数据挖掘第二次作业报告

姓名：欧穗新
学号：16340173
作业内容：
实现并行化的 GBDT/RF，完成 kaggle 比赛 Project2 of Data Mining 2019 spring
在这里自己选择实现并行化的 RF（随机森林）完成作业
实验环境：
Windows 环境，pycharm 工具/jupyter 工具完成编码（各有一份代码分别存在 pro2.py 和 pro2.ipynb）
编码语言：python

工作说明：

1. 数据读入处理，实现 Cache 友好
2. 实现并行化决策树（可选并行/不并行）
3. 实现并行化随机森林
4. 使用给出的数据集训练自实现的随机森林、预测 r2 score 得分，调整参数在训练集得到更高的分数
5. 使用训练好的随机森林预测 test.csv 下的数据，并将结果写入文件、提交
6. 实验代码文件位于 pro2.py、pro2.ipynb
   pro2.ipynb 中还附带可视化实验结果
   RF.py 中是自己写的随机森林类的实现，为了让多进程程序在 jupyter 中运行，需要将相应代码放到 py 文件中，然后在 jupyter 中 import，所以自己为了在 jupyter 中展示随机森林运行结果，自己将 pro2.py 中部分代码放到了 RF.py 中，然后在 jupyter 中 import
7. 报告中**红色**标明核心算法原理部分，**加粗**标明实现的关键步骤

数据读入处理，实现 Cache 友好

1. 先将原本的 6 个 train1.csv-train6.csv 合并成一个

```
dfs = []
for i in range(1, 6):
    df = pd.read_csv('./data/train' + str(i) + '.csv', header=None)
    lb = pd.read_csv('./data/label' + str(i) + '.csv', header=None)
    df[13] = lb[0]
    dfs.append(df)

df = pd.concat(dfs)
```

2. 然后随机打乱这 10000000 多行的样本数据
3. 最后将打乱的样本集合以 csv 格式存入磁盘（文件名为 train.csv）

```
df = df.sample(frac=1, random_state=random.randint(0,100))
df.to_csv('./data/train.csv')
```

4. 后续的读取操作都在磁盘文件 train.csv 上按序进行，从而实现 cache 友好

```
df = pd.read_csv('./data/train.csv')
df = df.drop(['Unnamed: 0'],axis=1)
print(df[0:5])
```

说完了操作，下面解释一下 cache 友好有什么用，以及为什么这样做就能够实现 cache 友好：

cache 友好有什么用：

根据计算机体系结构理论知识，我们知道读存取度关系 register>cache>memory>disk，也就是存取速度最快的是寄存器，其次是 cache 缓存，然后是内存，最后是 disk 磁盘，但是存储空间上来说则是反过来的，寄存器容量最小，而磁盘容量最大。为了加快程序运行速度，我们应该充分利用高速设备，因为寄存器我们操作不了（归操作系统管），所以尽量操作一下 cache，但是 cache 虽快，容量却很小，一帧可能就能存几十个条目，如果我们需要访问磁盘上的数据，首先将其从磁盘会读取到内存，如果这段数据最近经常被用到则会从内存加载到 cache，然后从 cache 读到 cpu 寄存器。**如果我们按顺序访问这些数据**，那么同一帧上的几十个数据按序被使用，然后继续操作后面的数据，**每一帧都只会被加载到 cache 中一次**，但是**如果我们因为某些原因不得不随机对这些数据进行访问**，那么访问完这一帧（记为 A）中的某个数据之后，需要另一帧（记为 B）上的数据，这一帧将被另一帧替代，等到下次需要 A 帧的另一个数据，则会再次加载 A 帧，即**每一帧可能反复被加载到 cache 中**，因为访问不是按顺序的，我们可能在很久之后才再次需要同一帧上的其他数据，这时如果该帧已经从 cache 移除，那么就需要重复读入。本来一次读入就能完成，现在因为随机取用，导致需要多次读入，肯定会导致用时加倍。**因此我们应该尽量避免这样的随机访问，尽可能每次都访问 cache/内存/磁盘中位置相近的数据，尽可能按序访问**。

比如本次项目中，一共 10000000 多个样本，自己粗略看了一下，有很多标签值相近/相等的样本被放到一起，按序访问将影响训练效果（可能模型会记住和顺序相关的信息，然后根顺序、位置信息进行预测，这并不是我们想要的，我们希望的是模型根据特征进行预测）。即不得不进行随机访问。

所以为了加快程序的速度，提升性能，可以使用 1-4 步骤实现 cache 友好，即能够完成随机访问，访问的速度也不会慢于按序访问。

下面紧接着解释为什么这样做能够实现 cache 友好：

我们将 10000000 多行的样本数据打乱，然后存到磁盘，之后的读取直接按序读取即可，虽然这些样本在磁盘上是按序排列的，但是这个“按序”的顺序相对原来的样本队列是乱序的（这些样本在相近位置的样本特征、标签都不是相近的，不会影响训练效果）

所以我们最终能够在磁盘/内存上按序读取样本到 cache，并且这种按序访问的效果与乱序访问相当（按序访问乱序的样本队列相当于乱序访问顺序排列的样本队列）

实现并行化决策树（可选并行/不并行）

1. 决策树初始化

   feature_rate 参数控制分支节点时，随机选取的特征数量比例，使得即使给定的样本集合不变，生成的树也不同，增加决策树的多样性，防止过拟合

   max_depth 参数控制树的深度，同样用于防止过拟合；

   min_samples_leaf 参数控制每个节点最小样本树，这个参数可以防止发生过拟合；

   parallel 参数控制并行化，为 True 时并行，使用多进程寻找最佳分割属性/特征，每个进程针对一个特征寻找最佳分割点，返回给主进程，再由主进程选出最佳分支方案----使得子树 mse 估值最低的分割特征+分割点

```
class DecisionTree(object):
    def __init__(self, feature_rate=1.0, max_depth=None, min_leaf_size=1, parallel=False):
        self._x = None
        self._y = None
        self._col_names = None
        self._feature_rate = feature_rate
        self._max_depth = max_depth
        self._min_leaf_size = min_leaf_size
        self._parallel = parallel

        self._split_feature = None
        self._split_point = None
        self._mse_score = None
        self._val = None

        self._left_node = None
        self._right_node = None
```

2. 决策树评估标准

   使用 mse 即整棵树的均方误差，整棵树 mse 数值越低越好，计算方法入下述公式所示，其中x_i是样本点标签值，μ是样本集标签平均值

$$\Sigma \frac{节点样本数}{总样本书} \times (\frac{1}{节点样本数} \Sigma_{i=1}^{节点样本数} (x_i - \mu)^2)$$

```
@staticmethod
def calculate_mse(y_square_sum, y_sum, y_n):
    return (y_square_sum/y_n) - (y_sum**2)/(y_n**2)
```

   上面用到了一个概率论的简单计算来简化编码：

$$E((x-\mu)^2) = E(x^2) + E(\mu^2) - 2E(x \times \mu) = E(x^2) + \mu^2 - 2\mu \times E(x) = E(x^2) + \mu^2 - 2\mu^2$$

3. **并行/串行生成单棵随机决策树**（并行/串行通过参数 parallel 控制）

   输入参数 x，y，x 是样本集特征，y 是样本标签集，x 与 y 按位置对应

   采用 dfs 的方式生成一颗决策树，每次为当前节点寻找最佳分支方案，然后按顺序由子节点生成子树，后代节点分支完毕时返回到父节点。树达到限定深度/节点中样本数到达设定下限/继续分支无法优化决策树时将停止分支，下面是单个节点分支的步骤：

   a) **随机选取若干样本特征**作为候选集，选出其中最佳的分支特征，随机选取特征保证每次生成的决策树都会不同，增加森林的多样性，避免过拟合

```
# 随机选取feature_rate*col_nums个属性（避免过拟合），选出其中最优的
feature_num = int(self._feature_rate*len(self._col_names))
features = np.random.choice(len(self._col_names), feature_num)
```

b) **并行/遍历寻找各个特征的最佳分割点，此处可以有并行化操作，即开多个线程/进程，每个线程/进程寻找单个特征的最佳分割点**，然后将找到的分割点返回主进程，主进程比较各个特征对应的最佳分支方案（分支特征+分割点），选出最佳方案（找出使得决策树 mse 估值最低的方案，也就是使得本节点 mse 估值最低的方案）

每个并行进程/每次迭代中的操作如下：

i. **将该节点所有样本按照选中的特征的值排序，排序后每个不同的特征值都可以作为候选分隔值**，比如一组排序后的特征值 1，2，2，4，5 中，分割点有 1，2，4，5 这四个不同的值。注意这里排序的是样本点的特征值，而不是标签值，因为我们要根据特征值来预测。

ii. **遍历每个候选分割点，寻找最佳分割点**，将样本集分为两部分，由于样本集已经按照选定特征值排序了，所以被分开的两部分就是队列中分割点前面的部分、分割点后面的部分，计算这两部分样本标签的加权均方差，与当前最优分割方案对应均方差比较，如果前者小于后者，说明找到了更好的分割方案，记录该分割属性、分割点、分割后整体的均方差数值。遍历完成后将最佳分割方案返回主进程。注意这里用来计算均方差的是样本点的标签值，因为我们的训练目标是找出一棵拥有尽量小的标签均方差的树。

寻找最佳分割点代码：

```
def find_best_split_point(self, feature):
    # 根据选定特征feature寻找最佳分割点

    points = list(np.argsort(self._x.iloc[:, feature]))
    start = self._x.iloc[points[self._min_leaf_size-1], feature]

    y_square_sum = np.power(self._y, 2).sum()
    y_sum = self._y.sum()
    y_n = len(self._y)
    left_square_sum = np.power(self._y[0:self._min_leaf_size], 2).sum()
    left_sum = self._y[0:self._min_leaf_size].sum()
    left_n = self._min_leaf_size
    right_square_sum = y_square_sum - left_square_sum
    right_sum = y_sum - left_sum
    right_n = y_n - left_n

    tmp_split_feature = None
    tmp_split_point = None
    tmp_mse_score = self._mse_score
    for i in range(self._min_leaf_size, len(self._y)-self._min_leaf_size+1):
        xi, yi = self._x.iloc[points[i], feature], self._y[points[i]]
        tmp = 0
        if xi == start:
            tmp = 1
        if tmp == 0:
            left_score = self.calculate_mse(left_square_sum, left_sum, left_n)
            right_score = self.calculate_mse(right_square_sum, right_sum, right_n)
            score_after_split = left_score*(left_n/y_n) + right_score*(right_n/y_n)
            # 必须要比当前的mse score小才能分裂，不然就有可能出现y都一样但是仍然分裂的状况，因为此时score是相等的
            if score_after_split < tmp_mse_score:
                tmp_split_feature = feature
                tmp_split_point = xi
                tmp_mse_score = score_after_split
        left_square_sum = left_square_sum + yi ** 2
        left_sum = left_sum + yi
        left_n = left_n + 1
        right_square_sum = right_square_sum - yi ** 2
        right_sum = right_sum - yi
        right_n = right_n - 1
        start = xi
    return tmp_split_feature, tmp_split_point, tmp_mse_score
```

串行方案中，遍历随机选出的若干个样本特征，对每个特征寻找最佳分割点后选出各个“最佳分割点+对应分割特征”中最好的方案，代码如下：

```
else:
    for feature in features:
        tmp_split_feature, tmp_split_point, tmp_mse_score = self.find_best_split_point(feature)
        if tmp_mse_score < self._mse_score:
            self._split_feature = tmp_split_feature
            self._split_point = tmp_split_point
            self._mse_score = tmp_mse_score
```

并行方案中，随机选出多少个特征，就开多少个进程，每个进程负责查找一个特征对应的最佳分割点，各个进程完成自己工作后，返回主进程，主进程拿到各个子进程的工作成果，从中选取使得节点分裂后 mse 值最低的方案，代码如下：

```
if self._parallel:
    self.find_split_parallel(features)
else:
```

```
def find_split_parallel(self, features):
    # 并行化寻找最佳分割点
    workers = cpu_count()
    try:
        workers = min(workers, len(features))
    except Exception:
        print('please input correct n_job')
    pool = Pool(processes=workers)
    result = []
    for feature in features:
        result.append(pool.apply_async(self.find_best_split_point, (feature, )))
    # 关闭进程池，等待子进程退出后，拿出结果
    pool.close()
    pool.join()
    # 由于开启不同进程训练树时会发生tree变量的拷贝（而不是引用），所以最后还需要将结果值赋给trees
    for res in result:
        item = res.get()
        if item[2] < self._mse_score:
            self._split_feature = item[0]
            self._split_point = item[1]
            self._mse_score = item[2]
```

c) **上一步 b 中找出了最佳分裂方案**，我们将这个方案对应的标签均方差与本节点的标签均方差比较（即不分裂情况下的 mse），**如果前者不小于后者**，说明选定的分割特征中**不存在分裂方案能够继续优化**该决策树，应该停止分支，返回父节点（即该节点成为叶节点）

至于不存在更优方案原因有几个，如下面注释所言

```
# 如果是寻找最优属性过程中，没能够完成分裂（原因很多种，可能是min_leaf_size限制、mse无法减小限制等）
if (self._split_feature is None) or (self._split_point is None):
    return self
```

i. 可能 min_leaf_size 限制，该节点如果再分裂则子节点样本数不足 min_leaf_size，所以不能继续分裂

ii. mse 无法减小限制，即该节点继续分裂，mse 估值无法减小，继续分裂也没有意义

iii. 当然还有可能是收到 max_depth 限制，到达限定深度，不能继续分裂，当然这在之前的代码中就处理了，但是这里说到不能分裂了，就附带提一下：

```
# 如果到达一定深度，停止分裂
if (self._max_depth is not None) and self._max_depth < 2:
    return self
```

否则存在分裂方案能够继续优化该决策树，那么就**使用该最佳分裂方案（分割特征+分割点），分裂该节点，并递归分裂子节点**：所有样本中分割特征小于分割点的分为一类记为 A 集合，其他分为一类记为 B 集合，然后构建左右两颗子树，并使用 A 样本集合生成左子树，B 样本集合生成右子树，按顺序开始各个子节点（子树）的分支操作----进入左子节点的训练，回到步骤 a)，继续分裂，直到完全生成一棵子树，所有子节点都完成分裂或者不能继续分裂，返回父节点，由父节点再进入右子节点的训练，回到步骤 a)重复操作，等到右子节点训练完成，返回父节点，从父节点返回祖先节点……直到整棵树训练完毕。

代码如下：

分成 A、B 两个集合：

```
# 选出最优的分割属性、分割点后分裂节点，递归生成子节点
left_nodes = np.nonzero(np.array(self._x.iloc[:, self._split_feature]) < self._split_point)[0]
right_nodes = np.nonzero(np.array(self._x.iloc[:, self._split_feature]) >= self._split_point)[0]
```

递归生成、训练左右子节点：

```
max_depth = self._max_depth - 1 if self._max_depth is not None else None
self._left_node = DecisionTree(feature_rate=self._feature_rate, max_depth=max_depth,
                               min_leaf_size=self._min_leaf_size)
self._left_node.fit(self._x.iloc[left_nodes], self._y[left_nodes])
self._right_node = DecisionTree(feature_rate=self._feature_rate, max_depth=max_depth,
                                min_leaf_size=self._min_leaf_size)
self._right_node.fit(self._x.iloc[right_nodes], self._y[right_nodes])
```

4. **使用生成的决策树预测样本**

预测过程就是一个深度优先搜索的过程，这里我通过**递归查询二叉树**实现预测：

a) **待预测样本分割特征值小于当前节点的分割点，就递归调用左子节点预测该样本**

b) **否则递归调用右子节点预测该样本**

c) **直到到达叶子节点，此时返回节点全部样本标签均值，作为预测值**，返回的数值一直向上传递直到根节点，返回给调用处（尾递归）。

上面的“分割特征”指的是当前节点分支方案中的分支特征。

```
def predict(self, x):
    # 预测决策树的参数如下：
    # x，类型为array或者dataFrame，其中的每一项表示一个预测对象的特征
    if type(x) != np.ndarray:
        x = np.array(x)
    return [self.predict_one(one) for one in x]

def predict_one(self, one):
    if (self._split_feature is None) or (self._split_point is None):
        return self._val
    if one[self._split_feature] < self._split_point:
        return self._left_node.predict_one(one)
    return self._right_node.predict_one(one)
```

5. 测试/使用方法示范：

a) 使用参数构造决策树：

```
dt = DecisionTree(feature_rate=0.4, min_leaf_size=5)
```

b) 训练决策树

```
In [9]: dt.fit(df.iloc[0:100000, 0:13], np.array(df.iloc[0:100000, 13]))
```

```
Out[9]: sample size: 100000      value: -0.0027179999999999995   mse score: 0.3426780942286538
        split feature: 7         split point: 0.020381
```

c) 使用决策树预测样本

```
In [14]: dt.predict([[0,2,3,4,5,1,2,3,4,5,1,2,3], [21,22,3,-114,-5,1,2,1,4,5,1,2,3]])
```

```
Out[14]: [1.0250000000000001, 0.62]
```

实现并行化随机森林

1. 随机森林初始化

   以下是森林的参数

   n_job 参数控制并行程度，表示使用的并行进程个数，n_job 是多少，就是用多少个并行进程生成决策树，默认为 None，此时不使用并行，只有一个进程来生成决策树

   n_tree 参数控制森林规模，表示最终森林中决策树的数量

   sample_size 参数控制输入每棵树的样本数量，从原始样本中进行有放回采样的采样次数

   以下是每棵树的初始化参数

   feature_rate 参数控制每棵决策树的 feature_rate 参数，保证即使输入决策树的样本相同，产生的决策树也不同，从而使每棵树都是“随机决策树”

   min_leaf_size 参数控制每棵决策树的 min_leaf_size

   max_depth 参数控制决策树的 max_depth 即最大深度

```
class RandomForest(object):
    def __init__(self, feature_rate=1.0, max_depth=None, min_leaf_size=1, n_job=None, n_tree=1, sample_size=None):
        self._feature_rate = feature_rate
        self._max_depth = max_depth
        self._min_leaf_size = min_leaf_size
        self._n_job = n_job
        self._n_tree = n_tree
        self._sample_size = sample_size

        self._trees = [DecisionTree(self._feature_rate, self._max_depth, self._min_leaf_size) for i in
                       range(self._n_tree)]
        self._x = None
        self._y = None
```

2. **并行训练随机森林**

   a) 并行是可选项，若未设置 n_jobs，则遍历森林中的树逐个训练，否则并行训练：

```
def fit(self, x, y):
    # 训练森林，参数说明如下
    # x，类型为dataFrame，列数为特征数
    # y，类型为dataFrame，列数为1
    self._x = x
    self._y = y
    if self._n_job:
        self.fit_tree_parallel()
    else:
        for tree in self._trees:
            self.fit_one_tree(tree, self._trees.index(tree))
```

   b) **并行训练若干棵树**

      i. 首先创建 workers=n_jobs 的进程池

      ii. 然后每个 worker 都开始各自训练、生成一棵决策树，一旦某个 worker 完成一棵树的训练，马上开始下一棵树的训练

      iii. 所有的树训练完毕后，遍历进程池取出所有训好的树，释放进程池

      代码如下：

```
def fit_tree_parallel(self):
    # 并行训练森林中的树
    workers = cpu_count()
    try:
        workers = min(workers, self._n_job)
    except Exception:
        print('please input correct n_job')
    pool = Pool(processes=workers)
    result = []
    for tree in self._trees:
        result.append(pool.apply_async(self.fit_one_tree, (tree, self._trees.index(tree))))
    # 关闭进程池，等待子进程退出后，进行赋值
    pool.close()
    pool.join()
    # 由于开启不同进程训练树时会发生tree变量的拷贝（而不是引用），所以最后还需要将结果值赋给trees
    self._trees = [res.get() for res in result]
```

c) **单棵树的训练步骤**:

i. 首先从样本中进行**有放回随机抽样**，抽出 sample_size 个样本下标

ii. 为了实现 cache 友好，**将抽样出的下标进行排序**，然后再根据下标按序获得对应的样本及其标签（避免随机下标访问导致大量 cache 页重复读取）

iii. 最后**使用上述抽样样本对该树进行训练**

使用随机采样获得的样本集合进行训练，进一步增加了每一棵决策树的多样性，避免过拟合。

```
def fit_one_tree(self, tree, index):
    # 训练单颗树的参数如下：
    # tree，单棵树的引用，但被其他进程调用时是拷贝
    # index，树的序号
    start = datetime.datetime.now()
    print('fit the {} th tree'.format(index))
    # 首先进行有放回的随机筛选，选出sample_size个样本
    sample_indexes = np.sort(np.random.choice(len(self._x), self._sample_size))
    # 然后调用DecisionTree.fit进行训练
    tree.fit(self._x.iloc[sample_indexes], self._y.iloc[sample_indexes])
    print('{} th tree fit over, time used: {}'.format(index, (datetime.datetime.now()-start).seconds))
    return tree
```

3. **使用训练好的随机森林预测样本**

采取森林决策树投票的方式预测样本，遍历随机森林的每一棵随机决策树，调用其 predict 方法，然后**将获得的 n_trees 个数取平均值作为预测结果**

```
def predict(self, x):
    # 预测决策树的参数如下：
    # x，类型为dataFrame，其中的每一项表示一个预测对象的特征
    all_predictions = [tree.predict(x) for tree in self._trees]
    return np.mean(all_predictions, axis=0)
```

4. 测试/使用方法示范：

a) 使用参数构造决策树：

b) 训练决策树

```
__name__ == "__main__":
freeze_support()
rf = RF.RandomForest(feature_rate=0.3, max_depth=32, min_leaf_size=2, n_job=4, n_tree=10, sample_size=50000)
rf.fit(df.iloc[0:100000, 0:13], df.iloc[0:100000, 13])
```

c) 使用决策树预测样本

```
rf.predict([[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3], [1, 2, 3, 14, 5, 1, 2, 1, 4, 5, 1, 2, 3]])
```

```
array([0.29666667, 0.29666667])
```

d) 预测 r2 score 得分

R2 score 参考维基百科定义实现：按照公式计算出 SSR 和 SST，然后取 score=1-

SSR/SST

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

于是可以得到总平方和

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2,$$

回归平方和

$$SS_{\text{reg}} = \sum_i (f_i - \bar{y})^2,$$

残差平方和

$$SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2$$

由此，决定系数可定义为

$$R^2 \equiv 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}.$$

```
def r2_score(preds, labels):
    assert len(preds) == len(labels)
    y_mean = np.array(labels).mean()
    SSR = (np.power(np.array(preds) - np.array(labels), 2)).sum()
    SST = (np.power(np.array(labels) - y_mean, 2)).sum()
    print('score', 1-SSR/SST)
```

这里使用训练好的随机森林预测 100000 个训练样本（训练也是用这 100000 个样本），我们看到，不调任何参数并且森林规模很小（n_tree=10）的时候，训练效果很差，只有 0.21 分

```
r2_score(rf.predict(df.iloc[0:100000, 0:13]), np.array(df.iloc[0:100000, 13]))
```

```
score 0.2095079674884449
```

实验（调参&参数分析）

1. 决策树参数调整对训练结果的影响
   基础参数为 n_job=8, n_tree=16, sample_size=50000, feature_rate=0.3, min_leaf_size=2, max_depth=32，样本数为 100000，后面的 a，b，c，d，e 在此基础上分别调整 n_job、n_tree、sample_size、min_leaf_szie、max_depth 的值，每次仅仅改变一个参数来控制变量
   a) n_jobs 参数：

N_job	1	2	4	8
训练用时（单位 s）	5211.57	2743.13	1508.72	891.51

从上述结果明显看到并行化加快了训练的用时，**虽然不是按比例减少，但是仍然符合并行进程越多，省时越多**，之所以无法完全按照并行进程数量加快相应倍数，有其他原因，个人认为很重要的一个原因是自己的电脑在进行实验的同时还在使用诸如 qq、浏览器等程序，这样一定程度影响了其中一些进程的 cpu 占用，比如 n_jobs=8 时，自己电脑只有 8 个虚拟核心，即使什么都不做，也有系统程序在运

行，所以各个进程的 cpu 占用受到影响，不如 n_job=2 时单个进程的 cpu 占用（因为还有 6 个核心可以提供给其他程序）。n_job=1 和 n_job=2 时用时接近 2 倍也能说明也能说明这一点（这两个进程分别可以占满两个核心）

最终方案中选择 n_job=8，这是自己电脑最大核心数，再增大 n_job 也是无效的

b) n_tree 参数：

N_tree	16	50	100
训练用时	891.51	2948.86	5486.26
得分	0.1478	0.3639	0.9415

N_tree 越大，训练用时相应越长，得分越高，最终的训练方案中应该根据自己计算机算力情况尽量选择最大的森林规模（最后自己选了 n_tree=200，因为 n_tree=400 的时候内存炸了，程序崩溃，需要重新训练，只得减小 n_tree 数值）

c) sample_size 参数

sample_size	100000	80000	50000	20000
训练用时	2263.08	1714.45	891.51	274.31
得分	0. 2825	0.2639	0.1478	0.0778

Sample_size 越大训练得分越高，但是训练时间越长

sample_size 本应该越大越好，但是计算资源限制+问题规模过大，只能够设置得小一些，以加快计算速度

d) feature_rate 参数

根据自己的尝试，这个参数太小了会导致欠拟合，如果太大训练速度太慢，所以最终决定设置为 0.3

e) min_leaf_size 参数

根据自己的尝试，这个参数应该调整的大一些，加快训练速度（不然训练速度太慢了），最终方案中设置为 50

f) max_depth 参数

max_depth	None	32	16
训练用时	1097.2532	891.51	822.93
得分	0.2899	0.1478	0.0759

max_depth 越大，得分越高（但是容易过拟合），参数值太小容易导致欠拟合，分数很低，比如这里 depth=16，仅有 0.075 分，所以不应该太小

这个参数可以用于避免过拟合，但是这个问题中更应该考虑欠拟合，但是考虑到计算资源有限，自己最终设置改参数为 32，即每棵树的最大深度为 32

2. 使用/不使用并行策略训练单棵决策树的比较

在使用多个进程训练单棵决策树的时候，用时远多于仅用单线程训练，推测是因为进程创建和释放的开销超过了并行化所节约的计算时间。所以最后的方案中自己仅仅采用了并行化生成多棵树，每棵树训练时并未采取并行

注意到我们的并行化操作----每个进程负责查找使用特定特征分裂节点时的最佳分割点，如果树的规模非常大，比如有几百上千个节点，每次分裂一个节点都需要开启 int(13*0.3)=3 个进程来做最佳分割方案查找，那么就需要反复创建、释放进程成百上千次，这带来了极大的额外开销，这些开销甚至超过了计算本身，所以即使并行化减少了计算用时，但是带来了更多的创建、释放进程的用时，得不偿失。

因为进程创建、释放带来巨大开销，所以这里其实应该避开使用进程，而用线程实现并行化，线程创建销毁、切换简单，速度极快，如果线程创建带来的开销小于并行化减少

的计算用时，那么并行化在此处就有用武之地了，但是因为自己此次试验采用 python 进行编程，python 的多线程 thread 并不能实现真正的计算并行，python 的 thread 适用于 io 密集型编程（比如服务器并发访问等），不适用于本题的计算密集型编程，所以无法尝试使用多线程并行化单棵决策树，即使用 python 的多线程写了相应的代码，也不会有速度提升，因为 python 多线程无法利用计算机 cpu 的多个核心。
可以参考这个博客 https://www.cnblogs.com/stubborn412/p/4033651.html
而更为底层的语言 c\c++自然能够做到，下次有机会再做 c/c++的随森实现尝试

3. 使用 cache 优化操作前后的用时比较
使用拥有 40 棵树的随机森林，针对 100000 条数据进行训练，其他参数分别为 n_job=8，sample_size=50000，feature_rate=0.3，min_leaf_size=2，max_depth=32，训练样本数 100000
在为每棵树做有放回抽样，选取训练样本时将选取的样本索引排序后获取样本用时：

```
fit the 37 th tree
34 th tree fit over, time used: 205.73497500000002
fit the 38 th tree
36 th tree fit over, time used: 137.15665
fit the 39 th tree
35 th tree fit over, time used: 205.73497500000002
37 th tree fit over, time used: 137.15665
38 th tree fit over, time used: 137.15665
39 th tree fit over, time used: 137.15665
all th tree fit over, time used: 2194.5064
score 0.086980909543641 35

Process finished with exit code 0
```

最终用时单位是 s

不排序直接按照随机索引获取样本用时：

```
34 th tree fit over, time used: 137.15665
fit the 36 th tree
36 th tree fit over, time used: 137.15665
fit the 37 th tree
35 th tree fit over, time used: 137.15665
fit the 38 th tree
37 th tree fit over, time used: 205.73497500000002
38 th tree fit over, time used: 137.15665
fit the 39 th tree
39 th tree fit over, time used: 137.15665
all th tree fit over, time used: 3771.8078750000004
score 0.10288354152725243

Process finished with exit code 0
```

最终用时单位是 s

从结果上看，使用 cache 优化之后，用时减少了一倍，这还是在样本数比较小，随机访问导致的分页切换频率较低的情况下，所以如果是使用全部 1000 万多条样本进行训练，优化应该不止一倍

预测&提交

经过上面的实验、参数调整，最终方案如下：

1. 构建一个拥有 200 棵树的随机森林；
2. 使用 8 个并行的进程进行训练（自己电脑一共 8 个虚拟核心，开更多进程电脑容易卡死）；
3. 有放回随机采样样本数 2000000 用于训练每棵树（占总训练样本数约 1/5）；
4. 每棵树最大深度为 32；
5. 叶子节点样本数不小于 50；
6. 搜索最佳分支方案时选取的特征数占比 0.3（本问题一共 13 个特征，所以节点分支时随机选取 int(13*0.3)=3 个特征）；

其实自己并不是根据实验现象来决定参数，更多的是根据计算资源限制来确定参数，尽可能加快计算出结果的速度，因为问题规模实在太大+计算资源仅有自己的 PC。

使用如上参数初始化森林，然后使用全部的训练样本
最后使用训练好的随机森林遍历 test1.csv-test6.csv，预测每一个样本，加入到 preds 数组。
最后按照指定格式 id、Predicted 写入 sub.csv 并提交。

```
import RF
from multiprocessing import Pool, cpu_count, freeze_support
# 这里其实是有输出提示（当前正在生成第几棵树，用时多少，但是由于代码在RF.py中，因此只能在jupyter的命令行那里看到）
if __name__ == "__main__":
    freeze_support()
    rf = RF.RandomForest(feature_rate=0.3, max_depth=32, min_leaf_size=50, n_job=8, n_tree=200, sample_size=2000000)
    rf.fit(df.iloc[:, 0:13], df.iloc[:, 13])

    preds = []
    for i in range(1, 7):
        test_data = pd.read_csv('./data/test'+str(i)+'.csv', header=None)
        tmp = rf.predict(test_data)
        preds.extend(tmp)

    ids = np.arange(1, len(preds)+1)
    res = pd.DataFrame({'id':ids, 'Predicted':preds})
    res.to_csv('./sub.csv', index=False)
```

```
submission = pd.read_csv('./sub.csv')
print(submission)
```

```
          id  Predicted
0          1  -0.181492
1          2  -0.123607
2          3  -0.156439
3          4  -0.210649
4          5  -0.116018
5          6  -0.042638
6          7  -0.043169
```