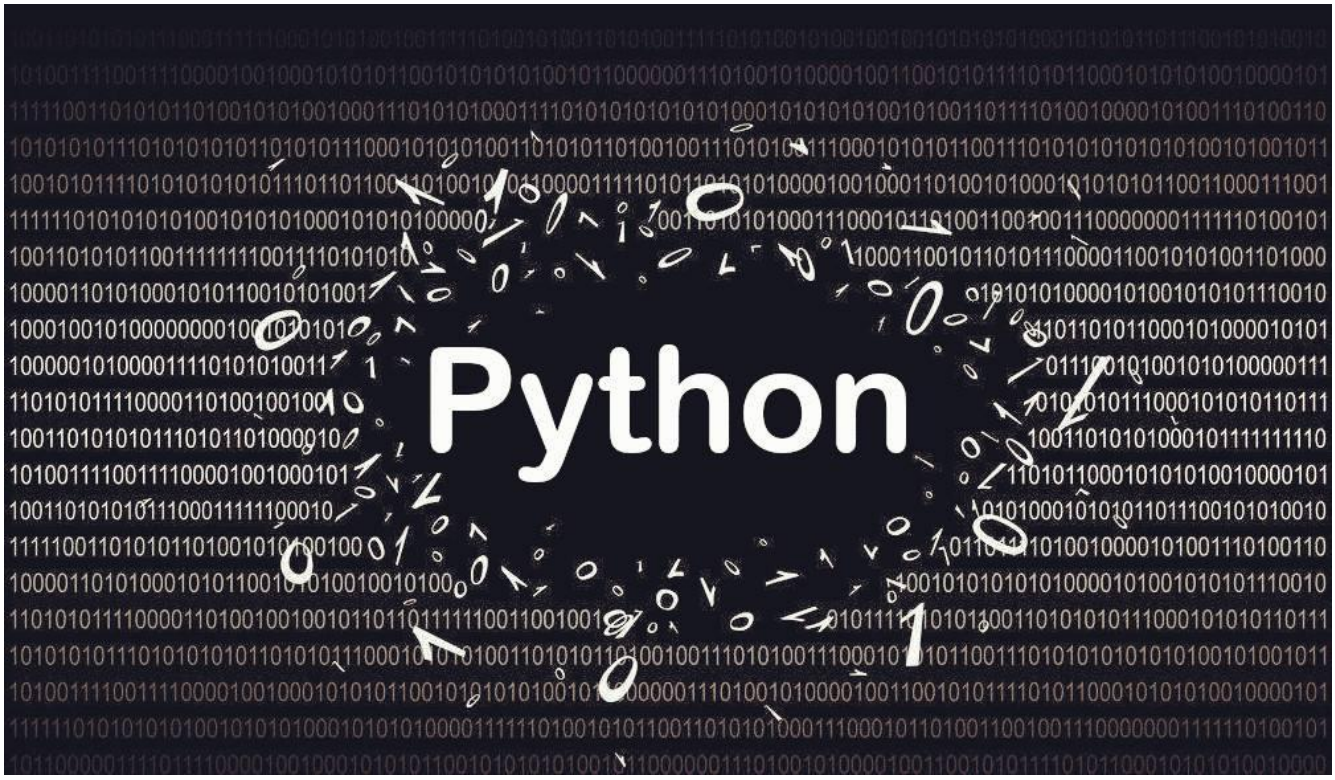
用 Python 实现随机森林算法



拥有高方差使得决策树（secision tress）在处理特定训练数据集时其结果显得相对脆弱。bagging（bootstrap aggregating 的缩写）算法从训练数据的样本中建立复合模型，可以有效降低决策树的方差，但树与树之间有高度关联（并不是理想的树的状态）。

随机森林算法（Random forest algorithm）是对 bagging 算法的扩展。除了仍然根据从训练数据样本建立复合模型之外，随机森林对用做构建树（tree）的数据特征做了一定限制，使得生成的决策树之间没有关联，从而提升算法效果。

本文章旨在探讨如何用 Python 实现随机森林算法。通过本文，我们可以了解到：

- bagged decision trees 与随机森林算法的差异；
- 如何构建含更多方差的装袋决策树；
- 如何将随机森林算法运用于预测模型相关的问题。

算法描述

随机森林算法

决策树运行的每一步都涉及到对数据集中的最优分裂点（best split point）进行贪婪选择（greedy selection）。

这个机制使得决策树在没有被剪枝的情况下易产生较高的方差。整合通过提取训练数据库中不同样本（某一问题的不同表现形式）构建的复合树及其生成的预测值能够稳定并降低这样的高方差。这种方法被称作引导聚集算法（bootstrap aggregating），其简称 bagging 正好是装进口袋，袋子的意思，所以被称为「装袋算法」。该算法的局限在于，由于生成每一棵树的贪婪算法是相同的，那么有可能造成每棵树选取的分裂点（split point）相同或者极其相似，最终导致不同树之间的趋同（树与树相关联）。相应地，反过来说，这也使得其会产生相似的预测值，降低原本要求的方差。

我们可以采用限制特征的方法来创建不一样的决策树，使贪婪算法能够在建树的同时评估每一个分裂点。这就是随机森林算法（Random Forest algorithm）。

与装袋算法一样，随机森林算法从训练集里撷取复合样本并训练。其不同之处在于，数据在每个分裂点处完全分裂并添加到相应的那棵决策树当中，且可以只考虑用于存储属性的某一固定子集。

对于分类问题，也就是本教程中我们将要探讨的问题，其被考虑用于分裂的属性数量被限定为小于输入特征的数量之平方根。代码如下：

```
num_features_for_split = sqrt(total_input_features)
```

这个小更改会让生成的决策树各不相同（没有关联），从而使得到的预测值更加多样化。而多样的预测值组合往往会比一棵单一的决策树或者单一的装袋算法有更优的表现。

声纳数据集（Sonar dataset）

我们将在本文里使用声纳数据集作为输入数据。这是一个描述声纳反射到不同物体表面后返回的不同数值的数据集。60 个输入变量表示声纳从不同角度返回的强度。这是一个二元分类问题（binary classification problem），要求模型能够区分出岩石和金属柱体的不同材质和形状，总共有 208 个观测样本。

该数据集非常易于理解——每个变量都互有连续性且都在 0 到 1 的标准范围之间，便于数据处理。作为输出变量，字符串'M'表示金属矿物质，'R'表示岩石。二者需分别转换成整数 1 和 0。通过预测数据集（M 或者金属矿物质）中拥有最多观测值的类，零规则算法（Zero Rule Algorithm）可实现 53% 的精确度。

此次教程分为两个步骤。

- **分裂次数的计算。**
- **声纳数据集案例研究**

这些步骤能让你了解为你自己的预测建模问题实现和应用随机森林算法的基础。

分裂次数的计算

在决策树中，我们通过找到一些特定属性和属性的值来确定分裂点，这类特定属性需表现为其所需的成本是最低的。

分类问题的成本函数（cost function）通常是基尼指数（Gini index），即计算由分裂点产生的数据组的纯度（purity）。对于这样二元分类的分类问题来说，指数为 0 表示绝对纯度，说明类值被完美地分为两组。

从一棵决策树中找到最佳分裂点需要在训练数据集中对每个输入变量的值做成本评估。在装袋算法和随机森林中，这个过程是在训练集的样本上执行并替换（放回）的。因为随机森林对输入的数据要进行行和列的采样。对于行采样，采用有放回的方式，也就是说同一行也许会在样本中被选取和放入不止一次。我们可以考虑创建一个可以自行输入属性的样本，而不是枚举所有输入属性的值以期找到获取成本最低的分裂点，从而对这个过程进行优化。该输入属性样本可随机选取且没有替换过程，这就意味着在寻找最低成本分裂点的时候每个输入属性只需被选取一次。

如下的代码所示，函数 get*split() 实现了上述过程。它将一定数量的来自待评估数据的输入特征和一个数据集作为参数，该数据集可以是实际训练集里的样本。辅助函数 test*split() 用于通过候选的分裂点来分割数据集，函数 gini_index() 用于评估通过创建的行组（groups of rows）来确定的某一分裂点的成本。

以上我们可以看出，特征列表是通过随机选择特征索引生成的。通过枚举该特征列表，我们可将训练集中的特定值评估为符合条件的分裂点。

```
# Select the best split point for a dataset
def get_split(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1)
        if index not in features:
            features.append(index)
    for index in features:
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups
    return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

至此，我们知道该如何改造一棵用于随机森林算法的决策树。我们可将之与装袋算法结合运用到真实的数据集当中。

关于声纳数据集的案例研究

在这个部分，我们将把随机森林算法用于声纳数据集。本示例假定声纳数据集的 csv 格式副本已存在于当前工作目录中，文件名为 sonar.all-data.csv。

首先加载该数据集，将字符串转换成数字，并将输出列从字符串转换成数值 0 和 1. 这个过程是通过辅助函数 load*csv()*、*str*column*to*float() 和 str*column*to_int() 来分别实现的。

我们将通过 K 折交叉验证（k-fold cross validatio）来预估得到的学习模型在未知数据上的表现。这就意味着我们将创建并评估 K 个模型并预估这 K 个模型的平均误差。评估每一个模型是

由分类准确度来体现的。辅助函数 cross*validation*split()、accuracy*metric() 和 evaluate*algorithm() 分别实现了上述功能。

装袋算法将通过分类和回归树算法来满足。辅助函数 test*split() 将数据集分割成不同的组；gini*index() 评估每个分裂点；前文提及的改进过的 get*split() 函数用来获取分裂点；函数 to*terminal()、split() 和 build*tree() 用以创建单个决策树；predict() 用于预测；subsample() 为训练集建立子样本集； bagging*predict() 对决策树列表进行预测。

新命名的函数 random_forest() 首先从训练集的子样本中创建决策树列表，然后对其进行预测。

正如我们开篇所说，随机森林与决策树关键的区别在于前者在建树的方法上的小小的改变，这一点在运行函数 get_split() 得到了体现。

完整的代码如下：

```
# Random Forest Algorithm on Sonar Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# 加载 CSV 文件
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
```

```
    csv_reader = reader(file)
    for row in csv_reader:
        if not row:
            continue
        dataset.append(row)
  return dataset

# 将字符串列转换为 float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# 将字符串列转换为整数
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# 将数据集拆分成 k 个折叠
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = len(dataset) / n_folds
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
        index = randrange(len(dataset_copy))
        fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split
```

```python
# 计算精度百分比
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

# 使用交叉验证分割来评估算法
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores

# 基于属性和属性值拆分数据集
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right
```

```python
# 计算分割数据集的基尼系数
def gini_index(groups, class_values):
  gini = 0.0
  for class_value in class_values:
    for group in groups:
      size = len(group)
      if size == 0:
        continue
      proportion = [row[-1] for row in group].count(class_value) / float(size)
      gini += (proportion * (1.0 - proportion))
  return gini

# 选择数据集的最佳分割点
def get_split(dataset, n_features):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  features = list()
  while len(features) < n_features:
    index = randrange(len(dataset[0])-1)
    if index not in features:
      features.append(index)
  for index in features:
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# 创建终端节点值
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)

# 为节点或终端创建子分割
```

```python
def split(node, max_depth, min_size, n_features, depth):
    left, right = node['groups']
    del(node['groups'])
    # 检查不分裂
    if not left or not right:
        node['left'] = node['right'] = to_terminal(left + right)
        return
    # 检查最大深度
    if depth >= max_depth:
        node['left'], node['right'] = to_terminal(left), to_terminal(right)
        return
    # 处理左孩子
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left, n_features)
        split(node['left'], max_depth, min_size, n_features, depth+1)
    # 过程权的孩子
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
        node['right'] = get_split(right, n_features)
        split(node['right'], max_depth, min_size, n_features, depth+1)

# 构建决策树
def build_tree(train, max_depth, min_size, n_features):
    root = get_split(dataset, n_features)
    split(root, max_depth, min_size, n_features, 1)
    return root

# 用决策树进行预测
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
```

```
        return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

# 从替换的数据集创建一个随机子样本
def subsample(dataset, ratio):
    sample = list()
    n_sample = round(len(dataset) * ratio)
    while len(sample) < n_sample:
        index = randrange(len(dataset))
        sample.append(dataset[index])
    return sample

# 用袋装树木列表进行预测
def bagging_predict(trees, row):
    predictions = [predict(tree, row) for tree in trees]
    return max(set(predictions), key=predictions.count)

# 随机森林算法
def random_forest(train, test, max_depth, min_size, sample_size, n_trees, n_features):
    trees = list()
    for i in range(n_trees):
        sample = subsample(train, sample_size)
        tree = build_tree(sample, max_depth, min_size, n_features)
        trees.append(tree)
    predictions = [bagging_predict(trees, row) for row in test]
    return(predictions)

# 测试随机森林算法
seed(1)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
```

```
# convert string attributes to integers
for i in range(0, len(dataset[0])-1):
  str_column_to_float(dataset, i)
# 将类列转换为整数
str_column_to_int(dataset, len(dataset[0])-1)
# evaluate algorithm
n_folds = 5
max_depth = 10
min_size = 1
sample_size = 1.0
n_features = int(sqrt(len(dataset[0])-1))
for n_trees in [1, 5, 10]:
  scores = evaluate_algorithm(dataset, random_forest, n_folds, max_depth, min_size, sample_si
ze, n_trees, n_features)
  print('Trees: %d' % n_trees)
  print('Scores: %s' % scores)
     print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

这里对第 197 行之后对各项参数的赋值做一个说明。

将 K 赋值为 5 用于交叉验证，得到每个子样本为 208/5 = 41.6，即超过 40 条声纳返回记录会用于每次迭代时的评估。

每棵树的最大深度设置为 10，每个节点的最小训练行数为 1. 创建训练集样本的大小与原始数据集相同，这也是随机森林算法的默认预期值。

我们把在每个分裂点需要考虑的特征数设置为总的特征数目的平方根，即 sqrt(60)=7.74，取整为 7。

将含有三组不同数量的树同时进行评估，以表明添加更多的树可以使该算法实现的功能更多。

最后，运行这个示例代码将会 print 出每组树的相应分值以及每种结构的平均分值。如下所示：

Trees: 1

Scores: [68.29268292682927, 75.60975609756098, 70.73170731707317, 63.41463414634146, 65.85365853658537]

Mean Accuracy: 68.780%

Trees: 5

Scores: [68.29268292682927, 68.29268292682927, 78.04878048780488, 65.85365853658537, 68.29268292682927]

Mean Accuracy: 69.756%

Trees: 10

Scores: [68.29268292682927, 78.04878048780488, 75.60975609756098, 70.73170731707317, 70.73170731707317]

Mean Accuracy: 72.683%