3 LLM 性能评估方法

前面的文章中，我们讨论了LLM训练及推理方法。在训练完成后，我们需要对模型进行评测，以评估其性能。评测方法可分为内在评测 (Intrinsic Evaluation) 和外在评测 (Extrinsic Evaluation) 两类。内在评测不依赖具体任务，直接通过语言模型的输出来评估其生成能力。外在评测则通过机器翻译、摘要生成等具体任务来评估语言模型的处理能力。

3.1 内在评测 (Intrinsic Evaluation)

内在评测使用与预训练数据独立同分布的测试文本，其中最常用的指标是困惑度 (Perplexity)。困惑度衡量了语言模型对测试文本感到"困惑"的程度。设：

- 测试文本为 $s_{test} = w_{1:N}$
- 词典为 D
- w_i为基于前面的词序列 $w_{<i}$ 下一个词为w_i的概率

则困惑度 PPL 的计算公式为：

$$PPL(s_{test}) = P(w_{1:N})^{-\frac{1}{N}} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{<i})}} \tag{1}$$

困惑度值越小，表示语言模型对测试文本越"肯定"（生成概率越高）；反之，困惑度值越大，则表示模型越"不确定"（生成概率越低）。由于测试文本和预训练文本同分布，困惑度可以在一定程度上衡量语言模型的生成能力。

困惑度可以改写为如下等价形式：

$$PPL(s_{test}) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{<i})\right) \tag{2}$$

其中，$-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{<i})$ 可视为生成模型的词分布与测试样本真实词分布间的交叉熵 (Cross Entropy)，即 $-\frac{1}{N}\sum_{i=1}^{N}\sum_{d=1}^{|D|} I(\hat{w}_d = w_i)\log o_{i-1}[w_i]$，其中 D 为词典。由于 $P(w_i|w_{<i}) \leq 1$，此交叉熵是生成模型词分布信息熵 (Information Entropy) 的上界：

$$-\frac{1}{N}\sum_{i=1}^{N} P(w_i|w_{<i})\log P(w_i|w_{<i}) \leq -\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{<i}) \tag{3}$$

因此，困惑度的减小意味着熵减，即模型"胡言乱语"的可能性降低。

困惑度示例代码：

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

def calculate_perplexity(model, test_text, tokenizer, device='cuda'):
    """计算语言模型在测试文本上的困惑度

    Args:
```

```
        model: 语言模型
        test_text: 测试文本
        tokenizer: 分词器
        device: 运行设备

    Returns:
        float: 困惑度值
    """
    model.eval()

    # 对测试文本进行编码
    inputs = tokenizer(test_text, return_tensors='pt')
    input_ids = inputs['input_ids'].to(device)

    with torch.no_grad():
        # 获取模型输出概率分布
        outputs = model(input_ids)
        logits = outputs.logits  # [batch_size, seq_len, vocab_size]

        # 获取每个位置的预测概率，并直接取对数
        # log_softmax(x) = log(softmax(x))
        probs = F.log_softmax(logits, dim=-1)

        # 获取真实token的概率
        shift_probs = probs[..., :-1, :].contiguous()
        shift_labels = input_ids[..., 1:].contiguous()

        # 获取每个位置对应的真实token概率
        # 1. shift_probs的形状为[batch_size, seq_len-1, vocab_size]，表示每个位置上所有词的预测概率
        # 2. shift_labels的形状为[batch_size, seq_len-1]，表示每个位置的真实token索引
        # 3. 使用torch.gather函数从预测概率中提取真实token的概率：
        #    - input: shift_probs，预测概率矩阵
        #    - dim=2，表示在vocab_size维度上收集
        #    - index: shift_labels.unsqueeze(-1)，将真实token索引扩展一维以匹配输入形状
        # 4. 示例说明torch.gather的工作原理：
        #    输入tensor:                索引tensor:
        #    [[1, 2, 3],               [[0],
        #     [4, 5, 6],     dim=1      [1],
        #     [7, 8, 9]]                [2]]
        #    结果: [[1],    # 第0行取第0个元素
        #           [5],    # 第1行取第1个元素
        #           [9]]    # 第2行取第2个元素
        # 5. 最后用squeeze(-1)去掉收集后多余的维度，得到形状[batch_size, seq_len-1]
        token_probs = torch.gather(
            shift_probs,
            2,
            shift_labels.unsqueeze(-1)
        ).squeeze(-1)
```

```
        # 计算序列长度(去掉padding)
        # 1. input_ids != tokenizer.pad_token_id 会生成一个形状为[batch_size, seq_len]的布
尔张量
        # 2. 使用sum()方法计算每个序列中非padding token的数量
        # 3. 减去1是因为序列长度不包括起始token
        seq_length = (input_ids != tokenizer.pad_token_id).sum().item() - 1

        # 计算困惑度
        ppl = torch.exp(-token_probs.sum() / seq_length)

    return ppl.item()

# 使用示例
"""
tokenizer = AutoTokenizer.from_pretrained('model-name')
model = AutoModelForCausalLM.from_pretrained('model-name')

test_text = "今天天气真不错"
ppl = calculate_perplexity(model, test_text, tokenizer)
print(f"困惑度: {ppl}")
"""
```

3.2 外在评测 (Extrinsic Evaluation)

外在评测依赖于具体任务，使用包含问题和标准答案的测试文本来评判语言模型的任务处理能力。外在评测主要包括基于统计指标的评测方法 (Statistical Metrics-based Evaluation) 和基于语言模型的评测方法 (Language Model-based Evaluation)。

3.2.1 基于统计指标的评测 (Statistical Metrics-based Evaluation)

基于统计指标的评测通过构造统计指标来评估语言模型输出与标准答案的契合程度。其中最广泛使用的是精度导向的BLEU和召回导向的ROUGE指标。

3.2.1.1 BLEU

BLEU(Bilingual Evaluation Understudy，双语评估替身)主要用于评估机器翻译任务，通过计算生成的翻译与参考翻译在词级别上的重合程度来评分。设：

- 生成翻译文本集合为 $S_{gen} = \{S_{gen}^i\}_{i=1}^{|S_{gen}|}$
- 对应的参考翻译集合为 $S_{ref} = \{S_{ref}^i\}_{i=1}^{|S_{ref}|}$
- 原始的n-gram精度定义为：

$$Pr(g_n) = \frac{\sum_{i=1}^{|S_{gen}|} \sum_{g_n \in S_{gen}^i} Count_{match}(g_n, S_{ref}^i)}{\sum_{i=1}^{|S_{gen}|} \sum_{g_n \in S_{gen}^i} Count(g_n)} \tag{4}$$

其中：

- g_n 为n-gram
- $Count(g_n)$ 为n-gram g_n 在生成翻译文本集合 S_{gen} 中出现的次数
- $Count_{match}(g_n, S^i_{ref})$ 为n-gram g_n 在参考翻译文本集合 S_{ref} 中出现的次数

这个公式的意思是：将生成翻译文本集合 S_{gen} 中的每个翻译文本 S^i_{gen} 与对应参考翻译文本 S^i_{ref} 进行比较，计算n-gram g_n 在生成翻译文本集合 S_{gen} 中出现的次数 $Count(g_n)$ 和在参考翻译文本集合 S_{ref} 中出现的次数 $Count_{match}(g_n, S^i_{ref})$，然后计算n-gram精度 $Pr(g_n)$。

例如，$n = 1$时，模型生成的文本是 "I love cats"，参考翻译文本是 "I love dogs"，则：

- 生成翻译文本集合 S_{gen} 中包含 "I"、"love"、"cats" 三个词
- 参考翻译文本集合 S_{ref} 中包含 "I"、"love"、"dogs" 三个词
- 根据生成翻译文本集合的长度，算出$Count(g_1) = 3$
- "I"和"love"都在参考翻译文本集合 S_{ref} 中出现1次，算出$Count_{match}(g_1, S_{ref}) = 1 + 1 = 2$
- 因此，$Pr(g_1) = \frac{2}{3}$。

如果$n = 2$，则：

- 生成翻译文本集合 S_{gen} 中包含 "I love" 和 "love cats" 两个词
- 参考翻译文本集合 S_{ref} 中包含 "I love" 和 "love dogs" 两个词
- 根据生成翻译文本集合的长度，算出$Count(g_2) = 2$
- 只有"I love" 在参考翻译文本集合 S_{ref} 中出现1次，算出$Count_{match}(g_2, S_{ref}) = 1$
- 因此，$Pr(g_2) = \frac{1}{2}$。

BLEU最终取会N个n-gram精度的几何平均作为评测结果：

$$BLEU = \sqrt[N]{\prod_{n=1}^{N} Pr(g_n)} = \exp\left(\sum_{n=1}^{N} \log Pr(g_n)\right) \tag{5}$$

基于以上原始的BLEU，还可以对不同的n-gram精度进行加权，或者对不同文本长度设置惩罚项来调整BLEU metric。

下面给出一个简化版的BLEU计算示例代码，注意，这里只使用了split()方法进行分词，没有使用tokenizer，所以需要确保参考文本和生成文本都是以空格分隔的。

```
def compute_bleu(reference, candidate, n=1):
    """
    计算BLEU分数
    Args:
        reference: 参考翻译文本
        candidate: 生成的翻译文本
        n: n-gram的最大长度
    Returns:
        bleu: BLEU分数
    """
    # 将文本分词
```

```
    ref_tokens = reference.split()
    cand_tokens = candidate.split()

    # 计算各个n-gram的精度
    precisions = []
    for i in range(1, n+1):
        # 获取候选文本中的n-gram
        cand_ngrams = {}
        for j in range(len(cand_tokens)-i+1):
            ngram = tuple(cand_tokens[j:j+i])
            cand_ngrams[ngram] = cand_ngrams.get(ngram, 0) + 1

        # 获取参考文本中的n-gram
        ref_ngrams = {}
        for j in range(len(ref_tokens)-i+1):
            ngram = tuple(ref_tokens[j:j+i])
            ref_ngrams[ngram] = ref_ngrams.get(ngram, 0) + 1

        # 计算匹配的n-gram数量
        match_count = 0
        total_count = 0
        for ngram, count in cand_ngrams.items():
            total_count += count
            if ngram in ref_ngrams:
                match_count += min(count, ref_ngrams[ngram])

        # 计算精度
        if total_count == 0:
            precisions.append(0.0)
        else:
            precisions.append(match_count / total_count)

    # 计算BLEU分数
    if len(precisions) == 0:
        return 0.0
    return sum(precisions) / len(precisions)

# 使用示例
reference = "I love dogs"
candidate = "I love cats"
bleu_score = compute_bleu(reference, candidate, n=4)
print(f"BLEU score: {bleu_score}")  # 输出: BLEU score: 0.42083333333333334
```

3.2.1.2 ROUGE

ROUGE(Recall-Oriented Understudy for Gisting Evaluation，召回导向的评估替身)主要用于评估摘要生成任务，包含ROUGE-N（基于n-gram的召回）、ROUGE-L（基于最长公共子序列的召回）、ROUGE-W（带权重的ROUGE-L）和ROUGE-S（基于Skip-bigram的召回）。

ROUGE是和BLEU的主要区别是ROUGE基于召回率(recall)来评估生成文本与参考文本的相似度,而BLEU基于精确率(precision)来评估,因此ROUGE更适合文本摘要等任务的评估,BLEU更适合机器翻译等任务的评估。

precision的定义为:

$$precision = \frac{TP}{TP + FP} \tag{6}$$

recall的定义为:

$$recall = \frac{TP}{TP + FN} \tag{7}$$

- BLEU中的match指的是生成文本中的n-gram在参考文本中出现的次数/生成文本中的n-gram总数，也就是$TP/(TP+FP)$。
- ROUGE中的match指的是参考文本中的n-gram在生成文本中出现的次数/参考文本中的n-gram总数，也就是$TP/(TP+FN)$。

ROUGE-N的定义为:

$$ROUGE\text{-}N = \frac{\sum_{s \in S_{ref}} \sum_{g_n \in s} \mathrm{Count}_{\mathrm{match}}(g_n, s_{gen})}{\sum_{s \in S_{ref}} \sum_{g_n \in s} \mathrm{Count}(g_n)} \tag{8}$$

其中:

- s 为参考摘要
- s_{gen} 为生成摘要
- g_n 为n-gram
- $\mathrm{Count}_{\mathrm{match}}(g_n, s_{gen})$ 为n-gram g_n 在生成摘要 s_{gen} 中出现的次数
- $\mathrm{Count}(g_n)$ 为n-gram g_n 在参考摘要 s 中出现的次数

ROUGE-L的定义为:

$$ROUGE\text{-}L = \frac{(1+\beta^2) R_r^g R_g^r}{R_r^g + \beta^2 R_g^r} \tag{9}$$

其中:

$$\begin{aligned} R_r^g &= \frac{\mathrm{LCS}(s_{ref}, s_{gen})}{|s_{ref}|}, \\ R_g^r &= \frac{\mathrm{LCS}(s_{ref}, s_{gen})}{|s_{gen}|}, \end{aligned} \tag{10}$$

$\mathrm{LCS}(s_{ref}, s_{gen})$ 表示生成摘要和参考摘要之间的最长公共子序列 (Longest Common Subsequence, LCS) 长度，$\beta = R_g^r / R_r^g$。

基于统计指标的评测方法虽然简单直观，但难以完全适应生成任务中的表达多样性，特别是在生成样本具有较强创造性时，与人类评测差异较大。为解决这个问题，研究者提出了引入"裁判"语言模型的评测方法。

ROUGE-S的定义为：

$$ROUGE\text{-}S_n = \frac{\sum_{s \in S_{ref}} \sum_{\text{skip}_n \in s} \text{Count}_{\text{match}}(\text{skip}_n, s_{gen})}{\sum_{s \in S_{ref}} \sum_{\text{skip}_n \in s} \text{Count}(\text{skip}_n)} \tag{11}$$

其中：

- skip_n 为skip-bigram
- $\text{Count}_{\text{match}}(\text{skip}_n, s_{gen})$ 为skip-bigram在生成文本中出现的次数
- $\text{Count}(\text{skip}_n)$ 为skip-bigram在参考文本中出现的次数
- n 为允许跳过的最大单词数

ROUGE-S基于Skip-bigram的召回率来评估生成文本的质量。Skip-bigram是指在文本中任意两个单词的组合,这两个单词之间可以跳过其他单词。例如对于句子"the cat sat on the mat"，其Skip-bigram包括"the cat"、"the sat"、"the on"等。

ROUGE-S相比ROUGE-N的优势在于可以捕捉到文本中词序的变化，对于评估生成文本的流畅性和语序合理性具有重要意义。

ROUGE-N的计算示例代码：

```
def rouge_n(reference, candidate, n):
    """
    计算ROUGE-N分数

    Args:
        reference: 参考文本(字符串)
        candidate: 生成文本(字符串)
        n: n-gram的n值

    Returns:
        float: ROUGE-N分数
    """
    # 将文本分词
    ref_tokens = reference.split()
    can_tokens = candidate.split()

    # 获取参考文本中的n-gram
    ref_ngrams = {}
    for i in range(len(ref_tokens) - n + 1):
        ngram = tuple(ref_tokens[i:i+n])
        ref_ngrams[ngram] = ref_ngrams.get(ngram, 0) + 1

    # 获取生成文本中的n-gram
    can_ngrams = {}
    for i in range(len(can_tokens) - n + 1):
```

```
            ngram = tuple(can_tokens[i:i+n])
            can_ngrams[ngram] = can_ngrams.get(ngram, 0) + 1

    # 计算匹配的n-gram数量
    match_count = 0
    for ngram, count in can_ngrams.items():
        match_count += min(count, ref_ngrams.get(ngram, 0))

    # 计算参考文本中n-gram总数
    total_ref_ngrams = sum(ref_ngrams.values())

    # 避免除零错误
    if total_ref_ngrams == 0:
        return 0.0

    # 计算ROUGE-N分数
    rouge_n_score = match_count / total_ref_ngrams

    return rouge_n_score

# 使用示例
reference = "the cat sat on the mat"
candidate = "the cat was on the mat"
rouge_1 = rouge_n(reference, candidate, 1)
rouge_2 = rouge_n(reference, candidate, 2)

print(f"ROUGE-1分数: {rouge_1:.3f}")  # 输出: ROUGE-1分数: 0.833
print(f"ROUGE-2分数: {rouge_2:.3f}")  # 输出: ROUGE-2分数: 0.600
```

3.2.2 基于语言模型的评测 (Language Model-based Evaluation)

基于语言模型的评测主要分为基于上下文词嵌入 (Contextual Word Embedding) 的评测和基于生成模型 (Generative Model) 的评测两类。前者的代表是BERTScore，后者的代表是G-EVAL。相比BERTScore，G-EVAL无需人类标注的参考答案，更适合缺乏标注数据的场景。

3.2.2.1 BERTScore

BERTScore基于BERT的上下文词嵌入向量计算生成文本与参考文本的相似度。

对于生成文本 $s_{gen} = \{w_g^i\}_{i=1}^{|s_{gen}|}$ 和参考文本 $s_{ref} = \{w_r^i\}_{i=1}^{|s_{ref}|}$，通过BERT获取词嵌入向量后，从精度 (Precision)、召回 (Recall) 和F1 (F-measure) 三个维度进行评测：

$$
\begin{aligned}
P_{BERT} &= \frac{1}{|v_{gen}|} \sum_{i=1}^{|v_{gen}|} \max_{v_r \in v_{ref}} {v_g^i}^T v_r^i, \\
R_{BERT} &= \frac{1}{|v_{ref}|} \sum_{i=1}^{|v_{ref}|} \max_{v_g \in v_{gen}} {v_r^i}^T v_g^i, \\
F_{BERT} &= \frac{2P_{BERT} \cdot R_{BERT}}{P_{BERT} + R_{BERT}}
\end{aligned} \tag{12}
$$

其中：

- v_{gen} 为生成文本的词嵌入向量
- v_{ref} 为参考文本的词嵌入向量
- v_g^i 为生成文本的第i个词的词嵌入向量
- v_r^i 为参考文本的第i个词的词嵌入向量

BERTScore的计算示例代码：

```
import torch
from transformers import AutoTokenizer, AutoModel

def bert_score(candidate, reference):
    # 加载预训练的模型和分词器
    tokenizer = AutoTokenizer.from_pretrained('model-name')
    model = AutoModel.from_pretrained('model-name')

    # 对文本进行分词和编码
    candidate_tokens = tokenizer(candidate, return_tensors='pt', padding=True,
truncation=True)
    reference_tokens = tokenizer(reference, return_tensors='pt', padding=True,
truncation=True)

    # 获取BERT词嵌入向量，并进行归一化
    with torch.no_grad():
        candidate_embeddings = model(**candidate_tokens).last_hidden_state.squeeze(0)
        reference_embeddings = model(**reference_tokens).last_hidden_state.squeeze(0)

        # L2归一化
        candidate_embeddings = torch.nn.functional.normalize(candidate_embeddings, p=2,
dim=-1)
        reference_embeddings = torch.nn.functional.normalize(reference_embeddings, p=2,
dim=-1)

    # 计算余弦相似度矩阵
    sim_matrix = torch.matmul(candidate_embeddings, reference_embeddings.transpose(0,
1))

    # 忽略[PAD]token的影响
```

```
    candidate_mask = candidate_tokens.attention_mask.squeeze(0).bool()
    reference_mask = reference_tokens.attention_mask.squeeze(0).bool()

    # 计算Precision (只考虑非padding的token)
    P = sim_matrix[candidate_mask].max(dim=1)[0].mean()

    # 计算Recall (只考虑非padding的token)
    R = sim_matrix[:, reference_mask].max(dim=0)[0].mean()

    # 计算F1分数
    F1 = 2 * (P * R) / (P + R)

    return {
        'precision': P.item(),
        'recall': R.item(),
        'f1': F1.item()
    }

# 使用示例
candidate = "今天天气真不错"
reference = "今天是个好天气"

scores = bert_score(candidate, reference)
print(f"BERTScore评分结果:")
print(f"Precision: {scores['precision']:.3f}")
print(f"Recall: {scores['recall']:.3f}")
print(f"F1: {scores['f1']:.3f}")
```

3.2.2.2 G-EVAL

相较于统计评测指标，BERTScore通过利用上下文词嵌入向量的语义信息，能够更好地捕捉文本的语义相似度，因此评测结果更接近人类判断。

然而，BERTScore仍然依赖于人类标注的参考文本作为评测基准，这限制了其在缺乏标注数据的场景中的应用。随着生成式大语言模型的快速发展，研究人员提出了G-EVAL(GPT-based Evaluation)评测方法。该方法巧妙地利用GPT-4的强大语言理解和评判能力，无需参考文本即可对生成文本进行评分。G-EVAL通过精心设计的Prompt Engineering来引导GPT-4理解评测标准并输出合理的评测分数。

G-EVAL的Prompt设计分为3个部分:

任务描述与评分标准

- 明确说明评测任务类型(如摘要生成, 翻译, 问答等)
- 定义评分范围和评判标准(如1-5分)
- 列举需要考虑的关键因素(如流畅性, 准确性, 完整性等)

评测步骤

- 基于任务描述与评分标准的内容

- 由GPT-4自动生成思维链(Chain-of-Thoughts, CoT)，给出评价结果

待评测文本

- 包含源文本(如原始文章)
- 包含生成文本(如模型生成的摘要)

将这三部分内容组织成完整的prompt后输入GPT-4,模型即可给出评分结果。但由于直接使用GPT-4的评分可能存在区分度不足的问题,G-EVAL采用了加权平均机制来优化最终评分,以提高评测的准确性和可靠性。

除 G-EVAL 外，近期还涌现出多种基于生成模型的评测方法，其中最具代表性的是 InstructScore。该方法会训练一个可解释的评估模型，包括分数预测器 (Score Predictor) 和解释生成器 (Explanation Generator)。因此，InstructScore不仅能给出量化的评分结果，还能提供详细的评分理由和建议，包括错误位置l，错误类型t，严重性标签s_e和解释e。与传统的统计指标和基于上下文词嵌入的评测方法相比，基于生成模型的评测方法在准确性、灵活性、可解释性和无需参考答案等方面具有显著优势。它能更好地理解和评估文本的语义内容，可适应不同类型的生成任务，提供清晰的评分依据和改进建议，并且降低了对标注数据的依赖。随着大语言模型技术的不断进步，基于生成模型的评测方法将在未来发挥越来越重要的作用，并有望成为评估文本生成质量的主流方法。