

# 2025 自然语言处理 课程设计 1

人工智能学院 221300079 王俊童

2025.4.1

综述，首先观察代码结构，逻辑如下：

- 命令行参数解析。有 method，是否 analyze，statistical 里面方法的选取。
- 加载数据和数据分析（需要我们实现数据分析）
- 三个方法的训练：
  - rule: 基于一些规则得到的一个实现。train 有四种纠错规则：
    - \* `_extract_confusion_pairs`: 字符混淆对提取。
    - \* `_extract_punctuation_rules`: 标点符号规则提取
    - \* `_extract_grammar_rules`: 语法规则提取
    - \* `_extract_word_confusion`: 词汇混淆对提取然后以上四种错误的纠错发生在 correct 里面。
  - statistical: 基于统计学习方法的纠错。这个里面又分为两个模型：
    - \* ngram 模型: 初始化了一堆数据结构，1-4 的 gram 方法，字符混淆矩阵和错误率等
    - \* ml 模型: 用机器学习方法去做。
  - 集成学习方法，在框架代码的 ensemble 部分有留给我们实现。
- 三个方法对应的纠错和评估。跟上面一样了，可以实现很多的 correct 方法，都有对应接口。

可以看出整个代码框架都还是比较整齐的，我们需要完成的 TODO 任务如下：

- 数据的 analyze 分析部分和画图。
- rule: 完成规则方法的实现。完成对应规则方法的纠错改正。
- statistical: 完成 ngram 和 ml 方法的对应修正和改正。
- main: 完成集成学习方法。
- 其余可以加一些深度学习之类的方法实现。

## 1 实现方法及其简单描述，遇到的问题 and 解决方案（全包含，就不单独列了，按照我的编程和问题思考思路来写的）

### 1.1 数据分析部分

数据分析部分，我们将原来的 args 做了一点点修改，然后我们首先可以根据原词典数据进行统计，把 label 为 1 的错误数据中的错误字符全部统计出来，而且可以得到错误率最高的 10 个的错误模式和错误字符，这更方便我们后续处理：

```
# 只看错的
if label == 1:
    error_count += 1

    if len(source) == len(target):
        for i, (s_char, t_char) in enumerate(zip(source, target)):
            if s_char != t_char:
                char_error_freq[s_char] += 1
                error_patterns[(s_char, t_char)] += 1
```

然后我们把它可视化, 同时, 由于 matplotlib 不支持中文字体, 需要更换自己电脑里面的路径。这个在对应 data analysis 的 python 文件里面有讲。

可以看一个我做出来的效果, 还是蛮不错的。可以看到的和地的错误最多, 还有的和得之类的, 一般都

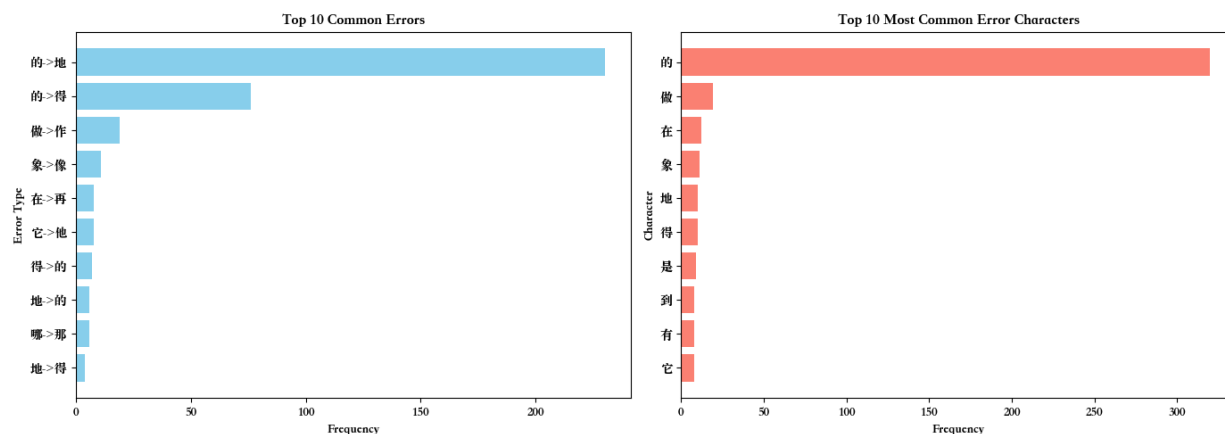


Figure 1: error distribution

是些同音不同意的字。

## 1.2 3 个方法部分

### 1.2.1 方法 1:rule

rule 这个方法还蛮简单的, 基本是基于人类的常识性的方法, 有点像是打表。但是肯定有补全不了的规则, 这个是硬伤。共有如下的需要填补的方法:

- self.extract\_confusion\_pairs: 这个方法已经给我们补全了。意思是提取了混淆字符对。但是这个一眼就存在一些问题:
  - 没有考虑插入和删除的错误
  - 没有考虑很强烈的上下文特征
  - 不同的 count 对于噪声过滤效果不一样, 可以产生不一样的效果

我们首先修改这个混淆对的做法: 我们加入一个 insert, delete 标识, 这样就考虑了重复和缺失两个新的情况, 然后对于 count 的噪声过滤, 我们设一个 min\_count, 这样就可以随时调控。但是, 我写了之后发现还降低了, 真没绷住啊, 说明好像不需要考虑这么多, 就注释掉了。但是我们保留一个 count 的修改

同时我们还尝试让他考虑上下文: `self.confusion_pairs[s_char][(t_char, context)] += 1`, 但是好像这样效果更差。我们的思考是, 引入上下文增加了噪声, 反而让这个效果不好了, 说明成对出现的错词可能多, 而且可能存在故意的行为, 我们就干脆不要好了。

BTW, 经过我反复尝试, mincount 去噪为 3 效果好, 这也证实了我的猜想, 上下文反而会增加噪声。

- self.\_extract\_punctuation\_rules: 我的思路是将每一句的标点单独的提取出来, 然后对比, 看哪一个位置的标点不一样, 然后就记录。但是事实上好像训练集的标点正确率有点高吧, 我只找到了一组有错误的。''' : defaultdict(< class'int' >, '' : 1) 那这个就很恶心了, 我可能需要自己去补充一些修正规则了。
- self.\_extract\_grammar\_rules: 我的想法是, 可以用一定的词性分析工具来看语法的正确或者错误。我们引入 import jieba.posseg。这个库可以拿来作词性分析。然后记录每种词性替换 (pos\_replace) 和单词替换 (word\_replace) 出现的次数。我们设置一个置信度, 这样就不会乱换一些东西, 跟之前的 count 其实是一个东西。我们认为:

$$confidence = \frac{appear\ counts}{training\ set\ length}$$

这样就可以避免一些问题。

经过实验, 其实我发现这个的作用率很小, 因为如果我按照 rule 直接去设定规则, 整个会有能修改的, 但是同时也会违背一些规则本身, 句法反而被改变了, 所以高置信度虽然可以保证不会错改, 但是修改及其的少。

- self.\_extract\_word\_confusion: 同理我们还是分词, 用高阈值过滤 (置信度 90% + 最小出现次数 3 次)。而且如果一个错误词可能对应多个正确词 (如 “的” 可能改为 “得” 或 “地”), 只保留最可能的修正。我们可以看到他的一些提取:

提取到 14 条易混淆词规则

```
'象' → '像' (置信度: 100.00%)
'唯个' → '唯一' (置信度: 100.00%)
'看做' → '看作' (置信度: 100.00%)
'当做' → '当作' (置信度: 100.00%)
'自己' → '自己' (置信度: 100.00%)
'好象' → '好像' (置信度: 100.00%)
'其它' → '其他' (置信度: 100.00%)
'纪录' → '记录' (置信度: 100.00%)
'来自' → '外地' (置信度: 100.00%)
'外地' → ',' (置信度: 100.00%)
```

其实还真像那么一回事。但是肯定有问题, 我们加两个约束就好, 首先长度要一样, 而且第一个字一般相同。

```
'象' → '像' (置信度: 100.00%)
'唯个' → '唯一' (置信度: 100.00%)
'看做' → '看作' (置信度: 100.00%)
'当做' → '当作' (置信度: 100.00%)
'自己' → '自己' (置信度: 100.00%)
'好象' → '好像' (置信度: 100.00%)
'其它' → '其他' (置信度: 100.00%)
```

这下确实对劲了

上面全部是提取, 那么下面给出修改的规则:

- self.\_correct\_punctuation(text): 那么根据上面说的其实标点错的很少, 我们就着重处理标点成对的问题, 这也是观察发现的。我们用 stack 来处理匹配问题, 做自动补全, 然后还调整引号和句号的顺序。
- self.\_correct\_confusion\_chars(corrected): 除了已经实现的, 其实基于最开始的实现, 我们也有加入 insert 或者 delete 的处理, 只不过到后面发现没啥用 QAQ, 但是我们发现重复字词还是蛮多的, 所以可以进行一个去重。那么要考虑的基本就是单字去重和单词去重。

- `self._correct_grammar(corrected)`: 结合上面的就是先进性序列化修正，然后替换可能错误的词性，对于高置信度的语法错误，可以直接换。
- `self._correct_word_confusion(corrected)`: 这个结合上面的修正规则，用一个词的 window 去做过滤，然后进行修正。

最后实现的效果如下：

```

===== Chinese Text Correction Evaluation Results =====

Sample-level Evaluation:
Accuracy: 0.3734
Character Accuracy: 0.8999

Detection Evaluation:
Precision: 0.0855
Recall: 0.0697
F1 Score: 0.0768
F0.5 Score: 0.0818

Correction Evaluation:
Precision: 0.0808
Recall: 0.0661
F1 Score: 0.0727
F0.5 Score: 0.0774

Final Score:
F0.5 Score: 0.0774
=====
    
```

Figure 2: rule

说实话，rule 这个方法是真的笨比，感觉不下降就很好了。

### 1.2.2 方法 2:ml 模型

这个板块分为 ngram 和 ml 方法，先看 ngram 吧：btw 我稍微修改了一下类别的问题，单独把两个拆开了，变成了两个单独的类别，这样好修改

#### 1.ngram 模型:

ngram 模型主要是要修改 `_train_ngram_model`，对于这个东西，首先看他干了啥：很显然，现在是最简单的，一个 gram 的实现。

- `train` 部分：这个模块主要是对训练样本进行一元，二元，三元，四元的统计并更新 counter。然后对错误 `label=1` 的进行分析，然后和目标文本对比，加入误差矩阵。并且储存每一个字的一个错误概率
- `correct`：主要是逐字去看错误概率，如果小于某个阈值就跳过。然后根据上下文纠错，看是否要纠正，然后用 ngram 进行选择。对比原字符看替换哪一个

那么对于这个，涉及到我到底看哪一个字符的评分，这个问题比较重要。事实上，好像把补全了之后没有很明显的提升，那就说明参数要调，还有一个问题是除了参数，阈值这个东西在我看的里面确实做的不够。然后我更换了阈值和调参数，具体操作：`python3 main.py -method statistical -analyze 0 -statistical_method ngram -statistical_optparam 1`。然后做出来的结果如下

从结果来看，这个任务用 ngram 的统计方法可能有点鸡肋了。

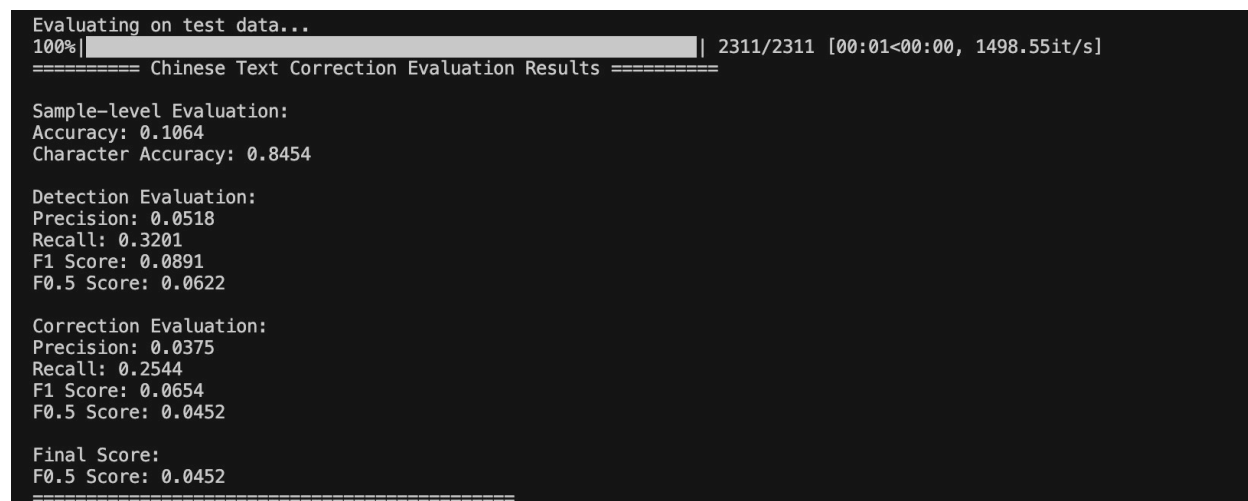


Figure 3: ngram

**2.ml 模型:**

1

### 1.2.3 方法 3:ensemble 模型

### 1.3 其余方法

## 2 如何复现结果和代码环境依赖问题

rulebase: 复现指令:

```
python3 main.py --method rule --analyze 0
```

statistical ngram: 复现指令:

```
python3 main.py --method statistical --analyze 0 --statistical_method ngram
--statistical_optparam 0
```

statistical ml: **复现指令:**

```
python3 main.py --method statistical --analyze 0 --statistical_method ml
```

### 3 不同实验方法的对比结果

## 4 一些简单思考