简述 GBDT 原理。

梯度提升树的训练过程大致是这样的：

1. 根据训练集训练一颗初始决策树；
2. 计算之前所有树在此数据集上预测结果之和与真实结果的差值，又叫做残差。
3. 把残差作为当前树作为拟合的目标在训练集上训练。
4. 重复 2,3 步骤，直到达到设置的阈值（树的个数，早停策略等）

采用伪代码表示如下：

输入：训练数据集 $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$，其中$x_i \in x \subseteq R^n, y_i \in y \subseteq R$;

输出：提升树$f_M(x)$.

1）初始化$f_0(x) = 0$

2）对$m = 1, 2, \dots, M$

a. 计算残差

$$r_{mi} = y_i - f_{m-1}(x_i), \quad i = 1, 2, \dots, N$$

b. 拟合残差r_{mi}学习一个回归树，得到$T(x, \theta_m)$

c. 更新 $f_m(x) = f_{m-1}(x) + T(x, \theta_m)$

3）得到回归问题的提升树

$$f_M(x) = f_m(x) = \sum_{m=1}^{M} T(x, \theta_m)$$

上述的伪代码描述中，我们对 GBDT 算法做了一下三个简化：

1. 用残差来表示提升树的负梯度；
2. 假设所有树的贡献权重都相同；
3. 没有把回归树的在叶子节点上的拟合信息体现出来。

接下来我们将简化的信息补全，得到下面 GBDT 算法的伪代码：

1）通过最小化损失最小化初始模型：

$$F_0(x) = \arg\min_{r}(\sum_{n=1}^{N} L(y_i, \gamma))$$

2）对$m = 1, 2, \dots, M$

a. 计算负梯度：

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1(x)}}, i = 1, 2, \dots, n$$

b. 训练一个回归树去拟合目标值r_{im}，树的终端区域为$R_{jm}(j = 1, 2, \dots, J_m)$

c. 对$j = 1, 2, \dots, J_m$，计算步长γ_{jm}

$$\gamma_{jm} = \arg\min_{\gamma}\left(\sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)\right)$$

d. 更新模型：

$$F_m(x) = F_{m-1}(x) + \sum_{j=1}^{Jm} \gamma_{jm} I(x \in R_{jm})$$

3）输出$F_M(x)$.

其中每个树的终端区域代表当前树上叶子节点所包含的区域，步长γ_{jm}为第 m 棵树上第 j 个叶子节点的预测结果。

GBDT 如何用于分类？

GBDT 在做分类任务时与回归任务类似，所不同的是损失函数的形式不同。我们以二分类的指数损失函数为例来说明：

我们定义损失函数为：

$$L(y, f(x)) = \exp(-yf(x))$$

其中$y_i \in y = \{-1, +1\}$，则此时负梯度为

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1(x)}} = y_i \exp(-y_i f(x_i)), i = 1, 2, \dots, n$$

对于各个叶子节点最佳拟合的值为：

$$\gamma_{jm} = \arg\min_{\gamma}\left(\sum_{x_i \in R_{jm}} \exp(-y_i(F_{m-1}(x_i) + \gamma))\right)$$

通过与 GBDT 伪代码中步骤 b 和步骤 c 类比来理解。

GBDT 常用损失函数有哪些？

回归问题：

MAE，MSE，RMSE（见**回归问题常用的性能度量指标？**）

补充

Huber Loss （MAE 和 MSE 结合）

1）在 0 附近可导

2）Huber 大部分情况下为 MAE，只在误差很小的时候变成了 MSE。

$$L_\delta(y, f(x)) = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\frac{1}{2}(y_i - f(x_i))^2, & |y_i - f(x_i)| \le \delta \\ \delta|y_i - f(x_i)| - \frac{1}{2}\delta, & otherwise\end{cases}$$

分类问题：

对数似然损失函数，应对二分类和多分类问题（见**逻辑回归的损失函数？逻辑回归处理多标签分类问题时，一般怎么做？**）

补充

指数损失函数：$y_i \in y = \{-1, +1\}$

$$L(y, f(x)) = \frac{1}{N}\sum_{i=1}^{N}\exp(-y_i f(x_i))$$

为什么 GBDT 不适合使用高维稀疏特征？

高维稀疏的 ID 类特征会使树模型的训练变得极为低效，且容易过拟合。

- 树模型训练过程是一个贪婪选择特征的算法，要从候选特征集合中选择一个使分裂后收益函数增益最大的特征来分裂。按照高维的 ID 特征做分裂时，子树数量非常多，计算量会非常大，训练会非常慢。
- 同时，按 ID 分裂得到的子树的泛化性也比较弱，由于只包含了对应 ID 值的样本，样本稀疏时也很容易过拟合。

GBDT 算法的优缺点？

优点：

- 预测阶段的计算速度快，树与树之间可并行化计算（注意预测时候可并行）。
- 在分布稠密的数据集上，泛化能力和表达能力都很好。

- 采用决策树作为弱分类器使得 GBDT 模型具有 1）较好的解释性和鲁棒性，2）能够自动发现特征间的高阶关系，并且也 3）不需要对数据进行特殊的预处理如归一化等。

缺点

- GDBT 在高维稀疏的数据集上，表现不佳（见**为什么 GBDT 不适合使用高维稀疏特征?**）
- 训练过程需要串行训练，只能在决策树内部采用一些局部并行的手段提高训练速度。

附加题:

CV:

7.LSTM为什么能解决梯度消失/爆炸的问题

LSTM把原本RNN的单元改造成一个叫做CEC的部件，这个部件保证了误差将以常数的形式在网络中流动，并在此基础上添加输入门和输出门使得模型变成非线性的，并可以调整不同时序的输出对模型后续动作的影响。

NLP:

5、Self-Attention 的时间复杂度是怎么计算的?

Self-Attention时间复杂度： $O(n^2 \cdot d)$ ，这里，n是序列的长度，d是embedding的维度。

Self-Attention包括**三个步骤：相似度计算，softmax和加权平均**，它们分别的时间复杂度是：

相似度计算可以看作大小为(n,d)和(d,n)的两个矩阵相乘： $(n,d) * (d,n) = O(n^2 \cdot d)$ ，得到一个(n,n)的矩阵

softmax就是直接计算了，时间复杂度为 $O(n^2)$

加权平均可以看作大小为(n,n)和(n,d)的两个矩阵相乘： $(n,n) * (n,d) = O(n^2 \cdot d)$ ，得到一个(n,d)的矩阵

因此，Self-Attention的时间复杂度是 $O(n^2 \cdot d)$ 。

这里再分析一下Multi-Head Attention，它的作用类似于CNN中的多核。

多头的实现不是循环的计算每个头，而是通过 transposes and reshapes，用矩阵乘法来完成的。

> In practice, the multi-headed attention are done with transposes and reshapes rather than actual separate tensors. —— 来自 google BERT 源码

Transformer/BERT中把 **d** ，也就是hidden_size/embedding_size这个维度做了reshape拆分，可以去看Google的TF源码或者上面的pytorch源码：

> hidden_size (d) = num_attention_heads (m) * attention_head_size (a)，也即 d=m*a

并将 num_attention_heads 维度transpose到前面，使得Q和K的维度都是(m,n,a)，这里不考虑batch维度。

这样点积可以看作大小为(m,n,a)和(m,a,n)的两个张量相乘，得到一个(m,n,n)的矩阵，其实就相当于(n,a)和(a,n)的两个矩阵相乘，做了m次，时间复杂度（感谢评论区指出）是 $O(n^2 \cdot m \cdot a) = O(n^2 \cdot d)$ 。

张量乘法时间复杂度分析参见：矩阵、张量乘法的时间复杂度分析

因此Multi-Head Attention时间复杂度也是 $O(n^2 \cdot d)$ ，复杂度相较单头并没有变化，主要还是 transposes and reshapes 的操作，相当于把一个大矩阵相乘变成了多个小矩阵的相乘。

BI:

问题: 推荐系统中为什么要有召回?在召回和排序中使用的深度学习算法有什么异同?

答案:

首先,在实际应用中,推荐算法往往是线上使用,可用的设备资源和响应时间都是有限的.而整个物品集的规模往往十分庞大,在线上对大量物品进行排序是对性能的较大挑战,很难实现.召回可以是为一个粗排序的过程,这个过程的主要目的是在有限的资源条件下提供尽可能准确的一个小候选集,从而减轻排序阶段的计算负担和耗时.

其次,即使资源和时间足够对整个物品集进行扫描,先使用较为简单的方法进行一次召回往往也是有利的.先进行召回意味着可以排除大部分无关物品,从而允许在排序阶段对更小的候选集使用更多的特征和更复杂的模型,以提高排序的准确率.

上述理由也是推荐系统在召回和排除阶段有不同侧重的原因.在召回阶段,推荐系统一般更侧重于大量物品的筛选效率,可以接受一些牺牲一定准确性的优化算法,而在排序阶段推荐系统通常对预测准确率更加重视.在使用传统算法时,召回和排序阶段使用的算法种类通常有所区别,例如可以在召回阶段使用协同过滤,而在排序阶段使用逻辑回归或者梯度提升决策树.

在使用深度学习算法时,可以把相似的深度神经网络同时用于召回和排序.但由于召回和排序阶段的目标有一定差别,在应用深度神经网络时也有相应的区别.在特征方面,排序阶段的算法会更多的使用当前上下文特征,用户行为相关特征,时序相关特征等,对特征的处理比召回阶段更加精细.在结构方面,召回阶段使用深度神经网络时,比较常用的方案是用softmax作为网络的最后一层,而在排序阶段的最后一层是单个神经元的情况较多.由于可用的特征更多更详细,排序算法可以使用更多精巧的结构,例如时序特征相关的结构(LSTM,GRU等)和注意力机制,而召回算法通常不用这些结构.