LAB3 XGBoost

袁雨 PB20151804

一、实验目的

实现基模型是决策树的XGBoost。并在数据集上进行训练与测试。

二、实验原理

XGBoost

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 要优化的目标函数为:

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

其中 n 表示训练样本的数量, $penalty(f_k)$ 表示对第 k 个模型的复杂度的惩罚项, $loss(y_i, \hat{y}_i^{(t)})$ 表示损失函数，

例如二分类问题的

$$loss(y_i, \hat{y}_i^{(t)}) = -y_i \cdot \log p(\hat{y}_i^{(t)} = 1|x_i) - (1 - y_i) \log \left(1 - p(\hat{y}_i^{(t)} = 1|x_i)\right)$$

回归问题

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

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

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

其中 $g_i = \frac{\partial\, loss(y_i, \hat{y_i}^{(t-1)})}{\partial\, \hat{y_i}^{(t-1)}}$, $h_i = \frac{\partial^2 loss(y_i, \hat{y_i}^{(t-1)})}{\partial\, (\hat{y_i}^{(t-1)})^2}$，即 g_i 为一阶导数，h_i 为二阶导数。

此时的优化目标变为

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

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

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

决策树（回归树）

本实验中，我们以决策树（回归树）为基，因此还需要写出决策树的算法。

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 $f(x_i) = w_3$。

我们对于某一棵决策树，他的惩罚为

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

其中 γ, λ 为我们可调整的超参数，T 为叶子数，w 为权重向量. 由于显示问题，$\|w\|$ 实际上为 w 的范数，且 $\|w\|^2 = \sum_{i=1}^{dim} w_i^2$

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

综上，我们在树结构确定（你可以自行确定）时，可以进行如下优化：

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

简单起见，我们简记 $G_j = \sum_{i \in I_j} g_i, H_j = \sum_{i \in I_j} h_i$

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

对 w_j 求导，得最优的$w_j^* = -\frac{G_j}{H_j + \lambda}$，带入得$Obj^{(t)} = -\frac{1}{2}\left(\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda}\right) + \gamma T$，也就是每个叶节点得得分 $-\frac{1}{2} \cdot \frac{G_j^2}{H_j + \lambda} + \gamma$ 之和。

构造过程

对于每一棵决策树，即每一个基的训练，我们可以按照以下步骤划分结点

1. 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。
2. 根据划分前后的收益划分结点，收益为

$$Gain = Obj_P - Obj_L - Obj_R = -\frac{1}{2} \cdot \frac{G_j^2}{H_j + \lambda} + \gamma - [-\frac{1}{2}(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda}) + 2\gamma]$$

其中 Obj_P 为父结点的得分，Obj_L, Obj_R 为左右孩子的得分.

3. 选择最大增益进行划分

   选择最大增益的过程如下：

   1. 选出所有可以用来划分的特征集合 $\mathcal{F}$；
   2. For feature in $\mathcal{F}$:

      将节点分配到的样本的特征 feature 提取出来并升序排列，记作 sorted_f_value_list；
      For f_value in sorted_f_value_list：

      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；

      计算划分后的增益；

   返回最大的增益所对应的 feature 和 f_value。

停止策略

对于如何决定一个节点是否还需要继续划分，我们提供下列策略，你可以选择一个或多个，或自行设定合理的策略。

- 划分后增益小于某个阈值则停止划分；
- 划分后树的深度大于某个阈值停止划分；
- 该节点分配到的样本数目小于某个阈值停止分化。

  对于整个算法如何终止，我们提供下列策略，你可以选择一个或多个，或自行设定合理的策略。
- 学习 M 个颗决策树后停下来；
- 当在验证集上的均方误差小于某个阈值时停下来；
- 当验证集出现过拟合时停下来。

评价指标

你可以在实验中以下列指标来验证你的算法效果和不同参数对于结果的影响

- $RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_{test}^{(i)} - \hat{y}_{test}^{(i)})^2}$，越小越好
- $R^2 = 1 - \frac{\sum_{i=1}^{m}(y_{test}^{(i)} - \hat{y}_{test}^{(i)})^2}{\sum_{i=1}^{m}(\bar{y}_{test} - \hat{y}_{test}^{(i)})^2} = 1 - \frac{MSE(\hat{y}_{test}, y_{test})}{Var(y_{test})}$，越大越好
- 运行时间

三、实验步骤

1. **读取并获取数据集信息**

   使用pandas的read_csv来读取数据，因为该数据集没有标签，故设置"head=None"。

   查看数据集信息，可知该数据集有7154 条 41 维的数据，其中前 40 列为 feature，最后一列为 label。均为float或int型，且不含缺失值。

2. 构建模型

主要实现了三个类：`class Node`、`class Decision Tree`、`class XGBoost`。

(1) class Node

`class Node`对应XGBoost中基模型决策树的节点。

```
class Node(object):
    def __init__(self):
        self.left = None  # 左孩子
        self.right = None  # 右孩子
        self.feature = None  # 划分特征
        self.value = None  # 划分值
        self.w = None  # 权值
        self.isleaf = False  # 是否为叶节点
        self.depth = None  # 节点深度
```

(2) class DecisionTree

`class DecisionTree`为XGBoost的基模型，定义如下：

```
class DecisionTree(object):
    def __init__(self, data, gamma, Lambda, max_depth, min_gain, min_num):
        self.gamma = gamma
        self.Lambda = Lambda
        self.max_depth = max_depth
        self.feature = None
        self.value = None
        self.min_gain = min_gain
        self.min_num = min_num
        self.root = self.BuildTree(data, 0)
```

模型参数如上，其中data为$[X, y, y_{t-1}]$组成的数组。

该类中包含以下函数。

```
def GetObj(self, G, H):
    return -0.5 * (G ** 2) / (H + self.Lambda) + self.gamma

def GetWeight(self, data):
    G = -2 * np.sum(data[:, -2] - data[:, -1])
    H = 2 * data.shape[0]
    return -G / (H + self.Lambda)
```

`GetObj(self, G, H)`计算对应的要优化的目标函数，`GetWeight(self, data)`计算最优的权值。

```
def ChooseBestSplit(self, data, depth):
    # 树的深度大于某个阈值，停止划分
    if depth > self.max_depth:
        return None, self.GetWeight(data)

    y_t, y = data[:, -1], data[:, -2]
```

```
    # 所有特征值都相同，停止划分
    if len(set(y.T.tolist())) == 1:
        return None, self.GetWeight(data)

    m, n = np.shape(data)

    # 样本数目小于某个阈值，停止划分
    if m < self.min_num:
        return None, self.GetWeight(data)

    G = np.sum(-2 * (y - y_t))
    H = 2 * m
    Obj_p = self.GetObj(G, H)
    max_gain = float("-inf")
    BestFeature = 0
    BestValue = 0

    # 遍历属性
    for feature in range(n - 2):
        # 特征值排序
        tmp = np.c_[data[:, feature], -2 * (y - y_t)]
        sorted_value_list = tmp[np.argsort(tmp[:, 0])]
        Gl, Gr, Hl, Hr = 0, G, 0, H
        # 遍历特征值
        for i in range(sorted_value_list.shape[0]):
            # <=i,划分到左子树
            Gl += sorted_value_list[i, -1]
            Gr = G - Gl
            Hl += 2
            Hr = H - Hl
            Obj_l = self.GetObj(Gl, Hl)
            Obj_r = self.GetObj(Gr, Hr)
            gain = Obj_p - Obj_l - Obj_r
            if gain > max_gain:
                max_gain = gain
                BestFeature = feature
                BestValue = sorted_value_list[i, 0]

    # 增益小于某个阈值，停止划分
    if max_gain < self.min_gain:
        return None, self.GetWeight(data)

    return BestFeature, BestValue
```

`ChooseBestSplit(self, data, depth)` 遍历data中所有可以用来划分的特征集合。对每个特征，先对特征值进行排序，接着遍历排序后的特征值，以该特征值为分界线将data划分为左右子树，计算划分后的增益，返回最大的增益所对应的 feature 和 value 。

排序时使用tmp数组存储特征值与残差，并使用np.argsort()函数提取排列前对应的索引。第i次遍历时，因为值已经是排好序的，故`Gl += sorted_value_list[i, -1],Gr = G - Gl,Hl += 2,Hr = H - Hl`，大大减少了计算Obj的耗时。

在所有特征值都相同、树的深度大于某个阈值、样本数目小于某个阈值、增益小于某个阈值时，停止划分。

```
def BuildTree(self, data, depth):
```

```
    # 选择最优化划分
    feature, value = self.ChooseBestSplit(data, depth)

    # 满足停止条件，返回叶节点
    if feature == None:
        leaf = Node()
        leaf.depth = depth
        leaf.isleaf = True
        leaf.w = self.GetWeight(data)
        return leaf

    # 划分后赋值
    else:
        root = Node()
        root.depth = depth
        root.feature = feature
        root.value = value
        left = data[np.nonzero(data[:, feature] <= value)[0], :]
        right = data[np.nonzero(data[:, feature] > value)[0], :]
        root.left = self.BuildTree(left, depth + 1)
        root.right = self.BuildTree(right, depth + 1)
        return root
```

`BuildTree(self, data, depth)` 递归创建一棵决策树。若满足停止条件，则返回叶节点。

```
def infer(self, x):
    p = self.root
    while not p.isleaf:
        if x[p.feature] <= p.value:
            p = p.left
        elif x[p.feature] > p.value:
            p = p.right
    return p.w
```

`infer(self, x)` 根据x的feature进行搜索，得到其对应权值。

(3) class XGBoost

XGBoost定义如下：

```
class XGBoost(object):
    def __init__(self, gamma, Lambda, max_depth, tree_num, min_gain,
min_num):
        self.gamma = gamma
        self.Lambda = Lambda
        self.max_depth = max_depth
        self.tree_num = tree_num  # 决策树个数
        self.min_gain = min_gain
        self.min_num = min_num
        self.TreeList = []
```

模型参数如上，其中TreeList保存了得到的决策树。

```
def fit(self, X, y):
    y_t = np.zeros(y.shape)
    data = np.c_[X, y, y_t]
    LossList = []
    for i in range(self.tree_num):
        print(f"Bulid {i + 1} tree")
        tree = DecisionTree(data, self.gamma, self.Lambda, self.max_depth,
self.min_gain, self.min_num)
        self.TreeList.append(tree)
        data[:, -1] = self.predict(X)
        loss = np.mean((y - data[:,-1]) ** 2)
        LossList.append(loss)
    return LossList, data[:, -1]
```

`fit`对模型在X和y上进行训练。初始化y_t为全0，每生成一棵决策树，将其加入TreeList，并使用self.predict(X)，更新y_t。学习num棵树后停止，返回训练过程中的loss列表。

```
def predict(self, X):
    if len(self.TreeList) == 0:
        print("TreeList is empty!")
    else:
        m = X.shape[0]
        y_pred = np.zeros(m)
        for i in range(m):
            for tree in self.TreeList:
                y_pred[i] += tree.infer(X[i, :])
        return y_pred
```

`predict`遍历TreeList，对tree.infer的结果相加，得到X的预测值。

3. 数据集划分

将数据集划分为X_train, X_test, y_train, y_test，其中训练集：测试集 = 7 ： 3。

4. 训练

将模型在X_train、y_train上进行训练，并计算训练过程的loss、耗时、$RMSE$、R^2，画出loss曲线。

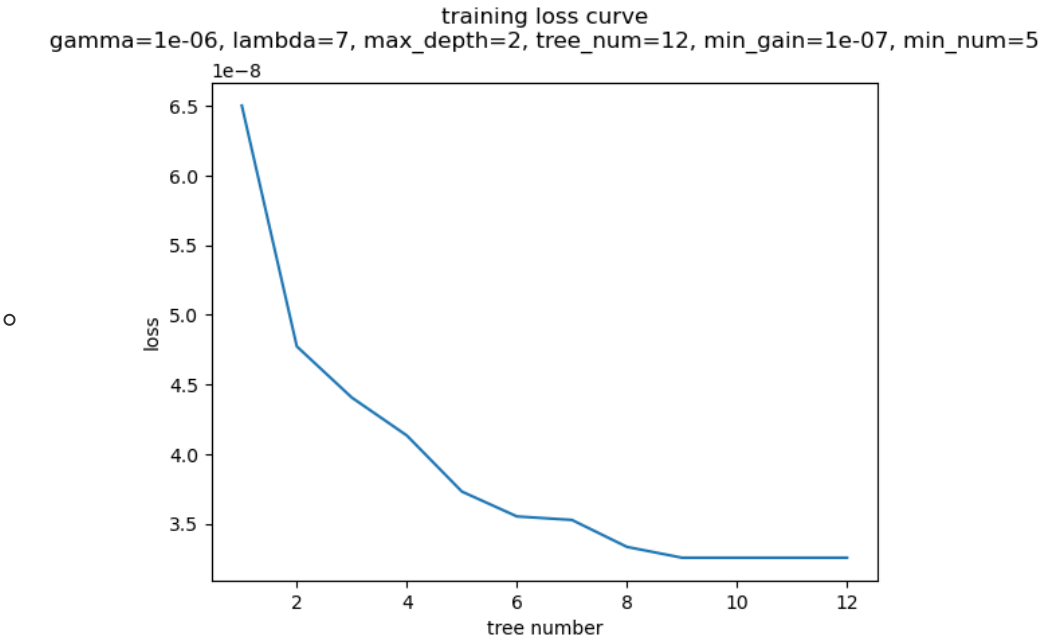
5. 测试

将模型在X_test、y_test上进行测试，并计算测试集的$RMSE$、R^2。

四、实验结果与分析

（1）最佳模型

```
gamma = 1e-6, Lambda = 7, max_depth = 2, tree_num = 12, min_gain = 1e-7, min_num
= 5
```

- 训练耗时：13.44s
- 训练集
  - $RMSE$: 0.0001804
  - R^2 : 0.8120
  - 

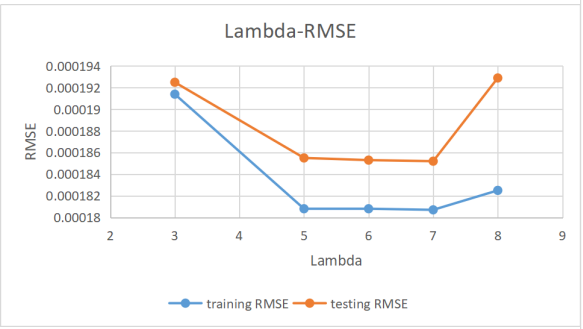
- 测试集
  - $RMSE$: 0.0001845
  - R^2 : 0.7896

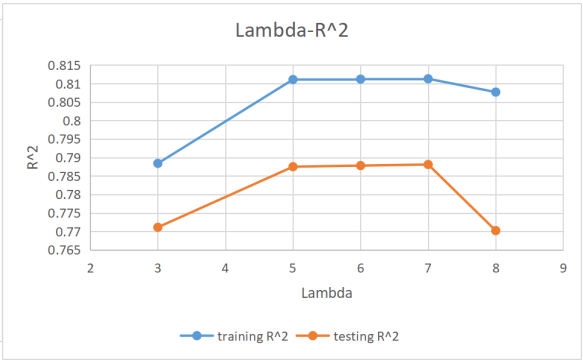
(2) 不同参数的比较

gamma	training $RMSE$	training $RMSE$	training R^2	testing R^2
1e-5	0.0002252	0.0002204	0.7070	0.6998
1e-6	0.0001807	0.0001852	0.8113	0.7881
1e-7	0.0001838	0.0001915	0.8049	0.7733

由表可见在gamma=1e-6左右效果最好。

Lambda	training $RMSE$	testing $RMSE$	training R^2	testing R^2
3	0.0001914	0.0001925	0.7884	0.7711
5	0.0001808	0.0001855	0.8111	0.7875
6	0.0001808	0.0001853	0.8112	0.7878
7	0.0001807	0.0001852	0.8113	0.7881
8	0.0001825	0.0001929	0.8077	0.7702



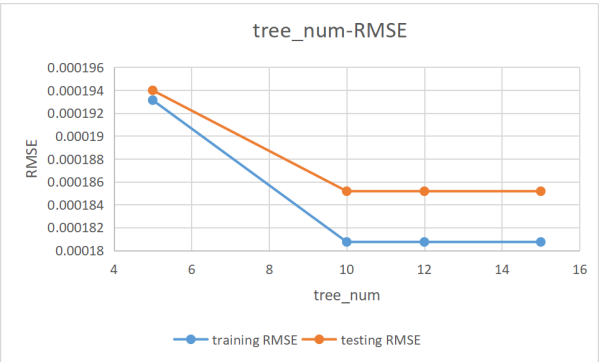


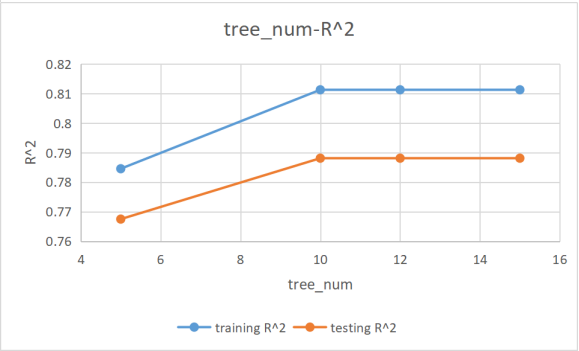
由图表可见，随着Lambda的增加，RMSE先下降后上升，R^2先上升后下降。且测试集上变化明显，产生了过拟合。取Lambda = 7。

max_depth	training $RMSE$	testing $RMSE$	training R^2	testing R^2
1	0.0001928	0.0001906	0.7853	0.7755
2	0.0001807	0.0001852	0.8113	0.7881
3	0.0001891	0.0002051	0.7934	0.7401

由表可见，随着max_depth的增加，RMSE下降后上升，R^2先上升后下降。取max_depth=2。

tree_num	training $RMSE$	testing $RMSE$	training R^2	testing R^2
5	0.000193132	0.000193988	0.784565	0.767480
10	0.000180740	0.000185174	0.811327	0.788130
12	0.000180739	0.000185173	0.811327	0.788131
15	0.000180739	0.000185173	0.811327	0.788131





由图表可见在tree_num>10左右，模型收敛。取tree_num = 12。

min_gain	training $RMSE$	testing $RMSE$	training R^2	testing R^2
1e-6	0.0001925	0.0001955	0.7860	0.7640
1e-7	0.0001804	0.0001845	0.8120	0.7896
1e-8	0.0001807	0.0001852	0.8113	0.7881
1e-9	0.0001807	0.0001852	0.8113	0.7881

由表可见在max_gain=1e-7左右效果最好。

min_num	training $RMSE$	training R^2	testing $RMSE$	testing R^2
2	0.0001807	0.8113	0.0001851	0.7881
5	0.0001807	0.8113	0.0001851	0.7881
10	0.0001807	0.8113	0.0001851	0.7881
100	0.0001817	0.8092	0.0001906	0.7756

由表可见min_num对结果的影响不大，取min_num=5。

（3）与决策树比较

该模型与sklearn里的决策树比较如下：

	Decision Tree	XGBoost
$RMSE$	0.0002027	0.0001845
R^2	0.7460	0.7896

可见集成提升了性能。

五、结果分析

XGBoost是一种高效的梯度提升决策树算法。作为一种前向加法模型，核心是采用boosting思想，将多个弱学习器通过一定的方法整合为一个强学习器。

该数据集不大，故使用XGBoost容易产生过拟合。

可以通过增大λ，减小树的深度和数目等来缓解过拟合。