.2.1.1 归一化之前

- DEAP 数据集
  - 5-fold：97.6562 98.8281 98.0469 99.2188 98.4375
  - average：98.4375
- MAHNOB-HCI 数据集
  - 5-fold：98.1132 96.2264 98.1132 96.2264 96.3303
  - average：97.0019

2.2.1.2 归一化之后

- DEAP 数据集
  - 5-fold：94.9219 86.3281 98.4375 11.3281 89.5833
  - average：76.1198
- MAHNOB-HCI 数据集
  - 5-fold：98.1132 83.0189 96.2264 79.2453 95.4128
  - average：90.4033

2.2.1.3 结果分析

- 朴素贝叶斯在此任务上分类效果很好，说明了 **各个属性对于标签 id 的类条件概率 $P(x_i|id)$ 基本满足 “相互独立” 的假设**，这对于之后在以 valabel / emotion 进行分类的时候是一种先验知识（但是时间不够，没有具体探究并应用）
- 在这个分类器之前做数据预处理出现了很大的问题，具体出在分类第 4 个划分时准确率特别低
  - 经过单步调试自己实现的 Mix_NB 模块，问题差不多定位在一下这个高斯概率公式

$$P(x_i \mid c) = \frac{1}{\sqrt{2\pi}\,\sigma_{c,i}} exp(-\frac{(x_i - \mu_{c,i})^2}{2\sigma_{c,i}^2})$$

  - 如果像我这样简单地把数据压缩到 0~1 范围内，$\sigma_{c,i}$ **变小的幅度是 $\mu_{c,i}$ 的平方**，这会导致连续值属性的概率急剧增大，从而影响到其它属性队整体分类预测的正常参与。

2.2.2 Bayes_emotion

- MAHNOB-HCI 数据集
  - 5-fold：12.2642 13.2075 14.1509 11.3208 13.7615
  - average：12.9410

MLP

1. 调包代码

我用的是 keras 包，版本在开头写了。下面以 MLP_id 为例

```
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
```

```
......

def MLP_id(root_dir):
        ...
        # create model
        model = Sequential()
        model.add(Dense(12, input_dim=input_shape, activation='relu'))
        model.add(Dropout(0.2))
        model.add(Dense(8, activation='relu'))
        model.add(Dense(output_shape, activation='softmax'))
        # Compile model
        model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
        # Fit the model
        model.fit(x_train, y_train,  validation_data=(x_test,y_test),
                    epochs=50, batch_size=10, verbose=0)
        # evaluate the model
        scores = model.evaluate(x_test, y_test, verbose=0)
        print("     %s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
        cvscores.append(scores[1] * 100)
```

2. 分类结果

2.1 MLP_id

- DEAP 数据集
  - 5-fold：96.8750 96.8750 96.9238 96.9360 96.8750
  - average：96.8970
- MAHNOB-HCI 数据集
  - 5-fold：96.6667 96.6667 96.6667 96.7924 96.6667
  - average：96.6918

2.2 MLP_valabel

- DEAP 数据集
  - 5-fold：76.8555 75.0000 75.5859 76.4648 74.8698
  - average：75.7552
- MAHNOB-HCI 数据集
  - 5-fold：75.0000 75.0000 75.0000 75.0000 75.0000
  - average：75.0000

2.3 MLP_emotion

- MAHNOB-HCI 数据集
  - 5-fold：88.8889 88.8889 88.8889 88.8889 88.8889
  - average：88.8889

Decision Tree

1. 调包代码

用了 sklearn.tree.DecisionTreeClassifier，其中使用了 “基尼系数” 作为决策树分支评估参数

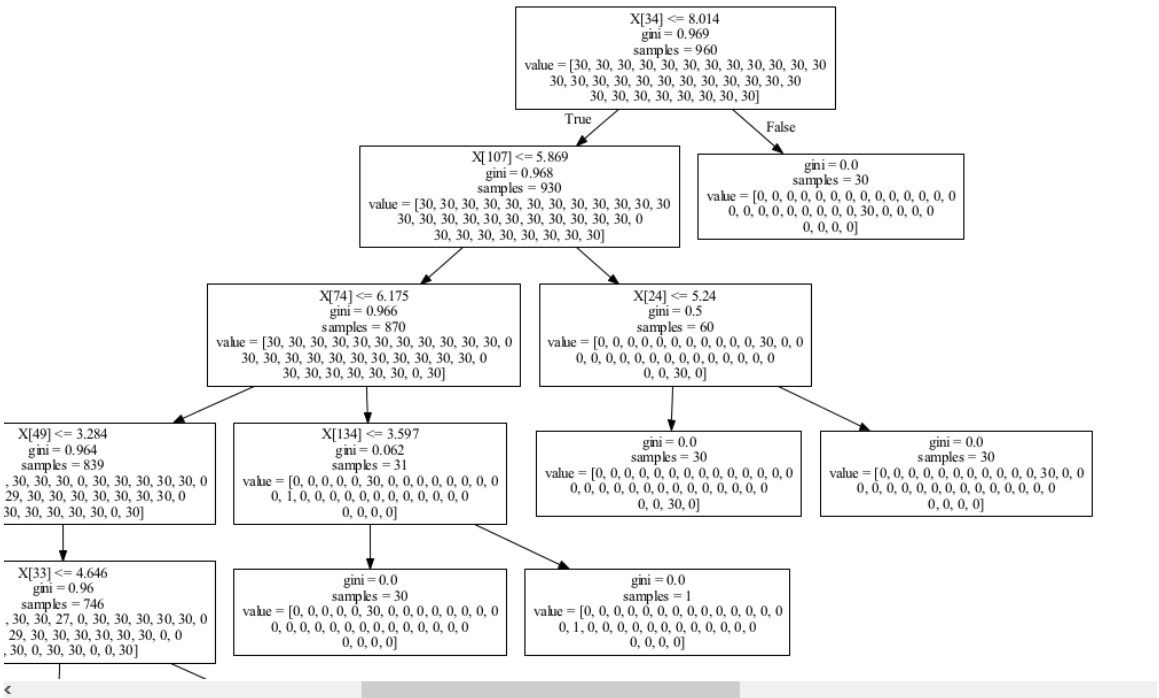
```
def DT_id(root_dir):
        ...
        model = tree.DecisionTreeClassifier(criterion='gini')
        model.fit(x_train, y_train)
```

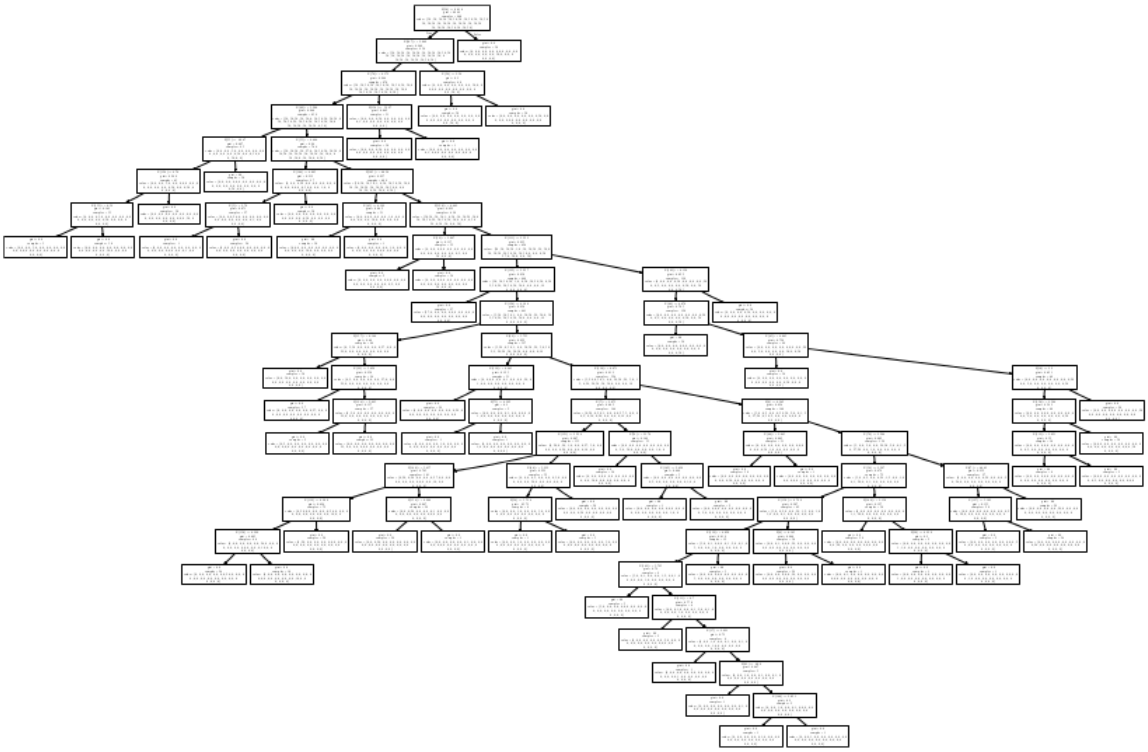
2. 可视化代码

为了方便进一步观察参数的和调参，我把决策树可视化了出来，代码如下

```
dot_data = tree.export_graphviz(model, out_file=None)
graph = pydotplus.graph_from_dot_data(dot_data)
graph.write_pdf(root_dir+'DT/Decision_Tree_fold'+str(i+1)+'.pdf')
```

分别存储在类似 data/DEAP/DT/Decision_Tree_foldi.pdf 的路径，示例图如下：





3. 分类结果

3.1 DT_id

- DEAP 数据集
  - 5-fold：0.8945 0.9375 0.9258 0.9453 0.8958
  - average：0.9198
- MAHNOB-HCI 数据集
  - 5-fold：0.8774 0.8962 0.8868 0.8679 0.9174
  - average：0.8891