算法工程师（NLP/搜索推荐/机器学习）常见面试题

我自己收集的面试题和答案，如果发现有不对的地方请随时联系我
（b 站/小红书/抖音：一只甜药，wb：炸酱子）
ChatGPT 相关的问题在最后

【SVM 原理】

SVM 是一个二分类算法，在空间里找一个超平面，将空间分为两块，其中某一类的点都在超平面上方，另一类都在超平面下方。

算法的原理是找一个超平面，使得两类的点中离平面最近的那个点离平面的距离最远。这个点称为支持向量，所以 SVM 的全名是支持向量机。

【LR 逻辑回归】

- **原理介绍**

二分类算法，也可以用于多分类问题。

逻辑回归分为两部分：逻辑和回归。

线性回归模型里的因变量是连续变量，而逻辑回归里的因变量可以理解为分类变量，现在我们需要用回归模型去表示这个分类，比如说大或小，黑或白。那么考虑类别之间相对的关系，我们可以用概率来表示这个问题，比如说分类为黑的概率大于白的概率时，就把样本预测为黑。所以我们可以定制一个映射关系，将负无穷到正无穷之间的数值映射成概率值，即 0-1 之间，就可以解决线性回归到分类问题的过渡。

最常见的激活函数是 Sigmoid 函数。

- **为什么激活函数用 sigmoid？**

因为伯努利的指数族分布形式为

$$p(x|\mu) = \mathrm{Bern}(x|\mu) = e^{\ln(\mu^x(1-\mu)^{1-x})} = (1-\mu)\exp\{x\ln\left(\frac{\mu}{1-\mu}\right)\}$$

在这个式子中，对照指数族分布的形式

$$\mu = \frac{1}{1+e^{-\eta}}$$

- **为什么要用指数族分布？**

因为指数族分布是给定某些统计量下熵最大的分布，例如伯努利分布就是只有两个取值且给定期望值为下的熵最大的分布。

- **为什么要使用熵最大的分布？**

最大熵理论

https://zhuanlan.zhihu.com/p/59137998

- **交叉熵公式推导**

问题描述

对逻辑回归交叉熵损失函数

$$
\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= -\sum_{i=1}^{m}\left[\nabla y_i \ln \hat{y}_i + \nabla(1-y_i)\ln(1-\hat{y}_i)\right] \\
&= -\sum_{i=1}^{m}\left[\frac{y_i \nabla \hat{y}_i}{\hat{y}_i} + \frac{(1-y_i)\nabla(1-\hat{y}_i)}{1-\hat{y}_i}\right] \\
&= -\sum_{i=1}^{m}\left[\frac{y_i \nabla \hat{y}_i}{\hat{y}_i} + \frac{(y_i-1)\nabla \hat{y}_i}{1-\hat{y}_i}\right] \\
&= -\sum_{i=1}^{m}\left[\frac{y_i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)}\right]\nabla \hat{y}_i
\end{aligned}
$$

注意到

$$
d\sigma = \frac{e^{-z}dz}{(1+e^{-z})^2} = \sigma(1-\sigma)dz
$$

即有

$$
\nabla \hat{y}_i = \hat{y}_i(1-\hat{y}_i)\nabla z_i
$$

那么

$$
\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= -\sum_{i=1}^{m}\left[\frac{y_i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)}\right]\nabla \hat{y}_i \\
&= -\sum_{i=1}^{m}\left[\frac{y_i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)}\right]\hat{y}_i(1-\hat{y}_i)\nabla z_i \\
&= -\sum_{i=1}^{m}(y_i-\hat{y}_i)\nabla\left(\boldsymbol{\theta}^T \mathbf{x}_i\right) \\
&= -\sum_{i=1}^{m}(y_i-\hat{y}_i)\mathbf{x}_i
\end{aligned}
$$

● 逻辑回归的参数能不能初始化为 0

神经网络里不可以，LR 里可以，因为反向传播的时候(y^-y)*x，参数更新的时候求导的值会因为 x 的不同而不同，而且不为 0，模型的权重可以得到更新。

当 b 初始化为 0 的时候，同理，loss 对 b 的导数也不为 0，可以得到更新。

● 为什么用交叉熵？

两个角度，

1、从极大似然估计的角度理解

根据逻辑回归的计算公式，我们可以知道对应为 1 和 0 的样本的概率：

$$P(Y=1|x)=\frac{e^{wx+b}}{1+e^{wx+b}}=p(x)$$
$$P(Y=0|x)=\frac{1}{1+e^{wx+b}}=1-p(x)$$

然后就可以计算出现这些样本的似然函数，就是把每一个样本的概率乘起来：

$$L(w;b)=\prod_{i=1}^{n}[p(x_i)^{y_i}(1-p(x_i))^{1-y_i}$$

但是这个形式是连乘的，并不好求，所以一般我们会把他取对数，转化为累加的形式，就得到对数似然函数：

$$L'(w;b)=\sum_{i=1}^{n}[y_i\log(p(x_i))+(1-y_i)log(1-p(x_i))]$$

这时候呢，我们就可以通过最大化这个对数似然函数的方式来求得逻辑回归模型中的 w 和 b，把上面的式子加个负号，就是通过最小化这个负对数似然函数来求得 w 和 b，就可以通过梯度下降法来进行求解了。

可以发现，通过数理统计中的极大似然估计方法，也可以得到逻辑回归的损失函数。

2、从 KL 散度的角度理解

如果对于同一个随机变量 X 有两个单独的概率分布 p(X)和 q(X) ，那么我们就可以用 KL 散度来衡量这两个分布的差异：

$$D_{KL}(p||q) = \sum_{i=1}^{n} p(x_i) \log(\frac{p(x_i)}{q(x_i)})$$

我们将 p(x) 定义为真实的概率分布， q(x) 定义为模型预测的概率分布，我们希望预测的概率分布与真实的概率分布差异越小越好，也就是使得 KL 散度越小越好，而 p(x) 是在数据集确定之后就确定下来的了，所以我们只要使得 q(x) 尽可能地接近 p(x) 就可以了。

将这个 KL 散度的公式展开可以得到：

$$\begin{aligned} D_{KL}(p||q) &= \sum_{i=1}^{n} p(x_i) \log(\frac{p(x_i)}{q(x_i)}) \\ &= \sum_{i=1}^{n} p(x_i) \log(p(x_i)) - \sum_{i=1}^{n} p(x_i) \log(q(x_i)) \\ &= -H(p(x)) - \sum_{i=1}^{n} p(x_i) \log(q(x_i)) \end{aligned}$$

在信息论中， $-\log(p(x))$ 代表的就是信息量，某一随机事件发生的概率越小，反映的信息量就越大，而 $-\sum_{i=1}^{n} p(x) \log(p(x))$ 代表的就是信息量的期望，也就是信息熵，然后如果把这个 log 里面的 p(x)换成另一个分布的概率 q(x)，也就是 $-\sum_{i=1}^{n} p(x) \log(q(x))$ ，这个就是交叉熵。

所以根据上面那个展开的公式，就可以发现 KL 散度=交叉熵-真实分布的信息熵，而这个真实分布的信息熵是根据 p(x) 计算得到的，而这个 p(x) 是在数据集确定之后就确定下来的了，这一项就可以当成一个常数项，所以我们如果想让 KL 散度越小，只需要让交叉熵越小越好了，因此就可以直接将逻辑回归的损失函数直接定义为交叉熵。

https://zhuanlan.zhihu.com/p/386406590

【集成学习】

● bagging 和 boosting 的区别

1）样本选择上：

Bagging：训练集是在原始集中有放回选取的，从原始集中选出的各轮训练集之间是独立的。

Boosting：每一轮的训练集不变，只是训练集中每个样例在分类器中的权重发生变化。而权值是根据上一轮的分类结果进行调整。

2）样例权重：

Bagging：使用均匀取样，每个样例的权重相等

Boosting：根据错误率不断调整样例的权值，错误率越大则权重越大。

3）预测函数：

Bagging：所有预测函数的权重相等。

Boosting：每个弱分类器都有相应的权重，对于分类误差小的分类器会有更大的权重。

4）并行计算：

Bagging：各个预测函数可以并行生成

Boosting：各个预测函数只能顺序生成，因为后一个模型参数需要前一轮模型的结果。

5）方差偏差分解：

Bagging：降低方差

Boosting：降低偏差

对于 Bagging 算法来说，由于我们并行的训练很多的分类器的目的就是降低这个方差。所以对于每个基分类器的目的就是如何降低这个偏差，所以我们会采用深度很深并且不剪枝的决策树。这也是为什么随机森林的树的深度往往大于 GBDT 的树的深度的原因。

● 为什么 Bagging 能降低方差

对每个样本 x，假设单模型（如决策树）在不同数据集上学习得到的模型对样本 x 的输出服从某种分布，Bagging 的集成策略为对弱学习器求平均

$$\mathrm{F}(\mathrm{x}) = \frac{\mathrm{G}_1(\mathrm{x}) + \mathrm{G}_2(\mathrm{x}) + \cdots \mathrm{G}_\mathrm{n}(\mathrm{x})}{\mathrm{n}}$$

由于子样本集的相似性以及使用的是同种模型，因此各模型有近似相等的 bias 和 variance（事实上，各模型的分布也近似相同，但不独立）。

- **为什么偏差不变**

单个模型和集成之后的模型关于样本的预测值的期望一样，因此单个模型的偏差决定集成之后的模型的偏差，因此要尽量选择偏差比较小的单模型，通常来说模型越复杂偏差越小，因此尽量选择比较复杂的单模型，如深度很深或者不剪枝的决策树。

算法通过迭代构建一系列的分类器，每次分类都将上一次分错的数据权重提高一点再进行下一个分类器分类，这样最终得到的分类器在测试数据与训练数据上都可以得到比较好的成绩。其代表算法为 AdaBoost、GBDT、XGBoost。

而对于 Boosting 来说，每一步我们都会在上一轮的基础上更加的拟合原数据，所以可以保证偏差，所以对于每个基分类器来说，问题就是如何选择方差更小的分类器，即更简单的弱分类器，所以我们选择深度很浅的决策树。

从优化角度来看，是用 forward-stagewise 这种贪心法去最小化损失函数。因此 boosting 是在 sequential 地最小化损失函数，其 bias 自然逐步下降。但由于是采取这种 sequential、adaptive 的策略，各子模型之间是强相关的，于是子模型之和并不能显著降低 variance。所以说 boosting 主要还是靠降低 bias 来提升预测精度。

- **方差偏差分解**

Bias：Bias 是 “用所有可能的训练数据集训练出的所有模型的输出的平均值”与 “真实模型”的输出值之间的差异；

方差：Variance 则是“不同的训练数据集训练出的模型”的输出值之间的差异。

Boosting：从偏差—方差分解角度看，降低偏差。

Bagging：从偏差—方差分解角度看，降低方差。

偏差指的是算法的期望预测与真实值之间的偏差程度，反映了模型本身的拟合能力；方差度量了同等大小的训练集的变动导致学习性能的变化，刻画了数据扰动所导致的影响。

https://cloud.tencent.com/developer/article/1483259

- **随机森林和 GBDT**

https://zhuanlan.zhihu.com/p/37676630

- **GBDT 算法以及 GBDT 做分类**

GBDT 无论用于分类还是回归，一直使用的是 CART 回归树。

GBDT 二分类算法，就是用多棵 CART 回归树拟合结果为 1 的对数几率，损失函数使用对数损失函数，只是需要将预测值 yi'(预测为 1 的概率)用回归树预测

的对数几率 F(x)来代替，然后每轮再去拟合残差即可，得到最终分类器输出之后，将输出经过 sigmoid 函数即可得到预测结果为 1 的概率。

https://zhuanlan.zhihu.com/p/508713509

- GBDT，XGBoost，LightGBM

https://zhuanlan.zhihu.com/p/148050748

【决策树】

https://zhuanlan.zhihu.com/p/266880465

https://zhuanlan.zhihu.com/p/267368825

【EM 算法】

- **原理**

已知的是观察数据，未知的是隐含数据和模型参数

EM 算法解决这个的思路是使用启发式的迭代方法，既然我们无法直接求出模型分布参数，那么我们可以先猜想隐含参数（EM 算法的 E 步），接着基于观察数据和猜测的隐含参数一起来极大化对数似然，求解我们的模型参数（EM 算法的 M 步)。由于我们之前的隐含参数是猜测的，所以此时得到的模型参数一般还不是我们想要的结果。我们基于当前得到的模型参数，继续猜测隐含参数（EM 算法的 E 步），然后继续极大化对数似然，求解我们的模型参数

（EM 算法的 M 步)。以此类推，不断的迭代下去，直到模型分布参数基本无变化，算法收敛，找到合适的模型参数。

一个最直观了解 EM 算法思路的是 K-Means 算法。在 K-Means 聚类时，每个聚类簇的质心是隐含数据。我们会假设 K 个初始化质心，即 EM 算法的 E 步；然后计算得到每个样本最近的质心，并把样本聚类到最近的这个质心，即 EM 算法的 M 步。重复这个 E 步和 M 步，直到质心不再变化为止，这样就完成了 K-Means 聚类。

- **收敛性**

EM 算法可以保证收敛到一个稳定点，但是却不能保证收敛到全局的极大值点，因此它是局部最优的算法，当然，如果我们的优化目标是凸的，则 EM 算法可以保证收敛到全局极大值，这点和梯度下降法这样的迭代算法相同。

【过拟合】

- **解决过拟合的方法**

1、模型角度：正则化、BatchNorm 和 LayerNorm、Dropout

2、数据角度：增加训练数据、数据增强、标签平滑、引入先验知识

3、其他方法：交叉验证，预训练等

- **Dropout 为什么可以解决过拟合？**

  每次做完 dropout，对于某些神经网络单元，相当于从原始的网络以概率 P 暂时从网络中进行丢弃，找到一个更瘦的网络。

  Dropout 的意义在于，减小了不同神经元的依赖。有些中间输出，在给定的训练集上，可能发生只依赖某些神经元的情况，这就会造成对训练集的过拟合。而随机关掉一些神经元，可以让更多神经元参与到最终的输出当中。我觉得 dropout 方法也可以看成，联合很多规模比较小的网络的预测结果，去获取最终的预测。

【归一化】

- **为什么要做归一化**

把数据映射到 0-1 范围内,使得处理过程更加便捷;提高不同数据特征之间的可比性。

- **各种归一化的区别和优缺点**

最常用的有 z 标准化和最小最大标准化方法。z 标准化一般在处理近似正态分布的数据比较合适，否则使用最小最大标准化，将数据的范围压缩到 0~1 区间。

- **为什么 NLP 不用 batchnorm？**

LayerNorm 是对特征维度，也就是句子中一个 Tokens 的向量表示进行归一化，BatchNorm 是对 batch_size*seq_size 维度范围，也就是一个 batch 中所有 Tokens 向量的第 i 维进行归一化。NLP 不使用 BatchNorm 有以下几个原因：

（1）一个 batch 中不同句子字符长度不等，虽然通过补 0 或截断后能达到相同的句子长度，对这样一个 batch 进行 BatchNorm，反而会加大特征的方差。

（2） NLP 中一个 batch 中所有 Tokens 的第 i 维向量关联性不大，对他们进行 BatchNorm 会损失 Tokens 之间的差异性。而我们想保留不同 Tokens 之间的差异性，所以不在该维度进行 Norm

（3）有实验证明，将 LayerNorm 换为 BatchNorm 后，会使得训练中 Batch 的统计量以及统计量贡献的梯度不稳定，Batch 的统计量就是 Batch 中样本的均值和方差，Batch 的统计量不稳定也就是当前 Batch 的均值和运行到当前状态累加得到的均值间的差值有的大，有的小，方差也是类似的情况。

（4）LayerNorm 对 self-attention 处理累加得到向量进行归一化，降低其方差，加速收敛。

【正则化】

● 正则化怎么做，L1 和 L2

模型复杂度如果过高，会出现过拟合的情况，正则化的主要目的就是控制模型复杂度，减小过拟合。最基本的正则化方法是在原目标（代价）函数中添加惩罚项，对复杂度高的模型进行“惩罚”。这个惩罚项是一个参数 a 乘以一个函数，a 是用于控制正则化强弱的。对于 l1 正则化来说，函数是 l1 范数，对于 l2 来说是 l2 范数。

L1 范数：向量中各个元素绝对值之和

L2 范数：向量各元素的平方和然后求平方根

- **现象**

l2 正则化的效果是对原最优解的每个元素进行不同比例的放缩；l1 正则化则会使原最优解的元素产生不同量的偏移，并使某些元素为 0，从而产生稀疏性。

- **先验**

l1 正则化可通过假设权重 w 的先验分布为拉普拉斯分布，由最大后验概率估计导出；

l2 正则化可通过假设权重 w 的先验分布为高斯分布，由最大后验概率估计导出。

【初始化】

- **不同网络的初始化有什么区别**

全 0 初始化

全 0 初始化是最差的初始化方法，只适用于单神经元神经网络。对于任何具有隐藏层的神经网络，如果初始化时所有的权重都是相同的，则任意一层多神经元的效果都与单神经元相同，无法对多特征进行学习，白白浪费计算力。

而全 0 初始化又是相同权重初始化的一种特殊情况，根据反向传播的链式法则，具有隐藏层的神经网络参数将永远不会更新，cost 函数永远不会下降。

随机初始化

随机初始化是最常见的一种初始化方法：W 参数全部随机初始化，b 参数全部初始化为 0。在 python 中，使用 np.random.randn 和 np.zeros 函数初始化即可。

Xavier 初始化

条件：正向传播时，激活值的方差保持不变；反向传播时，关于状态值的梯度的方差保持不变。其中，激活值是激活函数输出的值，状态值是输入激活函数的值。

初始化方法为，

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}]$$

He 初始化（Kaiming 初始化）

条件：正向传播时，状态值的方差保持不变；反向传播时，关于激活值的梯度的方差保持不变。

初始化方法为（适用于 ReLU），

$$W \sim N[0, \sqrt{\frac{2}{\hat{n}_i}}]$$

在 python 中，实现方式是在随机初始化的基础上，乘以 np.sqrt(u / layers_dims[l - 1])。layers_dims[l - 1]即上一层神经元的个数，u 是一个可调的参数。经实践，对于 tanh，u 的取值为 1；对于 relu，u 的取值为 2。

He 初始化方法主要适用于 ReLU 和 Leaky ReLU 等激活函数。

- **神经网络的隐层可以全部初始化为 0 吗**

将偏置项 b 初始化为 0 实际上是可行的，但把 W 初始化成全零就成问题了，它的问题在于给神经网络输入任何的样本，隐藏层的两个单元都是相同的值。每个单元的节点的权重是一样的，即权重矩阵的每一行是相同的。那么在反向传播的时候，所有节点的梯度改变是一样的，最后计算出来输入层与隐藏层之间的参数更新都是一样的。

一直这样迭代下去，神经网络的每一次正向和反向传播，隐藏层的每个激活单元其实都在计算同样的东西，其实也就是一个"对称网络"，这种网络无法学习到什么有趣的东西，因为每个隐藏层的激活单元都在计算同样的东西，那么隐藏层其实相当于只有一个单元，因此神经网络的 W 参数矩阵不能随机初始化为 0。

【激活函数】

一般会问你某个激活函数（比如 sigmoid）的公式和优缺点，比如在问逻辑回归相关的题的时候混杂着问。

- **作用**

网络只是线性模型，只能把输入线性组合再输出，不能学到复杂的映射关系，所以用激活函数做非线性的转换。

【损失函数】

- **二分类的损失函数**

  交叉熵

- **为什么分类不用 MSE**

用方差容易出现多个局部最优解（即非凸函数），这样很难找到全局最优训练出好的模型，这样很依赖初始权值的起点。反之，用交叉熵就很容易找到全局最优，因为是代价函数是大部分情况是凸函数。

【信息论】

https://www.zhihu.com/question/591173254/answer/2950347352?utm_id=0

https://zhuanlan.zhihu.com/p/292434104?utm_id=0

【样本不均衡的解决方案】

- **过采样**

  随机采样：从少数类的样本中进行随机采样来增加新的样本

  SMOTE: 对于少数类样本 a, 随机选择一个最近邻的样本 b, 然后从 a 与 b 的连线上随机选取一个点 c 作为新的少数类样本;

  ADASYN: 关注的是在那些基于 K 最近邻分类器被错误分类的原始样本附近生成新的少数类样本

- **降采样**

（1）随机降采样

从多数类中随机选择一些样样本组成样本集。然后将新样本集与少数类样本集合并。

（2）Edited Nearest Neighbor (ENN)

遍历多数类的样本，如果他的大部分 k 近邻样本都跟他自己本身的类别不一样，我们就将他删除；

（3）Repeated Edited Nearest Neighbor（RENN）

重复以上 ENN 的过程直到没有样本可以被删除；

（4）Tomek Link Removal

其思想是，类别间的边缘可能增大分类难度，通过去除边缘中的多数类样本可以使得类别间 margin 更大，便于分类。具体方法是：如果有两个不同类别的样本，它们的最近邻都是对方，也就是 A 的最近邻是 B，B 的最近邻是 A，那么 A,B 就是 Tomek link，我们要做的就是将所有 Tomek link 都删除掉。那么一个删除 Tomek link 的方法就是，将组成 Tomek link 的两个样本，如果有一个属于多数类样本，就将该多数类样本删除掉。这样我们可以发现正负样本就分得更开了。

- **带权重的 loss**

https://blog.csdn.net/dfly_zx/article/details/123152907

【数据预处理】

连续特征归一化，离散特征 one-hot 编码

【梯度消失和梯度爆炸】

● 梯度消失和梯度爆炸的原因

当梯度消失发生时，最后一个隐层梯度更新基本正常，但是越往前的隐层内更新越慢，甚至有可能会出现停滞，此时，多层深度神经网络可能会退化为浅层的神经网络（只有后面几层在学习），因为浅层基本没有学习，对输入仅仅做了一个映射而已。

当梯度爆炸发生时，最后一个隐层梯度同样更新正常，但是向前传播的梯度累计过程中，浅层网络可能会产生剧烈的波动，从而导致训练下来的特征分布变化很大，同时输入的特征分布可能与震荡幅度不同，从而导致最后的损失存在极大的偏差。

梯度消失和梯度爆炸本质上是一样的，均因为网络层数太深而引发的梯度反向传播中的连乘效应。

● 解决方案

权重初始化+激活函数

Batchnorm

【优化器】

- **作用**

  用来更新和计算影响模型训练和模型输出的网络参数,使其逼近或达到最优值,从而最小化(或最大化)损失函数。

- **各种优化器原理、公式、公式符号的意思**

https://zhuanlan.zhihu.com/p/64113429

- **从 SGD 到 Adam 做了哪些改进**

  自适应的学习率、动量

- **从 SGD 到 Adam 做了哪些改进**

Adamw 即 Adam + weight decate ,效果与 Adam + L2 正则化相同，但是计算效率更高,因为 L2 正则化需要在 loss 中加入正则项，之后再算梯度，最后反向传播，而 Adamw 直接将正则项的梯度加入反向传播的公式中，省去了手动在 loss 中加正则项这一步。

【评价指标】

Precision，也叫精确率。是指被预测为阳性的样本中，有多少是预测正确的（针对于预测结果来说的）

Precision=TP/（TP+FP）

准确率 Accuracy = （TPR+TNR）/2

AUC（area under Curve），是 ROC 曲线下的面积，具体概念这里不赘述了。AUC 能反映模型的排序能力，他反应的是一个相对性，即 item a 排在 item b 之前的能力；但是它不反应绝对性，例如，0.9 排在 0.1 前面和 0.5 排在 0.1 前面对他来说是一样的。

为什么 AUC 和 logloss 比 accuracy 更常用呢？因为很多机器学习的模型对分类问题的预测结果都是概率，如果要计算 accuracy，需要先把概率转化成类别，这就需要手动设置一个阈值，如果对一个样本的预测概率高于这个预测，就把这个样本放进一个类别里面，低于这个阈值，放进另一个类别里面。所以这个阈值很大程度上影响了 accuracy 的计算。使用 AUC 或者 logloss 可以避免把预测概率转换成类别。（前面讲了 ROC 是怎么画图的，画图的时候用的就是概率，并没有转换成类别，AUC 就是 ROC 曲线下的面积）

- **工业界为什么用 auc？**

1. AUC 指标本身和模型预测 score 绝对值无关，只关注排序效果，因此特别适合排序业务，为何与模型预测 score 值无关为何是很好的特性呢?假设你采用 precision、F1 等指标，而模型预测的 score 是个概率值，就必须选择一个阈值来决定哪些样本预测是 1 哪些是 0，不同的阈值选择，precision 的值会不同，而 AUC 可以直接使用 score 本身，参考的是相对顺序，更加好用。

2. AUC 对均匀正负样本采样不敏感。正由于 AUC 对分值本身不敏感，故常见的正负样本采样，并不会导致 auc 的变化。比如在点击率预估中，处于计算资源的考虑，有时候会对负样本做负采样，但中于采样完后并不影响正负样本的顺序分布。即假设采样是随机的，采样完成后，给定一条正样本，模型预测为 score1，由于采样随机，则大于 score1 的负样本和小于 score1 的负样本的比例不会发生变化。

- **代码**

https://zhuanlan.zhihu.com/p/82978513?utm_id=0

【BERT 和 Transformer】

- **介绍一下** BERT/Transformer
- BERT **的两个训练任务？**

  MLM 和 Next Sentence Prediction

- BERT **的优化器（**AdamW**）？和** Adam **的区别（前面写了）？**
- Attention **和** self-attention **的区别**

  self-attention 比 attention 约束条件多了两个：

  1. Q=K=V（同源）

  2. Q，K，V 需要遵循 attention 的做法

- Self-attention **的公式，计算过程**

首先 QKV 是由输入序列 X 经过矩阵变换得到的，可以认为是对原始输入 X 做了某种特征提取，不直接使用 X 是为了提高模型的可学习性。其中 Q 和 K 是用来计算词语之间相似度的，即关注程度，V 可以理解为某个词的特征表征，与 QK 计算到的权重系数相乘，然后求和，即实现了加权求和，实现了注意力机制。

为什么要除以一个维度：缩放后后注意力分数矩阵中分数的方差由原来的缩小到了 1，方差越小，点积的数量级越小。

- **多头的意义**

多头注意力增加复杂度吗？不增加

多头的意义？多头注意力允许模型在不同位置共同关注来自不同表示子空间的信息。使用一个单注意力头，会抑制这种情况。

在这篇论文中 http://arxiv.org/pdf/1905.0941 讨论了多头的作用，发现并不是头越多越好，去掉一些头效果依然有不错的效果（而且效果下降可能是因为参数量下降），这是因为在头足够的情况下，这些头已经能够有关注位置信息、关注语法信息、关注罕见词的能力了，再多一些头，无非是一种 enhance 或 noise 而已。

- **对比 LSTM，Transformer 的优点？**

上下文感知、并行计算

- **BERT 怎么解决 OOV out of vobulary**

如果一个单词不在词表中，则按照 subword 的方式逐个拆分 token，如果连逐个 token 都找不到，则直接分配为[unknown]

【ChatGPT】

● 对 ChatGPT 发展的看法

自己编！！！

● 对比 GPT-3 的性能提升

ChatGPT 对比 GPT-3 的性能提升主要来源于以下四个方面：

（1）大模型。大模型拥有更多的文本数据和代码数据，这是 ChatGPT 具有优秀推理能力的基础。

（2）用代码进行预训练。代码和文本相比，需要更长的语言依赖如函数调用。因此，在代码上进行预训练，拆分代码流程，这使得 ChatGPT 的语言理解能力更为强大，能够得到更好的训练效果。

（3）Prompt/Instruction Tuning。用提示/指令模版去进行微调，使得语言模型能够适应人的语言逻辑，而不是人类去理解语言模型的参数。

（4）人类反馈的强化学习（RLHF）。ChatGPT 使用了真实的用户反馈数据，使得生成结果在多样性和安全性等方面更符合人类的期望。

● RLHF 原理

RLHF 原理：

RLHF 的训练过程可以分解为三个核心步骤：

1、选择一个经典的预训练语言模型作为初始模型，例如 OpenAI 在 InsturctGPT 中使用小规模参数版本的 GPT-3，

2、收集上述预训练语言模型产生的数据来训练一个奖励模型，这个模型可以看作一个判别式的语言模型，输入是 prompt 和模型的回答，输出是人类的满意度，但是这里标注人员的任务是对生成的回答进行排序，比如说给定同一个 prompt，让两个语言模型同时生成文本，然后比较这两段文本哪个好

3、通过强化学习来基于奖励模型优化初始的语言模型，强化学习的策略就是基于该语言模型，接收 prompt 作为输入，然后输出一系列文本（或文本的概率分布）；动作空间就是词表所有 token 在所有输出位置的排列组合；观察空间则

是可能的 prompt 序列；奖励函数则是刚才的奖励模型的计算结果再叠加一个约束项。

（约束项：假设用一个 prompt 得到两个模型的输出 y1 和 y2，再用奖励模型打分，这两个打分的差值作为一个信号，用 KL 散度计算奖励/惩罚的大小，y2 打分比 y1 高得越多奖励越大，反之惩罚越大。这样是为了防止当前模型围着初始模型“绕圈”，避免模型通过一些取巧的方式获得高额 reward）

最后根据 Proximal Policy Optimization(PPO)算法更新模型参数。

NLP 算法工程师面试
大模型相关

基础部分

（这部分答案请参考我之前的帖子）

1、self-attention 的公式及参数量，为什么用多头，为什么要除以根号 d

2、BERT 和 GPT 的训练方式（预训练任务、训练细节）

3、transformer 架构

*注意要会**手撕** self-attention，熟知公式里的参数是做什么的，其实没有大模型之前面试就常考，大模型出来以后更是常考……

大模型

1、大模型的模型结构

一般指一亿参数以上的模型。

目前以 Transformer 为基础自回归生成大致可以分为三种架构：

- Encoder-only 的模型，如 BERT
- Encoder-Decoder 的模型，如 T5。
- Decoder-Only 的模型，如 GPT 系列。

用 GPT-4 举例，它使用 Transformer 的 Decoder 结构，并对 Transformer Decoder 进行了一些改动。->介绍 Transformer Decoder 的结构

2、ChatGPT 对比 GPT-3 的性能提升主要来源于以下四个方面：

[1] 大模型。大模型拥有更多的文本数据和代码数据，这是 ChatGPT 具有优秀推理能力的基础。

[2] 用代码进行预训练。代码和文本相比，需要更长的语言依赖如函数调用。因此，在代码上进行预训练，拆分代码流程，这使得 ChatGPT 的语言理解能力更为强大，能够得到更好的训练效果。

[3] Prompt/Instruction Tuning。用提示/指令模版去进行微调，使得语言模型能够适应人的语言逻辑，而不是人类去理解语言模型的参数。

[4] 人类反馈的强化学习（RLHF）。ChatGPT 使用了真实的用户反馈数据，使得生成结果在多样性和安全性等方面更符合人类的期望。

3、InstructGPT 和 ChatGPT 模型中使用的关键技术（SFT->RLHF）

训练过程可以分解为三个核心步骤：

[1] SFT：生成模型 GPT 的有监督精调（supervised fine-tuning）。选择一个经典的预训练语言模型作为初始模型，例如 OpenAI 在 InstructGPT 中使用小规模参数版本的 GPT-3，

->RLHF

[2] RM：奖励模型的训练（reward model training）。收集上述预训练语言模型　产生的数据来训练一个奖励模型，这个模型可以看作一个判别式的语言模型，输入是 prompt 和模型的回答，输出是人类的满意度，但是这里标注人员的任务是对生成的回答进行排序，比如说给定同一个 prompt，让两个语言模型同时生成文本，然后比较这两段文本哪个好

[3] PPO：近端策略优化模型（Proximal Policy Optimization）。通过强化学习来基于奖励模型优化初始的语言模型，强化学习的策略就是基于该语言模型，接收 prompt 作为输入，然后输出一系列文本（或文本的概率分布）；动作空间就是词表所有 token 在所有输出位置的排列组合；观察空间则是可能的 prompt 序列；奖励函数则是刚才的奖励模型的计算结果再叠加一个约束项。（约束项：假设用一个 prompt 得到两个模型的输出 y1 和 y2，再用奖励模型打分，这两个打分的差值作为一个信号，用 KL 散度计算奖励/惩罚的大小，y2 打分比 y1 高得越多奖励越大，反之惩罚越大。这样是为了防止当前模型围着初始模型“绕圈”，避免模型通过一些取巧的方式获得高额 reward）

最后根据 PPO 算法更新模型参数。

4、大模型中常用的位置编码

改写自某乎：归来仍是少年

详解点链接 https://zhuanlan.zhihu.com/p/631003833

位置编码分为绝对位置编码和相对位置编码，以前的语言模型如 BERT 主要使用绝对位置编码，比如三角函数位置编码，但最大的长度是 512，所以需要截断输入，所以后续的大模型主要研究相对位置编码。

（考虑到面试一般时间比较紧，介绍常用的两三个和他们的区别就行）

- ROPE（旋转位置编码）

谷歌 PaLM 和 LLaMA 中采用的位置编码，通过复数形式来改进三角函数绝对位置编码。

• Alibi 位置编码（Attention with Linear Biases）

Alibi 位置编码主要是 Bloom 模型采用，Alibi 的方法也算较为粗暴，是直接作用在 attention score 中，给 attention score 加上一个预设好的偏置矩阵，相当于 q 和 k 相对位置差 1 就加上一个 -1 的偏置。其实相当于假设两个 token 距离越远那么相互贡献也就越低。

区别：

Alibi 位置编码的外推性比旋转位置编码外推性要好一些，旋转位置编码也是基于正余弦三角式位置编码改进融入相对位置信息，但是正余弦三角式位置编码外推性缺点也很明显，看起来是不需要训练可以直接推演无限长度位置编码，但是忽略了一点就是周期性函数必须进行位置衰减，到远处的位置信息趋于直线震荡，基本很难有位置信息区分了，所以外推性比训练式的好不了多少，旋转位置编码基于此改进的自然也是如此。

Alibi 相当于在 k 和 q 向量内积上加入分数上的偏置，来体现出来位置差异性，针对于远距离衰减问题，则是通过 softmax 函数特性进行差异软放大，将 token 之间的位置差异性拉大，避免远距离时被衰减无限接近于 0，因为直接作用在 attention 分数上，拉大远距离内积值，在训练的时候带来的位置差异性减少的问题会大大缓解，从而获得更远距离的外推性能。

5、大模型的微调方法

改写自某乎：YBH
详解 https://zhuanlan.zhihu.com/p/636481171

• LoRA

基于大模型的内在低秩特性，增加旁路矩阵来模拟 full finetuning。具体来说，LoRA 冻结一个预训练模型的矩阵参数，并选择用 A 和 B 矩阵来替代，在下游任务时只更新 A 和 B。

• Adapter

在预训练模型每一层（或某些层）中添加 Adapter 模块，微调时冻结预训练模型主体，由 Adapter 模块学习特定下游任务的知识。每个 Adapter 模块由两个 Feed-Forward 层组成，通常情况下将原始输入降维再升维，通过降维后的维度控制 Adapter 的参数量大小。

• Prefix-tuning

前缀微调（prefix-tunning），用于生成任务的轻量微调。前缀微调将一个连续的特定于任务的向量序列添加到输入，称之为前缀，如下图中的红色块所示。与提示（prompt）不同的是，前缀完全由自由参数组成，与真正的 token 不对应。相比于传统的微调，前缀微调只优化了前缀。因此，我们只需要存储一个大型 Transformer 和已知任务特定前缀的副本，对每个额外任务产生非常小的开销。

• Prompt-tuning

给每个任务定义自己的 Prompt，拼接到数据上作为输入，同时 freeze 预训练模型进行训练，在没有加额外层的情况下，随着模型体积增大，Prompt-tuning 的效果越来越好，最终追上精调的效果。