

# Random Forest

## 1.Random Forest Algorithm

Bagging是通过bootstrap的方式，从原始的数据集 $D$ 中得到新的 $\hat{D}$ ；然后再使用一些base algorithm对每个 $\hat{D}$ 都得到相应的 $g_t$ ；最后将所有的 $g_t$ 通过投票uniform的形式组合成一个 $G$ ， $G$ 即为我们最终得到的模型。

Decision Tree是通过递归形式，利用分支条件，将原始数据集 $D$ 切割成一个个子树结构，长成一棵完整的树形结构。Decision Tree最终得到的 $G(x)$ 是由相应的分支条件 $b(x)$ 和分支树 $G_c(x)$ 递归组成。

| Bagging   | Decision Tree  |
|---|--|
| <pre>function Bag(<math>\mathcal{D}, \mathcal{A}</math>) For <math>t = 1, 2, \dots, T</math>   ① request size-<math>N'</math> data <math>\tilde{\mathcal{D}}_t</math> by     bootstrapping with <math>\mathcal{D}</math>   ② obtain base <math>g_t</math> by <math>\mathcal{A}(\tilde{\mathcal{D}}_t)</math> return <math>G = \text{Uniform}(\{g_t\})</math></pre> <p>—reduces variance<br/>by voting/averaging</p> | <pre>function DTree(<math>\mathcal{D}</math>) if termination return base <math>g_t</math> else   ① learn <math>b(x)</math> and split <math>\mathcal{D}</math> to     <math>\mathcal{D}_c</math> by <math>b(x)</math>   ② build <math>G_c \leftarrow \text{DTree}(\mathcal{D}_c)</math>   ③ return <math>G(x) =</math>     <math>\sum_{c=1}^C \mathbb{I}[b(x) = c] G_c(x)</math></pre> <p>—large variance<br/>especially if fully-grown</p> |

Bagging和Decision Tree算法各自有一个很重要的特点。

Bagging具有减少不同 $g_t$ 的方差variance的特点。这是因为Bagging采用投票的形式，将所有 $g_t$  uniform结合起来，起到了求平均的作用，从而降低variance。

Decision Tree具有增大不同 $g_t$ 的方差variance的特点。这是因为Decision Tree每次切割的方式不同，而且分支包含的样本数在逐渐减少，所以它对不同的资料 $D$ 会比较敏感一些，从而不同的 $D$ 会得到比较大的variance。

所以说，Bagging能减小variance，而Decision Tree能增大variance。

那么可以使用Bagging的方式把众多的Decision Tree进行uniform结合起来。

这种算法就叫做随机森林（Random Forest），它将完全长成的C&RT决策树通过bagging的形式结合起来，最终得到一个庞大的决策模型。

**random forest (RF) = bagging + fully-grown C&RT decision tree**

Random Forest算法流程图如下所示：

| function RandomForest( $\mathcal{D}$ )   | function DTree( $\mathcal{D}$ )  |
|--|--|
| <pre>For <math>t = 1, 2, \dots, T</math>   ① request size-<math>N'</math> data <math>\tilde{\mathcal{D}}_t</math> by     bootstrapping with <math>\mathcal{D}</math>   ② obtain tree <math>g_t</math> by DTree(<math>\tilde{\mathcal{D}}_t</math>) return <math>G = \text{Uniform}(\{g_t\})</math></pre> | <pre>if termination return base <math>g_t</math> else   ① learn <math>b(x)</math> and split <math>\mathcal{D}</math> to     <math>\mathcal{D}_c</math> by <math>b(x)</math>   ② build <math>G_c \leftarrow \text{DTree}(\mathcal{D}_c)</math>   ③ return <math>G(x) =</math>     <math>\sum_{c=1}^C \mathbb{I}[b(x) = c] G_c(x)</math></pre> |

Random Forest算法的优点主要有三个：

- 1.不同决策树可以由不同主机并行训练生成，效率很高
- 2.随机森林算法继承了C&RT的优点
- 3.将所有的决策树通过bagging的形式结合起来，避免了单个决策树造成过拟合的问题。

- highly **parallel/efficient** to learn
- **inherit pros** of C&RT
- **eliminate cons** of fully-grown tree

以上是基本的Random Forest算法。

Bagging中，通过bootstrap的方法得到不同于 $D$ 的 $D'$ ，使用这些**随机抽取的资料**得到不同的 $g_t$ 。除了随机抽取资料获得不同 $g_t$ 的方式之外，还有另外一种方法，就是**随机抽取一部分特征**。例如，原来有100个特征，现在只从中随机选取30个来构成决策树，那么

每一轮得到的树都由不同的30个特征构成，每棵树都不一样。

假设原来样本维度是 $d$ ，则只选择其中的 $d'$  ( $d' < d$ ) 个维度来建立决策树结构。这类似是一种从 $d$ 维到 $d'$ 维的特征转换，相当于是从高维到低维的投影，也就是说 $d'$ 维 $z$ 空间其实就是 $d$ 维 $x$ 空间的一个随机子空间 (subspace)。通常情况下， $d'$ 远小于 $d$ ，从而保证算法更有效率。

Random Forest算法的作者建议在构建C&RT每个分支 $b(x)$ 的时候，都可以重新选择子特征来训练，从而得到更具有多样性的决策树。

another possibility for **diversity**:

randomly **sample  $d'$  features** from  $\mathbf{x}$

- when sampling index  $i_1, i_2, \dots, i_{d'}: \Phi(\mathbf{x}) = (x_{i_1}, x_{i_2}, \dots, x_{i_{d'}})$
- $\mathcal{Z} \in \mathbb{R}^{d'}: \text{a random subspace of } \mathcal{X} \in \mathbb{R}^d$
- often  $d' \ll d$ , efficient for large  $d$   
—can be generally applied on other models
- original RF **re-sample new subspace for each  $b(x)$  in C&RT**

所以说，这种增强的Random Forest算法增加了random-subspace。

RF = **bagging** + **random-subspace C&RT**

上面我们讲的是随机抽取特征，除此之外，还可以将现有的特征 $x$ ，通过数组 $p$ 进行线性组合，来保持多样性：

$$\phi_i(x) = p_i^T x$$

这种方法使每次分支得到的不再是单一的子特征集合，而是子特征的线性组合（权重不为1）。

好比在二维平面上不止得到水平线和垂直线，也能得到各种斜线。这种做法使子特征选择更加多样性。

不同分支 $i$ 下的 $p_i$ 是不同的，而且向量 $p_i$ 中大部分元素为零，因为我们选择的只是一部分特征，这是一种低维映射。

more **powerful features for diversity**: row  $i$  other than natural basis

- **projection** (combination) with random row  $\mathbf{p}_i$  of  $\mathbf{P}$ :  $\phi_i(\mathbf{x}) = \mathbf{p}_i^T \mathbf{x}$
- often consider **low-dimensional projection**:  
only  $d''$  **non-zero** components in  $\mathbf{p}_i$
- includes **random subspace** as **special case**:  
 $d'' = 1$  and  $\mathbf{p}_i \in \text{natural basis}$
- original RF consider  $d'$  random **low-dimensional projections for each  $b(x)$  in C&RT**

所以，这里的Random Forest算法又有增强，由原来的random-subspace变成了random-combination。

这里的random-combination类似于perceptron模型。

RF = **bagging** + **random-combination C&RT**  
—**randomness everywhere!**

## 2. Out-Of-Bag Estimate

通过bootstrap得到新的样本集 $D'$ ，再由 $D'$ 训练不同的 $g_t$ 。

我们知道 $D'$ 中包含了原样本集 $D$ 中的一些样本，但也有些样本没有涵盖进去。

如下表所示，不同的 $g_t$ 下，红色的表示在 $\hat{D}_t$ 中没有这些样本。

每个 $g_t$ 中，红色表示的样本被称为out-of-bag(OOB) example。

|                       | $g_1$         | $g_2$         | $g_3$         | $\dots$ | $g_T$         |
|-----------------------|---------------|---------------|---------------|---------|---------------|
| $(\mathbf{x}_1, y_1)$ | $\tilde{D}_1$ | *             | $\tilde{D}_3$ |         | $\tilde{D}_T$ |
| $(\mathbf{x}_2, y_2)$ | *             | *             | $\tilde{D}_3$ |         | $\tilde{D}_T$ |
| $(\mathbf{x}_3, y_3)$ | *             | $\tilde{D}_2$ | *             |         | $\tilde{D}_T$ |
| $\dots$               |               |               |               |         |               |
| $(\mathbf{x}_N, y_N)$ | $\tilde{D}_1$ | $\tilde{D}_2$ | *             |         | *             |

计算OOB样本到底有多少。

假设bootstrap的数量 $N' = N$ ，那么某个样本 $(x_n, y_n)$ 是OOB的概率是：

$$\left(1 - \frac{1}{N}\right)^N = \frac{1}{\left(\frac{N}{N-1}\right)^N} = \frac{1}{\left(1 + \frac{1}{N-1}\right)^N} \approx \frac{1}{e}$$

其中,  $e$  是自然对数,  $N$  是原样本集的数量。

由上述推导可得, 每个  $g_t$  中, OOB 数目大约是  $\frac{1}{e} N$ , 即大约有三分之一的样本没有在 bootstrap 中被抽到。

然后, 我们将 OOB 与之前介绍的 Validation 进行对比:

| OOB                   |               |               |               |         |               |
|-----------------------|---------------|---------------|---------------|---------|---------------|
|                       | $g_1$         | $g_2$         | $g_3$         | $\dots$ | $g_T$         |
| $(\mathbf{x}_1, y_1)$ | $\tilde{D}_1$ | *             | $\tilde{D}_3$ |         | $\tilde{D}_T$ |
| $(\mathbf{x}_2, y_2)$ | *             | *             | $\tilde{D}_3$ |         | $\tilde{D}_T$ |
| $(\mathbf{x}_3, y_3)$ | *             | $\tilde{D}_2$ | *             |         | $\tilde{D}_T$ |
| $\dots$               |               |               |               |         |               |
| $(\mathbf{x}_N, y_N)$ | $\tilde{D}_1$ | *             | *             |         | *             |

| Validation  |             |         |             |
|-------------|-------------|---------|-------------|
| $g_1^-$     | $g_2^-$     | $\dots$ | $g_M^-$     |
| $D_{train}$ | $D_{train}$ |         | $D_{train}$ |
| $D_{val}$   | $D_{val}$   |         | $D_{val}$   |
| $D_{val}$   | $D_{val}$   |         | $D_{val}$   |
|             |             |         |             |
| $D_{train}$ | $D_{train}$ |         | $D_{train}$ |

在 Validation 表格中, 蓝色的  $D_{train}$  用来得到不同的  $g_m^-$ , 而红色的  $D_{val}$  用来验证各自的  $g_m^-$ 。  $D_{train}$  与  $D_{val}$  没有交集, 一般  $D_{train}$  是  $D_{val}$  的数倍关系。

再看左边的 OOB 表格, 之前我们也介绍过, 蓝色的部分用来得到不同的  $g_t$ , 而红色的部分是 OOB 样本。刚推导过, 红色部分大约占  $N$  的  $\frac{1}{e}$ 。

如何使用 OOB 来验证 G 的好坏。方法是先看每一个样本  $(x_n, y_n)$  是哪些  $g_t$  的 OOB 资料, 然后计算其在这  $g_t$  上的表现, 最后将所有样本的表现求平均即可。

例如, 样本  $(x_N, y_N)$  是  $g_2, g_3, g_T$  的 OOB, 则可以计算  $(x_N, y_N)$  在  $G_N^-(x)$  上的表现为:

$$G_N^-(x) = \text{average}(g_2, g_3, g_T)$$

像是 Leave-One-Out Cross Validation, 每次只对一个样本进行  $g^-$  的验证一样, 只不过这里选择的是每个样本是哪些  $g_t$  的 OOB, 然后再分别进行  $G_n^-(x)$  的验证。每个样本都当成验证资料一次 (与留一法相同), 最后计算所有样本的平均表现:

$$E_{oob}(G) = \frac{1}{N} \sum_{n=1}^N \text{err}(y_n, G_n^-(x_n))$$

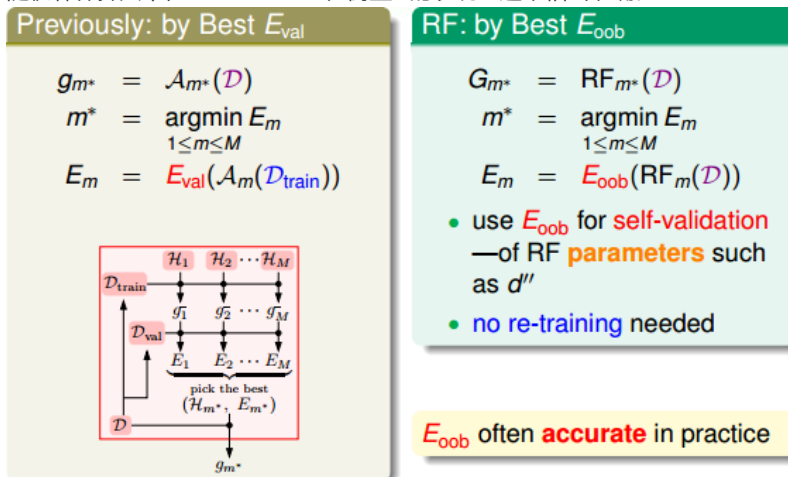
$E_{oob}(G)$  估算的就是 G 的表现好坏。我们把  $E_{oob}$  称为 bagging 或者 Random Forest 的 self-validation。

这种 self-validation 相比于 validation 来说还有一个优点就是它不需要重复训练。

如下图左边所示, 在通过  $D_{val}$  选择到表现最好的  $g_{m^*}$  之后, 还需要在  $D_{train}$  和  $D_{val}$  组成的所有样本集  $D$  上重新对该模型  $g_{m^*}$  训练一次, 以得到最终的模型系数。

但是 self-validation 在调整随机森林算法相关系数并得到最小的  $E_{oob}$  之后, 就完成了整个模型的建立, 无需重新训练模型。

随机森林算法中, self-validation 在衡量 G 的表现上通常相当准确。



### 3.Feature Selection

需要移除的特征分为两类:

一类是冗余特征, 即特征出现重复, 例如“年龄”和“生日”;

另一类是不相关特征, 例如疾病预测的时候引入的“保险状况”。

这种从d维特征到d'维特征的subset-transform  $\Phi(x)$ 称为Feature Selection，最终使用这些d'维的特征进行模型训练。

for  $\mathbf{x} = (x_1, x_2, \dots, x_d)$ , want to remove

- **redundant** features: like keeping one of 'age' and 'full birthday'
- **irrelevant** features: like insurance type for cancer prediction

and only 'learn' **subset-transform**  $\Phi(\mathbf{x}) = (x_{i_1}, x_{i_2}, x_{i_{d'}})$   
with  $d' < d$  for  $g(\Phi(\mathbf{x}))$

| 特征选择的优点               | 特征选择的缺点         |
|-----------------------|-----------------|
| 提高效率，特征越少，模型越简单       | 筛选特征的计算量较大      |
| 正则化，防止特征过多出现过拟合       | 不同特征组合，也容易发生过拟合 |
| 去除无关特征，保留相关性大的特征，解释性强 | 容易选到无关特征，解释性差   |

advantages:

- **efficiency**: simpler hypothesis and shorter prediction time
- **generalization**: 'feature noise' removed
- **interpretability**

disadvantages:

- **computation**: 'combinatorial' optimization in training
- **overfit**: 'combinatorial' selection
- **mis-interpretability**

在decision tree中，decision stump切割方式也是一种feature selection。  
可以通过计算出每个特征的重要性（即权重），然后再根据重要性的排序进行选择即可。

idea: if possible to calculate

**importance(i)** for  $i = 1, 2, \dots, d$

then can select  $i_1, i_2, \dots, i_{d'}$  of top- $d'$  **importance**

这种方法在线性模型中比较容易计算。  
线性模型的score是由每个特征经过加权求和而得到的，而加权系数的绝对值 $|w_i|$ 正好代表了对应特征 $x_i$ 的重要性为多少。 $|w_i|$ 越大，表示对应特征 $x_i$ 越重要，则该特征应该被选择。  
 $w$ 的值可以通过对已有的数据集 $(x_i, y_i)$ 建立线性模型而得到。

**importance by linear model**

$$\text{score} = \mathbf{w}^T \mathbf{x} = \sum_{i=1}^d w_i x_i$$

- intuitive estimate: **importance(i) = |w<sub>i</sub>|** with some 'good' **w**
- getting 'good' **w**: learned from data
- non-linear models? often **much harder**

在非线性模型（Random Forest）下进行特征选择有点困难。

RF中，特征选择的核心思想是random test。  
random test的做法是对于某个特征，如果用另外一个随机值替代它之后的表现比之前更差，则表明该特征比较重要，所占的权重应该较大，不能用一个随机值替代。  
相反，如果随机值替代后的表现没有太大差别，则表明该特征不那么重要，可有可无。  
所以，通过**比较某特征被随机值替代前后的表现**，就能推断出该特征的权重和重要性。

random test中的随机值选择通常有两种方法：  
一是使用uniform或者gaussian抽取随机值替换原特征；  
一是通过permutation的方式将原来的所有N个样本的第i个特征值重新打乱分布（相当于重新洗牌）。  
比较而言，第二种方法更加科学，保证了特征替代值与原特征的分布是近似的（只是重新洗牌而已）。这种方法叫做permutation test（随机排序测试），即在计算第i个特征的重要性的时候，将N个样本的第i个特征重新洗牌，然后比较D和 $D^{(p)}$ 表现的差异性。如果差异很大，则表明第i个特征是重要的。

- which random values?
  - uniform, Gaussian, ...:  $P(x_i)$  changed
  - bootstrap, **permutation** (of  $\{x_{n,i}\}_{n=1}^N$ ):  $P(x_i)$  approximately remained
- **permutation test**:
 
$$\text{importance}(i) = \text{performance}(\mathcal{D}) - \text{performance}(\mathcal{D}^{(p)})$$

with  $\mathcal{D}^{(p)}$  is  $\mathcal{D}$  with  $\{x_{n,i}\}$  replaced by **permuted**  $\{x_{n,i}\}_{n=1}^N$

知道了permutation test的原理后，接下来要考虑的问题是如何衡量上图中的performance，即替换前后的表现。

performance可以用  $E_{\text{oob}}(G)$  来衡量。

但是，对于N个样本的第i个特征值重新洗牌重置的  $\mathcal{D}^{(p)}$ ，要对它进行重新训练，而且每个特征都要重复训练，然后再与原D的表现进行比较，过程非常繁琐。

为了简化运算，RF的作者提出了一种方法，就是把permutation的操作从原来的training上移到了OOB validation上去，记为  $E_{\text{oob}}(G^{(p)}) \rightarrow E_{\text{oob}}^{(p)}(G)$ 。

也就是说，在训练的时候仍然使用D，但是在OOB验证的时候，将所有的OOB样本的第i个特征重新洗牌，验证G的表现。这种做法大大简化了计算复杂度，在RF的feature selection中应用广泛。

- **performance( $\mathcal{D}^{(p)}$ )**: needs re-training and **validation** in general
- **'escaping' validation? OOB in RF**
- original RF solution:  $\text{importance}(i) = E_{\text{oob}}(G) - E_{\text{oob}}^{(p)}(G)$ , where  $E_{\text{oob}}^{(p)}$  comes from replacing each request of  $x_{n,i}$  by a **permuted OOB** value

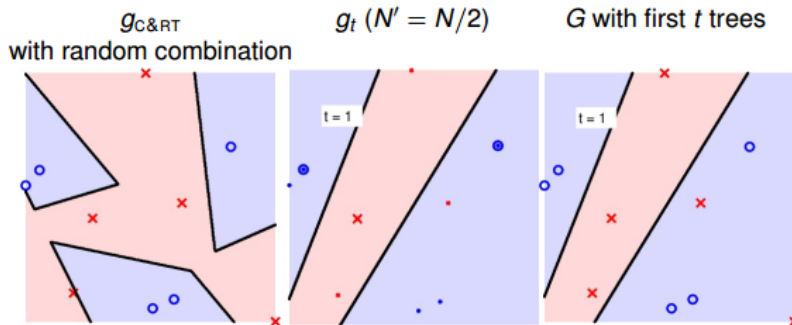
## 4. Random Forest in Action

一个二元分类的例子。

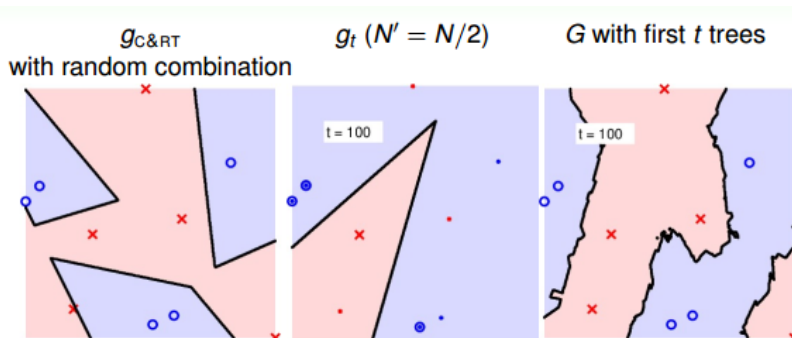
如下图所示，左边是一个C&RT树没有使用bootstrap得到的模型分类效果，其中不同特征之间进行了随机组合，所以有斜线作为分类线；

中间是由bootstrap ( $N'=N/2$ ) 后生成的一棵决策树组成的随机森林，图中加粗的点表示被bootstrap选中的点；

右边是将一棵决策树进行bagging后的分类模型，效果与中间图是一样的，都是一棵树。

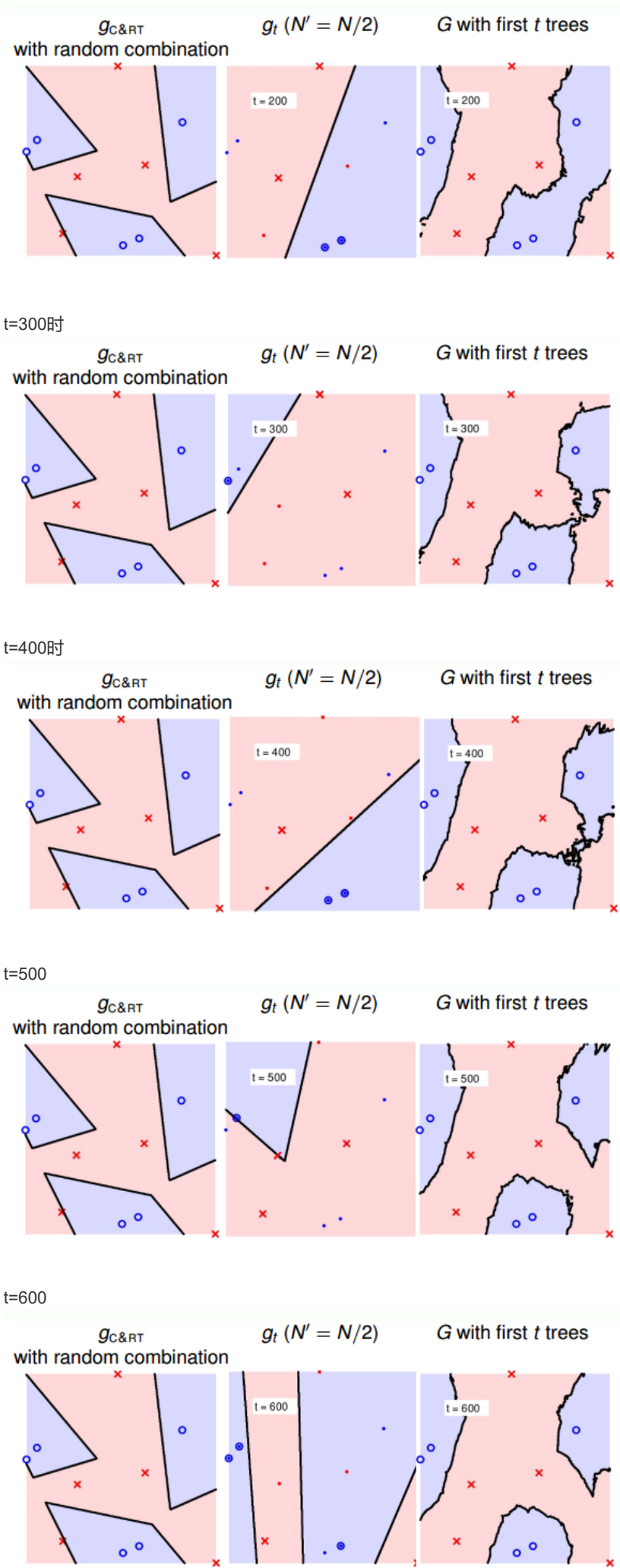


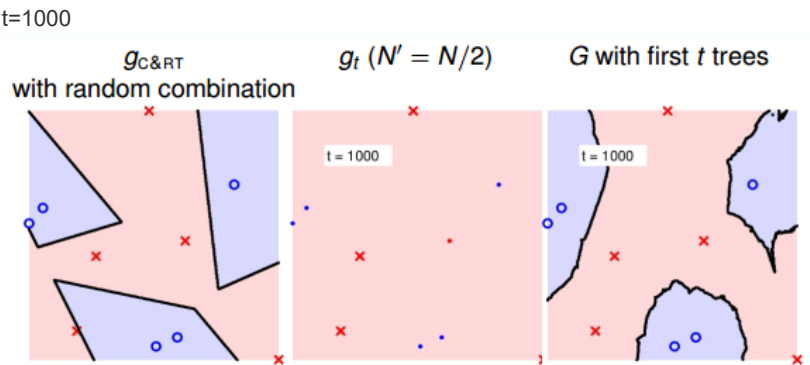
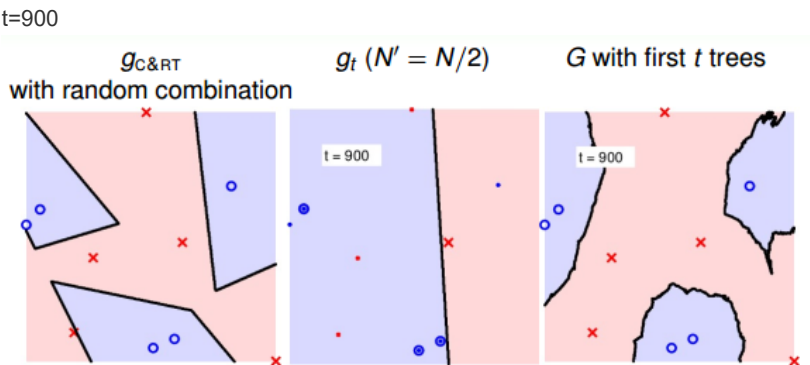
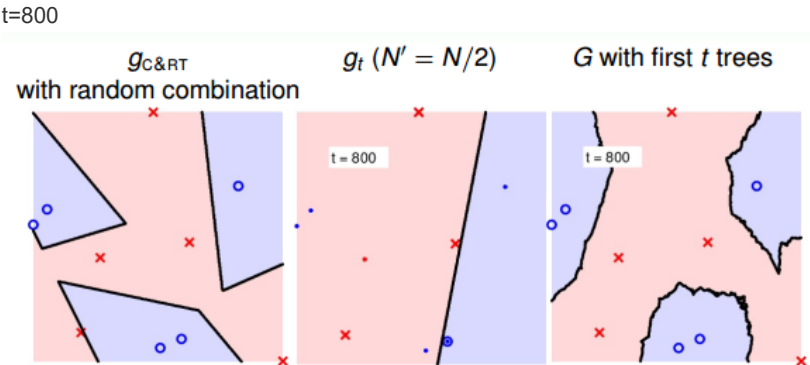
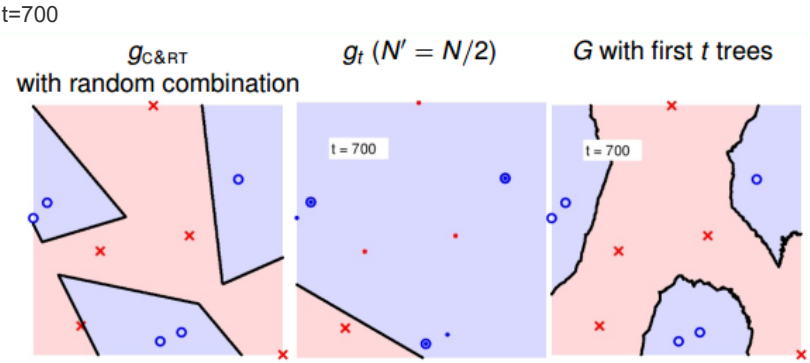
当  $t=100$ ，即选择了100棵树时，中间的模型是第100棵决策树构成的，还是一棵树；右边的模型是由100棵决策树bagging起来的，如下图所示：



t=200时

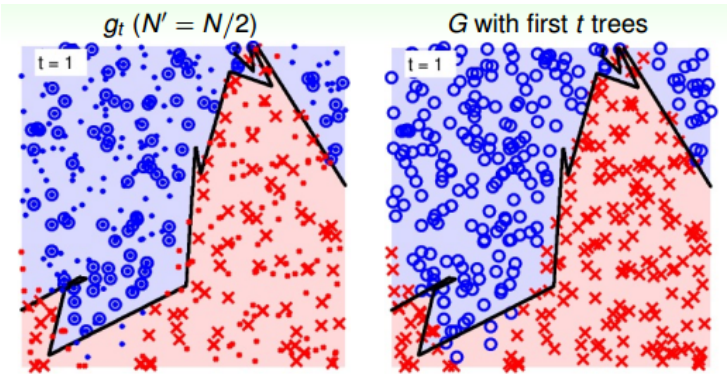




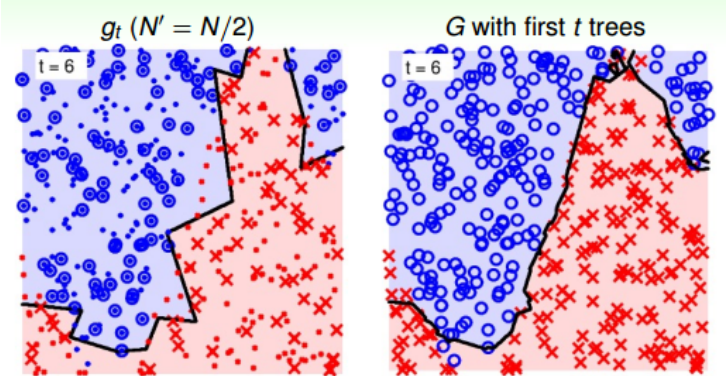


随着树木个数的增加，我们发现，分界线越来越光滑而且得到了large-margin-like boundary，类似于SVM一样的效果。也就是说，树木越多，分类器的置信区间越大。

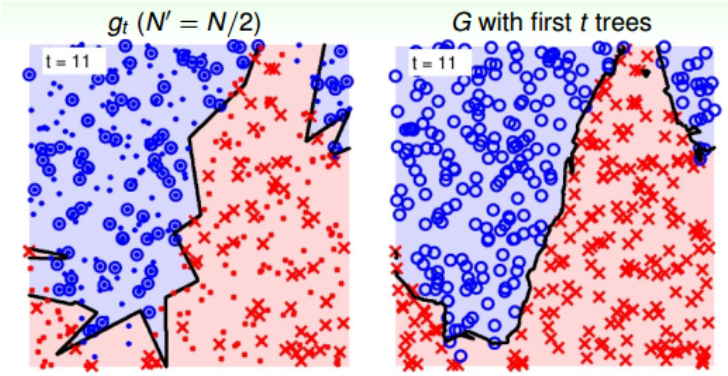
然后，我们再来看一个比较复杂的例子，二维平面上分布着许多离散点，分界线形如sin函数。当只有一棵树的时候（t=1），下图左边表示单一树组成的RF，右边表示所有树bagging组合起来构成的RF。因为只有一棵树，所以左右两边效果一致。



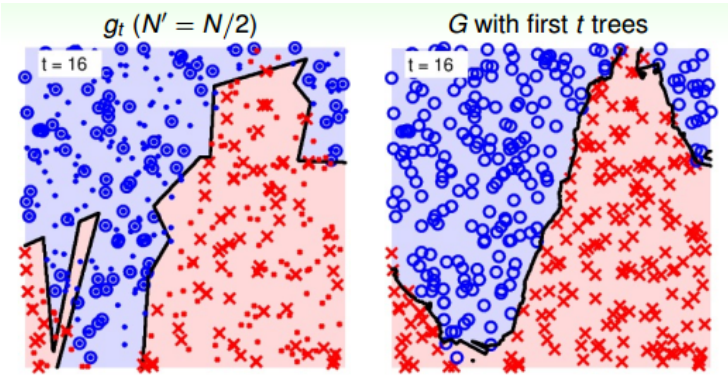
t=6时



t=11时

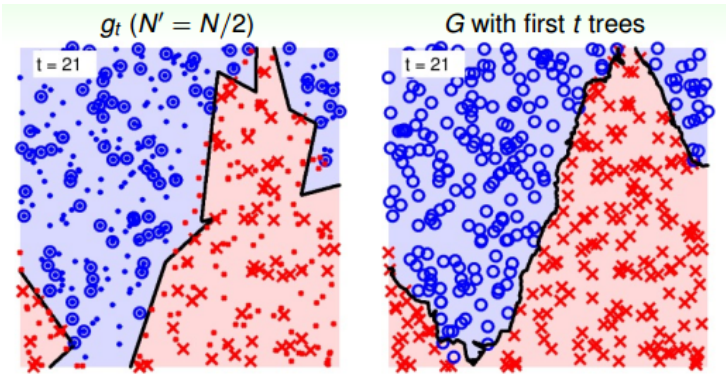


t=16时



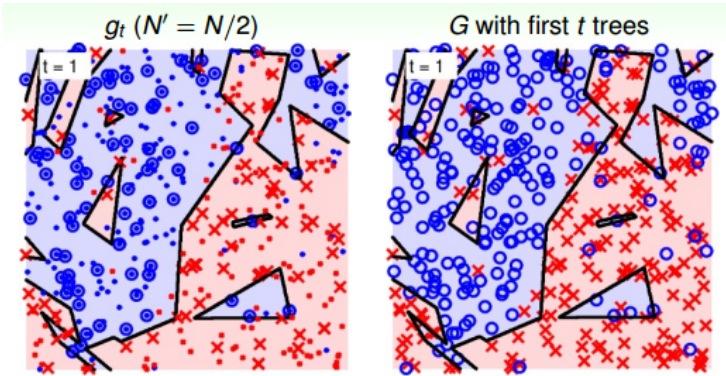
t=21时



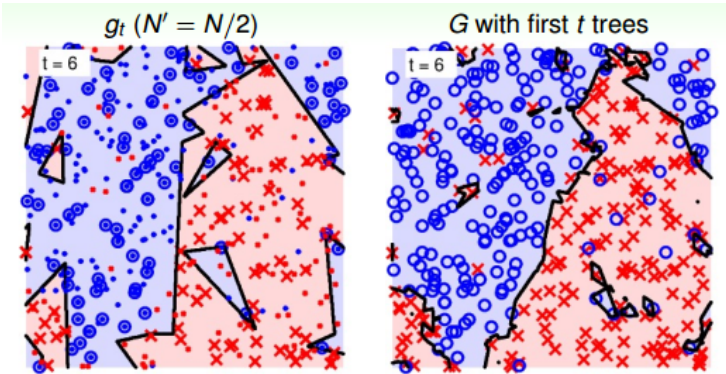


可以看到，当RF由21棵树构成的时候，分界线就比较平滑了，而且它的边界比单一树构成的RF要robust得多，更加平滑和稳定。

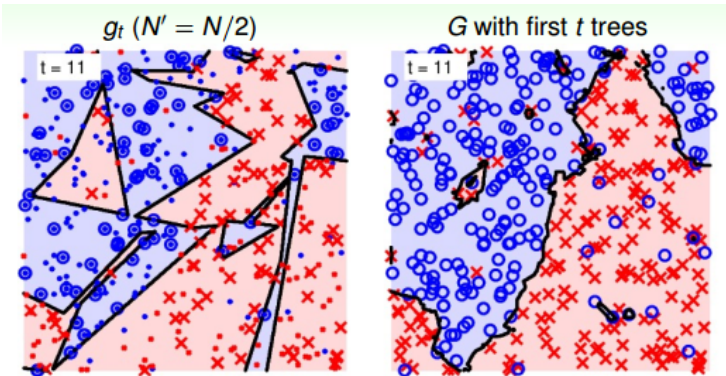
最后，基于上面的例子，再让问题复杂一点：在平面上添加一些随机噪声。当t=1时，如下图所示：



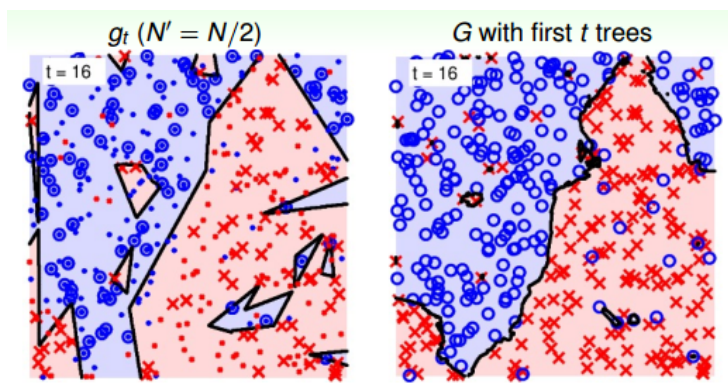
t=6



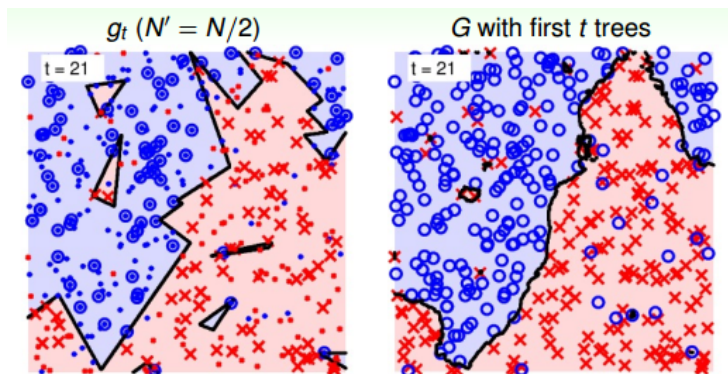
t=11



t=16



$t=21$



从上图中，我们发现21棵树的时候，随机noise的影响基本上能够修正和消除。这种bagging投票的机制能够保证较好的降噪性，从而得到比较稳定的结果。

经过以上三个例子，我们发现RF中，树的个数越多，模型越稳定越能表现得越好。在实际应用中，应该尽可能选择更多的树。值得一提的是，RF的表现同时也与random seed有关，即随机的初始值也会影响RF的表现。

cons of RF: may need lots of trees if the whole random process too unstable  
—should double-check stability of  $G$  to ensure enough trees