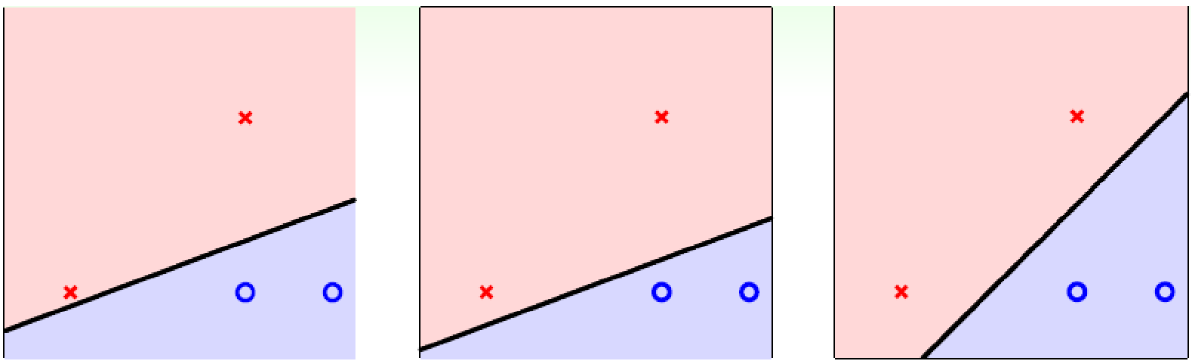
hw3 机器学习分类算法

学号：191250016　　　　　姓名：陈梓俊

SVM（支撑向量机）

simple hard-margin SVM 原理与实现

当我们做二分类问题的时候，例如使用逻辑回归或者感知器，假如训练集线性可分，那么我们只要找到一个可以分开的超平面就可以了。我们不会管这个超平面的其他性质。在优化这个超平面的问题上，我们最原始的想法，就是令超平面距离样本点尽可能的远，如下图所示：



显然我们希望的超平面（直线）是最右边的那一条，而不是其他两条，尽管他们都是可以将样本点分开。那么怎么解决这个问题呢？我们就要定义我们的margin，然后来做优化问题。

$$
\begin{aligned}
\max_{\mathbf{w}} \quad & \text{margin}(\mathbf{w}) \\
\text{subject to} \quad & \text{every } y_n \mathbf{w}^T \mathbf{x}_n > 0 \\
& \text{margin}(\mathbf{w}) = \min_{n=1,\ldots,N} \text{distance}(\mathbf{x}_n, \mathbf{w})
\end{aligned}
$$

上面的式子中，w 是超平面的法向量，y_n代表样本点的label，值为-1或者1。我们将w分成新的w和截距b，经过一系列数学推导，我们可以得到一个形式化的问题，这就是我们要求解的问题：

$$
\begin{aligned}
\min_{b,\mathbf{w}} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} \\
\text{subject to} \quad & y_n(\mathbf{w}^T\mathbf{x}_n + b) \geq 1 \text{ for all } n
\end{aligned}
$$

上面的这个问题由于存在约束条件，无法像逻辑回归那样用梯度下降求解，但是很幸运的是，上面求最小值的表达式是b, w的一个凸二次函数，而且约束是b, w的线性约束，恰好规约到了二次规划（quadratic programming）问题。二次规划问题形式化表达为

$$
QP(P, q, A, l, u) = min_x \ (1/2) * x^T Q x + p^T x
$$
$$
subject\ to\ l <= Ax <= u
$$
$$
Q, A\ is\ a\ matrix,\ p, l, u\ is\ a\ vector
$$

然后我们就可以构造相应的矩阵，放入求解凸二次规划问题的算法中求解。对应的代码如下

```
def fit(self, X, y):
    self._dim = len(X[0]) + 1
    N = len(X)

    Q = np.identity(self._dim)
    Q[0][0] = 0
    p = np.zeros(self._dim)

    A = np.zeros((N, self._dim))
    for i in range(N):
        X_tmp = np.zeros(self._dim)
        X_tmp[0] = 1
        X_tmp[1:] = X[i]
        A[i] = y[i] * X_tmp
    l = np.array([1 for i in range(N)])
    x = cp.Variable(self._dim)
    prob = cp.Problem(cp.Minimize((1/2) * cp.quad_form(x, Q) + p.T @ x), [A @ x >= 1])
    prob.solve(solver = 'OSQP', max_iter = 2000)
    self._w = x.value
    # cprint(x.value)
    return
```

在这里我们使用了`cvxpy`模块来求解二次规划。当我们的样本不是线性可分的时候，该算法无法正常运行，该二次规划问题无解。

这个算法对应的实现在`linear_hard_margin_svm.py`中。

kernel hard-margin SVM 原理与实现

dual hard-margin SVM原理

linear hard-margin的SVM存在一些问题。首先是在线性不可分的样本中无法使用，而且其求解过程与样本空间的维度d相关，Q矩阵为$(d+1)*(d+1)$的矩阵。因此当我们做特征转换（feature transform）到高纬度空间Z时，求解速度会存在问题。因此我们希望求解只和样本大小有关，而与样本空间的维度无关。要完成这个事情，我们就需要做这个二次规划问题的拉格朗日对偶问题（Lagrange duality），从而求解原问题。这些都是约束最优化问题中的数学算法，我们只将一下大概地数学证明过程。在这个过程中，我们将会知道为什么这个算法会被叫做SVM（支撑向量机）。

首先要解决在线性不可分的样本中使用SVM，我们进行了特征转换，转换为$z_n = \phi(x_n)$。原问题变为

$$\min_{b,\mathbf{w}} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

$$\text{s. t.} \quad y_n(\mathbf{w}^T \underbrace{\mathbf{z}_n}_{\Phi(\mathbf{x}_n)} + b) \geq 1,$$

$$\text{for } n = 1, 2, \ldots, N$$

在微积分中我们使用拉格朗日算子（Lagrange Multipliers）来做约束下的多元函数极值求解，同理我们可以对原问题做如下转换：

$$\mathcal{L}(b, \mathbf{w}, \alpha) =$$

$$\underbrace{\frac{1}{2}\mathbf{w}^T\mathbf{w}}_{\text{objective}} + \sum_{n=1}^{N} \alpha_n \underbrace{(1 - y_n(\mathbf{w}^T\mathbf{z}_n + b))}_{\text{constraint}}$$

然后取最优值：

$$\text{SVM} \equiv \min_{b,\mathbf{w}} \left(\max_{\text{all } \alpha_n \geq 0} \mathcal{L}(b, \mathbf{w}, \alpha) \right) = \min_{b,\mathbf{w}} \left(\infty \text{ if violate} ; \frac{1}{2}\mathbf{w}^T\mathbf{w} \text{ if feasible} \right)$$

- any ‘violating’ $(b, \mathbf{w})$: $\max_{\text{all } \alpha_n \geq 0} \left(\square + \sum_n \alpha_n (\text{some positive}) \right) \to \infty$
- any ‘feasible’ $(b, \mathbf{w})$: $\max_{\text{all } \alpha_n \geq 0} \left(\square + \sum_n \alpha_n (\text{all non-positive}) \right) = \square$

其中，不满足约束的b, w会被max转变为∞，从而取min的时候会被丢弃。故条件隐藏在max中。再转换成其拉格朗日对偶问题

$$\underbrace{\min_{b,\mathbf{w}} \left(\max_{\text{all } \alpha_n \geq 0} \mathcal{L}(b, \mathbf{w}, \boldsymbol{\alpha}) \right)}_{\text{equiv. to original (primal) SVM}} \geq \underbrace{\max_{\text{all } \alpha_n \geq 0} \left(\min_{b,\mathbf{w}} \mathcal{L}(b, \mathbf{w}, \boldsymbol{\alpha}) \right)}_{\text{Lagrange dual}}$$

由于SVM中的QP问题满足如下条件:

- convex primal
- feasible primal (true if $\mathbf{\Phi}$-separable)
- linear constraints

因此(b, w, a)对原问题和对偶问题都是最优的解，为强对偶，故$>=$可以变成$=$。我们利用$Karush-Kuhn-Tucker(KKT)conditions$的条件

- primal feasible: $y_n(\mathbf{w}^T\mathbf{z}_n + b) \geq 1$
- dual feasible: $\alpha_n \geq 0$
- dual-inner optimal: $\sum y_n\alpha_n = 0$; $\mathbf{w} = \sum \alpha_n y_n \mathbf{z}_n$
- primal-inner optimal (at optimal all 'Lagrange terms' disappear):

$$\alpha_n(1 - y_n(\mathbf{w}^T\mathbf{z}_n + b)) = 0$$

我们将对偶问题规约到如下形式:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n\alpha_m y_n y_m \mathbf{z}_n^T\mathbf{z}_m - \sum_{n=1}^{N}\alpha_n \\ \text{subject to} \quad & \sum_{n=1}^{N} y_n\alpha_n = 0; \\ & \alpha_n \geq 0, \text{for } n = 1, 2, \ldots, N \end{aligned}$$

这也是一个二次规划问题，我们继续使用求解二次规划问题的算法既可以解出a向量。

当我们解除a向量之后，我们要求解回我们的b, w，利用$KKT\ condition$，求解过程如下:

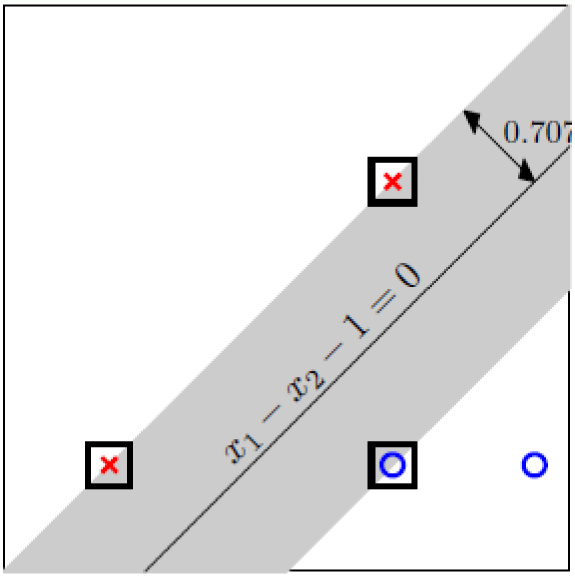
- dual-inner optimal: $\sum y_n\alpha_n = 0$; $\boxed{\mathbf{w} = \sum \alpha_n y_n \mathbf{z}_n}$
- primal-inner optimal (at optimal all 'Lagrange terms' disappear):

$$\alpha_n(1 - y_n(\mathbf{w}^T\mathbf{z}_n + b)) = 0 \text{ (**complementary slackness**)}$$

我们在求解b的时候，会发现，如果a_n不等于0的话，那么$b = y_n - w^T z_n$。同时在求解w的时候，只需要a_n不等于0的向量即可，故那些a_n不等于0的向量被称为支撑向量（support vector）。我们有：

- only SV needed to compute $\mathbf{w}$: $\mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{z}_n = \sum_{\text{SV}} \alpha_n y_n \mathbf{z}_n$
- only SV needed to compute b: $b = y_n - \mathbf{w}^T \mathbf{z}_n$ with any SV $(\mathbf{z}_n, y_n)$

我们不难发现，支撑向量一定在margin的边界上，例如：



且我们超平面的系数只取决于那些支撑向量，因此这个算法就被成为支撑向量机。

kernel hard-margin SVM 原理

在上面的算法中，存在一个问题，就是计算Q矩阵的时候，我们仍有$d+1$个维度的计算来得到，即$q_{n,m} = y_n y_m z_n^T z_m$。我们希望把转换和内积能够合成一步，那么我们就使用到了我们的核函数。以二次转换为例子：

$$\mathbf{\Phi}_2(\mathbf{x}) = (1, x_1, x_2, \ldots, x_d, x_1^2, x_1 x_2, \ldots, x_1 x_d, x_2 x_1, x_2^2, \ldots, x_2 x_d, \ldots, x_d^2)$$

$$\begin{aligned}
\mathbf{\Phi}_2(\mathbf{x})^T \mathbf{\Phi}_2(\mathbf{x}') &= 1 + \sum_{i=1}^{d} x_i x_i' + \sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j x_i' x_j' \\
&= 1 + \sum_{i=1}^{d} x_i x_i' + \sum_{i=1}^{d} x_i x_i' \sum_{j=1}^{d} x_j x_j' \\
&= 1 + \mathbf{x}^T \mathbf{x}' + (\mathbf{x}^T \mathbf{x}')(\mathbf{x}^T \mathbf{x}')
\end{aligned}$$

我们将转换和做内积合成了一步，因此我们定义核函数

$$\mathbf{\Phi}_2 \Longleftrightarrow K_{\mathbf{\Phi}_2}(\mathbf{x}, \mathbf{x}') = 1 + (\mathbf{x}^T \mathbf{x}') + (\mathbf{x}^T \mathbf{x}')^2$$

那么矩阵Q，向量b的计算以及最后结果的预测都可以使用核函数完成

1. $q_{n,m} = y_n y_m K(\mathbf{x}_n, \mathbf{x}_m)$; $\mathbf{p} = -\mathbf{1}_N$; $(\mathrm{A}, \mathbf{c})$ for equ./bound constraints
2. $\alpha \leftarrow \mathrm{QP}(\mathrm{Q_D}, \mathbf{p}, \mathrm{A}, \mathbf{c})$
3. $b \leftarrow \left(y_s - \sum_{\text{SV indices } n} \alpha_n y_n K(\mathbf{x}_n, \mathbf{x}_s) \right)$ with SV $(\mathbf{x}_s, y_s)$
4. return SVs and their α_n as well as b such that for new $\mathbf{x}$,

$$g_{\mathrm{SVM}}(\mathbf{x}) = \mathrm{sign}\left(\sum_{\text{SV indices } n} \alpha_n y_n K(\mathbf{x}_n, \mathbf{x}) + b \right)$$

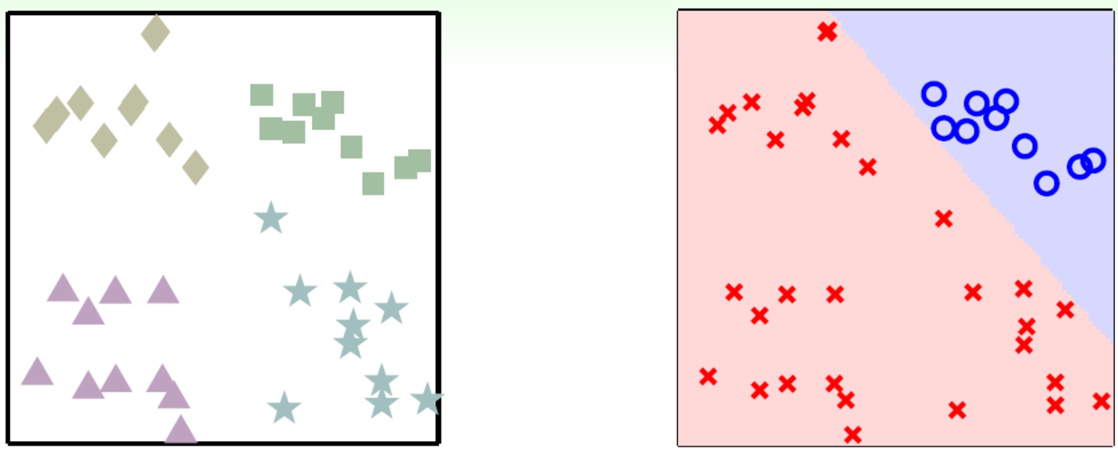
因此我们就得到了kernel hard-margin SVM算法，其中常见的核函数为多项式核函数和高斯核函数

$$\begin{aligned} K_2(\mathbf{x}, \mathbf{x}') &= (\zeta + \gamma \mathbf{x}^T \mathbf{x}')^2 \text{ with } \gamma > 0, \zeta \geq 0 \\ K_3(\mathbf{x}, \mathbf{x}') &= (\zeta + \gamma \mathbf{x}^T \mathbf{x}')^3 \text{ with } \gamma > 0, \zeta \geq 0 \\ &\vdots \\ K_Q(\mathbf{x}, \mathbf{x}') &= (\zeta + \gamma \mathbf{x}^T \mathbf{x}')^Q \text{ with } \gamma > 0, \zeta \geq 0 \end{aligned}$$

$$\text{Gaussian kernel } K(\mathbf{x}, \mathbf{x}') = \exp\left(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2 \right)$$

多分类实现

上面的kernel hard-margin SVM只适用于二分类，而要进行多分类，我们就要做一些特殊处理。多分类中使用二分类主要用两种机制，一种是OVO(One Versus One)，另一种是OVA(One Versus All)。在实现中我采用的是OVA。下图为OVA的演示：



OVA每次将每一个类训练时分为是自己的类和不是自己的类，因此N个类需要N个二分类器，这个算法存在一个问题，那么就是当一个训练样本进去，为多个类或者一个类也不是的时候，这个算法无法工作了。因此我们需要通过$sigmoid(x) = 1/(1 + exp(-1))$函数来将$w^T x$的值转换到区间$[0, 1]$之间，来表示是这个类的概率。因为$|w^T x|$表示到这个超平面的距离，距离越远概率就越高或越低，那么我们就可以选取概率最大的情况，作为这个测试样本的label。

OVO则是通过每次选取两个类得到C_N^2个分类器，每个分类器做二分类，然后这C_N^2个分类器进行投票，票数最多的类作为该测试样本的label。

具体代码实现在`kernel_hard_margin_svm.py`中。

```
'''
uses ova(one versus all) to implement multiple classification using svm
'''
def fit(self, X, y, kernel = 'gaussian'):
    if kernel == 'gaussian':
        self._kernel_func = self.__gaussian_kernel
    else:
        self._kernel_func = self.__poly_kernel

    self.__clear()
    self._N = len(X)

    for i in y:
        if i not in self._label_list:
            self._label_list.append(i)
    # print(self._label_list)


    for i, label in enumerate(self._label_list):
        x_ova_train, y_ova_train = self.__get_ova_data(X, y, label)
        self.__fit_bin(x_ova_train, y_ova_train)
    return
```

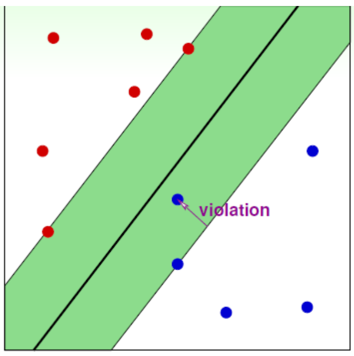
kernel soft-margin SVM 原理与实现

在linear hard-margin SVM和kernel hard-margin SVM中，我们允许出现训练样本在超平面的margin内或者出现错误。但是由于噪音的存在，这样会导致过拟合，使得我们的算法没有通用性，因此，我对原来的最优化的求解作出改变。

- record '**margin violation**' by ξ_n
- penalize with **margin violation**

$$\min_{b,\mathbf{w},\boldsymbol{\xi}} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w} + C \cdot \sum_{n=1}^{N} \xi_n$$

$$\text{s.t.} \quad y_n(\mathbf{w}^T\mathbf{z}_n + b) \geq 1 - \xi_n \text{ and } \xi_n \geq 0 \text{ for all } n$$



- parameter C: trade-off of large margin & margin violation
  - large C: want less margin violation
  - small C: want large margin

其中C是用户自定义的参数，用来控制margin的遵守程度。在这里我们引入了新的变量ξ_n，表示训练样本偏离margin的程度。同理使用对偶问题求解，得到

$$\min_{\alpha} \quad \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m y_n y_m \mathbf{z}_n^T\mathbf{z}_m - \sum_{n=1}^{N}\alpha_n$$

$$\text{subject to} \quad \sum_{n=1}^{N} y_n\alpha_n = 0;$$

$$0 \le \alpha_n \le C, \text{for } n = 1, 2, \ldots, N;$$

相对于kernel hard-margin，只是增加了N个条件，同样是一个二次规划问题。

求解完a，我们可以得到b。

soft-margin SVM

complementary slackness:

$$\alpha_n(1 - \xi_n - y_n(\mathbf{w}^T\mathbf{z}_n + b)) = 0$$

$$(C - \alpha_n)\xi_n = 0$$

- SV ($\alpha_s > 0$)

  $\Rightarrow b = y_s - y_s\xi_s - \mathbf{w}^T\mathbf{z}_s$
- free ($\alpha_s < C$)

  $\Rightarrow \xi_s = 0$

因此

solve unique b with free SV $(\mathbf{x}_s, y_s)$:

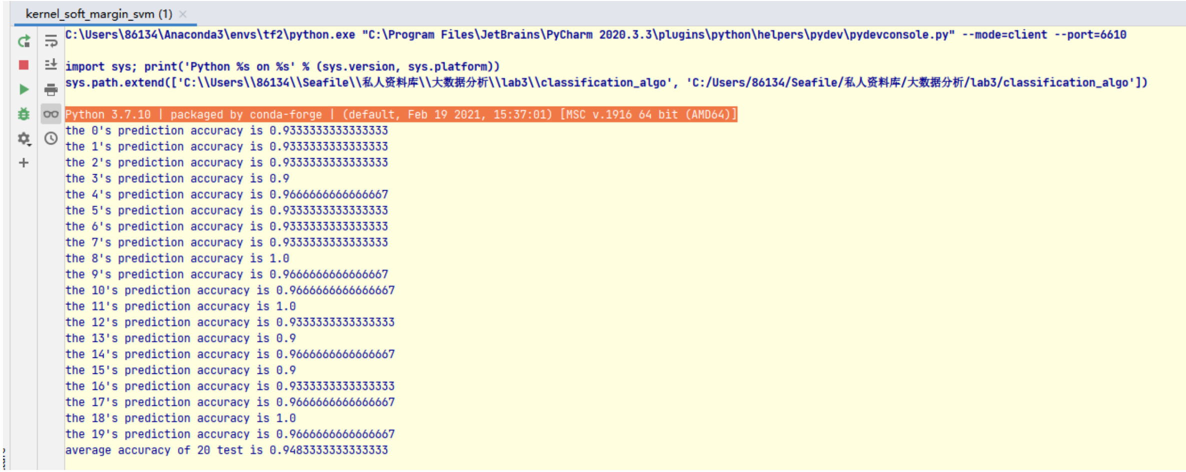
$$b = y_s - \sum_{\text{SV indices } n} \alpha_n y_n K(\mathbf{x}_n, \mathbf{x}_s)$$

具体实现在`kernel_soft_margin_svm.py`

数据处理和SVM自实现算法测试

使用Iris数据集来进行对SVM的测试。由于Iris数据集只有150个数据，因此我们需要预留出一部分来做测试（在SVM中无进行验证过程来选取模型参数和核函数）。我们使用0.2的整体数据集做测试，剩余的0.8做训练。使用sklearn.module_selection中的train_test_split每次随机的抽取20%作为训练集，重复20次。

由于实践中一般选择soft-margin SVM，在测试中我们使用$C = 1$，高斯核函数中的$\gamma = 1$作为参数，得到结果如下：



```
kernel_soft_margin_svm (1)
C:\Users\86134\Anaconda3\envs\tf2\python.exe "C:\Program Files\JetBrains\PyCharm 2020.3.3\plugins\python\helpers\pydev\pydevconsole.py" --mode=client --port=6610

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['C:\\Users\\86134\\Seafile\\私人资料库\\大数据分析\\lab3\\classification_algo', 'C:/Users/86134/Seafile/私人资料库/大数据分析/lab3/classification_algo'])

Python 3.7.10 | packaged by conda-forge | (default, Feb 19 2021, 15:37:01) [MSC v.1916 64 bit (AMD64)]
the 0's prediction accuracy is 0.9333333333333333
the 1's prediction accuracy is 0.9333333333333333
the 2's prediction accuracy is 0.9333333333333333
the 3's prediction accuracy is 0.9
the 4's prediction accuracy is 0.9666666666666667
the 5's prediction accuracy is 0.9333333333333333
the 6's prediction accuracy is 0.9333333333333333
the 7's prediction accuracy is 0.9333333333333333
the 8's prediction accuracy is 1.0
the 9's prediction accuracy is 0.9666666666666667
the 10's prediction accuracy is 0.9666666666666667
the 11's prediction accuracy is 1.0
the 12's prediction accuracy is 0.9333333333333333
the 13's prediction accuracy is 0.9
the 14's prediction accuracy is 0.9666666666666667
the 15's prediction accuracy is 0.9
the 16's prediction accuracy is 0.9333333333333333
the 17's prediction accuracy is 0.9666666666666667
the 18's prediction accuracy is 1.0
the 19's prediction accuracy is 0.9666666666666667
average accuracy of 20 test is 0.9483333333333333
```

平均有约95%的正确率，在没有具体参数调整的情况下，是一个客观的结果。

Decision Tree（决策树）

决策树原理

决策树作为最基础、最常见的有监督学习模型，常被用于分类问题和回归问题，将决策树应用集成思想可以得到随机森林、梯度提升决策树等模型。其主要优点是模型具有可读性，分类速度快。决策树的学习通常包括三个步骤：特征选择、决策树的生成和决策树的修剪，下面对特征选择算法进行描述和区别

ID3

首先要了解熵(Entropy)的概念。在热力学中，熵被用于表示系统的混乱程度；而在信息论中，熵用于表示信息量的大小。

在一个有K个类别的样本D中，假设类别Y的概率分布为：

$$P(Y = k) = p_k$$

那么这个具有K个类别的样本的信息熵为：

$$H(D) = -\sum_{k=1}^{K} p_k \log p_k$$

当整个数据集只有一个类别时，熵最低，为

$$H_{min}(D) = -1 \cdot log1 = 0$$

当数据集各类别为均匀分布时，熵最大，为

$$H_{max}(D) = -K \cdot \frac{1}{K} \cdot \log \frac{1}{K} = -\log K$$

假设某一离散属性A有V个不同的的取值，那么当以特征A来划分时可以将数据集D分为V个子集：

$$D = \sum_{v}^{V} D_v$$

那么划分之后的V个子数据集的加权熵为：

$$H(D|A) = \sum_{v}^{V} \frac{|D_v|}{|D|} H(D_v)$$

Iternative Dichotomizer，是最早的决策树算法，其根据**信息增益**(Information Gain)来寻找最佳决策特征，当按特征A来划分数据集时的信息增益定义为：

$$G(D, A) = H(D) - H(D|A)$$

首先，树的根节点中包含整个数据集，ID3算法会遍历所有特征分别做**test**来计算划分后的信息增益，然后选择信息增益最大的那个test，按该特征的类别数将数据集划分到下一层的各个子节点中；对各个子节点中的数据递归进行这个过程，直到信息增益足够小或者无特征可用。

在只考虑信息增益的情况下，ID3算法有一个致命缺陷，就是会倾向于选择类别数多的特征来做划分。假设有一列特征(如样本ID)类别数与样本数相等，如果以该特征来进行划分数据集，则数据集被划分成了单样本节点，每个节点的熵均为0，总熵也为0，这样一来得到了最大的信息增益，但是这种划分显然是不合理的。

C4.5

为了修正ID3算法的缺陷，C4.5算法应运而生。首先，C4.5在ID3的基础上改进了生成树算法，不再使用信息增益，而是使用**增益比**(gain ratio)来决定使用哪个特征来划分数据集。增益比定义为：

$$GR(D, A) = \frac{G(D, A)}{IV(A)}$$

其中$IV(A)$被称为**固有值**(intrinsic value)，它等价于某一特征在数据集中的熵。假设在一个数据集D中有某个特征A，其有K个不同的取值，那么特征A在数据集中的概率分布为：

$$P(A = v) = \frac{|D_v|}{|D|} = p_v$$

那么数据集中该特征的固有值(熵)为：

$$IV(A) = -\sum_{v=1}^{V} p_v \log p_v$$

这样一来就减少了信息增益在做决策时的比重。此外，C4.5算法还加入了对连续值的处理。

CART

CART是一棵二叉树，每次test都将问题空间分成两个区域，可处理连续变量与类别型变量。

CART分类树的生成类似上面两种算法，不过做test时选取特征的依据是**基尼指数**(Gini Index)。对于有K个类别的数据集D，某一样本属于类别k的概率等于该类别的分布概率：

$$P(Y=k)=p_k$$

那么该数据集D的基尼指数定义如下：

$$Gini(D)=1-\sum_{k=1}^{K}p_k^2$$

从公式易得，基尼指数表示的是从数据集中随机取两个不同类别样本的概率，其值越小则数据集纯度越高。

由于CART的特殊性，其在做test时与ID3、C4.5略有不同，CART是二叉树，并且在对类别型变量做划分时做得是非划分。如对一个数据集D以特征A的一个取值a来划分，那么数据集会被划分成$D_{A=a}$和$D_{A\neq a}$，那么数据集D依照特征$A=a$划分之后的加权基尼指数为：

$$G(D|A=a)=\frac{|D^{A=a}|}{|D|}Gini(D^{A=a})+\frac{|D^{A\neq a}|}{|D|}Gini(D^{A\neq a})$$

算法描述为：

输入是训练集D，基尼系数的阈值$\varepsilon 1$，样本个数阈值$\varepsilon 2$。

输出是决策树T。

我们的算法从根节点开始，用训练集递归的建立CART树。

1. 对于当前节点的数据集为D，如果样本个数小于阈值$\varepsilon 2$或者没有特征，则返回决策子树，当前节点停止递归。
2. 计算样本集D的基尼系数，如果基尼系数小于阈值$\varepsilon 1$，则返回决策树子树，当前节点停止递归。
3. 计算当前节点现有的各个特征的各个特征值对数据集D的基尼系数。
4. 在计算出来的各个特征的各个特征值对数据集D的基尼系数中，选择基尼系数最小的特征A和对应的特征值a。根据这个最优特征和最优特征值，把数据集划分成两部分$D1$和$D2$，同时建立当前节点的左右节点，做节点的数据集D为$D1$，右节点的数据集D为$D2$。
5. 对左右的子节点递归的调用1-4步，生成决策树

使用sklearn模块中的决策树实现分类

使用sklearn.tree包中的DecisionTreeClassifier来完成使用决策树对鸢尾花数据集进行分类任务，在criterion中选择了gini作为参数，因此使用的是CART算法。并且通过tree.plot_tree来刻画决策树。鸢尾花数据集的1/5作为测试集，4/5作为训练集。

参数调整

我们在训练集中取1/5作为验证集来进行参数的选取。

splitter、random_state

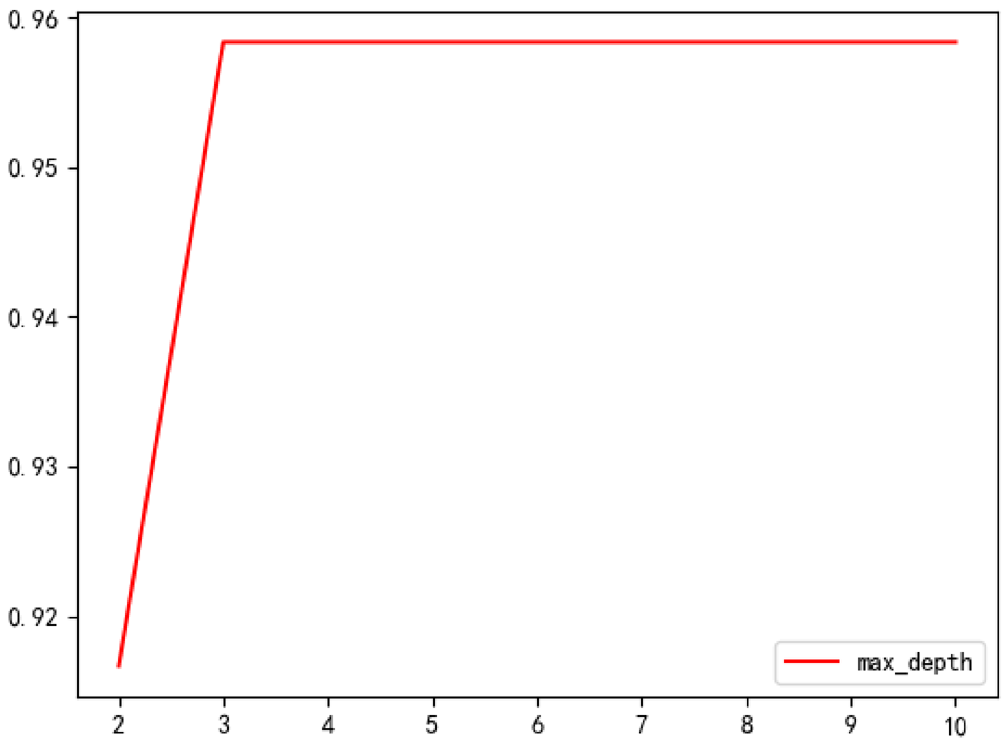
splitter也是用来控制决策树中的随机选项的，有两种输入值，输入"best"，决策树在分枝时虽然随机，但是还是会优先选择更重要的特征进行分枝（重要性可以通过属性feature_importances_查看），输入"random"，决策树在分枝时会更加随机，树会因为含有更多的不必要信息而更深更大，并因这些不必要信息而降低对训练集的拟合。这也是防止过拟合的一种方式。

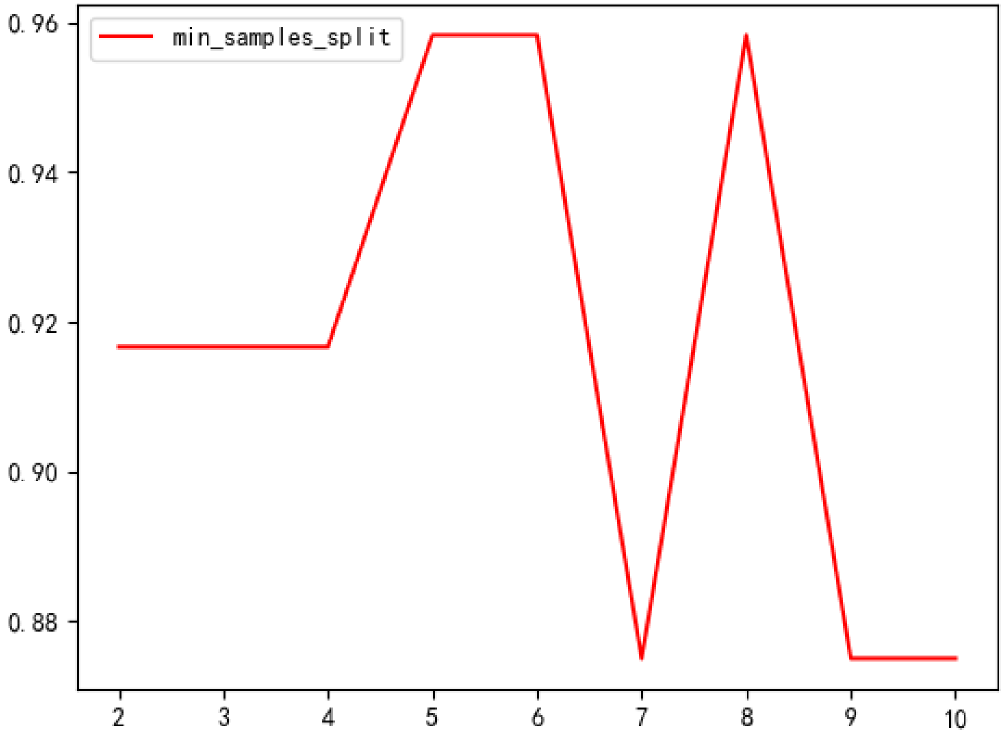
random_state用来设置分枝中的随机模式的参数，默认None，在高维度时随机性会表现更明显，低维度的数据鸢尾花数据集，随机性几乎不会显现。

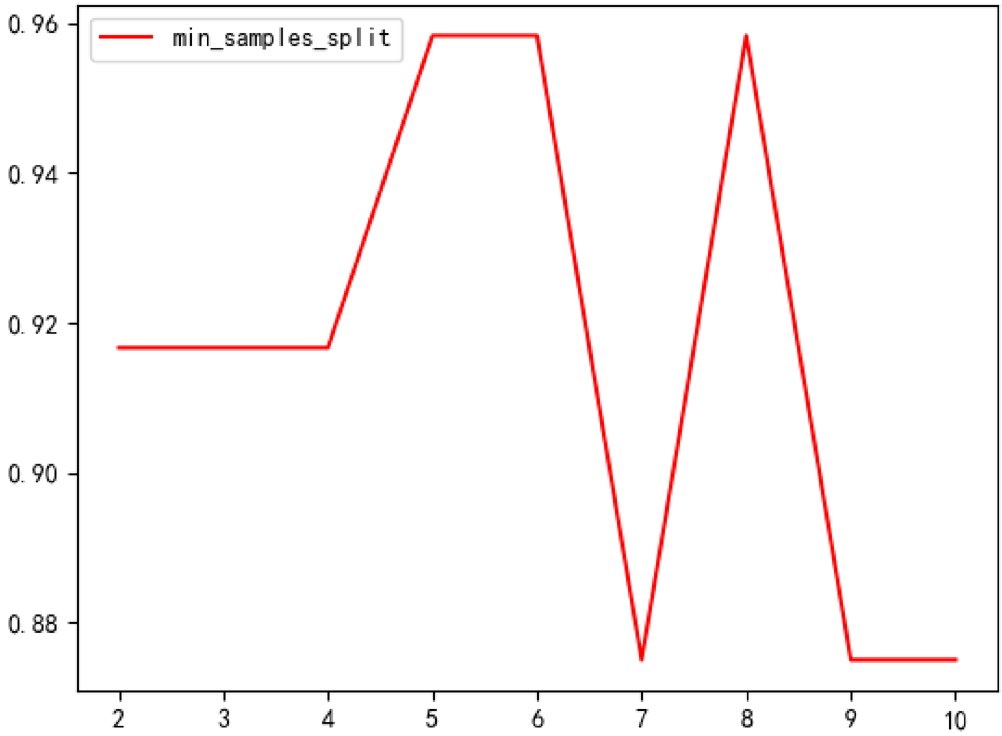
我们在splitter和random_state中分别使用best和None。

max_depth、min_samples_leaf、min_samples_split

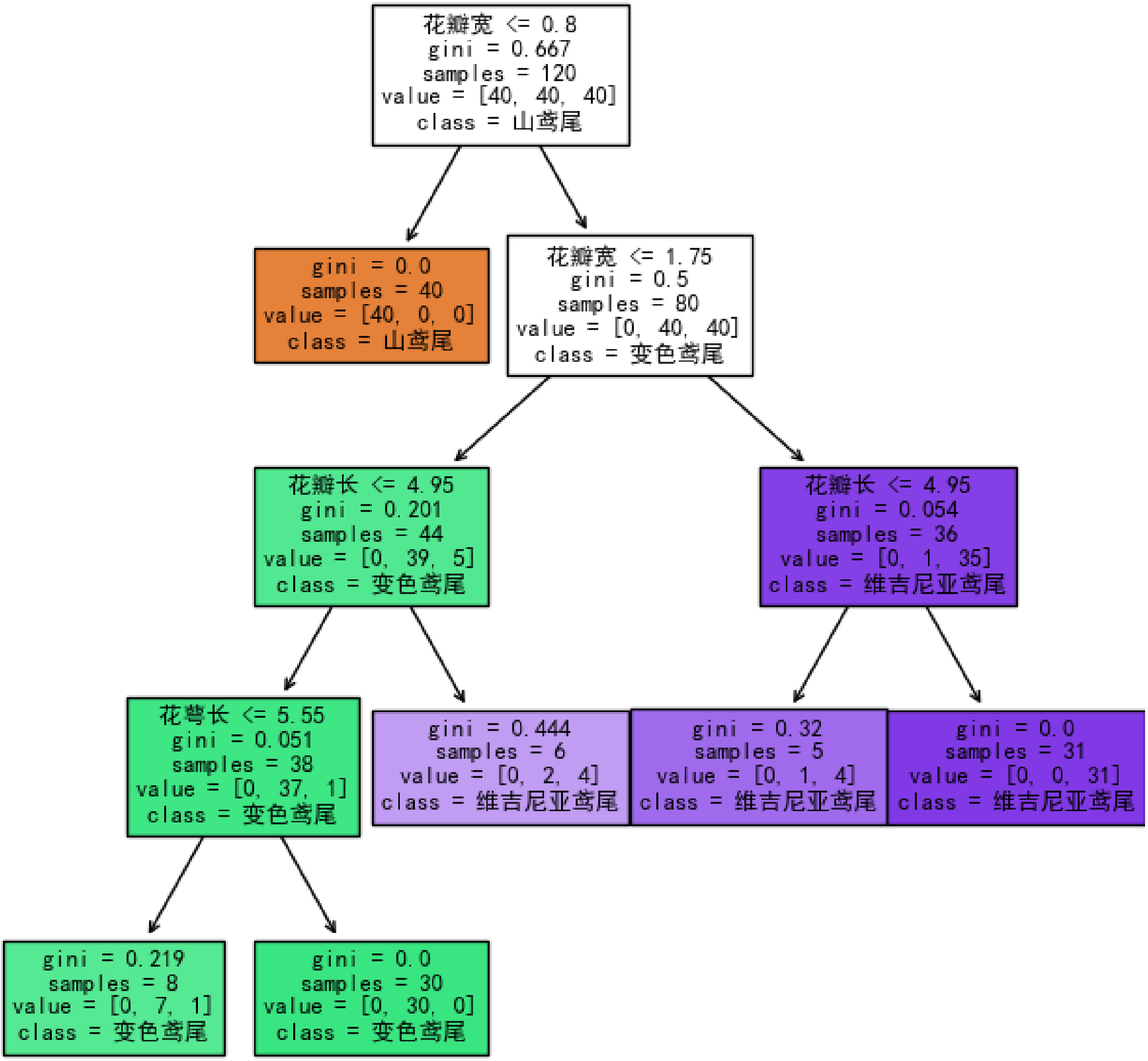
这三个参数中我们都从$[2, 10]$范围之间选取，分别在验证集中验证其正确率，选取正确率最大的参数作为最终在所有训练集上训练得到训练模型




min_samples_split
0.96
0.94
0.92
0.90
0.88
2
3
4
5
6
7
8
9
10


min_samples_split
0.96
0.94
0.92
0.90
0.88
2
3
4
5
6
7
8
9
10

测试结果



得到的决策时可视化结果如下。最终在测试集的正确率为100%

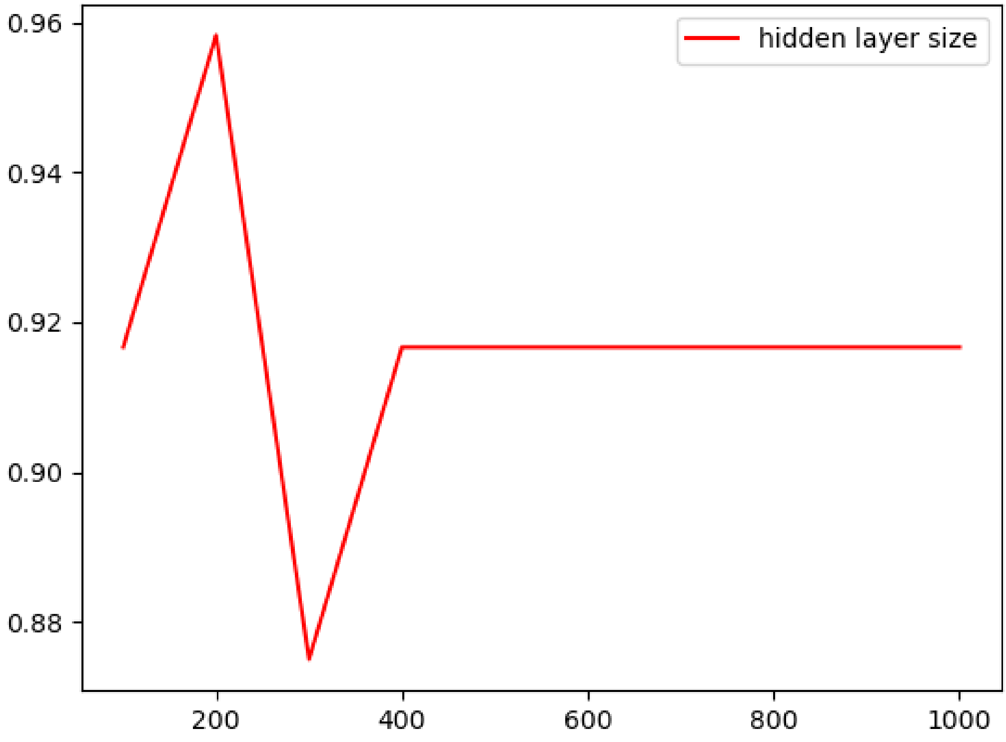
BPNN（BP神经网络）

使用sklearn模块中的神经网络实现分类

使用sklearn.neural_network中的MLClassifier来实现多分类，使用单层的隐藏层，激活函数使用将1/5的数据作为训练集，4/5作为测试集。由于训练的数据量不大，solver 'adam'在相对较大的数据集上效果比较好（几千个样本或者更多），对小数据集来说，lbfgs收敛更快效果也更好，因此我们使用lbfgs。最大迭代次数设置为2000，其他参数为默认参数。

hidden layer sizes调整

我们在训练集中取1/5作为验证集来进行参数的选取。hidden layer sizes 为单个值的元组，取值从100到1100，验证结果如下



可以知道在单层隐藏层的时候hidden layer sizes = (200,)的时候效果最好

测试结果

在测试集上正确率为100%