信息熵计算与数据采样方法指南

本文档详细介绍 6Analyst 中基于信息熵的数据排序和采样方法，包括原理、算法实现和使用示例。

目录

概述

在网络资产分析中，原始数据的质量参差不齐。有些记录包含丰富的服务信息（多个端口、详细的 Banner），而有些记录可能只有简单的 SSH 版本号。

传统方法按字符串长度筛选数据存在明显缺陷：

- ASCII 艺术图案字符多但信息量低

- 重复内容（如大量相同字符）被误判为高价值
- 无法区分真正有意义的信息

信息熵方法通过计算数据的信息含量，能够：

- 自动过滤重复、冗余内容
- 准确反映真实信息价值
- 优先选择对模型分析最有帮助的数据

信息熵原理

Shannon 信息熵

信息熵（Shannon Entropy）由克劳德·香农于 1948 年提出，用于量化信息的不确定性或信息含量。

数学定义：

```
H(X) = -Σ p(x) × log₂(p(x))
```

其中：

- `H(X)` 是随机变量 X 的熵
- `p(x)` 是字符 x 出现的概率
- 求和遍历所有可能的字符

熵值范围：

- 最小值 `0`：所有字符完全相同（如 "aaaaaaa"）
- 最大值 `log₂(n)`：所有字符均匀分布（n 为字符集大小）

直观理解：

文本示例	熵值	说明
`aaaaaaaaaa`	0.0	完全重复，无信息量
`abababab`	1.0	两种字符交替，信息量低
`MikroTik RouterOS 6.49`	~4.2	正常文本，信息量适中
`aZ9#kL2@mN`	~3.3	随机字符，信息量高

为什么使用信息熵

场景对比：

场景	字符长度	信息熵	实际价值
SSH Banner: `SSH-2.0-OpenSSH_8.9`	21	3.8	⭐⭐⭐ 高
ASCII 艺术 (100行重复字符)	5000	1.2	⭐ 低
详细设备信息页面	2000	4.5	⭐⭐⭐⭐ 很高
重复错误信息 `404 404 404...`	1000	1.5	⭐ 低

信息熵能够准确区分这些情况，而简单的长度判断会被 ASCII 艺术和重复内容误导。

优势

> **基于信息熵的筛选方法**能够**自动筛选、过滤掉信息量低，长度短的低质量数据**，保留**质量较高**的数据。信息熵排序过滤阶段**之前**经历了**数据清洗阶段**，在数据清洗阶段会去除掉每条ip数据中大部分的乱码、哈希值、基础代码、html标签，会**保留所有具有语义的字段，排除乱码造成高信息熵导致假高信息量数据的情况，因此结合信息熵的过滤是正确且高效的方法。**

信息熵计算方法

基础熵计算

位于 `6Analyst/exp/data_extractor.py`：

```python
from collections import Counter
import math

def calculate_entropy(text: str) -> float:
    """
    计算文本的 Shannon 信息熵

    Args:
        text: 输入文本

    Returns:
        信息熵值 (0 到 log2(字符集大小))
    """
    if not text or len(text) == 0:
        return 0.0

    # 统计字符频率
    char_counts = Counter(text)
    text_len = len(text)

    # 计算熵
    entropy = 0.0
    for count in char_counts.values():
        probability = count / text_len
        if probability > 0:
            entropy -= probability * math.log2(probability)

    return entropy
```

计算示例：

```
# 示例1：完全重复
```

```
text1 = "aaaaaaaaaa"
# 字符频率: {'a': 1.0}
# H = -1.0 × log₂(1.0) = 0.0

# 示例2：两种字符均匀分布
text2 = "abababababab"
# 字符频率: {'a': 0.5, 'b': 0.5}
# H = -0.5 × log₂(0.5) - 0.5 × log₂(0.5)
#   = -0.5 × (-1) - 0.5 × (-1)
#   = 1.0

# 示例3：正常 Banner
text3 = "SSH-2.0-OpenSSH_8.9p1 Ubuntu"
# 字符频率分布较均匀
# H ≈ 3.8
```

丰富度评分算法

基础熵只考虑单个文本，而网络资产数据包含多个服务和字段。

`calculate_richness_score` 函数综合评估整条记录的信息丰富度：

```
def calculate_richness_score(data: Dict) -> float:
    """
    计算数据的信息熵综合分数

    综合考虑：
    1. 服务数量（多样性）
    2. 每个服务的信息熵（信息含量）
    3. 字段的完整性（有多少有效字段）
    """
    services = data.get('Services', {})
    if not services or not isinstance(services, dict):
        return 0.0

    total_entropy = 0.0
    total_fields = 0
    service_count = len(services)

```

```
        # 遍历每个服务
        for service_name, service_data in services.items():
            if not isinstance(service_data, dict):
                continue

            # 计算每个字段的熵
            for field_name, field_value in service_data.items():
                if field_value is None:
                    continue

                # 转换为字符串
                if isinstance(field_value, (dict, list)):
                    text = json.dumps(field_value, ensure_ascii=False)
                else:
                    text = str(field_value)

                # 跳过太短的字段
                if len(text) < 5:
                    continue

                # 计算熵
                entropy = calculate_entropy(text)

                # 字段权重
                weight = 1.0
                if field_name in ['Banner', 'Body']:
                    weight = 2.0  # Banner 和 Body 更重要
                elif field_name in ['Banner Hash', 'Body sha256']:
                    weight = 0.1  # Hash 值信息量低

                # 长度因子：避免过度惩罚长文本
                length_factor = math.log2(len(text) + 1)

                total_entropy += entropy * length_factor * weight
                total_fields += 1

        if total_fields == 0:
            return 0.0

```

```
        # 综合分数
        avg_entropy = total_entropy / total_fields
        service_factor = math.log2(service_count + 1)    # 服务数量因子
        field_factor = math.log2(total_fields + 1)        # 字段数量因子

        return avg_entropy * service_factor * field_factor
```

评分公式：

```
丰富度分数 = 平均加权熵 × log₂(服务数+1) × log₂(字段数+1)
```

各因子作用：

因子	作用	说明
平均加权熵	衡量信息密度	过滤重复/冗余内容
服务数量因子	鼓励多样性	多服务的记录更有价值
字段数量因子	鼓励完整性	字段越多信息越全面
字段权重	区分重要性	Banner/Body 权重更高

实际评分示例：

记录类型	服务数	字段数	平均熵	丰富度分数
仅 SSH 版本号	1	1	2.5	~2.5
SSH + HTTP 基础	2	4	3.2	~15.8
多服务详细信息	5	12	4.0	~58.4
完整设备页面	8	20	4.2	~95.6

数据采样方法

isort 基础排序

`isort` 函数按信息熵对数据集排序，可选择保留前 N% 的高熵数据：

```python
from 6Analyst.entropy_sorter import isort, isort_with_entropy

# 示例数据
dataset = [
    {"192.168.1.1": {"Services": {...}}},
    {"192.168.1.2": {"Services": {...}}},
    # ...
]

# 按熵排序，返回全部数据
sorted_data = isort(dataset)

# 只保留信息熵最高的前 80%
top_80_data = isort(dataset, ratio=0.8)

# 同时返回熵值
sorted_with_entropy = isort_with_entropy(dataset, ratio=0.8)
for item, entropy in sorted_with_entropy:
    print(f"Entropy: {entropy:.2f}")
```

函数签名：

```python
def isort(
    dataset: List[Dict],
    ratio: Optional[float] = None
) -> List[Dict]:
    """
    根据信息熵对数据集排序和筛选

    Args:
        dataset: 输入数据集
        ratio: 保留比例 (0-1]，None 表示全部保留
```

```
11
12      Returns:
13          按熵从高到低排序的数据列表
14      """
```

厂商平衡采样

在实际场景中，数据集往往存在厂商分布不均的问题。例如 MikroTik 设备可能占 60%，而其他厂商各占很小比例。直接按熵排序会导致采样结果被主流厂商主导。

`isort_with_vendor_balance` 实现了厂商平衡采样：

```python
from 6Analyst.entropy_sorter import isort_with_vendor_balance

# 目标采样 1000 条，保持厂商平衡
sampled_data, stats = isort_with_vendor_balance(
    dataset,
    target_count=1000,
    major_ratio=0.9,                # 主流厂商占 90%
    max_single_vendor_ratio=0.40,   # 单厂商最高 40%
    min_single_vendor_ratio=0.10,   # 单厂商最低 10%
    top_n_vendors=5                 # 前 5 大厂商为主流厂商
)

# 查看统计信息
print(f"采样数量: {stats['sampled']}")
print(f"主流厂商: {stats['major_vendors']}")
print(f"厂商分布: {stats['vendor_distribution']}")
```

采样规则：

1. **动态识别主流厂商**：按数量排序，取前 N 个厂商
2. **其他厂商保护**：
   - 如果其他厂商占比 < 10%，则全采样其他厂商
   - 主流厂商目标 = 其他厂商数量 × 9
3. **配额分配**：
   - 每个主流厂商最低 10%（占主流目标）

    - 每个主流厂商最高 40%（占总目标）
4. **熵优先**：每个厂商内部按信息熵从高到低采样

厂商标准化：

系统内置厂商别名映射，自动将各种变体名称归一化：

```
VENDOR_ALIASES = {
        'cisco': 'Cisco',
        'cisco ios': 'Cisco',
        'cisco ios xr': 'Cisco',
        'mikrotik': 'MikroTik',
        'routeros': 'MikroTik',
        'juniper': 'Juniper',
        'junos': 'Juniper',
        # ... 更多映射
}
```

采样示例：

假设原始数据分布：

厂商	数量	占比
MikroTik	6000	60%
Cisco	2000	20%
Juniper	1000	10%
Huawei	500	5%
Fortinet	300	3%
其他	200	2%

目标采样 1000 条，采样结果：

厂商	配额	实际采样	说明
其他	200	200	全采样（占比<10%）
MikroTik	400	400	达到 40% 上限
Cisco	180	180	按比例分配
Juniper	100	100	最低 10% 保证
Huawei	80	80	最低 10% 保证
Fortinet	40	40	最低 10% 保证
总计	**1000**	**1000**	

使用示例

示例 1：基础熵排序

```python
import json
from 6Analyst.entropy_sorter import isort, get_entropy_statistics

# 加载数据
with open('data/input/devices.jsonl', 'r') as f:
      dataset = [json.loads(line) for line in f]

print(f"原始数据量: {len(dataset)}")

# 查看熵统计
stats = get_entropy_statistics(dataset)
print(f"熵值范围: {stats['min']:.2f} - {stats['max']:.2f}")
print(f"平均熵值: {stats['mean']:.2f}")
print(f"中位数熵值: {stats['median']:.2f}")

# 按熵排序，保留前 50%
filtered = isort(dataset, ratio=0.5)
print(f"筛选后数据量: {len(filtered)}")
```

示例 2：厂商平衡采样

```python
from 6Analyst.entropy_sorter import isort_with_vendor_balance

# 加载数据
dataset = load_dataset('data/input/routers.jsonl')

# 厂商平衡采样
sampled, stats = isort_with_vendor_balance(
    dataset,
    target_count=2000,
    top_n_vendors=5
)

# 输出统计
print("\n=== 采样统计 ===")
print(f"原始数据: {stats['total']} 条")
print(f"采样数据: {stats['sampled']} 条")
print(f"主流厂商: {stats['major_vendors']}")

print("\n=== 厂商分布 ===")
for vendor, count in stats['vendor_distribution'].items():
    original = stats['original_distribution'].get(vendor, 0)
    print(f"  {vendor}: {count} 条 (原始 {original} 条)")
```

示例 3：命令行使用 isort 模式

```bash
# 使用 isort 模式运行分析
python run_6analyst.py --isort

# 指定采样比例
python run_6analyst.py --isort --isort-ratio 0.8

# 指定目标数量
python run_6analyst.py --isort --isort-count 1000

# 厂商平衡采样
python run_6analyst.py --isort --isort-vendor-balance
```

API 参考

entropy_sorter 模块

`calculate_entropy_for_dataset(dataset)`

为数据集中每条数据计算信息熵。

参数:

- `dataset`: `List[Dict]` - 数据集列表

返回:

- `List[Tuple[Dict, float]]` - (数据, 熵值) 元组列表

`isort(dataset, ratio=None)`

按信息熵排序数据集。

参数:

- `dataset`: `List[Dict]` - 输入数据集
- `ratio`: `Optional[float]` - 保留比例 (0, 1]，None 表示全部

返回:

- `List[Dict]` - 排序后的数据列表

`isort_with_entropy(dataset, ratio=None)`

按信息熵排序，同时返回熵值。

参数:

- `dataset`: `List[Dict]` - 输入数据集
- `ratio`: `Optional[float]` - 保留比例

返回:

- `List[Tuple[Dict, float]]` - (数据, 熵值) 元组列表

`get_entropy_statistics(dataset)`

获取数据集的熵统计信息。

参数:

- `dataset`: `List[Dict]` - 输入数据集

返回:

- `Dict` - 包含 count, min, max, mean, median

`isort_with_vendor_balance(dataset, target_count, ...)`

厂商平衡采样。

参数:

- `dataset`: `List[Dict]` - 输入数据集
- `target_count`: `int` - 目标采样数量
- `major_ratio`: `float` - 主流厂商占比，默认 0.9
- `max_single_vendor_ratio`: `float` - 单厂商最高占比，默认 0.40
- `min_single_vendor_ratio`: `float` - 单厂商最低占比，默认 0.10
- `top_n_vendors`: `int` - 主流厂商数量，默认 5

返回:

- `Tuple[List[Dict], Dict]` - (采样数据, 统计信息)

---

data_extractor 模块

`calculate_entropy(text)`

计算文本的 Shannon 信息熵。

参数:

- `text`: `str` - 输入文本

返回:

- `float` - 熵值

`calculate_richness_score(data)`

计算数据记录的信息丰富度分数。

参数:

- `data`: `Dict` - 数据记录

返回:

- `float` - 丰富度分数

总结

信息熵方法为 6Analyst 提供了科学的数据质量评估手段:

方法	适用场景	优势
`isort`	快速筛选高质量数据	简单高效
`isort_with_vendor_balance`	保持厂商多样性	避免数据偏斜
`calculate_richness_score`	评估单条记录质量	综合考虑多因素

通过合理使用这些方法，可以:

- 优先处理信息量最大的数据
- 减少无效数据对模型的干扰
- 保持采样数据的多样性和代表性