极端正负例不平衡的应对办法

MLE 算法指北

2025 年 2 月 16 日

1 引言

点击率（Click-Through Rate, CTR）预测是在线广告、推荐系统等领域的重要任务。CTR 预测的目标是估计用户对某个广告或推荐内容的点击概率。然而，** 正负样本极度不均衡 ** 是 CTR 预测的核心挑战之一。通常情况下：

- 正样本（点击）比例极低，通常仅占 $0.1\% - 1\%$。
- 负样本（未点击）占据数据的绝大部分。

如果直接训练模型，模型可能会 ** 严重倾向于预测负样本 **，导致点击的召回率（Recall）极低，使广告和推荐系统的商业价值大打折扣。

为了解决此问题，我们从 ** 数据采样、损失函数调整、推理校准 ** 三个方面进行深入探讨，并提供相应的数学推导。

—

2 数据采样：负例采样与正例扩增

2.1 负例采样（Negative Sampling）

由于负样本远多于正样本，我们可以通过 ** 负例采样（Negative Sampling）** 和 ** 正例扩增（Positive Oversampling）** 来平衡数据分布。

2.1.1 随机负例采样（Random Undersampling）

随机负例采样（Random Undersampling, RUS）是最简单的负例采样策略。它的核心思想是：

- 在原始数据集中，** 随机选择一部分负样本 **，使其数量与正样本接近。
- 控制负样本比例 α，即：

$$\hat{N}_- = \alpha N_+ \tag{1}$$

其中：

  - N_+ 是正样本（点击）数量。
  - N_- 是原始负样本（未点击）数量，且 $N_- \gg N_+$。
  - $\hat{N}_-$ 是采样后的负样本数量。
  - α 是采样比例，一般取 **5 - 10**。

优缺点分析
优点：

- 简单易实现，计算成本低。
- 适用于数据量极大的情况，能快速减少训练数据大小，加速模型训练。

缺点：

- ** 随机性较强，可能丢失重要的负样本 **，导致模型泛化能力下降。
- ** 没有考虑负样本的重要性 **，可能保留对模型无价值的负样本。

Python 实现

```
from imblearn.under_sampling import RandomUnderSampler

# 负例采样比例 0.1 (即负样本数量 = 正样本数量 * 10)
undersampler = RandomUnderSampler(sampling_strategy=0.1, random_state=42)
X_resampled, y_resampled = undersampler.fit_resample(X_train, y_train)
```

—

2.1.2 重要性负例采样（Hard Negative Mining）

重要性负例采样（Hard Negative Mining, HNM）的核心思想是：

- 不是随机选择负样本，而是 ** 优先选择“难分类”的负样本 **。
- 计算所有负样本的 ** 预测得分 $\hat{y}$**，并选择接近阈值 0.5 的负样本：

$$\hat{N}_- = \{x \in N_- \mid |\hat{y} - 0.5| < \epsilon\} \tag{2}$$

其中 ϵ 是一个超参数，表示选择“难分类”样本的范围。

优缺点分析
优点：

- 只保留对模型最具挑战性的负样本，提高训练效率。
- 适用于 ** 对抗学习（Adversarial Learning）** 场景，例如目标检测、CTR 预测等。

缺点：

- 需要 ** 先训练一个初始模型 **，再进行 Hard Negative 采样，计算成本较高。
- 可能会导致 ** 模型对特定负样本过拟合 **，损害泛化能力。

Python 实现

```
import numpy as np

# 假设模型已经预测了负样本的点击概率
y_pred_neg = np.random.rand(10000)  # 负样本预测得分

# 选择得分接近 0.5 的负样本
epsilon = 0.05
hard_negatives = np.where((y_pred_neg > 0.5 - epsilon) & (y_pred_neg < 0.5 + epsilon))

X_hard_neg = X_neg[hard_negatives]
```

—

2.1.3 基于业务规则的负例采样（Business-driven Sampling）

在工业界，负例采样通常结合 ** 业务规则（Business Logic）** 进行。例如：

- ** 最近活跃用户更重要 **：优先采样最近 7 天有过浏览记录的用户。
- ** 广告相关性 **：优先选择与广告更匹配但未点击的样本。
- ** 页面曝光次数 **：选择 ** 曝光多次但未点击的用户 **，因为这些用户可能是潜在点击者。

优缺点分析
优点：

- 结合业务场景，提高模型的实际效果。
- ** 更符合实际用户行为模式 **，可提高 ROI（投资回报率）。

缺点：

- 需要深入了解业务数据，适用范围较窄。
- ** 依赖手工规则 **，难以适应复杂的数据模式。

Python 实现

```
# 选择最近 7 天活跃用户
X_active_users = X_neg[X_neg["last_active_days"] <= 7]

# 选择曝光次数 >= 5 但未点击的用户
X_high_exposure = X_neg[X_neg["impressions"] >= 5]
```

—

方法	优点	缺点
随机负例采样	简单易实现，计算效率高	丢失信息，可能影响模型性能
Hard Negative 采样	保留“最难”负样本，提高模型学习能力	计算开销大，可能过拟合
业务规则采样	结合业务，提高 ROI	依赖领域知识，适用范围有限

表 1: 负例采样方法对比

2.2 正例扩增（Positive Oversampling）

当正样本过少时，可以通过 ** 合成数据 ** 来增加正样本数量。

2.2.1 传统过采样（Random Oversampling）的问题

最简单的处理类别不均衡的方法是 ** 随机过采样（Random Oversampling）**，即：

- 直接 ** 复制少数类样本 **，以增加少数类样本数量，使其与多数类接近。

缺点：

- 复制样本 ** 不会增加数据多样性 **，导致模型容易过拟合（Overfitting）。
- 复制样本 ** 无法学习少数类数据的分布 **。

—

2.2.2 SMOTE 算法核心思想

**SMOTE 通过在少数类样本之间进行插值，生成新的合成样本 **，其核心思想为：

1. 对于每个少数类样本 x_i，找到其在少数类中的 k 个最近邻样本。
2. 随机选择一个邻居 x_j，并按照以下公式生成一个新的合成样本：

$$x_{\text{new}} = x_i + \lambda(x_j - x_i), \quad \lambda \sim U(0,1) \tag{3}$$

其中，λ 是一个随机数，服从均匀分布 $U(0,1)$。

3. 该过程重复，直到生成足够的新样本。

优点：

- 生成的新样本位于现有样本之间，避免了 ** 样本重复 ** 的问题，提高数据多样性。
- 保持少数类数据的 ** 局部结构 **，更好地表示少数类样本的分布。

2.2.3 SMOTE 详细数学推导

SMOTE 主要依赖于 **k 近邻（k-Nearest Neighbors, kNN）**，其数学推导如下：

1. 计算欧几里得距离（Nearest Neighbors）

给定一个少数类样本集 $\mathcal{X} = \{x_1, x_2, ..., x_N\}$，我们首先计算每个样本 x_i 与其他样本 x_j 的欧几里得距离：

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{d} (x_{ik} - x_{jk})^2} \tag{4}$$

其中：

- x_{ik} 表示样本 x_i 的第 k 维特征。
- d 是特征空间的维度。

然后，选择 k 个最近邻样本 $\{x_{j1}, x_{j2}, ..., x_{jk}\}$。

—

2. 生成新样本

对于每个样本 x_i，我们随机选择一个邻居 x_j，并按照以下方式生成新的样本：

$$x_{\text{new}} = x_i + \lambda(x_j - x_i), \quad \lambda \sim U(0,1) \tag{5}$$

直观理解：

- 这个公式表示从 x_i 到 x_j 之间的某个随机位置生成新样本。
- 通过引入 λ 这一随机变量，我们可以在 x_i 和 x_j 之间任意插值，而不是简单地复制数据点。

矩阵表示：如果我们需要生成 m 个新样本，那么：

$$X_{\text{new}} = X + \Lambda \odot (X' - X) \tag{6}$$

其中：

- X 是原始少数类样本矩阵。
- X' 是其最近邻矩阵。
- Λ 是一个随机矩阵，每个元素 $\lambda \sim U(0,1)$。
- $\odot$ 表示逐元素乘法（element-wise multiplication）。

—

2.2.4 SMOTE 的 Python 实现

使用 Scikit-Learn 进行 SMOTE 过采样：

```
from imblearn.over_sampling import SMOTE
import numpy as np

# 生成模拟数据
X = np.random.rand(100, 5)  # 100个样本，5个特征
y = np.array([1] * 10 + [0] * 90)  # 10个正样本，90个负样本

# 进行 SMOTE 过采样
smote = SMOTE(sampling_strategy=0.5, random_state=42)  # 生成正样本至 50%
X_resampled, y_resampled = smote.fit_resample(X, y)

print(f"原始数据集大小: {X.shape}")
print(f"过采样后的数据集大小: {X_resampled.shape}")
```

2.2.5 SMOTE 的优势与局限性

优势：

- 生成的数据比 ** 简单复制 ** 更自然，提高模型泛化能力。
- 适用于 ** 非线性分类器 **（如深度学习、SVM 等）。

局限性：

- 可能会引入 ** 噪声样本 **，导致过拟合。
- 对于 ** 高维数据 **，KNN 计算距离可能不够稳定。

3 训练阶段：损失函数调整

即使数据采样后，模型仍然可能 ** 对负样本更敏感 **，因为标准的二元交叉熵（BCE）损失不会考虑类别不均衡。因此，我们需要调整损失函数。

3.1 加权 BCE 损失（Weighted BCE Loss）

标准 BCE 损失为：

$$L_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)] \tag{7}$$

我们引入类别权重 α 和 β：

$$L_{\text{weighted}} = -\frac{1}{N}\sum_{i=1}^{N}[\alpha y_i \log \hat{y}_i + \beta(1-y_i)\log(1-\hat{y}_i)] \tag{8}$$

推导过程： 1. 我们希望正负样本的损失贡献均衡，即：

$$\sum_{\text{正例}} \alpha \log \hat{y} \approx \sum_{\text{负例}} \beta \log(1-\hat{y}) \tag{9}$$

2. 由于 N_+ 远小于 N_-，我们放大正样本的贡献，使其对损失的影响接近负样本。3. 归一化权重，使 $\alpha N_+ + \beta N_- \approx N$，则：

$$\alpha = \frac{N}{2N_+}, \quad \beta = \frac{N}{2N_-} \tag{10}$$

总结： - 当正样本较少时，α 较大，意味着 ** 放大正样本的损失贡献 **，增强模型对正样本的关注度。- 当负样本较多时，β 较小，意味着 ** 减少负样本的损失贡献 **，避免模型偏向多数类。**实现代码：**

```
import torch.nn.functional as F

def weighted_bce_loss(y_pred, y_true, pos_weight, neg_weight):
    loss = F.binary_cross_entropy(y_pred, y_true, reduction="none")
    weights = torch.where(y_true == 1, pos_weight, neg_weight)
    return (loss * weights).mean()
```

3.2 Focal Loss

Focal Loss 通过在 BCE 的基础上增加一个调节因子，使得：

- 对于容易分类的样本，降低它们的损失贡献。
- 对于难分类的样本，增加它们的损失权重，使模型更关注这些样本。

Focal Loss 的数学定义如下：

$$L_{\text{Focal}} = -\frac{1}{N} \sum_{i=1}^{N} \alpha_t (1-p_t)^{\gamma} \log p_t \tag{11}$$

其中：

- p_t 是样本 i 的预测概率：

$$p_t = \begin{cases} \hat{y}_i, & \text{若} y_i = 1 \\ 1-\hat{y}_i, & \text{若} y_i = 0 \end{cases} \tag{12}$$

- $(1-p_t)^{\gamma}$ 是 ** 调节因子（modulating factor）**，当 $\gamma > 0$ 时：
  - 若 p_t ** 接近 1（易分类）**，$(1-p_t)^{\gamma}$ 接近 0，损失贡献变小。
  - 若 p_t ** 接近 0（难分类）**，$(1-p_t)^{\gamma}$ 接近 1，损失贡献较大。
- $\gamma \geq 0$ 是 ** 聚焦参数（focusing parameter）**，常取 $\gamma \in [1,5]$：
  - 当 $\gamma = 0$ 时，Focal Loss 退化为标准 BCE 损失。
  - 较大的 γ 值使模型更关注难分类样本（但可能影响收敛速度）。
- α_t 是 ** 类别权重参数（class weight）**，用于平衡正负样本：

$$\alpha_t = \begin{cases} \alpha, & \text{若} y_i = 1 \\ 1-\alpha, & \text{若} y_i = 0 \end{cases} \tag{13}$$

- 通常 $\alpha \in [0,1]$，用于控制正负样本的损失贡献，避免负样本过多影响模型学习。

对易分类样本的影响

- 在 BCE 损失中，即使 $\hat{y}_i \approx 1$，损失仍然不为零。
- 在 Focal Loss 中，当 p_t 接近 1 时，$(1-p_t)^{\gamma}$ 接近 0，损失贡献极小，使得 ** 易分类样本对梯度更新影响很小 **。

对难分类样本的影响

- 当 p_t 较小时（接近 0.5 或 0），$(1-p_t)^{\gamma}$ 依然接近 1。
- 这意味着 ** 难分类样本的损失贡献更大 **，模型会更专注于学习它们。

超参数 γ 的影响

- 当 $\gamma = 0$ 时，Focal Loss 退化为 BCE Loss。
- 较大的 γ（如 2 或 5）会减少易分类样本的影响，使模型更关注难分类样本，但训练收敛会变慢。

使用 PyTorch 计算 Focal Loss：

```
import torch
import torch.nn.functional as F

def focal_loss(y_pred, y_true, alpha=0.25, gamma=2.0):
    """
    计算 Focal Loss
    :param y_pred: 预测值 (0-1 概率)
    :param y_true: 真实标签 (0 或 1)
    :param alpha: 类别权重参数
    :param gamma: 聚焦参数
    :return: 计算后的损失值
    """
    bce_loss = F.binary_cross_entropy(y_pred, y_true, reduction="none")
    p_t = torch.exp(-bce_loss)
    loss = alpha * (1 - p_t) ** gamma * bce_loss
    return loss.mean()

# 计算 Focal Loss
loss = focal_loss(y_pred, y_true, alpha=0.25, gamma=2.0)
```

4 推理阶段：概率校准

训练时，由于负例采样，模型预测出的点击率 P_{train} 并不等于真实点击率 P_{real}，因此需要在推理时进行校准。

4.1 方法 1：先验概率校正（Prior Probability Correction）

在训练数据中，由于我们进行了负例采样，导致训练数据的点击率 P_{train} 与真实分布 P_{real} 存在偏差：

$$P_{\text{train}} = \frac{N_+}{N_+ + \hat{N}_-}, \quad P_{\text{real}} = \frac{N_+}{N_+ + N_-} \tag{14}$$

由于训练数据的负样本数量 $\hat{N}_-$ 远少于真实负样本 N_-，导致 $P_{\text{train}} > P_{\text{real}}$，模型的预测概率 $\hat{y}$ 也因此偏大。

根据 Bayes 公式，我们可以对 $\hat{y}$ 进行校正：

$$P_{\text{corrected}} = \frac{P_{\text{train}}}{P_{\text{train}} + (1 - P_{\text{train}}) \cdot \frac{1-P_{\text{real}}}{P_{\text{real}}}} \cdot \hat{y} \tag{15}$$

```
def prior_correction(p_model, p_real=0.01, p_train=0.5):
    """
    先验概率校正
    :param p_model: 预测概率
    :param p_real: 真实CTR
    :param p_train: 训练数据CTR
    :return: 校正后的概率
    """
    return p_model / (p_model + (1 - p_model) * ((1 - p_real) / p_real) * (p_train / (1 - p_train)))

# 应用校正
y_pred_corrected = prior_correction(y_pred_model, p_real=0.01, p_train=0.5)
```

—

4.2 方法 2：Platt Scaling（Logistic Regression 校准）

Platt Scaling 是一种 ** 基于逻辑回归（Logistic Regression）** 的概率校准方法。它的核心思想是：

- 训练阶段，模型的预测值 $\hat{y}$ 可能偏离真实点击率。

- 在推理阶段，我们可以训练一个 **Logistic 回归模型 **，使其输出校正后的概率：

$$P_{\text{Platt}} = \frac{1}{1 + \exp(a\hat{y} + b)} \tag{16}$$

其中 a 和 b 通过最小化 ** 对数损失（Log Loss）** 进行拟合。

```
from sklearn.linear_model import LogisticRegression
from sklearn.calibration import CalibratedClassifierCV

# 训练 Platt Scaling 模型
calibrator = CalibratedClassifierCV(base_estimator=model, method='sigmoid')
calibrator.fit(X_valid, y_valid)

# 预测校正后的概率
y_pred_calibrated = calibrator.predict_proba(X_test)[:, 1]
```

—

4.3 方法 3：Isotonic Regression（非参数校准）

Isotonic Regression 是一种 ** 非参数校准方法 **，它的核心思想是：

- 训练一个 ** 单调递增的映射函数 **，使得模型输出 $\hat{y}$ 映射到真实概率空间。
- 适用于 ** 数据量较大、单调性较强 ** 的校准任务。

```
from sklearn.isotonic import IsotonicRegression

# 训练 Isotonic Regression 校准模型
iso_reg = IsotonicRegression(out_of_bounds="clip")
iso_reg.fit(y_pred_train, y_train)

# 预测校正后的概率
y_pred_corrected = iso_reg.transform(y_pred_test)
```

—

4.4 方法 4：Beta Calibration（Beta 分布校准）

Beta Calibration 通过 ** 拟合 Beta 分布参数 ** 对模型的预测概率进行校正：

$$P_{\text{Beta}} = \frac{1}{B(\alpha, \beta)} \int_0^{\hat{y}} t^{\alpha-1}(1-t)^{\beta-1} dt \tag{17}$$

其中 $B(\alpha, \beta)$ 是 Beta 函数，α 和 β 通过最大似然估计得到。

```
from betacal import BetaCalibration

# 训练 Beta 校准模型
beta_calibrator = BetaCalibration()
beta_calibrator.fit(y_pred_train, y_train)

# 预测校正后的概率
y_pred_corrected = beta_calibrator.predict(y_pred_test)
```

—

4.5 方法 5：Temperature Scaling（温度缩放）

Temperature Scaling 主要用于 ** 深度学习模型的概率校准 **，其核心思想是：

- 训练阶段，模型输出的 logits 可能较大，使得 softmax 输出的概率过度自信。
- 通过调整 logits 的温度参数 T，降低 softmax 输出的置信度：

$$P_{\text{scaled}} = \frac{\exp(z/T)}{\sum_j \exp(z_j/T)} \tag{18}$$

其中 $T > 1$ 使概率变得更平滑。

```
import torch

class TemperatureScaling(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.temperature = torch.nn.Parameter(torch.ones(1) * 1.5)

    def forward(self, logits):
        return logits / self.temperature

# 应用温度缩放
temperature_model = TemperatureScaling()
scaled_logits = temperature_model(logits)
```

—

4.6 方法对比与应用场景

方法	特点	适用场景
先验概率校正	适用于负例采样情况	CTR 预测
Platt Scaling	基于 Logistic 回归，适用于大部分模型	CTR 预测、信用评分
Isotonic Regression	非参数方法，适用于大数据量	医疗诊断、金融风控
Beta Calibration	拟合 Beta 分布，灵活性更强	目标检测
Temperature Scaling	适用于深度学习模型	图像分类、NLP

表 2: 概率校准方法对比