1. RF和GBDT的区别

相同点：

- 都是由多棵树组成，最终的结果都是由多棵树一起决定。

不同点：

- **集成学习**：RF属于$Bagging$思想，而$GBDT$是$Boosting$思想
- **偏差-方差权衡**：RF不断的降低模型的方差，而$GBDT$不断的降低模型的偏差
- **训练样本**：RF每次迭代的样本是从全部训练集中有放回抽样形成的，而$GBDT$每次使用全部样本
- **并行性**：RF的树可以并行生成，而$GBDT$只能顺序生成(需要等上一棵树完全生成)
- **最终结果**：RF最终是多棵树进行多数表决（回归问题是取平均），而$GBDT$是加权融合
- **数据敏感性**：RF对异常值不敏感，而$GBDT$对异常值比较敏感
- **泛化能力**：RF不易过拟合，而$GBDT$容易过拟合

2. 比较LR和GBDT，说说什么情景下GBDT不如LR

先说说LR和$GBDT$的区别：

- LR是线性模型，可解释性强，很容易并行化，但学习能力有限，需要大量的人工特征工程
- $GBDT$是非线性模型，具有天然的特征组合优势，特征表达能力强，但是树与树之间无法并行训练，而且树模型很容易过拟合；

当在**高维稀疏特征的场景下，LR的效果一般会比$GBDT$好。**原因如下：

先看一个例子：

> 假设一个二分类问题，label为0和1，特征有100维，如果有1w个样本，但其中只要10个正样本1，而这些样本的特征 f1的值为全为1，而其余9990条样本的f1特征都为0(在高维稀疏的情况下这种情况很常见)。 我们都知道在这种情况下，树模型很容易优化出一个使用f1特征作为重要分裂节点的树，因为这个结点直接能够将训练数据划分的很好，但是当测试的时候，却会发现效果很差，因为这个特征f1只是刚好偶然间跟y拟合到了这个规律，这也是我们常说的过拟合。

因为现在的模型普遍都会带着正则项，**而 LR 等线性模型的正则项是对权重的惩罚**，也就是 w_1一旦过大，惩罚就会很大，进一步压缩 w_1的值，使他不至于过大。但是，树模型则不一样，**树模型的惩罚项通常为叶子节点数和深度**等，而我们都知道，对于上面这种`case`，树只需要一个节点就可以完美分割9990和10个样本，一个结点，最终产生的惩罚项极其之小。

这也就是为什么在高维稀疏特征的时候，线性模型会比非线性模型好的原因了：**带正则化的线性模型比较不容易对稀疏特征过拟合。**

3. 简单介绍一下XGBoost

$XGBoost$是一种集成学习算法，属于3类常用的集成方法($Bagging$，$Boosting$，$Stacking$)中的$Boosting$算法类别。它是一个加法模型，基模型一般选择树模型，但也可以选择其它类型的模型如逻辑回归等。

$XGBoost$对$GBDT$进行了一系列优化，比如损失函数进行了二阶泰勒展开、目标函数加入正则项、支持并行、默认缺失值处理等，在可扩展性和训练速度上有了巨大的提升，但其核心思想没有大的变化。

4. XGBoost与GBDT有什么不同

- **基分类器**：$XGBoost$的基分类器不仅支持$CART$决策树，还支持线性分类器，此时$XGBoost$相当于带$L1$和$L2$正则化项的LR回归（分类问题）或者线性回归（回归问题）。
- **导数信息**：$XGBoost$对损失函数做了二阶泰勒展开，可以更为精准的逼近真实的损失函数，$GBDT$只用了一阶导数信息，并且$XGBoost$还支持自定义损失函数，只要损失函数一阶、二阶可导。
- **正则项**：$XGBoost$的目标函数加了正则项， 相当于预剪枝，使得学习出来的模型更加不容易过拟合。
- **列抽样**：$XGBoost$支持列采样，与随机森林类似，用于防止过拟合。
- **缺失值处理**：对树中的每个非叶子结点，$XGBoost$可以自动学习出它的默认分裂方向。如果某个样本该特征值缺失，会将其划入默认分支。
- **并行化**：注意不是树维度的并行，而是特征维度的并行。$XGBoost$预先将每个特征按特征值排好序，存储为块结构，分裂结点时可以采用多线程并行查找每个特征的最佳分割点，极大提升训练速度。
- **可扩展性**：损失函数支持自定义，只需要新的损失函数二阶可导。

5. XGBoost为什么可以并行训练

- **不是说每棵树可以并行训练**，$XGBoost$本质上仍然采用$Boosting$思想，每棵树训练前需要等前面的树训练完成才能开始训练。
- **而是特征维度的并行**：1)训练之前，每个特征按特征值对样本进行预排序，并存储为`block`结构，在后面查找特征分割点时可以重复使用，而且特征已经被存储为一个个`block`结构，那么在寻找每个特征的最佳分割点时，可以利用多线程对每个`block`并行计算。2)也可以进行列的并行。

6. XGBoost为什么快?

- **分块并行**：训练前每个特征按特征值进行排序并存储为`block`结构，后面查找特征分割点时重复使用，并且支持并行查找每个特征的分割点
- `block` **处理优化**：`block`预先放入内存；`block`按列进行解压缩；将`block`划分到不同硬盘来提高吞吐
- **候选分位点**：每个特征采用常数个分位点作为候选分割点
- **CPU cache 命中优化**： 使用缓存预取的方法，对每个线程分配一个连续的`buffer`，读取每个`block`中样本的梯度信息并存入连续的`buffer`中。

7. XGBoost中如何处理过拟合的情况?

- **目标函数中增加了正则项**：使用叶子结点的数目和叶子结点权重的$L2$模的平方，控制树的复杂度。
- **设置目标函数的增益阈值**：如果分裂后目标函数的增益小于该阈值，则不分裂。
- **设置最小样本权重和的阈值**：当引入一次分裂后，重新计算新生成的左、右两个叶子结点的样本权重和。如果任一个叶子结点的样本权重低于某一个阈值（最小样本权重和），也会放弃此次分裂。
- **设置树的最大深度**：$XGBoost$ 先从顶到底建立树直到最大深度，再从底到顶反向检查是否有不满足分裂条件的结点，进行剪枝。
- 调参：
  - 第一类参数：用于直接控制模型的复杂度。包括`max_depth，min_child_weight，gamma`等参数
  - 第二类参数：用于增加随机性，从而使得模型在训练时对于噪音不敏感。包括`subsample，colsample_by树`
  - 还有就是直接减小`learning rate`，但需要同时增加`estimator`参数。

8. XGBoost如何处理缺失值?

$XGBoost$模型的一个优点就是允许特征存在缺失值。对缺失值的处理方式如下：

- 在特征`k`上寻找最佳划分点时，不会对该列特征缺失的样本进行遍历，而只对该列特征值为无缺失的样本上对应的特征值进行遍历，通过这个技巧来减少了为稀疏离散特征寻找划分点的时间开销。
- 在逻辑实现上，为了保证完备性，会**将该特征值缺失的样本分别分配到左叶子结点和右叶子结点，两种情形都计算一遍后，选择分裂后增益最大**的那个方向（左分支或是右分支），作为预测时特征值缺失样本的默认分支方向。
- $XGBoost$**在构建树的节点过程中只考虑非缺失值的数据遍历**，而为每个节点增加了一个缺省方向，当样本相应的特征值缺失时，可以被归类到缺省方向上，最优的缺省方向可以从数据中学到。至于如何学到缺省值的分支，其实很简单，分别枚举特征缺省的样本归为左右分支后的增益，选择增益最大的枚举项即为最优缺省方向。
- 如果在**训练中没有缺失值而在预测中出现缺失，那么会自动将缺失值的划分方向放到右子结点。**缺失值处理的伪代码如下：

Algorithm 3: Sparsity-aware Split Finding

Input: I, instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: d, feature dimension
Also applies to the approximate setting, only collect statistics of non-missing entries into buckets
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I}, g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  // enumerate missing value goto right
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for j *in sorted*$(I_k,$ *ascent order by* $\mathbf{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
  // enumerate missing value goto left
  $G_R \leftarrow 0,\ H_R \leftarrow 0$
  for j *in sorted*$(I_k,$ *descent order by* $\mathbf{x}_{jk})$ **do**
    $G_R \leftarrow G_R + g_j,\ H_R \leftarrow H_R + h_j$
    $G_L \leftarrow G - G_R,\ H_L \leftarrow H - H_R$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split and default directions with max gain

- **树模型对缺失值的敏感度低，大部分时候可以在数据缺失时时使用。**

  原因就是：**一棵树中每个结点在分裂时，寻找的是某个特征的最佳分裂点（特征值）**，完全可以不考虑存在特征值缺失的样本，也就是说，如果某些样本缺失的特征值缺失，对寻找最佳分割点的影响不是很大。

9. XGBoost如何处理不平衡数据?

- 设置`scale_pos_weight`来平衡正样本和负样本的权重。例如，当正负样本比例为1:10时，`scale_pos_weight`可以取10；
- 你不能重新平衡数据集(会破坏数据的真实分布)的情况下，应该设置`max_delta_step`为一个有限数字来帮助收敛（基模型为LR时有效）。

10. XGBoost如何选择最佳分裂点?

- 训练前预先将特征**对特征值进行排序**，存储为`block`结构，以便在结点分裂时可以重复使用
- 采用**特征并行**的方法利用多个线程分别计算每个特征的最佳分割点，根据每次分裂后产生的增益，**最终选择增益最大的那个特征的特征值**作为最佳分裂点。
- $XGBoost$使用**直方图近似算法**，对特征排序后仅选择常数个候选分裂位置作为候选分裂点，极大提升了结点分裂时的计算效率。

11. XGBoost的Scalable性如何体现?

- **基分类器的scalability**：弱分类器可以支持$CART$决策树，也可以支持LR和Linear。
- **目标函数的scalability**：支持自定义loss function，只需要其一阶、二阶可导。有这个特性是因为泰勒二阶展开，得到通用的目标函数形式。
- **学习方法的scalability**：`block`结构支持并行化，支持 Out-of-core计算。

12. XGBoost如何评价特征的重要性?

常用的三种方法来评判模型中特征的重要程度：

- `freq`： 频率是表示特定特征在模型树中发生分裂的相对次数的百分比
- `gain`： 增益意味着相应的特征对通过对模型中的每个树采取每个特征的贡献而计算出的模型的相对贡献。与其他特征相比，此度量值的较高值意味着它对于生成预测更为重要。
- `cover`： 覆盖度量指的是与此功能相关的观测的相对数量。例如，如果您有100个观察值，4个特征和3棵树，并且假设特征1分别用于决定树1，树2和树3中10个，5个和2个观察值的叶节点;那么该度量将计算此功能的覆盖范围为$10+5+2=17$个观测值。这将针对所有4项功能进行计算，并将以17个百分比表示所有功能的覆盖指标。

$XGBoost$是根据`gain`来做重要性判断的。

13. XGBooost参数调优的一般步骤

- 确定`learning rate`和`estimator`的数量

  `learning rate`可以先用0.1，用cv来寻找最优的`estimators`

- `max_depth`和 `min_child_weight`

  我们调整这两个参数是因为，这两个参数对输出结果的影响很大。我们首先将这两个参数设置为较大的数，然后通过迭代的方式不断修正，缩小范围。

  `max_depth`: 每棵子树的最大深度，check from range(3，10，2)。

  `min_child_weight`: 子节点的权重阈值，check from range(1，6，2)。

  如果一个结点分裂后，它的所有子节点的权重之和都大于该阈值，该叶子节点才可以划分。

- `gamma`

  也称作最小划分损失`min_split_loss`，check from 0.1 to 0.5，指的是，对于一个叶子节点，当对它采取划分之后，损失函数的降低值的阈值。

  - 如果大于该阈值，则该叶子节点值得继续划分

    - 如果小于该阈值，则该叶子节点不值得继续划分
- `subsample`、 `colsample_by tree`

  `subsample`是对训练的采样比例

  `colsample_by tree`是对特征的采样比例both check from 0.6 to 0.9
- 正则化参数

  `alpha` 是$L1$正则化系数，try `1e-5， 1e-2， 0.1， 1， 100`

  `lambda` 是$L2$正则化系数
- 降低学习率

  降低学习率的同时增加树的数量，通常最后设置学习率为`0.01~0.1`

14. XGBoost的优缺点

- 优点
    - **精度更高：** $GBDT$ 只用到一阶泰勒展开，而 $XGBoost$ 对损失函数进行了二阶泰勒展开。$XGBoost$ 引入二阶导一方面是为了增加精度，另一方面也是为了能够自定义损失函数，二阶泰勒展开可以近似大量损失函数；
    - **灵活性更强：** $GBDT$ 以 $CART$ 作为基分类器，$XGBoost$ 不仅支持 $CART$ 还支持线性分类器，使用线性分类器的 $XGBoost$ 相当于带 和 正则化项的逻辑斯蒂回归（分类问题）或者线性回归（回归问题）。此外，$XGBoost$ 工具支持自定义损失函数，只需函数支持一阶和二阶求导；
    - **正则化：** $XGBoost$ 在目标函数中加入了正则项，用于控制模型的复杂度。正则项里包含了树的叶子节点个数、叶子节点权重的 范式。正则项降低了模型的方差，使学习出来的模型更加简单，有助于防止过拟合，这也是$XGBoost$优于传统$GBDT$的一个特性。
    - **Shrinkage（缩减）：** 相当于学习速率。$XGBoost$ 在进行完一次迭代后，会将叶子节点的权重乘上该系数，主要是为了削弱每棵树的影响，让后面有更大的学习空间。传统$GBDT$的实现也有学习速率；
    - **列抽样：** $XGBoost$ 借鉴了随机森林的做法，支持列抽样，不仅能降低过拟合，还能减少计算。这也是$XGBoost$异于传统$GBDT$的一个特性；
    - **缺失值处理：** 对于特征的值有缺失的样本，$XGBoost$ 采用的稀疏感知算法可以自动学习出它的分裂方向；
    - $XGBoost$**工具支持并行：** $Boosting$不是一种串行的结构吗?怎么并行的？注意$XGBoost$的并行不是树粒度的并行，$XGBoost$也是一次迭代完才能进行下一次迭代的（第次迭代的代价函数里包含了前面次迭代的预测值）。$XGBoost$的并行是在特征粒度上的。我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），$XGBoost$ 在训练之前，预先对数据进行了排序，然后保存为`block`结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个`block`结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行。
    - **可并行的近似算法：** 树节点在进行分裂时，我们需要计算每个特征的每个分割点对应的增益，即用贪心法枚举所有可能的分割点。当数据无法一次载入内存或者在分布式情况下，贪心算法效率就会变得很低，所以$XGBoost$还提出了一种可并行的近似算法，用于高效地生成候选的分割点。
- 缺点
    - 虽然利用预排序和近似算法可以降低寻找最佳分裂点的计算量，但**在节点分裂过程中仍需要遍历数据集**；
    - **预排序过程的空间复杂度过高**，不仅需要存储特征值，还需要存储特征对应样本的梯度统计值的索引，相当于**消耗了两倍的内存。**

15. XGBoost和LightGBM的区别

（1）树生长策略：XGB采用`level-wise`的分裂策略，LGB采用`leaf-wise`的分裂策略。XGB对每一层所有节点做无差别分裂，但是可能有些节点增益非常小，对结果影响不大，带来不必要的开销。Leaf-wise是在所有叶子节点中选取分裂收益最大的节点进行的，但是很容易出现过拟合问题，所以需要对最大深度做限制 。

（2）分割点查找算法：XGB使用特征预排序算法，LGB使用基于直方图的切分点算法，其优势如下：

- 减少内存占用，比如离散为256个bin时，只需要用8位整形就可以保存一个样本被映射为哪个bin(这个bin可以说就是转换后的特征)，对比预排序的exact greedy算法来说（用int_32来存储索引 + 用float_32保存特征值），可以节省7/8的空间。
- 计算效率提高，预排序的Exact greedy对每个特征都需要遍历一遍数据，并计算增益。而直方图算法在建立完直方图后，只需要对每个特征遍历直方图即可。
- LGB还可以使用直方图做差加速，一个节点的直方图可以通过父节点的直方图减去兄弟节点的直方图得到，从而加速计算

> 但实际上$XGBoost$的近似直方图算法也类似于$LightGBM$这里的直方图算法，为什么$XGBoost$的近似算法比$LightGBM$还是慢很多呢？ $XGBoost$在每一层都动态构建直方图， 因为$XGBoost$的直方图算法不是针对某个特定的feature，而是所有feature共享一个直方图(每个样本的权重是二阶导)，所以每一层都要重新构建直方图，而$LightGBM$中对每个特征都有一个直方图，所以构建一次直方图就够了。

（3）支持离散变量：无法直接输入类别型变量，因此需要事先对类别型变量进行编码（例如独热编码），而$LightGBM$可以直接处理类别型变量。

（4）缓存命中率：$XGBoost$使用`block`结构的一个缺点是取梯度的时候，是通过索引来获取的，而这些梯度的获取顺序是按照特征的大小顺序的，这将导致非连续的内存访问，可能使得CPU cache缓存命中率低，从而影响算法效率。而LGB是基于直方图分裂特征的，梯度信息都存储在一个个bin中，所以访问梯度是连续的，缓存命中率高。

（5）$LightGBM$ 与 $XGBoost$ 的并行策略不同：

- **特征并行** ：LGB特征并行的前提是每个`worker`留有一份完整的数据集，但是每个`worker`仅在特征子集上进行最佳切分点的寻找；`worker`之间需要相互通信，通过比对损失来确定最佳切分点；然后将这个最佳切分点的位置进行全局广播，每个`worker`进行切分即可。XGB的特征并行与LGB的最大不同在于XGB每个`worker`节点中仅有部分的列数据，也就是垂直切分，每个`worker`寻找局部最佳切分点，`worker`之间相互通信，然后在具有最佳切分点的`worker`上进行节点分裂，再由这个节点广播一下被切分到左右节点的样本索引号，其他`worker`才能开始分裂。二者的区别就导致了LGB中`worker`间通信成本明显降低，只需通信一个特征分裂点即可，而XGB中要广播样本索引。
- **数据并行** ：当数据量很大，特征相对较少时，可采用数据并行策略。LGB中先对数据水平切分，每个`worker`上的数据先建立起局部的直方图，然后合并成全局的直方图，采用直方图相减的方式，先计算样本量少的节点的样本索引，然后直接相减得到另一子节点的样本索引，这个直方图算法使得`worker`间的通信成本降低一倍，因为只用通信以此样本量少的节点。XGB中的数据并行也是水平切分，然后单个`worker`建立局部直方图，再合并为全局，不同在于根据全局直方图进行各个`worker`上的节点分裂时会单独计算子节点的样本索引，因此效率贼慢，每个`worker`间的通信量也就变得很大。
- **投票并行（LGB）** ：当数据量和维度都很大时，选用投票并行，该方法是数据并行的一个改进。数据并行中的合并直方图的代价相对较大，尤其是当特征维度很大时。大致思想是：每个`worker`首先会找到本地的一些优秀的特征，然后进行全局投票，根据投票结果，选择top的特征进行直方图的合并，再寻求全局的最优分割点。