MLlab2 实验报告

PB19010450 和泳毅

一、 实验要求

本次实验要求手动实现**XGBoost**（eXtreme Gradient Boosting），并在给定数据集上进行训练和验证/测试。具体需要完成以下部分：

- 读取训练数据集并自行划分验证集
- 手动实现**XGBoost**模型
- 设置模型停止运行的标准，决策树节点停止划分的标准
- 在训练数据集上进行模型的优化与参数选择
- 在测试数据集上进行测试，并输出预测结果
- 设置评价指标$RMSE$与R^2

二、数据集介绍

该数据集用于解决F16飞机的控制问题。其特征描述了飞机的状态，而目标是预测飞机副翼上的控制动作。

每个样本有40个特征，特征都是连续的，预测值也是连续的。

	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	...	f18	f19	f20	f21	f22	f23	f24	f39	f40	label
0	2	-56	-0.33	-0.09	0.90	0.2	-11	12	0.004	-0.1	...	0.032	0.032	0.032	0.032	0.032	0.032	0.032	0.9	0.032	-0.0009
1	470	-39	0.02	0.12	0.39	-0.6	-12	8	0.009	-1.6	...	0.034	0.034	0.034	0.034	0.034	0.034	0.034	0.9	0.034	-0.0011
2	165	4	0.14	0.14	0.78	0.4	-11	-9	-0.003	-0.2	...	0.034	0.034	0.034	0.034	0.034	0.034	0.034	1.0	0.034	-0.0012
3	-113	5	-0.12	0.11	1.06	0.6	-10	-7	-0.008	0.0	...	0.033	0.033	0.033	0.033	0.033	0.033	0.033	0.9	0.033	-0.0011
4	-411	-21	-0.17	0.07	1.33	-0.6	-11	0	0.002	0.1	...	0.030	0.030	0.030	0.030	0.030	0.030	0.032	0.9	0.032	-0.0008

三、 实验原理

1. 模型

XGBoost 是boosting族中的算法，遵从前向分步加法，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

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

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。

2.优化目标

XGBoost和GBDT两者都是boosting方法，除了工程实现、解决问题上的一些差异外，最大的不同就是目标函数的定义。模型的预测精度由模型的偏差和方差共同决定，损失函数代表了模型的偏差，想要方差小则需要在目标函数中添加惩罚项，用于防止过拟合。所以目标函数由模型的损失函数与抑制模型复杂度的惩罚项组成。在学习第t个基模型时，**XGBoost**要优化的目标函数:

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

其中n表示训练样本的数量，$\mathrm{penalty}(f_k)$表示对第k个模型的复杂度的惩罚项。由于依次学习每个基模型，所以当学习第t个基模型时，前$t-1$个基模型是固定的，其penalty是常数。$\mathrm{loss}\left(y_i, \hat{y}_i^{(t)}\right)$表示损失函数，在回归任务中损失函数是:

$$\mathrm{loss}\left(y_i, \hat{y}_i^{(t)}\right) = (y_i - \hat{y}_i^{(t)})^2$$

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

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

其中$g_i = \frac{\partial \mathrm{loss}\left(y_i, \hat{y}_i^{(t-1)}\right)}{\partial \hat{y}_i^{(t-1)}}, h_i = \frac{\partial^2 \mathrm{loss}\left(y_i, \hat{y}_i^{(t-1)}\right)}{\partial \left(\hat{y}_i^{(t-1)}\right)^2}$，即$g_i$是一阶导数，$h_i$是二阶导数。

此时的优化目标变为

$$Obj^{(t)} = \sum_{i=1}^{n} \left[\mathrm{loss}\left(y_i, \hat{y}_i^{(t-1)}\right) + g_i f_t\left(x_i\right) + \frac{1}{2} h_i f_t^2\left(x_i\right)\right] + \mathrm{penalty}(f_t) + \mathrm{constant}$$

去掉常数项$\mathrm{loss}\left(y_i, \hat{y}_i^{(t-1)}\right)$（学习第$t$个模型时候，$\mathrm{loss}\left(y_i, \hat{y}_i^{(t-1)}\right)$也是一个固定值）和$constant$，可得目标函数为

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

下面解决回归问题，即用基模型$f(x_i)$拟合标签数据y_i。那么

$$\text{loss}\left(y_i,\hat{y}_i^{(t-1)}\right)=(y_i-\hat{y}_i^{(t-1)})^2$$

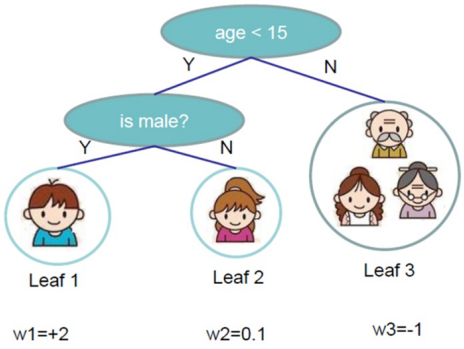
则

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

所以我们只需要求出每一步损失函数的一阶导g_i和二阶导h_i的值（由于前一步的$\hat{y}_i^{(t-1)}$是已知的，所以这两个值是常数），然后最优化目标函数Obj，就可以得到每一步的$f(x)$，最后根据加法模型得到一个整体模型。

3.基模型——决策树

XGBoost的基模型不仅支持决策树，还支持线性模型，本实验主要使用基于决策树的目标函数。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。



即$f(x_i)=w_{q(x_i)}$，w是一个要学的T维的向量其中$q(x_i)$表示把输入x_i映射到的叶子节点的索引。例如：$q(x_i)=3$，那么模型输出第三个叶子节点的权重，即$f(x_i)=w_3$。

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

其中T为叶子节点数目，γ和λ是可调的超参数。

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

由前面可知:

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

其中$G_j=\sum_{i \in I_j} g_i, \quad H_j=\sum_{i \in I_j} h_i$。这里注意$G_j$和$H_j$是前$t-1$步得到的结果，其值为常数，只有最后一棵树的叶子节点w_j不确定。

于是求解出w_j的闭式解$w_j^*=-\frac{G_j}{H_j+\lambda}$，得出目标值：

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

求出了每个叶子节点的权重w和整颗树对应的目标值后，可以度量树的好坏程度。根据划分前后的收益确定节点的划分。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。

$$
\begin{aligned}
Obj_1 &= -\frac{1}{2} \frac{G^2}{H+\lambda}+\gamma \\
Obj_2 &= -[\frac{1}{2} \frac{{G_L}^2}{H_L+\lambda}+\frac{1}{2} \frac{{G_R}^2}{H_R+\lambda}]+2\gamma \\
Gain &= Obj_1-Obj_2
\end{aligned}
$$

知道了父节点和子节点之间的增益，通过贪心算法实现最大增益划分。

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
for $k=1$ ***to*** m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted(I, by* $\mathbf{x}_{jk}$*)* **do**
    $G_L \leftarrow G_L+g_j$, $H_L \leftarrow H_L+h_j$
    $G_R \leftarrow G-G_L$, $H_R \leftarrow H-H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

最终预测值是每个基模型预测值的和，即

$$
\hat{y}_i=\sum_{t=1}^{M} f_t(x_i).
$$

4.模型停止标准

（1）基模型决策树节点是否划分的标准：

- 设置划分后增益小于0时停止划分；
- 划分后深度大于 max_depth 时停止划分， max_depth 是可调参数，本实验最后选择 max_depth =3；
- 该节点分配到的样本数目小于3个停止划分；

（2）算法停止运行标准：

- 学习 epoch 棵决策树后停下来， epoch 是可调参数，本实验最后选择 epoch =100搭配早停机制；
- 当在验证集上的均方误差连续 ESR 次没有降低时停下来， ESR 是可调参数，本实验最后选择 ESR =20搭配早停机制；

四、核心代码讲解

1.损失、评价、目标

```
# 均方误差
def MSE(ytrue, ypred):
    return np.sum((ytrue - ypred) ** 2) / ytrue.size

# 均方根误差
def RMSE(ytrue, ypred):
    return np.sqrt(MSE(ytrue, ypred))

# 相关系数
def R2(ytrue, ypred):
    return 1 - MSE(ytrue, ypred) / np.var(ytrue)

# 目标函数的一阶导和二阶导
def obj(ytrue, ypred):
    return 2 * (ypred - ytrue), 2
```

2.树结点结构

```python
# --------- 树结点结构 ---------
class TreeNode:
    """
    index: 结点包含样本的数组下标
    isleaf: 是否为叶子结点，是为TRUE，否为FALSE
    split_f: 划分点选取的特征
    split_v: 划分点选取的特征的值
    weight: 叶子结点的权值
    left: 左孩子结点
    right: 右孩子结点
    """
```

树结点结构包含传统的左右指针以及权重的存放功能，且还存放样本的数组下标列表、与该结点选择的特征及特征值。

3.决策树结构

```python
# --------- 决策树结构 ---------
class DecisionTree:
    """
    max_depth: 树最大深度
    gamma: 叶子结点数正则化系数
    Lambda: 二次正则化项
    subsample: 样本采样比例
    colsample: 特征采样比例
    seed: 随机种子
    lr: 学习率
    """
    def __init__(self,...):
        """..."""

    def fit(self, prior_y):
        """
        已知前t-1棵树输出，学习第t棵树
        :param prior_y:前t-1棵树输出
        :return:第t棵树
        """
        # 计算一阶导数g，二阶导数h
        self.__g, self.__h = np.array(
            [obj(y_true, y_pred) for y_true, y_pred in
             zip(DecisionTree.__y, prior_y)]).T
        # 随机采样，每棵树按照给予的比例随机选取样本和特征进行学习
        np.random.seed(self.seed)
        n, N = DecisionTree.__X.shape
```

```
        self.row_samples =
          np.random.choice(n, round(self.subsample * n), replace=False)
        self.col_samples =
          np.random.choice(N, round(self.colsample * N), replace=False)
        # 递归建树
        self.tree = self.__create_tree(TreeNode(self.row_samples), depth=0)
        return self

    def predict(self, X):
        """..."""

    def __split(self, index):
        """
        按照增益对该结点样本集index划分
        :param index: 需要划分达到样本集下标
        :return: 划分方案；(False,None)表示不划分；
        (True,(_,_,_,_))表示划分且返回(选择的特征，值，左子树样本下标，右子树样本
下标)
        """
        # 如果该结点的样本少于三个，停止划分
        if len(index) < 3:
            return False, None
        # 计算得分obj1
        gain = -np.inf
        _k = _i = None
        G, H = np.sum(self.__g[index]), np.sum(self.__h[index])
        obj1 = -1 / 2 * G ** 2 / (H + self.Lambda + self.eps) + self.gamma
        # 遍历特征
        for k in self.col_samples:
            Gleft, Hleft, Gright, Hright = 0, 0, G, H
            # 对该特征值升序排列
            sorted_index =
              sorted(index, key=lambda i: DecisionTree.__X[i, k])
            # 遍历该特征的值
            i = 0
            while i < len(sorted_index) - 1:
                idx = sorted_index[i]
                cur_value = DecisionTree.__X[idx, k]
                Gleft += self.__g[idx]
                Hleft += self.__h[idx]
                # 小于等于划分值的都划到左子树
                while i + 1 < len(sorted_index) - 1 and
                      DecisionTree.__X[sorted_index[i + 1], k] == cur_value:
```

```
                    i = i + 1
                    idx = sorted_index[i]
                    Gleft += self.__g[idx]
                    Hleft += self.__h[idx]
                Gright, Hright = G - Gleft, H - Hleft
                obj2 = -1 / 2 * (Gleft ** 2 / (Hleft + self.Lambda
                        + self.eps) +Gright ** 2 / (Hright + self.Lambda +
                         self.eps)) + self.gamma
                cur_gain = obj1 - obj2
                # 更新最大的增益
                if cur_gain > gain:
                    gain, _k, _i = cur_gain, k, i
                i = i + 1
        # 当增益小于阈值时停止划分
        if gain > 0:
            # 利用选择的特征进行值排序
            sorted_index =
              sorted(index, key=lambda i: DecisionTree.__X[i, _k])
            # 利用选择的特征值进行左右子树划分，返回特征，值，左子树样本下标，右子
树样本下标
            return True, (_k, DecisionTree.__X[sorted_index[_i], _k],
                          sorted_index[:_i + 1], sorted_index[_i + 1:])
        else:
            return False, None

    def __create_tree(self, node, depth):
        """
        递归建树
        :param node: 当前结点
        :param depth:当前深度
        :return:
        """
        global split_res
        stop_flag = True  # 结点停止划分标志
        if depth >= self.max_depth:  # 超过最大深度停止划分
            stop_flag = False
        if stop_flag:  # 划分结点
            stop_flag, split_res = self.__split(node.index)
        if stop_flag:  # 更新
            # 选择的特征，划分的值，左孩子样本下标，右孩子样本下标
            f, v, left_index, right_index = split_res
            node.split_f, node.split_v = f, v
            # 对左子树递归划分
```

```
            node.left = self.__create_tree(TreeNode(left_index), depth + 1)
            # 对右子树递归划分
            node.right = self.__create_tree(TreeNode(right_index), depth +
1)
        else:
            # 如果结点不再划分，标记为叶结点，并计算权重
            node.isleaf = True
            G, H =
              np.sum(self.__g[node.index]), np.sum(self.__h[node.index])
            node.weight = self.lr * -G / (H + self.Lambda + self.eps)
        return node
```

- `fit`：第t步时，由前$t-1$步的结果可以计算出一阶导与二阶导并存为数组。参考原论文与随机森林模型思想设置行采样（每轮训练不使用全部样本）与列采样（每轮训练不使用全部特征），使得算法更加保守，不易过拟合。最后递归建树。
- `__split`：结点划分函数，当结点的样本少于三个，停止划分。由于Obj_1为常数，先计算并存储Obj_1，使用 `self.eps` 防止计算出现下溢。接着采用贪心策略遍历特征选取增益最大的划分特征与划分值，如果最大增益小于0时停止划分。
- `__create_tree`：建树函数，当树深度超过 `self.max_depth` 时停止划分。根据 `__split` 函数返回的划分策略进行结点划分，并递归对所有结点调用 `__split` 直到叶子节点。在叶子节点将 `node.isleaf` 标志置位，并计算赋值叶子的权重，此处使用shrinkage思想，为该权重乘以一个比例以缩小该树对模型的影响，为后面的训练留出更多的学习空间，称为学习率：$w_j = \alpha \cdot \frac{-G_j}{H_j+\lambda+\epsilon}$。

4.XGBoost结构

```
# --------- XGBoost模型 -----------
class XGBoost(object):
    """
    epoch: 最大子树棵树，迭代次数
    max_depth: 每颗子树的最大深度
    lr: 学习率
    Lambda: 二次正则化系数
    gamma: 叶结点个数正则化系数
    subsample: 样本比例
    colsample: 特征比例
    seed: 随机种子
    """

    def __init__(self, ...):
        """..."""

    def fit(self, X, y, eval_set=None, ESR=None):
```

```
        """
        对epoch棵树训练
        :param X: 训练集特征
        :param y: 训练集标签
        :param eval_set: 验证集，用于评估泛化能力、调参以及早停机制。
        :param ESR: Early stopping rounds，早停机制。当模型在验证集上的损失连续
                    ESR次迭代都没有降低，则停止。并把损失最低的一次迭代次数记为
                    最佳迭代次数best_epoch。
        :return: self
        """
        self.err = {'train': [], 'val': []}
        self.best_epoch = self.epoch
        self.trees = []
        DecisionTree.set_data(X, y)

        if eval_set is not None:
            cur_val_y = np.zeros(eval_set[1].size)
            if ESR:
                min_val_loss = np.inf
                non_dec_rounds = 0

        # 学习第一棵树把先前输出当作0
        cur_y = np.zeros(y.size)
        # 开始学习
        for i in range(self.epoch):
            subtree = DecisionTree(self.max_depth, self.gamma, self.Lambda,
                       self.colsample, self.subsample, self.seed, self.lr)
            # 根据前i-1棵子树的输出学习第i棵树
            subtree.fit(cur_y)
            self.trees.append(subtree)
            # 更新预测值
            cur_y += subtree.predict(X)
            # 更新损失
            self.__cal_MSE(y, cur_y, flag='train')
            # 早停机制
            if eval_set is not None:
                cur_val_y += subtree.predict(eval_set[0])
                val_loss =
                   self.__cal_MSE(eval_set[1], cur_val_y, flag='val')
                if ESR:
                    if val_loss < min_val_loss:
                        min_val_loss = val_loss
                        non_dec_rounds = 0
```

```
                        self.best_epoch = i + 1
                    else:
                        non_dec_rounds += 1
                        if non_dec_rounds >= ESR:
                            break
        return self

    def predict(self, X):
        """
        输出预测结果
        :param X: 特征数组
        :return: 预测数组
        """
        pred = np.zeros(X.shape[0])
        for subtree in self.trees[:self.best_epoch]
                        if self.best_epoch else self.trees:
            pred += subtree.predict(X)
        return pred

    def __cal_MSE(self, ytrue, ypred, flag='train'):
        """..."""
```

- `fit`：学习第一棵树时将当前预测值当作0。通过迭代对 `self.epoch` 棵树进行训练学习，此处参考原论文的早停机制，在训练集上学习的同时在验证集进行预测验证，当模型在验证集上的损失连续 `ESR` 次迭代都没有降低，训练停止。并把损失最低的一次迭代次数记为最佳迭代次数 `best_epoch` 。
- `predict`：累加 `self.best_epoch` 棵树的输出作为该样本的预测值。

5.数据处理

```
# 读入数据
names = []
for i in range(40):
names.append('f' + str(i + 1))
names.append('label')
df = pd.read_csv("train.data", names=names)
# 选取去除的特征
columns = [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37]
df = df.drop(df.columns[columns], axis=1)
# 训练集
n, N = df.shape
x = df.iloc[:, 0:N - 1]
y = df['label']
x_train = np.array(x[:round(0.8 * n)])
```

```
y_train = np.array(y[:round(0.8 * n)])
# 验证集
x_eval = np.array(x[round(0.8 * n):])
y_eval = np.array(y[round(0.8 * n):])
# 测试集
df_ = pd.read_csv("ailerons.test", names=names)
df_ = df_.drop(df_.columns[columns], axis=1)
N = df_.shape[1]
x_test = np.array(df_.iloc[:, 0:N - 1])
y_test = np.array(df_['label'])
xgb = XGBoost(epoch=100, max_depth=3, learning_rate=0.15,
              gamma=0, Lambda=1, seed=123, subsample=1, colsample=1)
xgb.fit(x_train, y_train, eval_set=(x_eval, y_eval), ESR=20)
```

4：1划分训练集与验证集。并通过数据分析去除部分特征。

五、实验中遇到的问题及解决方案

- **模型出现过拟合。**

  参考原论文，设置行采样、列采样与早停机制。

- **调参时，模型对参数敏感度过高。**

  参考原论文shrinkage思想，为叶子节点权重设置学习率。

- **早停机制效果不明显。**

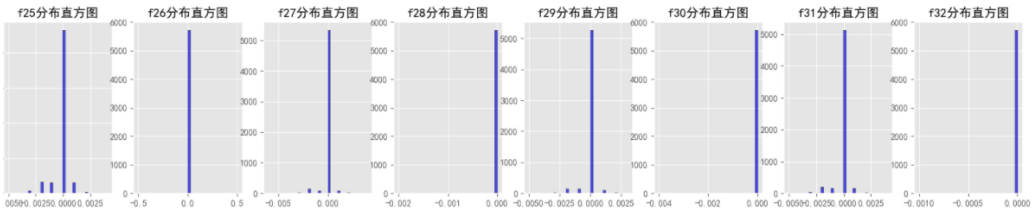
  参考网上相关实现与原论文，出现早停时保存损失最低的迭代次数，后续预测时采用该次数作为迭代次数。

- **结点划分运算时间过长。**

  使用numpy存储数据，并将划分样本内容改为划分样本下标，通过下标访问样本数据而不对数据做操作，极大减少运行时间。

- **数据维度过高导致复杂度高。**

  考虑PCA降维技术，操作后发现效果很不好。于是对数据本身进行分析，统计每一个特征的分布：



发现某些特征绝大部分数据为空，在节点划分过程中起的作用不大，考虑删去这些特征，尝试后发现效果不错。

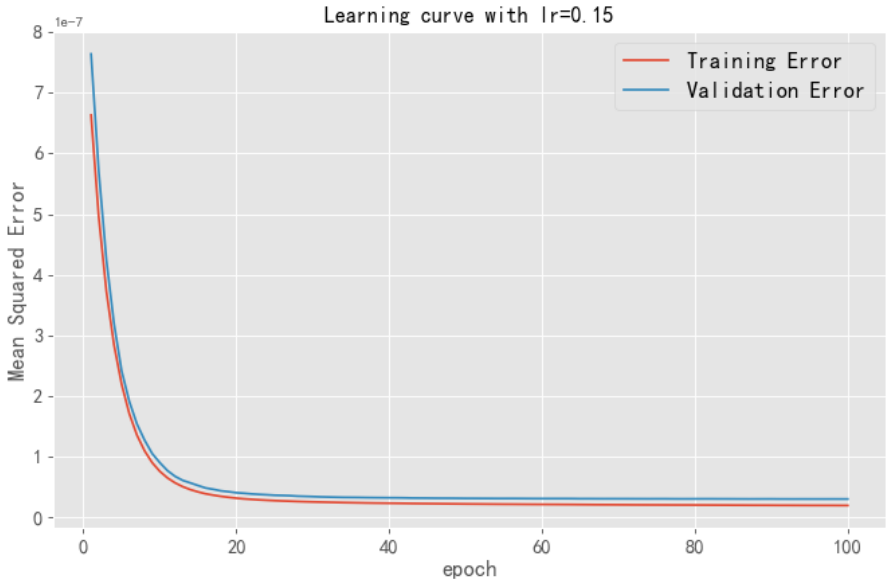
六、实验结果

在ailerons.test测试集上测试结果：

模型参数：

```
xgb = XGBoost(epoch=100, max_depth=3, learning_rate=0.15,
                    gamma=0, Lambda=1, seed=123, subsample=1, colsample=1)
```

损失曲线：



best epoch=97。

	RMSE	R^2
train	0.00014140509637199298	0.8804624284423571
eval	0.0001748888826375329	0.8256333093550854
test	0.0001640125908382933	0.8343408496503996

可以看到曲线下降趋势明显且光滑性良好，并没有出现过拟合现象。且测试集上的$RMSE$与R^2指标与验证集相当，结果良好。

模型对比：

	RMSE	R^2
this_XGBoost	0.0001640125908382933	0.8343408496503996
XGBoost	0.00017280357504902372	0.8161064495730901
Adaboost	0.00017207213974518117	0.817659907814704
GradientBoosting	0.00016356531713787806	0.8352431456233157
Bagging	0.00017226017922043855	0.8172611695913186
RandomForest	0.00017204771517297205	0.8177116682295609

	RMSE	R^2
DecisionTree	0.00024491183547524205	0.630613858481264
Linear	0.00017474453860084234	0.8119521925065031

可见该手动实现XGBoost效果良好。

七、结论

通过理解并实现XGBoost模型，学习到了集成学习的思想，也额外了解到了行采样、列采样、shrinkage等思想。实现的XGBoost 模型在测试集上表现良好。同时，通过不断地改进代码，模型计算效率也很高。