

MLlab2 实验报告

PB19030861 王湘峰

一、 实验要求

手动实现 XGBoost，在数据集 train.data 上自行划分训练集和测试集进行训练，并对模型进行评估。采用RMSE和 R^2 作为评估标准。

二、 实验原理

XGBoost 是由多个基模型组成的一个加法模型，假设第 k 个基本模型是 $f_k(x)$ ，那么前 t 个模型组成的模型的输出为

$$\hat{y}_i^{(t)} = \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$

其中 x_i 为第表示第 i 个训练样本， y_i 表示第 i 个样本的真实标签； $\hat{y}_i^{(t)}$ 表示前 t 个模型对第 i 个样本的标签最终预测值。

在学习第 t 个基模型时，XGBoost 要优化的目标函数：

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^n loss(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^t penalty(f_k) \\ &= \sum_{i=1}^n loss(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \sum_{k=1}^t penalty(f_k) \\ &= \sum_{i=1}^n loss(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + penalty(f_k) + const \end{aligned}$$

其中 $loss(y_i, \hat{y}_i^{(t)}) = (y_i - \hat{y}_i^{(t)})^2$.

将 $loss(y_i, \hat{y}_i^{(t-1)} + f_t(x_i))$ 在 $\hat{y}_i^{(t-1)}$ 处泰勒展开：

$$loss(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) \approx loss(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)$$

$$\text{其中 } g_i = \frac{\partial loss(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}, h_i = \frac{\partial^2 loss(y_i, \hat{y}_i^{(t-1)})}{\partial (\hat{y}_i^{(t-1)})^2}$$

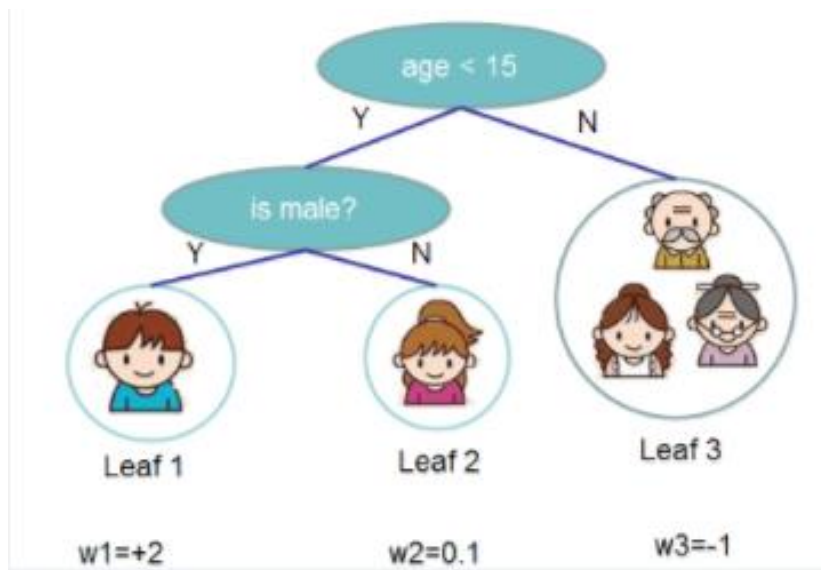
此时优化目标变成

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

$$g_i = -2(y_i - \hat{y}_i^{(t-1)}), h_i = 2$$

我们采用的基模型是决策树：

假设决策树有 T 个叶子节点，每个叶子节点对应有一个权重。决策树模型就是将输入 x_i 映射到某个叶子节点，决策树模型的输出就是这个叶子节点的权重。



$$\text{penalty}(f) = \gamma \cdot T + \frac{1}{2} \lambda \cdot ||w||^2$$

γ 和 λ 是可调的超参数。

我们将分配到第 j 个叶子节点的样本用 I_j 表示，即 $I_j = \{i | q(x_i = j)\} (1 \leq j \leq T)$

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^n \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \text{penalty}(f_t) \\ &= \sum_{j=1}^T \left[\left(\sum_{x \in I_j} g_i \right) \cdot w_j + \frac{1}{2} \cdot \left(\sum_{x \in I_j} h_i + \lambda \right) \cdot w_j^2 \right] + \gamma \cdot T \end{aligned}$$

记 $G_j = \sum_{x \in I_j} g_i, H_j = \sum_{x \in I_j} h_i$ ，求解得到 $w_j^* = -\frac{G_j}{H_j + \lambda}$ 解得

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

决策树的构建过程如下：

从根节点开始递归划分（初始情况下，所有的训练样本 x_i 都分配给根节点）。



如何确定节点的划分标准？答案是根据划分前后的收益。



假设划分前，该节点包含了若干个训练样本，要将训练样本划分为两部分，分别形成左孩子和右孩子。

划分前该节点的得分为：

$$Obj_1 = -\frac{1}{2} \cdot \frac{G^2}{H + \lambda} + \gamma$$

划分后左右子节点的得分和为：

$$Obj_2 = -\frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} \right] + 2\gamma$$

G 和 H 和前面的定义一样，即每个节点所分配到的样本对应的一阶导数和二阶导数的和。且有 $G = G_L + G_R, H = H_L + H_R$ 。

$$gain = Obj_2 - Obj_1$$

通过贪心算法来选择最大增益进行划分

Input：该节点分配到的样本集合；

Output：选定的划分特征和对应的划分阈值；

1. 选出所有可以用来划分的特征集合 F ；
2. For feature in F ：
3. 将节点分配到的样本的特征 feature 提取出来并升序排列，记做 sorted_f_value_list；
4. For f_value in sorted_f_value_list ：
5. 在特征 feature 上按照 f_value 为临界点将样本划分为左右两个集合；
6. 计算划分后的增益；
7. 返回最大的增益所对应的 feature 和 f_value。

关于评价指标

$$RMSE = \sqrt{\frac{1}{m} \sum_{i=1}^m (y_{test}^{(i)} - \hat{y}_{test}^{(i)})^2}$$

$$R^2 = 1 - \frac{MSE(\hat{y}_{test}, y_{test})}{Var(y_{test})}$$

上述指标越大越好

三、 实验代码细节

数据结构介绍：

决策树结构：嵌套字典

每个结点都是一个字典，对于叶子节点和非叶节点，有如下区别：

- ① 如果该节点的孩子是子树，则字典中的 key 有 feature value left

right 四种，feature 代表划分属性，value 代表该属性上的分割值，left 和 right 储存子树，即嵌套字典。

- ② 如果该节点的孩子是叶子节点，则其 left 和 right 直接储存叶子的权值。

数据集结构：nparray

数据集矩阵的倒数第二列是该样本的真实值，最后一列是预测值。

函数介绍：

- ① Obj 用于计算结点的目标函数值，其中每个 x_i 的真实值被记录在倒数第二列，预测值在倒数第一列。

```
def Obj(data, lamda, gamma):
    G = -2 * np.sum(data[:, -2] - data[:, -1])
    H = 2 * data.shape[0]
    return -0.5 * (G ** 2 / (H + lamda)) + gamma
```

- ② Leaf_weight 用于计算叶子结点的权重 ω_i

```
def Leaf_weight(data, lamda):
    # 返回叶节点的值
    return -(-2 * np.sum(data[:, -2] - data[:, -1])) / (2 * data.shape[0] + lamda)
```

- ③ get_w 用于决策树建立后取叶子节点的权值，使用递归函数实现

```
def get_w(x, Tree):
    if type(Tree) == dict:
        if x[Tree['feature']] > Tree['values']:
            return get_w(x, Tree['left'])
        else:
            return get_w(x, Tree['right'])
    else:
        return Tree
```

- ④ NodeSplit 用于将数据集按照节点的准则进行划分

```
def NodeSplit(data, feature, value):
    left = data[np.nonzero(data[:, feature] <= value)[0], :]
    right = data[np.nonzero(data[:, feature] > value)[0], :]
    return right, left
```

- ⑤ CreateTree 是创建决策树的函数。首先对当前的数据选择最佳的划分属性以及数值，如果经判断发现当前节点为叶子节点，则 feature 属性置空，直接返回权值。该函数也是递归实现的。

```
def createTree(data, gamma, lamda, depth=1):
    # 建树
    dept = depth
    feature, value = chooseBestSplit(data, gamma, lamda, dept)
    if feature == None:
        return value # 满足停止条件时返回叶结点值
    # 切分后赋值
    DecTree = {}
    DecTree['feature'] = feature
    DecTree['values'] = value
    # 切分后的左右子树
    left, right, = NodeSplit(data, feature, value)
    dept += 1
    DecTree['left'] = createTree(left, gamma, lamda, dept)
    DecTree['right'] = createTree(right, gamma, lamda, dept)
    return DecTree
```

- ⑥ chooseBestSplit 可以返回数据集的最佳划分属性和数值，如果不可划分则划分属性为空。划分前先进行判断，如果样本特征值一致或样本数量太少，则停止划分并生成叶节点。否则对每个属性的每个值进行遍历并计算划分后的目标函数值，最终得到最佳的划分属性和值。如果最佳划分的增益仍小于阈值，则不进行划分，直接生成叶节点。

```

def chooseBestSplit(data, gamma, lamda, depth=1):
    min_gain = thresh[0] # 允许的最小信息增益,小于则直接创建叶节点
    min_num = thresh[1] # 切分的最小样本数

    # 若所有特征值都相同, 停止切分
    if len(set(data[:, -2].T.tolist())) == 1:
        return None, Leaf_weight(data, lamda)

    # 若样本数目小于阈值, 则停止切分并生成叶节点
    if np.shape(data)[0] < min_num:
        return None, Leaf_weight(data, lamda)

    m, n = np.shape(data)
    parent = Obj(data, lamda, gamma) # 计算划分前该节点的得分
    max_gain = float("-inf")
    bestfeature = 0
    bestValue = 0

    for feature in range(n - 2): # 遍历数据的每个属性特征
        for splitVal in set((data[:, feature].T.tolist())): # 遍历每个特征里不同的特征值
            right, left = NodeSplit(data, feature, splitVal) # 对每个特征进行二元分类
            if (np.shape(right)[0] < 1) or (np.shape(left)[0] < 1):
                continue
            child = Obj(right, lamda, gamma) + Obj(left, lamda, gamma)
            score = parent - child
            if score > max_gain:
                bestfeature = feature
                bestValue = splitVal
                max_gain = score

    # 如果切分后的增益小于阈值, 直接创建叶节点
    if max_gain < min_gain or depth > maxdepth:
        return None, Leaf_weight(data, lamda)

    return bestfeature, bestValue # 返回特征编号和用于切分的特征值

```

⑦ fit 是拟合的主函数。在 main 函数中设置超参（如训练次数

tree_num, 学习率 lr 等）来调整拟合过程。函数结构为两层 for 循

环，外层为训练次数，内层为样本个数。每次训练得到的决策树都保存

在字典 f 中；每次训练完计算 RMSE 和 R^2 并保存在列表中。

```
def fit(train_data, tree_num, gamma, lamb):
    f = []
    RMSE = []
    R2 = []
    for j in range(tree_num):
        f.append(createTree(train_data, gamma=gamma, lamda=lamb, depth=1))
        for i in range(train_data.shape[0]):
            train_data[i, -1] += get_w(train_data[i, :], f[j]) * lr

        print("正在训练第{}棵决策树".format(j + 1))
        rmse = (np.sum((train_data[:, -2] - train_data[:, -1]) ** 2) / m) ** 0.5
        RMSE.append(rmse)
        print("训练集RMSE:", rmse)
        r2 = 1 - rmse ** 2 / np.var(train_data[:, -2])
        R2.append(r2)
        print('训练集R2:', r2)
    return f, RMSE, R2
```

- ⑧ predict 函数用于预测结果。预测方法是对每棵树给出的结果做求和
(乘以学习率)

```
def predict(test_data, f):
    for j in range(tree_num):
        for i in range(test_data.shape[0]):
            test_data[i, -1] += get_w(test_data[i, :], f[j]) * lr

    rmse = (np.sum((test_data[:, -2] - test_data[:, -1]) ** 2) / test_data.shape[0]) ** 0.5
    print("测试集RMSE:", rmse)
    r2 = 1 - rmse ** 2 / np.var(test_data[:, -2])
    print('测试集R2', r2)
```

四、实验困难及解决

- ① 数据结构的确定：第一次尝试的时候用 pd.DataFrame 保存数据，但是 pandas 的特性导致有些时候数据写入原数据失败（即赋值之后未发生变化，原因似乎与 pandas 底层的实现有关），遂穷则思变。
解决方案：改用 np.array 存储数据，并统一采用数字索引。对于 train_label 和 test_label，也直接保存在了原数据的最后两列。
- ② 训练结果不理想：初始时没有想到使用学习率（或者说学习率为 1），不论是增加训练次数还是改变树的深度都不能不能使测试集的 R^2 超过

0.75。

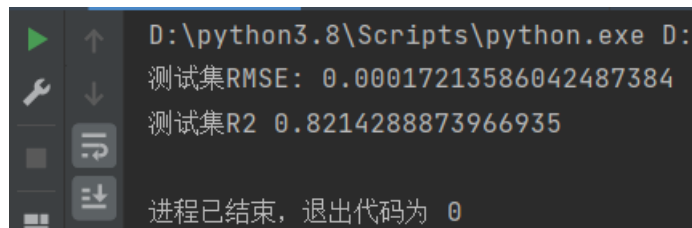
解决方案：后来想到了使用学习率，经测试将学习率调到0.1~0.3效果最佳，训练集的 R^2 基本上能稳定在0.8以上。

五、实验结果的展示

训练参数如下（训练集是随机划分80%的 train_data）

```
if __name__ == '__main__':  
    # 初始化部分  
    raw = pd.read_csv('train.data', header=None)  
    raw[41] = np.zeros(raw.shape[0])  
    df1 = raw.sample(frac=0.8)  
    df2 = raw[~raw.index.isin(df1.index)]  
    # 预处理  
    train_data = np.array(df1)  
    test_data = np.array(df2)  
    m, n = train_data.shape  
    # 参数部分  
    maxdepth = 3  
    tree_num = 25  
    gamma = 0  
    lamb = 1  
    lr = 0.2  
    thresh = (0, 3)  
    # 训练与预测部分  
    f, RMSE, R2 = fit(train_data, tree_num, gamma, lamb)  
    predict(test_data, f)
```

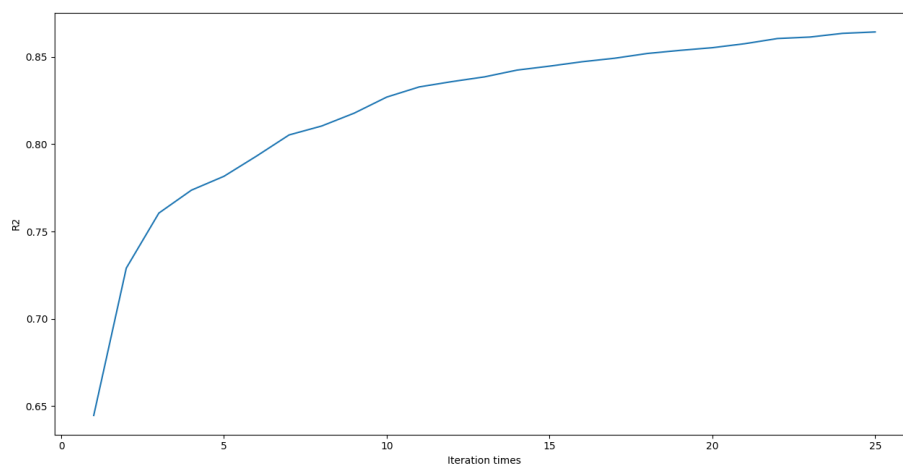
测试集RMSE与 R^2 : $RMSE = 0.000172$, $R^2 = 0.8214$



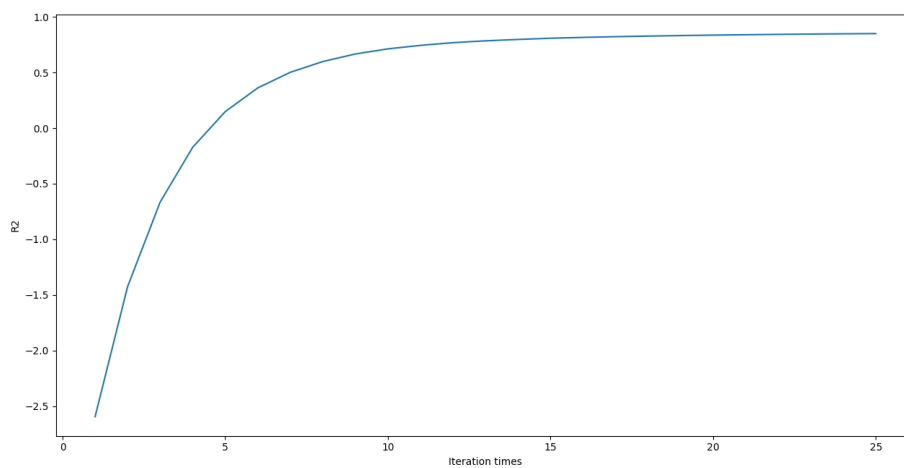
```
D:\python3.8\Scripts\python.exe D:  
测试集RMSE: 0.00017213586042487384  
测试集R2 0.8214288873966935  
进程已结束，退出代码为 0
```

训练过程中 R^2 随训练次数（树的个数）的变化：

① 不使用学习率（或者说 $lr=1$ ）：

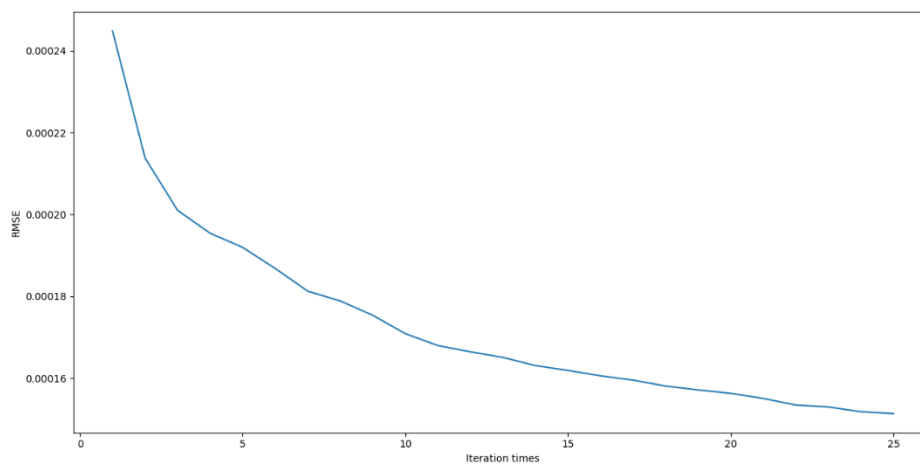


② 使用学习率的情况: (lr=0.15)

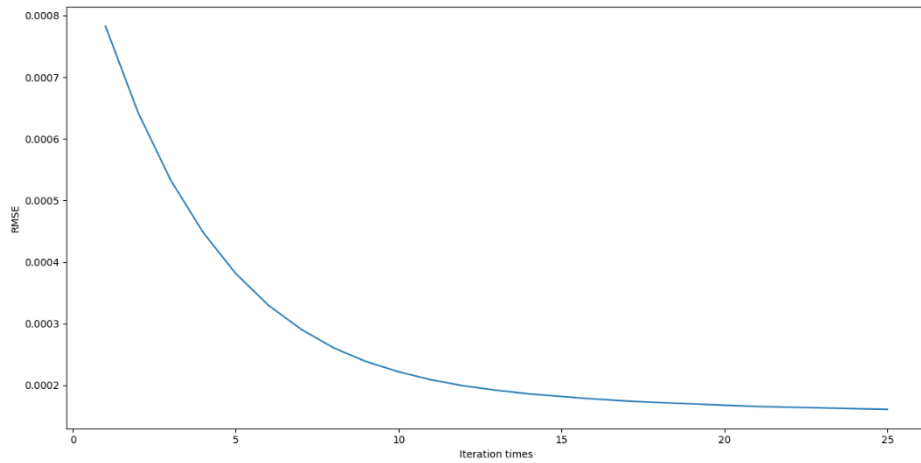


训练过程中RMSE随训练次数的变化:

① 不使用学习率:

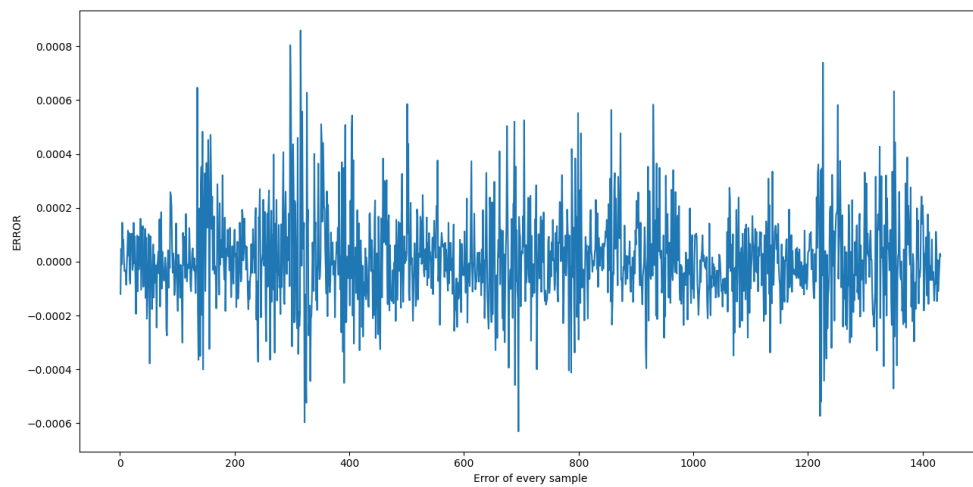


② 使用学习率: (lr=0.15)

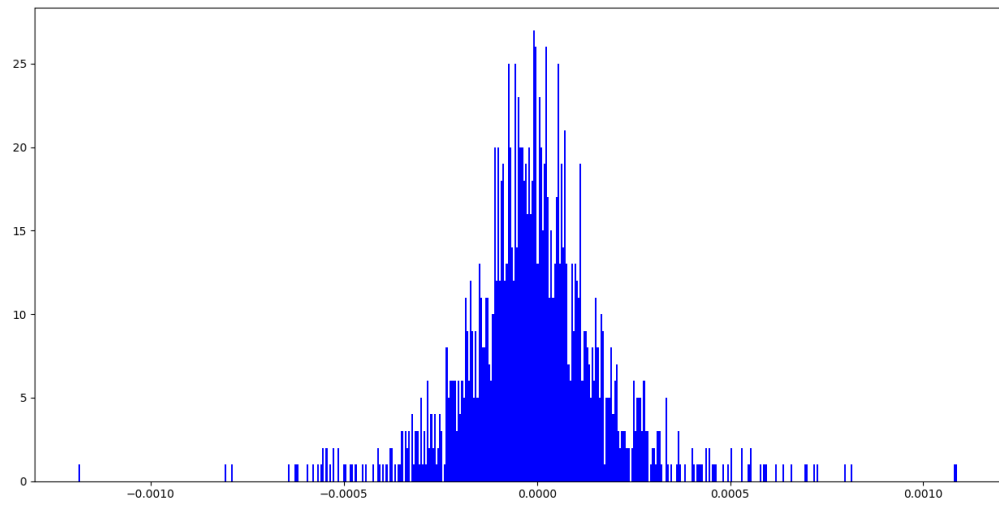


从上面两幅图可以看出，使用了学习率之后的曲线更为光滑，且效果更好（不使用学习率 R^2 大约在0.78左右）

训练完成后（使用已知最佳的参数），每个样本预测值与真实值的差值（即误差）：



对误差进行统计得到误差直方图：



可以看出绝对误差基本上在 0 ± 0.0005 之内，拟合较为良好。