得到充分学习。理想模式下，词表示模型应能覆盖绝大部分的输入词，并避免词表过大所造成的数据稀疏问题。

为了缓解未登录词问题，一些工作通过利用亚词级别的信息构造词表示向量。一种直接的解决思路是为输入建立字符级别表示，并通过字符向量的组合获得每个单词的表示，以解决数据稀疏问题。然而，单词中的词根、词缀等构词模式往往跨越多个字符，基于字符表示的方法很难学习跨度较大的模式。为了充分学习这些构词模式，研究人员提出了**子词词元化**（Subword Tokenization）方法，试图缓解上文介绍的未登录词问题。词元表示模型会维护一个词元词表，其中既存在完整的单词，也存在形如“c”“re”“ing”等单词的部分信息，称为**子词**（Subword）。词元表示模型对词表中的每个词元计算一个定长向量表示，供下游模型使用。对于输入的词序列，词元表示模型将每个词拆分为词表内的词元。例如，将单词“reborn”拆分为“re”和“born”。模型随后查询每个词元的表示，将输入重新组成词元表示序列。当下游模型需要计算一个单词或词组的表示时，可以将对应范围内的词元表示合成需要的表示。因此，词元表示模型能够较好地解决自然语言处理系统中未登录词的问题。**词元分析**（Tokenization）是将原始文本分割成词元序列的过程。词元切分也是数据预处理中至关重要的一步。

字节对编码（Byte Pair Encoding，BPE）[129] 是一种常见的子词词元算法。该算法采用的词表包含最常见的单词及高频出现的子词。使用时，常见词通常位于 BPE 词表中，而罕见词通常能被分解为若干个包含在 BPE 词表中的词元，从而大幅减小未登录词的比例。BPE 算法包括以下两个部分。

（1）词元词表的确定。

（2）全词切分为词元及词元合并为全词的方法。

BPE 中词元词表的计算过程如图3.4 所示。首先，确定数据库中全词的词表和词频，然后将每个单词切分为单个字符的序列，并在序列最后添加符号“</w>”作为单词结尾的标识。例如，单词“low”被切分为序列“l␣o␣w␣</w>”。所切分出的序列元素称为字节，即每个单词都切分为字节的序列。之后，按照每个字节序列的相邻字节对和单词的词频，统计每个相邻字节对的出现频率，合并出现频率最高的字节对，将其作为新的词元加入词表，并将全部单词中的该字节对合并为新的单一字节。在第一次迭代时，出现频率最高的字节对是 (e,s)，故将“es”作为词元加入词表，并将全部序列中相邻的 (e,s) 字节对合并为 es 字节。重复这一步骤，直至 BPE 词元词表的大小达到指定的预设值，或没有可合并的字节对为止。

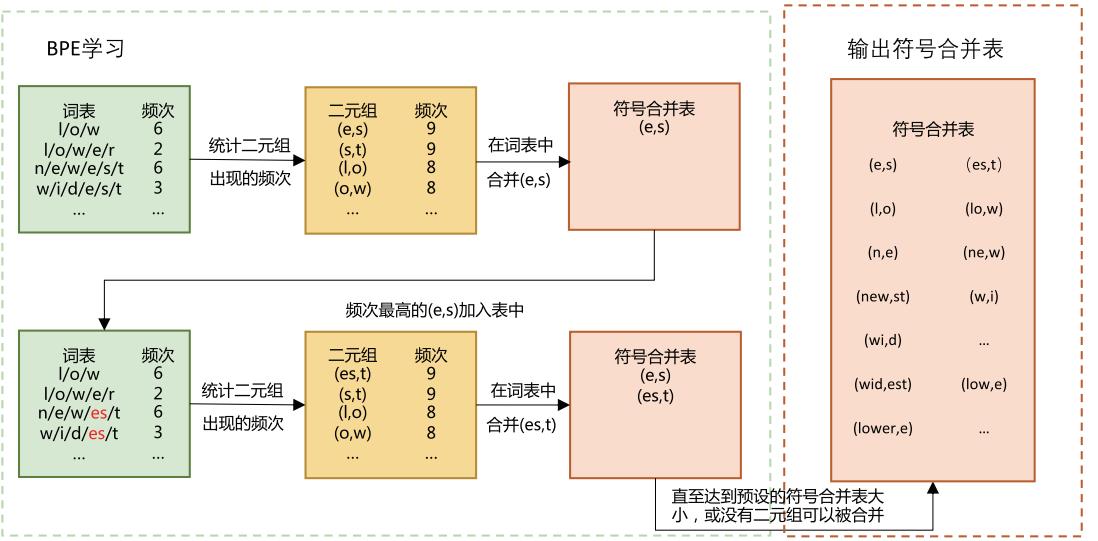


图 3.4 BPE 中词元词表的计算过程[129]

确定词元词表之后，对输入词序列中未在词表中的全词进行切分。BPE 算法对词表中的词元按从长到短的顺序进行遍历，将每一个词元与当前序列中的全词或未完全切分为词元的部分进行匹配，将其切分为该词元和剩余部分的序列。例如，对于单词“lowest</w>”，先通过匹配词元“est</w>”将其切分为“low”“est</w>”的序列，再通过匹配词元“low”，确定其最终切分结果为“low”“est</w>”的序列。通过这样的过程，使用 BPE 尽量将词序列中的词切分成已知的词元。

在遍历词元词表后，对于切分得到的词元序列，为每个词元查询词元表示，构成词元表示序列。若出现未登录词元，即未出现在 BPE 词表中的词元，则采取和未登录词类似的方式，为其赋予相同的表示，最终获得输入的词元表示序列。

此外，字节级（Byte-level）BPE 通过将字节视为合并的基本符号，改善多语言数据库（例如包含非 ASCII 字符的文本）的分词质量。GPT-2、BART、LLaMA 等大语言模型都采用了这种分词方法。原始 LLaMA 的词表大小是 32K①，并且主要根据英文进行训练，因此，很多汉字都没有直接出现在词表中，需要字节来支持所有的中文字符，2 个或者 3 个字节词元（Byte Token）才能拼成一个完整的汉字。

对于使用了 BPE 的大语言模型，其输出序列也是词元序列。对于原始输出，根据终结符 </w> 的位置确定每个单词的范围，合并范围内的词元，将输出重新组合为词序列，作为最终的结果。

WordPiece[130] 也是一种常见的词元分析算法，最初应用于语音搜索系统。此后，通常将该算法作为 BERT 的词元分析器[1]。WordPiece 与 BPE 有非常相似的思想，都是迭代地合并连续的词

① K，源于英文前缀 kilo，本书中指千，例如 10K 代表 1 万。

元，但在合并的选择标准上略有不同。为了进行合并，WordPiece 需要先训练一个语言模型，并用该语言模型对所有可能的词元对进行评分。在每次合并时，选择使得训练数据似然概率增加最多的词元对。Google 并没有发布其 WordPiece 算法的官方实现，HuggingFace 在其在线 NLP 课程中提供了一种更直观的选择度量方法：一个词元对的评分是根据训练数据库中两个词元的共现计数除以它们各自的出现计数的乘积。计算公式如下所示：

$$\text{score} = \frac{\text{词元对出现的频率}}{\text{第一个词元出现的频率} \times \text{第二个词元出现的频率}} \tag{3.1}$$

Unigram 词元分析[131] 是另一种应用于大语言模型的词元分析算法，T5 和 mBART 采用该算法构建词元分析器。不同于 BPE 和 WordPiece，Unigram 词元分析从一个足够大的可能词元集合开始，迭代地从当前列表中删除词元，直到达到预期的词汇表大小。词元删除基于训练好的 Unigram 语言模型，以从当前词汇表中删除某个字词后，训练数据库似然性的增加量为选择标准。为了估计一元语言（Unigram）模型，采用了期望最大化（Expectation–Maximization，EM）算法：每次迭代时，先根据旧的语言模型找到当前最佳的单词切分方式，然后重新估计一元语言单元概率以更新语言模型。在这个过程中，使用动态规划算法（如维特比算法）高效地找到给定语言模型时单词的最佳分解方式。

以 HuggingFace NLP 课程中介绍的 BPE 代码为例，介绍 BPE 算法的构建和使用，代码实现如下所示：

```
from transformers import AutoTokenizer
from collections import defaultdict

corpus = [
    "This is the HuggingFace Course.",
    "This chapter is about tokenization.",
    "This section shows several tokenizer algorithms.",
    "Hopefully, you will be able to understand how they are trained and generate tokens.",
]

# 使用GPT-2词元分析器将输入分解为单词
tokenizer = AutoTokenizer.from_pretrained("gpt2")

word_freqs = defaultdict(int)

for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
```

```
        word_freqs[word] += 1

# 计算基础词典，这里使用数据库中的所有字符
alphabet = []

for word in word_freqs.keys():
    for letter in word:
        if letter not in alphabet:
            alphabet.append(letter)
alphabet.sort()

# 在字典的开头增加特殊词元，GPT-2中仅有一个特殊词元"<|endoftext|>"，用来表示文本结束
vocab = ["<|endoftext|>"] + alphabet.copy()

# 将单词切分为字符
splits = {word: [c for c in word] for word in word_freqs.keys()}

# compute_pair_freqs函数用于计算字典中所有词元对的频率
def compute_pair_freqs(splits):
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            pair_freqs[pair] += freq
    return pair_freqs

# merge_pair函数用于合并词元对
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue

        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                split = split[:i] + [a + b] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits

# 迭代训练，每次选取得分最高词元对进行合并，直到字典大小达到设置的目标为止
vocab_size = 50
```

HuggingFace 的 transformer 类中已经集成了很多词元分析器，可以直接使用。例如，利用 BERT 词元分析器获得输入“I have a new GPU!”的词元代码如下所示：

```
>>> from transformers import BertTokenizer
>>> tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
>>> tokenizer.tokenize("I have a new GPU!")
["i", "have", "a", "new", "gp", "##u", "!"]
```

3.3 数据影响分析

大语言模型的训练需要大量的计算资源，通常不可能多次进行大语言模型预训练。有千亿级参数量的大语言模型进行一次预训练需要花费数百万元的计算成本。因此，在训练大语言模型之前，构建一个准备充分的预训练语料库尤为重要。本节将从数据规模、数据质量和数据多样性三个方面分析数据对大语言模型的性能影响。需要特别说明的是，截至本书成稿时，由于在千亿参数规模的大语言模型上进行实验的成本非常高，很多结论是在百亿甚至十亿规模的语言模型上进行的实验，其结果并不能完整地反映数据对大语言模型的影响。此外，一些观点仍处于猜想阶段，需要进一步验证。请各位读者甄别判断。

3.3.1 数据规模

随着大语言模型参数规模的增加，为了有效地训练模型，需要收集足够数量的高质量数据[34, 132]。在针对模型参数规模、训练数据量及总计算量与模型效果之间关系的研究[132] 被提出之前，大部分大语言模型训练所采用的训练数据量相较于 LLaMA 等最新的大语言模型都少很多。表3.1 给出了模型参数量与训练数据量的对比。在 Chinchilla 模型被提出之前，大部分大语言模型都在着重提升模型的参数量，所使用的训练数据量都在 3000 亿个词元左右，LaMDA 模型使用的训练参数量仅有 1370 亿个。虽然 Chinchilla 模型的参数量不足 LaMDA 模型的一半，但是训练数据的词元数达到 1.4 万亿个，是 LaMDA 模型的 8 倍多。

表 3.1 模型参数量与训练数据量的对比

模型名称	参数量（个）	训练数据量（个词元）
LaMDA[15]	1370 亿	1680 亿
GPT-3[39]	1750 亿	3000 亿
Jurassic [133]	1780 亿	3000 亿
Gopher [115]	2800 亿	3000 亿
MT-NLG 530B [134]	5300 亿	2700 亿
Chinchilla[132]	700 亿	14000 亿
Falcon[60]	400 亿	10000 亿
LLaMA[34]	630 亿	14000 亿
LLaMA-2[37]	700 亿	20000 亿
LLaMA-3[135]	4050 亿	150000 亿
Qwen2.5[136]	720 亿	180000 亿
GLM-4[137]	1300 亿	100000 亿

DeepMind 的研究人员在文献 [132] 中描述了他们训练 400 多个语言模型后得出的分析结果（模型的参数量从 7000 万个到 160 亿个，训练数据量从 5 亿个词元到 5000 亿个词元）。研究发现，如果希望模型训练达到计算最优（Compute-optimal），则模型大小和训练词元数量应该等比例缩放，即模型大小加倍则训练词元数量也应该加倍。为了验证该分析结果，他们使用与 Gopher 语言模型训练相同的计算资源，根据上述理论预测了 Chinchilla 语言模型的最优参数量与词元数量组合。最终确定 Chinchilla 语言模型具有 700 亿个参数，使用了 1.4 万亿个词元进行训练。通过实验发现，Chinchilla 在很多下游评估任务中都显著地优于 Gopher（280B）、GPT-3（175B）、Jurassic-1（178B）及 Megatron-Turing NLG（530B）。

图3.5 给出了在同等计算量情况下，训练损失随参数量的变化情况。针对 9 种不同的训练参数量设置，使用不同词元数量的训练数据，训练不同大小的模型参数量，使得最终训练所需浮点运算数达到预定目标。对于每种训练量预定目标，图3.5(a) 所示为平滑后的训练损失与参数量之间的关系。可以看到，训练损失值存在明显的低谷，这意味着对于给定训练计算量目标，存在一个最佳模型参数量和训练数据量配置。利用这些训练损失低谷的位置，还可以预测更大的模型的最佳模型参数量和训练词元数量，如图3.5(b) 和图3.5(c) 所示。图中绿色线表示根据 Gopher 训练的计算量预测的最佳模型参数量和训练数据词元数量。还可以使用幂律（Power Law）对计算量限制、损失最优模型参数量大小及训练词元数之间的关系进行建模。C 表示总计算量、N_{opt} 表示模型最优参数量、D_{opt} 表示最优训练词元数量，它们之间的关系如下：

$$N_{\text{opt}} \propto C^{0.49} \tag{3.2}$$

$$D_{\text{opt}} \propto C^{0.51} \tag{3.3}$$

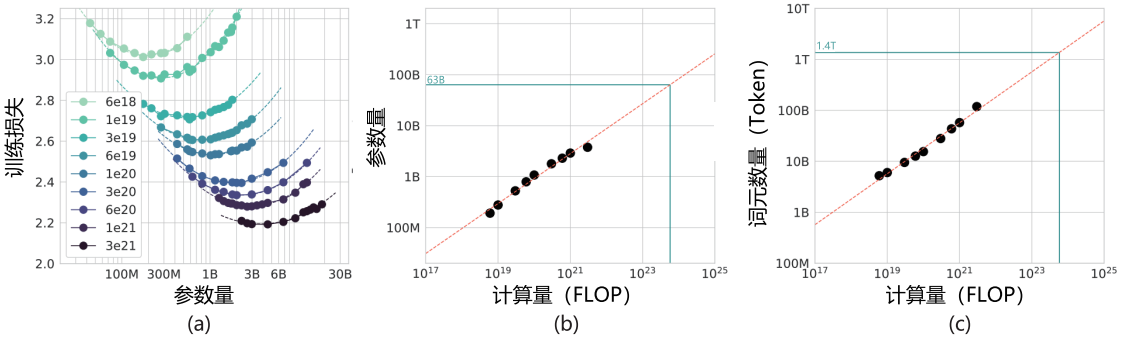


图 3.5　在同等计算量情况下，训练损失随参数量的变化情况[132]

LLaMA[34] 模型在训练时采用了与文献 [132] 相符的训练策略。研究发现，70 亿个参数的语言模型在训练超过 1 万亿个词元后，性能仍在持续增长。因此，Meta 的研究人员在 LLaMA-2[37] 模型训练中，进一步增大了训练数据量，训练数据量达到 2 万亿个词元。LLaMA-3[135] 模型训练中，则是进一步将训练数据量增大到了惊人的 15 万亿个词元。Qwen2.5[136] 的 720 亿参数的开源版本，也使用了 18 万亿个词元进行了训练。文献 [132] 给出了不同参数量的 LLaMA 模型在训练期间，随着训练数据量的增加，模型在问答和常识推理任务上的效果演变，如图3.6 所示。研究人员分别在 TriviaQA、HellaSwag、NaturalQuestions、SIQA、WinoGrande、PIQA 这 6 个数据集上进行了测试。可以看到，随着训练数据量的增加，模型在分属两类任务的 6 个数据集上的性能都在稳步提高。通过增加数据量和延长训练时间，较小的模型也能表现出良好的性能。

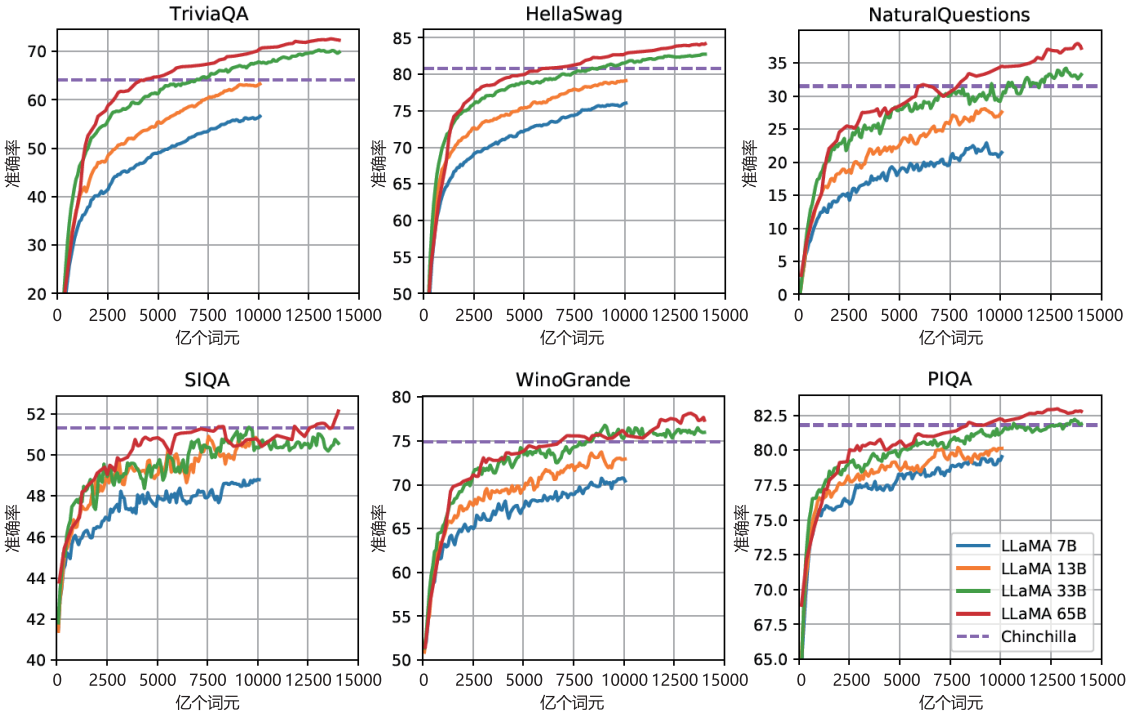


图 3.6 LLaMA 模型在问答和常识推理任务上的效果演变[34]

文献 [138] 对不同任务类型所依赖的语言模型训练数量进行了分析。针对分类探查（Classifier Probing）、信息论探查（Info-theoretic Probing）、无监督相对可接受性判断（Unsupervised Relative Acceptability Judgment）及应用于自然语言理解任务的微调（Fine-tuning on NLU Tasks）这四类任务，基于不同量级预训练数据的 RoBERTa[86] 模型进行了实验验证和分析。分别针对预训练了 1M①、10M、100M 和 1B②个词元的 RoBERTa 模型进行能力分析。研究发现，仅对模型进行 10M～100M 个词元的训练，就可以获得可靠的语法和语义特征。然而，需要更多的训练数据才能获得足够的常识知识和其他技能，并在典型的下游自然语言理解任务中取得较好的结果。

3.3.2 数据质量

数据质量通常被认为是影响大语言模型训练效果的关键因素之一。大量重复的低质量数据甚至导致训练过程不稳定，造成模型训练不收敛[122, 139]。现有的研究表明，训练数据的构建时间、包含噪声或有害信息情况、数据重复率等因素，都对语言模型性能产生较大影响[115, 122, 124, 140]。目前业界普遍的共识是语言模型在经过清洗的高质量数据上训练可以得到更好的性能。

① M，即 Million，表示百万。
② B，即 Billion，表示十亿。

文献 [115] 介绍了 Gopher 语言模型在训练时针对文本质量进行的相关实验。图3.7 所示为具有 140 亿个参数的模型在 OpenWebText、C4 及不同版本的 MassiveWeb 数据集上训练得到的模型效果对比。他们分别测试了利用不同数据训练得到的模型在 Wikitext103 单词预测、Curation Corpus 摘要及 Lambada 书籍级别的单词预测三个下游任务上的表现。图中纵坐标表示不同任务上的损失，数值越小表示性能越好。从结果可以看到，使用经过过滤和去重的 MassiveWeb 数据训练得到的语言模型在三个任务上都远好于使用未经处理的数据训练得到的模型。使用经过处理的 MassiveWeb 数据训练得到的语言模型在下游任务上的表现也远好于使用 OpenWebText 和 C4 数据集训练得到的结果。

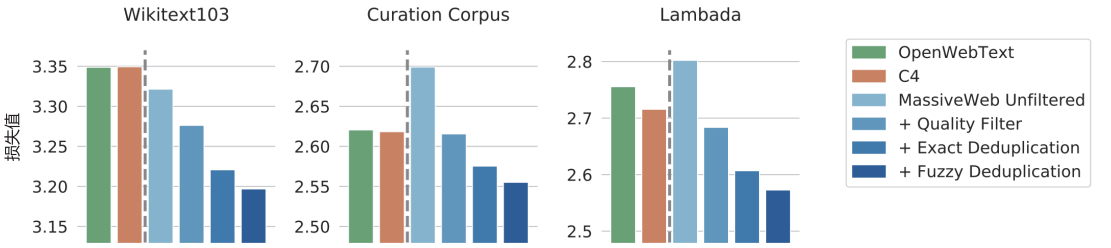
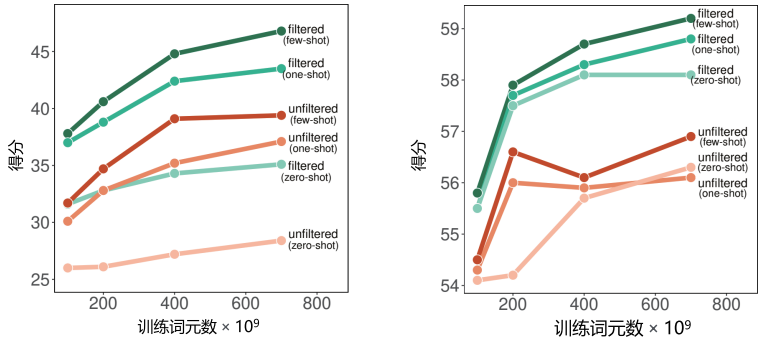


图 3.7　Gopher 语言模型使用不同数据质量的数据训练后的效果对比[115]

构建 GLaM[116] 语言模型时，也对训练数据质量的影响进行了分析。该项分析同样使用包含 17 亿个参数的模型，针对下游少样本任务的性能进行了分析。使用相同超参数，对使用原始数据集和经过质量筛选后的数据训练得到的模型效果进行了对比，实验结果如图3.8 所示。可以看到，使用高质量数据训练的模型在自然语言生成和自然语言理解任务上表现更好。特别是，高质量数据对自然语言生成任务的影响大于自然语言理解任务。这可能是因为自然语言生成任务通常需要生成高质量的语言，过滤预训练语料库对语言模型的生成能力至关重要。文献 [116] 的研究强调了预训练数据的质量在下游任务的性能中也扮演着关键角色。

Google Research 的研究人员针对数据构建时间、文本质量、是否包含有害信息进行了系统的研究[141]。他们使用包含不同时间、毒性水平、文本质量和领域的数据，训练了 28 个具有 15 亿个参数的仅解码器（Decoder-only）结构的语言模型。研究结果表明，大语言模型训练数据的时间、内容过滤方法及数据源对下游模型行为具有显著影响。



(a) 自然语言生成任务　　(b) 自然语言理解任务

图 3.8　使用不同数据质量的数据训练 GLaM 语言模型的效果对比分析[116]

针对数据时效性对于模型效果的影响问题，研究人员在 C4 数据集的 2013、2016、2019 和 2022 版本上训练了 4 个自回归语言模型。对于每个版本，研究人员删除了 CommonCrawl 数据集中截止年份之后的所有数据。使用新闻、Twitter 和科学领域的评估任务来衡量时间错配的影响。这些评估任务的训练集和测试集按年份划分，分别在每个按年份划分的数据集上微调模型，然后在 2013 年、2016 年、2019 年及 2022 年的测试集上进行评估。图3.9 给出了使用 4 个不同版本的数据集训练得到的模型在 5 个不同任务上的评测结果。热力图颜色（Heatmap Colors）根据每一列进行归一化得到。从图中可以看到，训练数据和测试数据的时间错配会在一定程度上影响模型的效果。

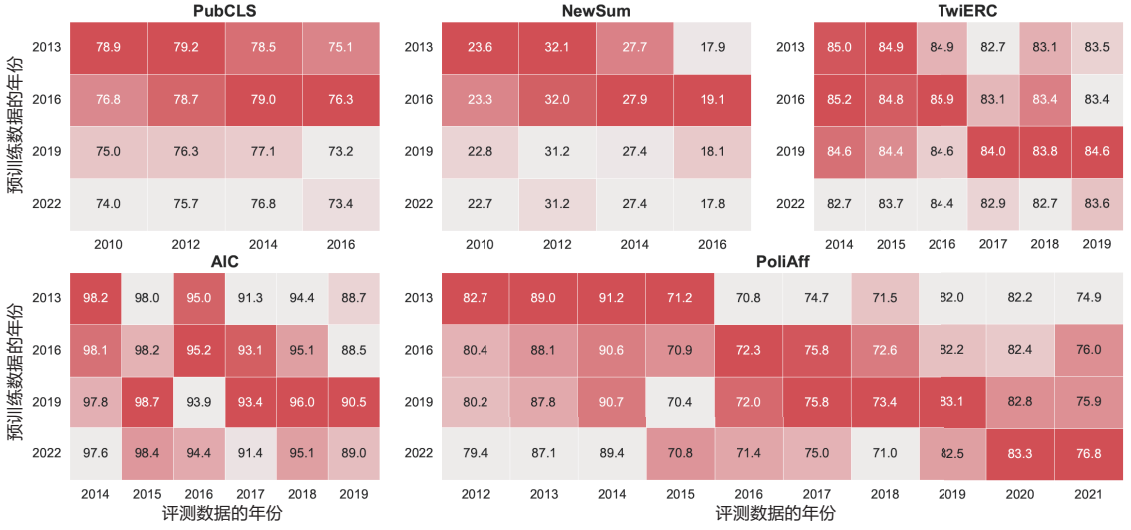


图 3.9　训练数据和测试数据在时间错配情况下的性能分析[141]

Anthropic 的研究人员针对数据集中的重复问题开展了系统研究[122]。为了研究数据重复对大语言模型的影响，研究人员构建了特定的数据集，其中大部分数据是唯一的，只有一小部分数据

被重复多次，并使用这个数据集训练了一组模型。研究发现了一个强烈的双峰下降现象，即重复数据可能会导致训练损失在中间阶段增加。例如，通过将 0.1% 的数据重复 100 次，即使其余 90% 的训练数据保持不变，一个参数量为 800M 的模型的性能也可能降低到与参数量为 400M 的模型相同。此外，研究人员还设计了一个简单的复制评估，即将《哈利·波特》（*Harry Potter*）的文字复制 11 次，计算模型在该段上的损失。在仅有 3% 的重复数据的情况下，训练过程中性能最差的轮次仅能达到参数量为其 1/3 的模型的效果。

文献 [14] 对大语言模型的记忆能力进行分析，根据训练样例在训练数据中出现的次数，显示了记忆率的变化情况，如图3.10 所示。可以看到，对于在训练中只见过一次的样例，PaLM 模型的记忆率为 0.75%，而其对见过 500 次以上的样例的记忆率超过 40%。这也在一定程度上说明重复数据对于语言模型建模具有重要影响。这也可能进一步影响使用上下文学习的大语言模型的泛化能力。由于 PaLM 模型仅使用了文档级别过滤，因此片段级别（100 个以上词元）可能出现非常高的重复次数。

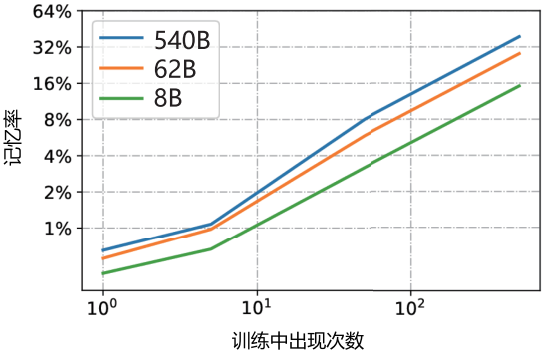


图 3.10　大语言模型记忆能力评测[14]

3.3.3 数据多样性

来自不同领域、使用不同语言、应用于不同场景的训练数据具有不同的语言特征，包含不同语义知识。通过使用不同来源的数据进行训练，大语言模型可以获得广泛的知识。表3.2 给出了 LLaMA 模型训练所使用的数据集。可以看到，LLaMA 模型训练混合了大量不同来源的数据，包括网页、代码、论文、图书等。针对不同的文本质量，LLaMA 模型训练针对不同质量和重要性的数据集设定了不同的采样概率，表中给出了不同数据集在完成 1.4 万亿个词元训练时的采样轮数。

表 3.2 LLaMA 模型训练所使用的数据集[37]

数据集	采样概率	训练轮数	存储空间
CommonCrawl	67.0%	1.10	3.3 TB
C4	15.0%	1.06	783 GB
GitHub	4.5%	0.64	328 GB
Wikipedia	4.5%	2.45	83 GB
Books	4.5%	2.23	85 GB
arXiv	2.5%	1.06	92 GB
Stack Exchange	2.0%	1.03	78 GB

Gopher 模型[115] 在训练过程中进行了对数据分布的消融实验，以便验证混合来源对下游任务的影响。针对 MassiveText 子集设置了不同权重的数据组合，并用于训练语言模型。利用 Wikitext103、Lambada、C4 和 Curation Corpus 测试不同权重组合训练得到的语言模型在下游任务上的性能。为了限制数据组合分布范围，实验中固定了 Wikipedia 和 GitHub 两个数据集的采样权重。对于 Wikipedia，要求对训练数据进行完整的学习，因此将采样权重固定为 2%；对于 GitHub，采样权重设置为 3%。对于剩余的 4 个子集（MassiveWeb、News、Books 和 C4）设置了 7 种不同的组合。图3.11 给出了 7 种不同子集采样权重训练得到 Gopher 模型在下游任务上的性能。可以看到，使用不同数量子集采样权重训练，获得的模型效果差别很大。在所有任务中表现良好且在 Curation Corpus 上取得最佳表现的绿色配置是 10% 的 C4、50% 的 MassiveWeb、30% 的 Books 和 10% 的 News。增加书籍数据的比例可以提高模型从文本中捕获长期依赖关系的能力，降低 Lambada 数据集[142] 上的损失，而使用更高比例的 C4 数据集[19] 则有助于在 C4 验证集[115] 上获得更好的表现。

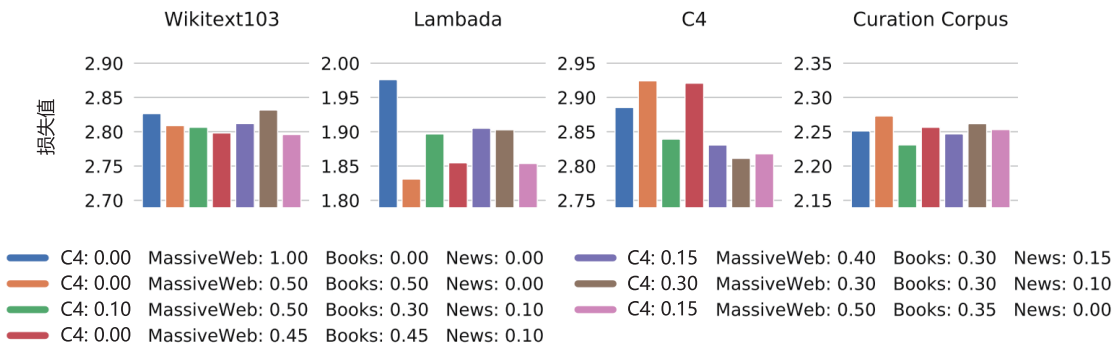


图 3.11 使用不同采样权重训练得到的 Gopher 语言模型在下游任务上的性能[115]

3.4 开源数据集

随着基于统计机器学习的自然语言处理算法的发展，以及信息检索研究的需求增加，特别是近年来对深度学习和预训练语言模型的研究更深入，研究人员构建了多种大规模开源数据集，涵盖了网页、图书、论文等多个领域。在构建大语言模型时，数据的质量和多样性对于提高模型的性能至关重要。同时，为了推动大语言模型的研究和应用，学术界和工业界也开放了多个针对大语言模型的开源数据集。本节将介绍典型的开源数据集。

3.4.1 Pile

Pile 数据集[87] 是一个用于大语言模型训练的多样性大规模文本数据库，由 22 个不同的高质量子集构成，包括现有的和新构建的，主要来自学术或专业领域。这些子集包括 Pile-CC（清洗后的 CommonCrawl 子集）、Wikipedia、OpenWebText2、arXiv、PubMed Central 等。Pile 的特点是包含了大量多样化的文本，涵盖了不同领域和主题，从而提高了训练数据集的多样性和丰富性。Pile 数据集包含 825GB 英文文本，其组成大体上如图3.12 所示，所占面积大小表示数据在整个数据集中的规模。

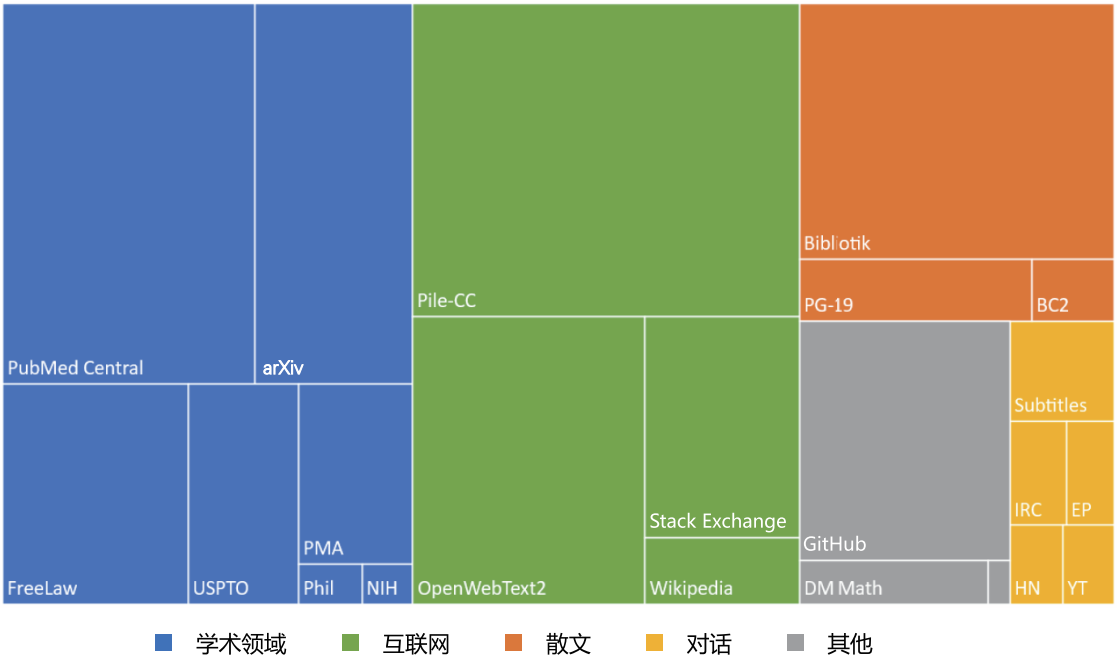


图 3.12 Pile 数据集的组成[87]

Pile 数据集由以下 22 个不同子集构成。

（1）Pile-CC 是基于 CommonCrawl 的数据集，该数据集通过在 Web Archive 文件上使用 jusText[143] 的方法进行提取，比直接使用 WET 文件产生更高质量的输出。

（2）PubMed Central（PMC）是由美国国家生物技术信息中心（NCBI）运营的 PubMed 生物医学在线资源库的一个子集，PubMed 是由美国国家医学图书馆运营的生物医学文章在线存储库，提供对近 500 万份出版物的开放全文访问。

（3）Books 3 是一个图书数据集，来自 Shawn Presser 提供的 Bibliotik。Bibliotik 由小说和非小说类书籍组成，几乎是图书数据集（BookCorpus 2）数据量的十倍。

（4）OpenWebText2 （OWT2）是一个基于 WebText[11] 和 OpenWebTextCorpus 的通用数据集。它包括来自多种语言的文本内容、网页文本元数据，以及多个开源数据集和开源代码库。

（5）arXiv 是一个自 1991 年开始运营的论文预印版本发布服务平台。发布在 arXiv 上的论文主要集中在数学、计算机科学和物理领域。arXiv 上的论文是用 LaTeX 编写的，其中公式、符号、表格等内容的表示非常适合语言模型学习。

（6）GitHub 是一个大型的开源代码库，对于语言模型完成代码生成、代码补全等任务具有非常重要的作用。

（7）FreeLaw 是一个非营利项目，为法律领域的学术研究提供访问和分析工具。CourtListener 是 FreeLaw 项目的一部分，包含美国联邦和州法院的数百万条法律意见，并提供批量下载服务。

（8）Stack Exchange 是一个围绕用户提供问题和答案的网站集合。Stack Exchange Data Dump 包含了 Stack Exchange 网站集合中所有用户贡献的内容的匿名数据集。它是截至 2023 年 9 月公开可用的最大的问题-答案对数据集之一，包括编程、园艺、艺术等主题。

（9）USPTO Backgrounds 是美国专利商标局授权的专利背景部分的数据集，来源于其公布的批量档案。由于专利通常包含任务背景介绍，给出了发明的背景和技术领域的概述，建立了问题空间的框架，因此该数据集包含了大量关于应用主题的技术内容。

（10）Wikipedia (English) 是维基百科的英文部分。维基百科是一部由全球志愿者协作创建和维护的免费在线百科全书，旨在提供各种主题的知识。它是世界上最大的在线百科全书之一，包含多种语言，如英语、中文、西班牙语、法语、德语，等等。

（11）PubMed Abstracts 是由 PubMed 中 3000 万份出版物的摘要组成的数据集。PubMed 还包含 MEDLINE，其包含 1946 年至今的生物医学摘要。

（12）Project Gutenberg 是一个包含西方经典文学的数据集。它使用的 PG-19 由 1919 年以前的 Project Gutenberg 中的书籍数据组成[144]，与更现代的 Books 3 和 BookCorpus 相比，它们代表了不同的风格。

（13）OpenSubtitles 是由英文电影和电视的字幕组成的数据集[145]。字幕是对话的重要来源，并且可以增强模型对虚构格式的理解，也可能对创造性写作任务（如剧本写作、演讲写作、交互式故事讲述等）有一定作用。

（14）DeepMind Mathematics 数据集由代数、算术、微积分、数论和概率等一系列数学问题组

成，并且以自然语言提示的形式给出[146]。大语言模型在数学任务上的表现较差[39]，这可能是由于训练集中缺乏数学问题。因此，Pile 数据集中专门增加了数学问题数据集，期望增强通过 Pile 数据集训练的语言模型的数学能力。

（15）BookCorpus 2 数据集是原始 BookCorpus[147] 的扩展版本，广泛应用于语言建模，甚至包括“尚未出版”的书籍。BookCorpus 与 Project Gutenberg、Books 3 几乎没有重叠。

（16）Ubuntu IRC 数据集是从 Freenode IRC 聊天服务器上提取的，包含所有与 Ubuntu 相关的频道的公开聊天记录。这些聊天记录数据提供了语言模型用于建模人类交互的可能性。

（17）EuroParl[148] 是一个多语言平行数据库，最初是为机器翻译任务构建的，也在自然语言处理的其他几个领域中得到了广泛应用[149–151]。Pile 数据集中所使用的版本包括 1996 年至 2012 年欧洲议会的 21 种欧洲语言的议事录。

（18）YouTube Subtitles 数据集是从 YouTube 上人工生成的字幕中收集的文本平行数据库。该数据集除了提供多语言数据，还包括教育内容、流行文化和自然对话的内容。

（19）PhilPapers 数据集由 University of Western Ontario 数字哲学中心（Center for Digital Philosophy）维护的国际数据库中的哲学出版物组成。它涵盖了广泛的抽象、概念性的话语，其文本写作质量也非常高。

（20）NIH 数据集包含 1985 年至今，所有获得美国 NIH 资助的项目申请摘要，是高质量的科学写作实例。

（21）Hacker News 数据集是初创企业孵化器和投资基金 Y Combinator 运营的链接聚合器。其目标是希望用户提交“任何满足一个人的知识好奇心的内容”，文章聚焦于计算机科学和创业主题。其中包含了一些小众话题的高质量对话和辩论。

（22）Enron Emails 数据集是由文献 [152] 提出的，它是用于研究电子邮件使用模式的数据集。该数据集的加入可以帮助语言模型建模电子邮件通信的特性。

Pile 中不同数据子集所占比例及训练时的采样权重有很大不同，高质量的数据会有更高的采样权重。例如，Pile-CC 数据集包含 227.12GB 数据，整个训练周期中采样 1 轮。虽然 Wikipedia (English) 数据集仅有 6.38GB 的数据，但是整个训练周期中采样 3 轮。具体的采样权重和采样轮数可以参考文献 [87]。

3.4.2 ROOTS

ROOTS（Responsible Open-science Open-collaboration Text Sources）数据集[128] 是 BigScience 项目在训练具有 1760 亿个参数的 BLOOM 大语言模型时使用的数据集。该数据集包含 46 种自然语言和 13 种编程语言，总计 59 种语言，整个数据集的大小约 1.6TB。ROOTS 数据集中各语言所占比例如图3.13 所示。图中左侧是以语言家族的字节为单位表示的自然语言占比树状图，其中欧亚大陆语言占据了绝大部分（1321.89GB）。右侧橙色矩形对应的是印度尼西亚语（18GB），它是巴布尼西亚大区唯一的代表。右下脚绿色矩形对应非洲语（0.4GB）。图中右侧是以文件数量为单

位的编程语言分布的华夫饼图（Waffle Plot），一个正方形大约对应 3 万个文件。

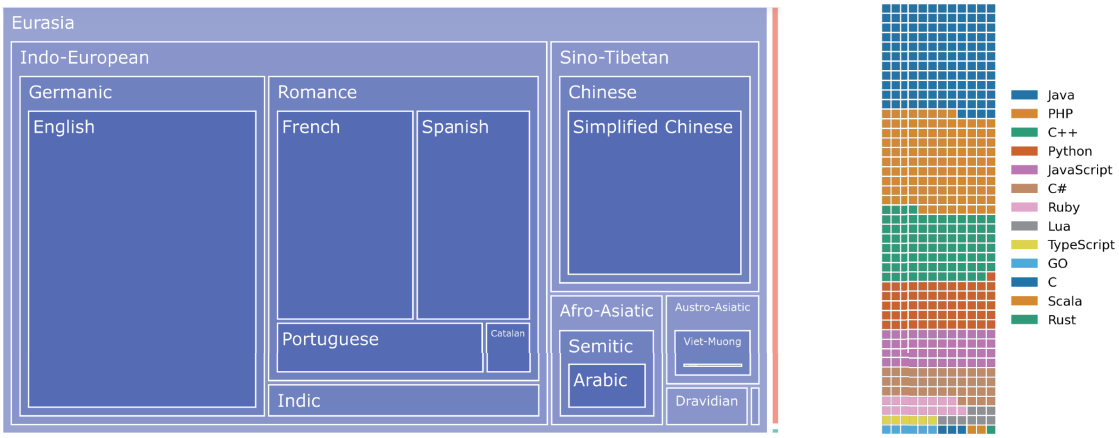


图 3.13　ROOTS 数据集中各语言所占比例[128]

ROOTS 中的数据主要来自四个方面：公开数据、虚拟抓取、GitHub 代码和网页数据。在**公开数据**方面，BigScience Data Sourcing 工作组的目标是收集尽可能多的各种类型的数据，包括自然语言处理数据集和各类型文档数据集。为此，还设计了 BigScience Catalogue[153] 用于管理和分享大型科学数据集，Masader Repository 用于收集阿拉伯语和文化资源的开放数据存储库。在收集原始数据集的基础上，进一步从语言和统一表示方面对收集的文档进行规范化处理。识别数据集所属语言并分类存储，将所有数据都按照统一的文本和元数据结构进行表示。由于数据种类繁多，ROOTS 数据集并没有公开其所包含数据集的情况，但是提供了 Corpus Map 及 Corpus Description 工具，以便查询各类数据集占比和数据情况。在 ROOTS 数据集中，中文数据集的种类及所占比例如图3.14 所示。其中，中文数据主要由 WuDao Corpora 和 OSCAR[154] 组成。在**虚拟抓取**方面，由于很多语言的现有公开数据集较少，因此这些语言的网页信息是十分重要的资源补充。在 ROOTS 数据集中，采用 CommonCrawl 网页镜像，选取了 614 个域名，从这些域名下的网页中提取文本内容补充到数据集中，以提升语言的多样性。在**GitHub 代码**方面，针对程序语言，ROOTS 数据集采用了与 AlphaCode[101] 相同的方法：从 BigQuery 公开数据集中选取文件长度在 100 到 20 万字符，字母符号占比在 15% 至 65%，最大行数在 20 至 1000 行的代码。训练大语言模型时，**网页数据**对于数据的多样性和数据量支撑起到重要的作用[2, 19]，ROOTS 数据集中包含了 OSCAR 21.09 版本，对应的是 CommonCrawl 2021 年 2 月的快照，占整体 ROOTS 数据集规模的 38%。

图 3.14　在 ROOTS 数据集中，中文数据集的种类及所占比例

在数据准备完成后，还要进行清洗、过滤、去重及隐私信息删除等工作，ROOTS 数据集处理流程如图3.15 所示。整个处理工作并非完全依赖自动计算，而是采用人工与自动相结合的方法。针对数据中存在的一些非自然语言的文本，例如预处理错误、SEO 页面或垃圾邮件（包括色情垃圾邮件），构建 ROOTS 数据集时会进行一定的处理。首先，定义一套质量指标，其中高质量的文本被定义为“由人类撰写，面向人类”（written by humans for humans），不区分内容（专业人员根据来源对内容进行选择）或语法正确性的先验判断。所使用的指标包括字母重复度、单词重复度、特殊字符、困惑度等。完整的指标列表可以参考文献 [128]。这些指标根据来源的不同，进行了两种主要的调整：针对每种语言单独选择参数，如阈值等；人工浏览每个数据来源，以确定哪些指标最可能识别出非自然语言。其次，针对冗余信息，采用 SimHash 算法[155]，计算文档的向量表示，并根据文档向量表示之间的海明距离（Hamming Distance）是否超过阈值进行过滤。最后，使用后缀数组（Suffix Array）删除存在 6000 个以上字符重复的文档。通过上述方法共发现 21.67% 的冗余信息。个人信息数据（包括邮件、电话、地址等）则使用正则表示的方法进行过滤。

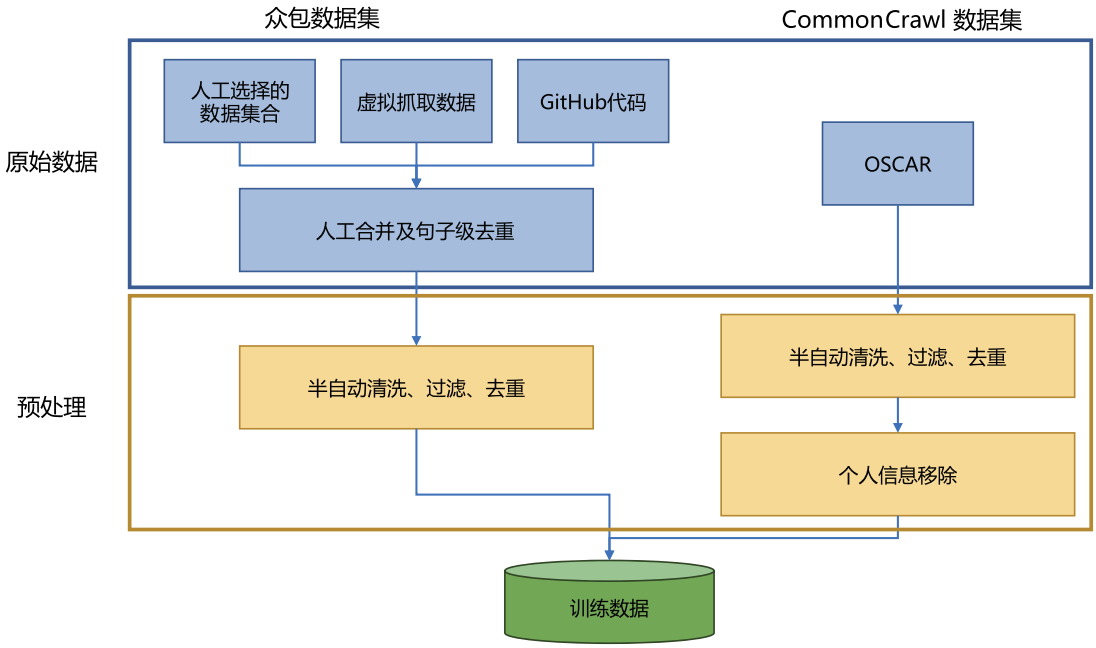
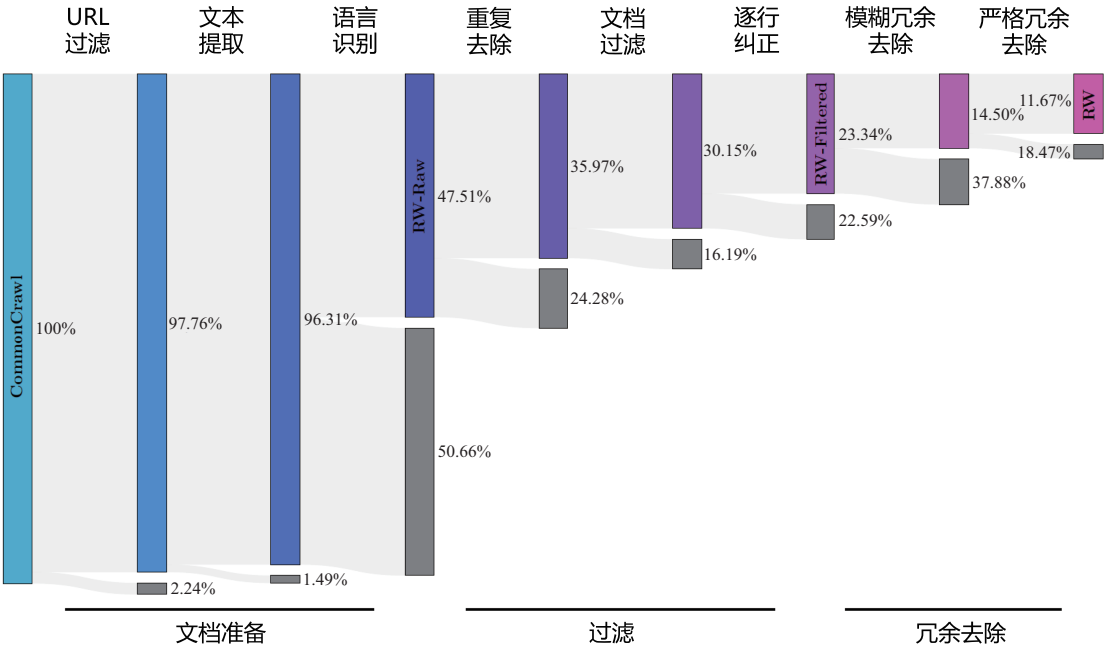


图 3.15 ROOTS 数据集处理流程[31]

3.4.3 RefinedWeb

RefinedWeb[60] 是由位于阿布扎比的技术创新研究院（Technology Innovation Institute，TII）在开发 Falcon 大语言模型时同步开源的大语言模型预训练集合，其主要由 CommonCrawl 数据集[156]过滤的高质量数据组成。CommonCrawl 数据集包含自 2008 年以来爬取的数万亿个网页，由原始网页数据、提取的元数据和文本提取结果组成，总数据量超过 1PB。CommonCrawl 数据集以 WARC（Web ARChive）格式或者 WET 格式进行存储。WARC 是一种用于存档 Web 内容的国际标准格式，包含了原始网页内容、HTTP 响应头、URL 信息和其他元数据。WET 文件只包含抽取出的纯文本内容。

文献 [60] 中给出了 RefinedWeb 中 CommonCrawl 数据集的处理流程和数据过滤百分比，如图3.16 所示。图中灰色部分是与前一个阶段相对应的移除率，阴影部分表示总体上的保留率。在文档准备阶段，移除率以文档数量的百分比进行衡量，过滤阶段和冗余去除阶段以词元为单位进行衡量。整个处理流程分三个阶段：文档准备、过滤和冗余去除。经过上述多个步骤，仅保留了大约 11.67% 的数据。RefinedWeb 一共包含 5 万亿个词元，开源公开部分包含 6 千亿个词元。



注：URL冗余去除未在图中体现。

图 3.16　RefinedWeb 中 CommonCrawl 数据集的过滤流程和数据过滤百分比[60]

文档准备阶段主要是进行 URL 过滤、文本提取和语言识别三个任务。**URL 过滤**（URL Filtering）主要针对欺诈和成人网站（指包含色情、暴力、赌博等内容的网站）。基于规则的过滤方法的使用如下。

（1）包含 460 万黑名单域名（Blacklist）。

（2）根据严重程度加权的词汇列表对 URL 评分。

文本提取（Text Extraction）的主要目标是仅提取页面的主要内容，同时去除菜单、标题、页脚、广告等内容。RefinedWeb 构建过程中使用 trafilatura 工具集[157]，并通过正则表达式进行部分后处理。**语言识别**（Language Identification）阶段使用 CCNet 提出的 fastText 语言分类器[125]。该分类器使用字符 n-gram 作为特征，并在 Wikipedia 上进行训练，支持 176 种语言识别。如图 3.16 所示，CommonCrawl 数据集中非英语数据占比超过 50%，经过语言识别后，过滤了所有非英语数据。通过文档准备阶段得到的数据集称为 RW-Raw。

过滤阶段主要包含重复去除、文档过滤、逐行纠正三个任务。**重复去除**（Repetition Removal）的主要目标是删除具有过多行、段落或 n-gram 重复的文档。这些文档主要由爬取错误或者低质重复的网页组成。这些内容会严重影响模型性能，使模型产生病态行为（Pathological Behavior），因此需要尽可能在早期阶段去除[123]。**文档过滤**（Document-wise Filtering）的目标是删除由机器生成

的垃圾信息，这些页面主要由关键词列表、样板文本或特殊字符序列组成。采用文献 [115] 中提出的启发式质量过滤算法，通过整体长度、符号与单词比率及其他标准剔除离群值，以确保文档是实际的自然语言。**逐行纠正**（Line-wise Correction）的目标是过滤文档中不适合语言模型训练的行（例如社交媒体计数器、导航按钮等）。使用基于规则的方法进行逐行纠正过滤，如果删除超过 5%，则完全删除该文档。经过过滤阶段，仅有 23.34% 的原始数据得以保留，所得的数据集称为 RW-Filtered。

冗余去除阶段包含模糊冗余去除、严格冗余去除及 URL 冗余去除三个任务。**模糊冗余去除**（Fuzzy Deduplication）的目标是删除内容相似的文档。RefinedWeb 构建时使用了 MinHash 算法[158]，能快速估算两个文档间的相似度。利用该算法可以有效过滤重叠度高的文档。RefinedWeb 数据集构建时，使用的是 5-gram 并分成 20 个桶，每个桶采用 450 个 Hash 函数。**严格冗余去除**（Exact Deduplication）的目标是删除连续相同的序列字符串。使用后缀数组进行逐个词元间的对比，并删除 50 个以上的连续相同词元序列。**URL 冗余去除**（URL Deduplication）的目标是删除具有相同 URL 的文档。CommonCrawl 数据集中存在一定量的具有重复 URL 的文档，并且这些文档的内容通常是完全相同的。构建 RefinedWeb 数据集时，对 CommonCrawl 数据集中不同部分之间相同的 URL 进行了去除。该阶段处理完成后的数据集称为 RefinedWeb，仅保留了原始数据的 11.67%。

以上三个阶段所包含的各个任务的详细处理规则可以参考文献 [60] 的附录部分。此外，文献 [60] 还利用三个阶段产生的数据分别训练 10 亿和 30 亿参数规模的模型，并使用零样本泛化能力对模型结果进行评测。评测后发现，RefinedWeb 的效果远好于 RW-Raw 和 RW-Filtered。这也在一定程度上说明高质量数据集对语言模型具有重要的影响。

3.4.4 CulturaX

CulturaX[159] 是一个可以用于预训练的多语言数据集，涵盖 167 种语言，包含 6.3 万亿个词元。它通过整合 mC4[160]（3.1.0 版本）和 OSCAR[161–163]（20.19、21.09、22.01 以及 23.01 版本）数据集，并经过语言识别、URL 过滤、基于度量的清洗、文档精炼以及数据去重等一系列严格的数据处理步骤，有效解决了现有多语言数据集存在的语言识别不准确、文档级去重缺失、数据清理不彻底等问题。该数据集具有多语言、开源、大规模和高质量的特点，旨在提升多语言场景下模型训练的数据质量，推动多语言学习的研究与发展，为训练高性能的多语言大语言模型提供了有力的数据支持，有助于打破训练数据不透明的现状。

mC4 最初是为训练多语言编码器-解码器模型 mT5[160] 而创建，涵盖 101 种语言，从 CommonCrawl 的 71 个月度快照中获取数据，经过去除短行页面、不良词汇页面及重复行去除等处理，其语言识别借助 cld3[164] 工具。OSCAR 数据集同样也来源于 CommonCrawl，开发了高性能的数据管道，对 166 种不同语言的网页数据进行分类和过滤。区别于以往依赖精选数据集（如 The Pile 和 BookCorpus）训练大语言模型的做法。在多语言场景下，网络爬虫数据集更具优势，它有助于高效收集多语言数据。尽管其原始数据质量参差不齐，但经清洗后可以很好应用于大语言模型训

练。二者组合后，为后续处理提供了多达 135 亿份文档。其中，mC4 占比 66%，OSCAR 23.01 占比 11%，OSCAR 22.01 占比 7%，OSCAR 21.09 占比 9%，OSCAR 20.19 占比 7%。

基于 mC4 和 OSCAR 合并后的数据集，CulturaX 研究团队通过一系列数据处理步骤来构造高质量的多语言数据集，包括语言识别、基于 URL 的过滤、基于指标的清洗、文档优化、冗余去除。具体清洗工作如下：

(1) 语言识别：在处理 mC4 和 OSCAR 数据集时，一个较为突出的问题是二者分别使用了 cld3 和 FastText 这两种不同的语言识别工具。此前的研究已经证实，cld3 在语言检测方面的表现远逊于 FastText，这使得 mC4 中出现了大量的语言检测错误[154]。因此，CulturaX 团队使用 FastText 对 mC4 中的文档语言重新进行预测。若文档的预测语言与 mC4 中原本提供的语言不一致，那么该文档将从数据集中剔除。这样做的目的在于避免那些会使 cld3 和 FastText 语言检测器产生混淆的文档，因为这些文档极有可能给数据带来噪声干扰。

(2) 基于 URL 的过滤：为了降低数据中的有害信息，CulturaX 研究团队使用了图卢兹大学（University of Toulouse）提供的最新 UT1 URL 和域名黑名单，将有毒和有害页面从数据中删除。该列表包含来自色情、抱怨和黑客攻击等不同主题的网站，名单每周更新两到三次。目前该黑名单包含超过 370 万条由人类和机器（如搜索引擎、已知地址和索引）共同贡献的记录[163]。mC4 数据集之前未使用过该黑名单进行过滤。OSCAR 数据集虽然使用过该黑名单进行数据清洗，但是可以根据更新的名单进一步进行清洗。

(3) 基于指标的清洗：受 ROOTS 语料库数据处理启发，CulturaX 数据集构建中也利用各种数据集指标的分布来识别和过滤异常文档。每个指标为数据集中的文档提供量化特定属性的单一值，根据指标值及其范围确定阈值，将其分为正常和异常范围，异常范围的文档被视为噪声，并从数据集中删除。使用一系列全面的指标，包括单词数量、字符和单词重复比率等。同时高困惑度分数的文档也会被视为噪声排除。由于重复信息会对训练大语言模型产生不利影响，CulturaX 研究团队利用不同语言的停用词和标记词列表计算比率以删除文档，还通过 FastText 获取语言识别置信度辅助过滤。

(4) 文档优化：由于 mC4 和 OSCAR 的文档是从互联网上抓取的 HTML 页面中提取的，其中很大一部分可能带有抓取和提取错误，包括长 JavaScript 行和无关内容。因此，对于每个文档，文档优化步骤的目标是通过一系列操作去除其噪声或不相关的部分。首先，去除每个文档末尾的短行，因为这些行通常包含页脚细节或来自网站的无用信息。其次，删除包含 JavaScript（JS）关键词列表中的单词（例如“<script>”）的行，以避免不相关和非语言信息。

(5) 冗余去除：尽管进行了全面的数据清洗，但由于信息在网络上重新发布、对同一文章的多次引用、样板内容和抄袭等各种原因，剩余数据集仍可能包含大量重复数据，这会导致大语言模型记忆和泛化能力受到影响，因此数据去重对保证训练数据质量至关重要。为此，CulturaX 研究团队利用 MinHash 和 URL 对数据集进行全面去重，并按语言独立进行。其中，MinHashLSH[165] 方法用于过滤相似文档，它基于 MinHash[158] 的多个哈希函数和 Jaccard 相似度，结合局部敏感哈

希提高效率；最后基于 URL 去除相同 URL 的文档，但避免删除仅含通用域的 URL。

3.4.5 SlimPajama

SlimPajama[166] 是由 CerebrasAI 公司针对 RedPajama 进行清洗和去重后得到的开源数据集。原始的 RedPajama 包含 1.21 万亿个词元，经过处理的 SlimPajama 数据集包含 6270 亿个词元。SlimPajama 还开源了用于对数据集进行端到端预处理的脚本。RedPajama 是由 TOGETHER 联合多家公司发起的开源大语言模型项目，试图严格按照介绍 LLaMA 模型的论文中的方法构造大语言模型训练所需的数据。虽然 RedPajama 数据集的数据质量较好，但是 CerebrasAI 的研究人员发现其存在以下两个问题。

（1）一些数据中缺少数据文件。

（2）数据集中包含大量重复数据。

为此，CerebrasAI 的研究人员针对 RedPajama 数据集开展了进一步的处理。

SlimPajama 数据集的处理过程如图3.17 所示。整体处理过程包括多个阶段：NFC 正规化、过滤短文档、全局去重、文档交错、文档重排、训练集和保留集拆分，以及训练集与保留集中相似数据去重等步骤。所有步骤都假定整个数据集无法全部装载到内存中，并分布在多个进程中进行处理。使用 64 块 CPU，大约花费 60 多个小时就可以完成 1.21 万亿个词元的处理。整个处理过程所需内存峰值为 1.4TB。

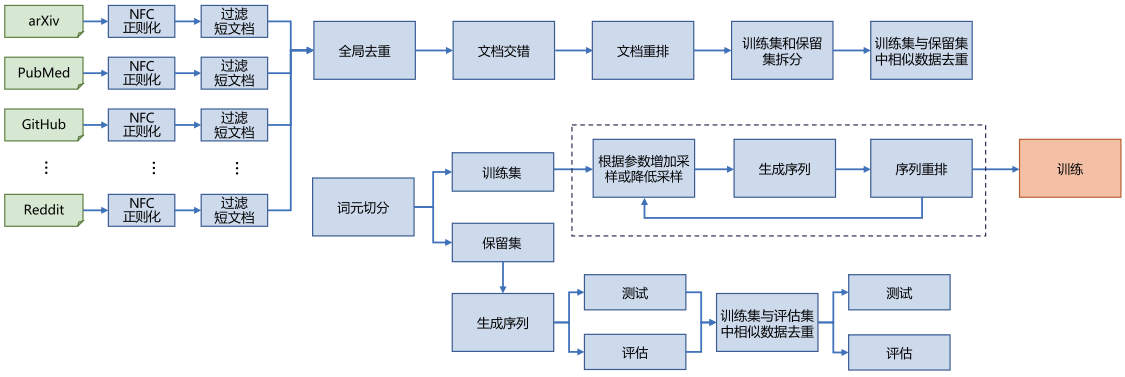


图 3.17　SlimPajama 数据集的处理过程[166]

SlimPajama 处理的详细流程如下。

（1）NFC 正则化（NFC Normalization）的目标是去除非 Unicode 字符，SlimPajama 遵循 GPT-2 的规范，采用 NFC（Normalization Form C）正则化方法。NFC 正则化的命令示例如下：

```
python preprocessing/normalize_text.py \
    --data_dir <prefix_path>/RedPajama/arxiv/  \
    --target_dir <prefix_path>/RedPajama_norm/arxiv/
```

（2）过滤短文档（Filter Short Documents）：RedPajama 的源文件中下载错误或长度非常短的内容占比为 1.85%，这些内容对模型训练没有作用。在去除标点、空格、换行和制表符后，过滤了长度少于 200 个字符的文档。查找需要过滤的文档的命令示例如下：

```
python preprocessing/filter.py \
      <prefix_path>/RedPajama_norm/<dataset_name>/ \
      <prefix_path>/RedPajama_filtered.pickle <n_docs> \
      <dataset_name> <threshold>
```

（3）全局去重（Deduplication）：为了对数据集进行全局去重（包括数据库内和数据库间的去重），SlimPajama 使用了 datasketch 库，并进行了一定的优化以减少内存消耗并增加并行性。SlimPajama 采用生产者-消费者模式，对运行时占主导地位的 I/O 操作进行了有效的并行。整个去重过程包括多个阶段：构建 MinHashLSH 索引、在索引中进行查询以定位重复项、构建图表示以确定重复连通域，最后过滤每个成分中的重复项。

（a）MinHash 生成（MinHash Generation）：为了计算每个文档的 MinHash 对象，先从每个文档中去除标点、连续空格、换行和制表符，并将其转换为小写。接下来，构建 13-gram 的列表，这些 n-gram 作为特征用于创建文档签名，并添加到 MinHashLSH 索引中。MinHash 生成的命令示例如下：

```
    python dedup/to_hash.py <dataset_name> \
         <prefix_path>/RedPajama_norm/<dataset_name>/\
         <prefix_path>/RedPajama_minhash/<dataset_name>/ \
         <n_docs> <iter> <index_start> <index_end> \
         -w <ngram_size> -k <buffer_size>
```

（b）重复对生成（Duplicate Pairs Generation）：使用 Jaccard 相似度计算文档之间的相似度，设置阈值为 0.8 来确定一对文档是否应被视为重复。SlimPajama 的实现使用了 –range 和 –bands 参数，可在给定 Jaccard 阈值的情况下使用 datasketch/lsh.py 进行计算。重复对生成的命令示例如下：