11+RandomForest

2019 年 3 月 23 日

1 随机森林

1.1 一、概念

随机森林 = bagging+ 决策树

随机森林的思想非常朴素，就是 ‘‘三个臭皮匠顶个诸葛亮’’。

1.1.1 1. bagging 抽样

bagging 的全称为 bootstrap aggregating。

bootstrap 是统计学中常用的抽样方法，简单而言就是从所有样本中**有放回地抽取一部分样本**进行实验，重复多次进行。

抽样好理解，是为了计算简便和节省时间空间，而有放回则是为了避免数据具有 ‘‘片面性’’。有放回的抽样会反复学习数据中特定的特征，而淡化数据的噪声。

譬如 100 个来自标准正态分布的样本中有 1 个离群值，影响到了这 100 个样本的均值使其偏离了原点。假设极端情况下无放回地抽取 10 次，每次抽取 10 个样本，则无放回的抽取再计算均值进行平均和直接计算所有样本的均值是等价的。而如果是有放回的抽取，则有一定的可能（$0.9^10 = 0.349$）不会抽取到离群值。

而 aggregating 则意味着我们将 bootstrap 抽样的结果结合起来，这种结合有简单的做平均，也有对样本进行加权再进行平均。

1.1.2 2. 决策树

此时的决策树称为随机树其实更为合适。回忆决策树部分中我们提到对于固定的样本，决策树也是确定的，因为每一步都在寻找最优的划分，针对一个固定的样本各种划分肯定也是固定的。

而随机森林中的树则相对随机，主要体现在四个方面：

- 样本随机：用来训练每棵树的样本都是不完全相同的
- 变量随机：用来训练每颗树的变量都是从所有变量中随机抽取的一部分变量
- 划分变量随机：随机森林中不要求划分的变量一定是最优的变量
- 划分节点随机：选定划分变量之后划分节点也不要求是最优的

可以见的这样训练出来的一棵决策树性能好不到哪里去，可能正确率只有 70%。但是如果有上百棵这样的树进行预测超过半数的树预测正确的概率却是可观的，也就是所谓 “三个臭皮匠顶个诸葛亮”。

1.1.3 3. 集成学习

类似于随机森林这样将很多个学习器组合起来的方法称为集成学习方法。

集成学习方法将很多个弱学习器组合在一起，获得能和强学习器媲美的性能。

集成学习方法仅适合用于弱学习器，像此前我们提过的逻辑回归等方法都是强学习器。将强学习器组合起来的收效是微乎甚微的，因为每个强学习器学习到的参数或者分布相差并不大。拿逻辑回归举例，可能我们用 bootstrap 抽样每次用一部分学习到的分离超平面是不同的，但是多个学习器学习到的超平面进行平均之后很可能和用全部数据学习到的超平面相差无几。

1.2 二、应用

这次的数据集和分类树一样用的是汽车数据集。

我们先把之前的分类树整合成一个类，以便后续 bagging 的时候调用。

```
In [2]: import copy
        import numpy as np
        import pandas as pd
        from scipy import stats

        title = ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety', 'acc']
        car = pd.read_csv('data_set/car.data', header=None, names=title)
        car = car.replace(['more', '5more'], 6)
        car['doors'] = car['doors'].astype('int')
        car['persons'] = car['persons'].astype('int')
        n = car.shape[0]

        np.random.seed(2099)
        index = np.random.permutation(n)
        train_index = index[0: int(0.7 * n)]
        test_index = index[int(0.7 * n): n]

        labels = np.unique(car['acc'])
```

```
In [3]: class Tree(object):

            def __init__(self, data, check=True):
                self.tree = pd.DataFrame({
                    'split_variable': [None],
                    'split_value': [None],
                    'discrete': [None],
                    'left_prediction': [None],
                    'right_prediction': [None]
                })
                tmp = data.dtypes
                tmp_set = {"continuous": [], "discrete": []}
                for n, i in enumerate(tmp):
                    if (i == "float32") or (i == "float64") or (i == "int32") or (i == "int64"):
                        tmp_set["continuous"].append(data.columns[n])
                    else:
                        tmp_set["discrete"].append(data.columns[n])
                self.variable_set = {
                    '1': tmp_set
                }
                if check:
                    print("make sure you get the variable type right")
                    print(self.variable_set)

            def fit(self, data, target, index, max_layer=5):
                try:
                    self.variable_set["1"]["continuous"].remove(target)
                except ValueError:
                    self.variable_set["1"]["discrete"].remove(target)

                def gini(data, labels, index, target):
                    data = data.loc[index]
                    n = data.shape[0]
                    g = 1
                    for label in labels:
                        p = np.sum(data[target] == label) / n
                        g -= p * p
                    return g

                def conditional_gini(data, labels, split_variable, split_value, target, discrete=T
                    n = data.shape[0]
                    if discrete:
                        index1 = data[data[split_variable] == split_value].index
                        index2 = data[data[split_variable] != split_value].index
                        g1 = gini(data=data, labels=labels, index=index1, target=target)
                g2 = gini(data=data, labels=labels, index=index2, target=target)
                g = len(index1) / n * g1 + len(index2) / n * g2
                return g
            else:
                index1 = data[data[split_variable] <= split_value].index
                index2 = data[data[split_variable] > split_value].index
                g1 = gini(data=data, labels=labels, index=index1, target=target)
                g2 = gini(data=data, labels=labels, index=index2, target=target)
                g = len(index1) / n * g1 + len(index2) / n * g2
                return g

        def split_val(data, variable, target, discrete=True):
            labels = np.unique(data[target])
            g = 1
            if discrete:
                values = np.unique(data[variable])
            else:
                values = data[variable].copy()
            split_value = None
            for value in values:
                tmp = conditional_gini(data, labels, variable, value, target, discrete)
                if tmp < g:
                    g = tmp
                    split_value = value
                else:
                    continue
            return g, split_value

        def split_var(data, target, variable_set):
            g = 1
            to_split_value = None
            to_split_variable = None
            tmp_gini = None
            tmp_value = None
            var_type = None
            for dtype in variable_set:
                discrete = dtype == 'discrete'
                for variable in variable_set[dtype]:
                    tmp_gini, tmp_value = split_val(data, variable, target, discrete)
                    if g > tmp_gini:
                        g = tmp_gini
                        to_split_value = tmp_value
                        to_split_variable = variable
                        var_type = discrete

        return g, to_split_variable, to_split_value, var_type

def build_tree(data, index, variable_set, tree, target, max_layer, node=1):
    """
    data: all the data
    index: the data on which shall be splited, data won't change in recursion but
    variable_set: variables to be split
    tree: a dataframe with node,split_variable,split_value,discrete,leaf,left and
    target: the label variable
    node: the node of the tree
    """
    tmp = data.iloc[index]
    leaf = len(variable_set[str(node)]['discrete']) == 0
    leaf = leaf and len(variable_set[str(node)]['continuous']) == 0
    leaf = leaf or len(np.unique(tmp[target])) == 1
    print(str(node) + ":" + str(leaf))
    if not leaf:
        g, variable, value, discrete = split_var(tmp, target, variable_set[str(nod
        if discrete:
            variable_set[str(node)]['discrete'].remove(variable)
            variable_set[str(node * 2)] = copy.deepcopy(variable_set[str(node)])
            variable_set[str(node * 2 + 1)] = copy.deepcopy(variable_set[str(node)

            index1 = tmp[tmp[variable] == value].index
            left_prediction = stats.mode(data.iloc[index1][target])[0][0]
            index2 = tmp[tmp[variable] != value].index
            right_prediction = stats.mode(data.iloc[index2][target])[0][0]
        else:
            variable_set[str(node)]['continuous'].remove(variable)
            variable_set[str(node * 2)] = copy.deepcopy(variable_set[str(node)])
            variable_set[str(node * 2 + 1)] = copy.deepcopy(variable_set[str(node)

            index1 = tmp[tmp[variable] <= value].index
```

```
                            left_prediction = stats.mode(data.iloc[index1][target])[0][0]
                            index2 = tmp[tmp[variable] > value].index
                            right_prediction = stats.mode(data.iloc[index2][target])[0][0]

                        tree.loc[node] = [variable, value, discrete, left_prediction, right_predic
                        if node < 2 ** max_layer - 1:
                            build_tree(data=data, index=index1, variable_set=variable_set,
                                       tree=tree, target=target, max_layer=max_layer, node=node *
                            build_tree(data=data, index=index2, variable_set=variable_set,
                                       tree=tree, target=target, max_layer=max_layer, node=node *

                build_tree(data, index, self.variable_set, self.tree, target, max_layer)

        def predict(self, data):
            n = data.shape[0]
            prediction = pd.Series()
            for i in range(n):
                tmp = data.iloc[i]
                leaf = False
                node = 1
                while not leaf:
                    variable = self.tree.loc[node]['split_variable']
                    discrete = self.tree.loc[node]['discrete']
                    if discrete:
                        left = tmp[variable] == self.tree.loc[node]['split_value']
                    else:
                        left = tmp[variable] <= self.tree.loc[node]['split_value']
                    if left:
                        prediction.loc[i] = self.tree.loc[node]['left_prediction']
                        node = node * 2
                    else:
                        prediction.loc[i] = self.tree.loc[node]['right_prediction']
                        node = node * 2 + 1
                    leaf = not (node in self.tree.index)
            return prediction
```

这里加了一个树的深度（层数）的参数并且不用手动设置变量类别了，可以用 *check=False* 参数来取消。

```
In [4]: tree = Tree(car)
```

```
make sure you get the variable type right
```

```
{'1': {'continuous': ['doors', 'persons'], 'discrete': ['buying', 'maint', 'lug_boot', 'safety', '
```

```
In [5]: tree.fit(data=car, index=train_index, target='acc', max_layer=10)
```

```
1:False


c:\users\37922\appdata\local\programs\python\python36\lib\site-packages\ipykernel_launcher.py:36:
c:\users\37922\appdata\local\programs\python\python36\lib\site-packages\scipy\stats\stats.py:248:
  "values. nan values will be ignored.", RuntimeWarning)


2:True
3:False
6:True
7:False
14:False
28:False
56:False
112:True
113:True
57:False
114:True
115:True
29:False
58:False
116:True
117:True
59:False
118:True
119:True
15:False
30:False
60:False
120:True
121:True
61:False
122:True
123:True
31:False
62:False
```

```
124:True
125:True
63:False
126:True
127:True
```

```
In [6]: tree.tree
```

```
Out[6]:    split_variable split_value discrete left_prediction right_prediction
        0            None        None     None            None             None
        1         persons           2    False           unacc            unacc
        3          safety         low     True           unacc              acc
        7           maint       vhigh     True           unacc              acc
        14         buying         med     True             acc            unacc
        28       lug_boot       small     True           unacc              acc
        56          doors           3    False             acc            unacc
        57          doors           2    False             acc              acc
        29       lug_boot       small     True           unacc            unacc
        58          doors           2    False           unacc            unacc
        59          doors           2    False           unacc            unacc
        15         buying         low     True             acc              acc
        30       lug_boot       small     True             acc            vgood
        60          doors           2    False             acc              acc
        61          doors           2    False             acc            vgood
        31       lug_boot       small     True           unacc              acc
        62          doors           2    False           unacc              acc
        63          doors           3    False             acc              acc
```

```
In [7]: prediction = tree.predict(car.iloc[test_index])
        prediction.index = test_index

        rate = np.sum(prediction == car.iloc[test_index]['acc'])/len(prediction)
        print('the accuracy rate is: '+str(rate))
```

```
the accuracy rate is: 0.8304431599229287
```

可以看到这里的结果和之前基本一致。

现在我们需要稍微修改一下选取变量和划分点的方法，并且在应用时让最深层数小于变量数，以便让这棵树随机起来。

```
In [12]: class Tree(object):

             def __init__(self, data, check=True):
                 self.tree = pd.DataFrame({
                     'split_variable': [None],
                     'split_value': [None],
                     'discrete': [None],
                     'left_prediction': [None],
                     'right_prediction': [None]
                 })
                 tmp = data.dtypes
                 tmp_set = {"continuous": [], "discrete": []}
                 for n, i in enumerate(tmp):
                     if (i == "float32") or (i == "float64") or (i == "int32") or (i == "int64"):
                         tmp_set["continuous"].append(data.columns[n])
                     else:
                         tmp_set["discrete"].append(data.columns[n])
                 self.variable_set = {
                     '1': tmp_set
                 }
                 if check:
                     print("make sure you get the variable type right")
                     print(self.variable_set)

             def fit(self, data, target, index, max_layer=5):
                 try:
                     self.variable_set["1"]["continuous"].remove(target)
                 except ValueError:
                     self.variable_set["1"]["discrete"].remove(target)

                 global_gini = 1

                 def gini(data, labels, index, target):
                     data = data.loc[index]
                     n = data.shape[0]
                     g = 1
                     for label in labels:
                         p = np.sum(data[target] == label) / n
                         g -= p * p
```

```
    return g

def conditional_gini(data, labels, split_variable, split_value, target, discrete=
    n = data.shape[0]
    if discrete:
        index1 = data[data[split_variable] == split_value].index
        index2 = data[data[split_variable] != split_value].index
        g1 = gini(data=data, labels=labels, index=index1, target=target)
        g2 = gini(data=data, labels=labels, index=index2, target=target)
        g = len(index1) / n * g1 + len(index2) / n * g2
        return g
    else:
        index1 = data[data[split_variable] <= split_value].index
        index2 = data[data[split_variable] > split_value].index
        g1 = gini(data=data, labels=labels, index=index1, target=target)
        g2 = gini(data=data, labels=labels, index=index2, target=target)
        g = len(index1) / n * g1 + len(index2) / n * g2
        return g

def split_val(data, variable, target, pre_gini, discrete=True):
    labels = np.unique(data[target])
    if discrete:
        values = np.unique(data[variable])
    else:
        values = np.array(data[variable])
    split_value = None
    np.random.shuffle(values)
    for value in values:
        tmp = conditional_gini(data, labels, variable, value, target, discrete)
        if tmp < pre_gini:
            pre_gini = tmp
            split_value = value
            break
        else:
            continue
    return pre_gini, split_value

def split_var(data, target, variable_set, pre_gini=global_gini):
    to_split_value = None
```

```
        to_split_variable = None
        tmp_gini = None
        tmp_value = None
        var_type = np.random.randint(2)
        if var_type:
            try:
                var = np.random.randint(len(variable_set["discrete"]))
            except ValueError:
                return split_var(data, target, variable_set, pre_gini)
            variable = variable_set['discrete'][var]
            tmp_gini, tmp_value = split_val(data, variable, target, pre_gini, var_typ
            if pre_gini > tmp_gini:
                pre_gini = tmp_gini
                to_split_value = tmp_value
                to_split_variable = variable
        else:
            try:
                var = np.random.randint(len(variable_set["continuous"]))
            except ValueError:
                return split_var(data, target, variable_set, pre_gini)
            variable = variable_set['continuous'][var]
            tmp_gini, tmp_value = split_val(data, variable, target, pre_gini, var_typ
            if pre_gini > tmp_gini:
                pre_gini = tmp_gini
                to_split_value = tmp_value
                to_split_variable = variable
        if tmp_gini is None:
            return split_var(data, target, variable_set, pre_gini)
        else:
            return pre_gini, to_split_variable, to_split_value, var_type

    def build_tree(data, index, variable_set, tree, target, max_layer, pre_gini=globa
        """
        data: all the data
        index: the data on which shall be splited, data won't change in recursion but
        variable_set: variables to be split
        tree: a dataframe with node,split_variable,split_value,discrete,leaf,left and
        target: the label variable
        node: the node of the tree
```

```
            """
            tmp = data.iloc[index]
            leaf = len(variable_set[str(node)]['discrete']) == 0
            leaf = leaf and len(variable_set[str(node)]['continuous']) == 0
            leaf = leaf or len(np.unique(tmp[target])) == 1
            # print(str(node) + ":" + str(leaf))
            if not leaf:
                tmp_gini, variable, value, discrete = split_var(tmp, target, variable_set
                if variable is not None:
                    if discrete:
                        variable_set[str(node)]['discrete'].remove(variable)
                        variable_set[str(node * 2)] = copy.deepcopy(variable_set[str(node
                        variable_set[str(node * 2 + 1)] = copy.deepcopy(variable_set[str(

                        index1 = tmp[tmp[variable] == value].index
                        left_prediction = stats.mode(data.iloc[index1][target])[0][0]
                        index2 = tmp[tmp[variable] != value].index
                        right_prediction = stats.mode(data.iloc[index2][target])[0][0]
                    else:
                        variable_set[str(node)]['continuous'].remove(variable)
                        variable_set[str(node * 2)] = copy.deepcopy(variable_set[str(node
                        variable_set[str(node * 2 + 1)] = copy.deepcopy(variable_set[str(

                        index1 = tmp[tmp[variable] <= value].index
                        left_prediction = stats.mode(data.iloc[index1][target])[0][0]
                        index2 = tmp[tmp[variable] > value].index
                        right_prediction = stats.mode(data.iloc[index2][target])[0][0]

                    tree.loc[node] = [variable, value, discrete, left_prediction, right_p
                    if node < 2 ** max_layer - 1:
                        build_tree(data=data, index=index1, variable_set=variable_set, tr
                                   target=target, max_layer=max_layer, pre_gini=tmp_gini,
                        build_tree(data=data, index=index2, variable_set=variable_set, tr
                                   target=target, max_layer=max_layer, pre_gini=tmp_gini,

        build_tree(data, index, self.variable_set, self.tree, target, max_layer)

    def predict(self, data):
        n = data.shape[0]
        prediction = pd.Series()
        for i in range(n):
            tmp = data.iloc[i]
            leaf = False
            node = 1
            while not leaf:
                variable = self.tree.loc[node]['split_variable']
                discrete = self.tree.loc[node]['discrete']
                if discrete:
                    left = tmp[variable] == self.tree.loc[node]['split_value']
                else:
                    left = tmp[variable] <= self.tree.loc[node]['split_value']
                if left:
                    prediction.loc[i] = self.tree.loc[node]['left_prediction']
                    node = node * 2
                else:
                    prediction.loc[i] = self.tree.loc[node]['right_prediction']
                    node = node * 2 + 1
                leaf = not (node in self.tree.index)
        return prediction
```

修改的部分主要有三个：

1. 在于 *split_var* 函数中，选取变量变为随机，此时可能出现某一类的变量被选取完之后无法抽样的可能，所以套用了一步 try-except 的调用。
2. 在 *split_val* 函数中，将可划分节点进行随机打乱
3. 在 *fit* 函数环境内加入一个局部的基尼系数，防止出现我们划分之后这个划分区间的基尼系数已经足够低，而当我们将这个子区间划分之后基尼系数反而比之前高的情况发生，在决策树的情况下每一步都是最优的划分，我们默认这种情况不会发生（其实也是当时我没有想到...）

```
In [13]: tree = Tree(car, check=False)
         tree.fit(data=car, index=train_index, target='acc', max_layer=5)

         prediction = tree.predict(car.iloc[test_index])
         prediction.index = test_index

         rate = np.sum(prediction == car.iloc[test_index]['acc'])/len(prediction)
         print('the accuracy rate is: '+str(rate))
```

```
c:\users\37922\appdata\local\programs\python\python36\lib\site-packages\ipykernel_launcher.py:38:
```

```
the accuracy rate is: 0.7129094412331407
```

准确率已经显著降下来了，而且这还是大样本情况，是件好事。试一试 bagging 一下。

```
In [ ]: def bagging(data, target, test, batch=256, n_tree=500, max_depth=4):
            forest = {}
            prediction = np.zeros((test.shape[0], 1))
            batch_index = data.sample(batch).index
            for i in range(n_tree):
                tree = Tree(car, check=False)
                tree.fit(data=car, index=batch_index, target=target, max_layer=max_depth)
                forest[i] = tree
                prediction = np.hstack((prediction, np.array(tree.predict(test)).reshape(test.shap
                print("the " + str(i) + "-th tree have been built")
            prediction = np.delete(prediction, 0, axis=1)
            return forest, prediction



        tmp, prediction = bagging(car.iloc[train_index], target='acc', test=car.iloc[test_index])
```

500 棵确实需要训练一段时间...其实耗时最长的并不是训练，而是每棵树对测试集进行预测，我觉得这棵树还能写得更快，这样在 bagging 的时候耗时就会更短。

这里我们的问题是多分类问题，在预测结果的呈现上考虑不一样的任务有不一样的需求，如果我们需要一个结果，那么我们可以对最后的结果逐行取众数。如果我们想要知道（所有分类的）概率值，譬如我们用似然函数评价模型时，那么我们需要统计每一行（也就是每个样本）各个类频次（也就是几百个弱学习器的结果）再做除法。

这里的效果提升并不明显，推测是因为数据有偏差，如果你有兴趣详细看一下 *acc* 变量，你会发现 *'acc'* 和 *'unacc'* 的比例特别大，导致另外两个类别 *'good'*，*'vgood'* 的样本比例较少。这种情况下很有可能我们采样时就会导致入样的 *'good'* 和 *'vgood'* 更少，甚至只有一两个，那么这种情况需要极细地分类才能使分类器产生这样的输出。而我们并不会将所有变量放进模型，这就进一步导致模型的偏差。这种情况需要靠抽样方法进行一定的调整，会在以后给出我了解的一些方法。

（我实在不想跑了要跑好久让我偷个懒吧 orz）

```
In [ ]:
```