

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

写在开头，请助教老师一定要看完这份报告。先说结果，方便老师评价，
正常方法 F0.5: 0.1991 ± 0.0049 ,

用了预训练模型（挺作弊的一个方法）F0.5: 0.4120

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，遇到的问题和解决方案（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

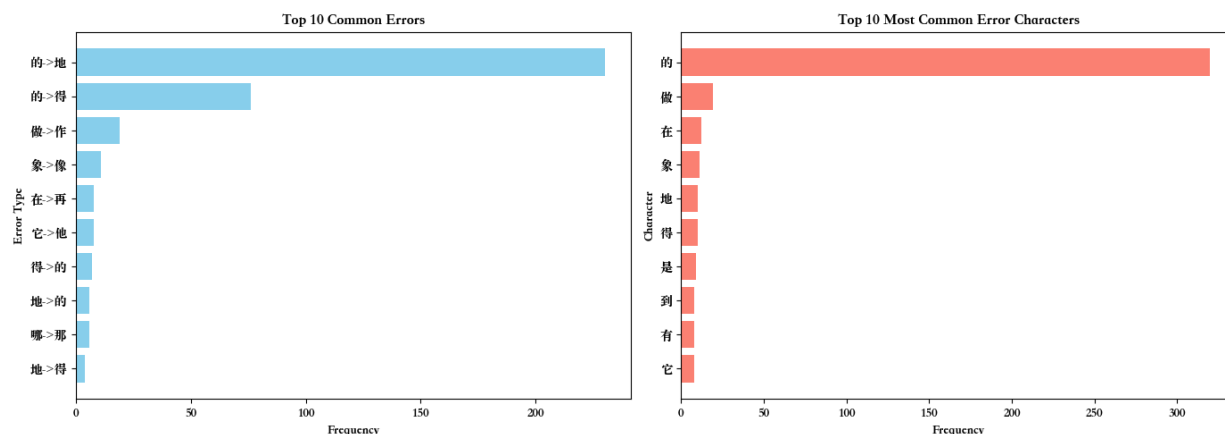


Figure 1: error distribution

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.possegaspgseg`。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

```
Evaluating on test data...
100% |██████████████████████████████████████████████████████████████████████████| 2311/2311 [00:01<00:00, 1498.55it/s]
===== Chinese Text Correction Evaluation Results =====

Sample-level Evaluation:
Accuracy: 0.1064
Character Accuracy: 0.8454

Detection Evaluation:
Precision: 0.0518
Recall: 0.3201
F1 Score: 0.0891
F0.5 Score: 0.0622

Correction Evaluation:
Precision: 0.0375
Recall: 0.2544
F1 Score: 0.0654
F0.5 Score: 0.0452

Final Score:
F0.5 Score: 0.0452
```

Figure 3: ngram

从结果来看，这个任务用 ngram 的统计方法可能有点鸡肋了。

2.ml 模型:

这个模型呢要基于传统机器学习模型来做，根据代码的提示呢，可以分为检测器和纠正器。具体如下：

- **上下文特征提取 (ContextWindowExtractor)**: 使用滑动窗口提取文本的局部上下文, 每个字符由其左右若干字符构成的上下文表示, 窗口大小可调, 默认为 3。
- **错误检测器 (self.detector)**: 使用 TF-IDF 特征编码字符级上下文, 结合 `SGDClassifier` 进行分类判断当前位置字符是否错误, 训练样本通过源-目标文本对生成, 包含数据增强。这个有说法, 数据增强在我其他所有方法都做了尝试, 只有 `ml` 方法有提升
- **字符纠正器 (self.corrector)**: 对检测为错误的位置, 提取其上下文窗口并使用 `RandomForestClassifier` 分类器预测正确字符。训练时只使用那些被标注为有误且原始与目标文本长度相同的样本。
- **数据增强机制**: 对于标注为有误的样本, 利用混淆集 `confusion_dict` 对正确字符引入干扰, 模拟更丰富的错误形式以提升模型鲁棒性。
- **纠错流程**: 给定输入文本, 首先滑窗提取上下文并检测错误位置, 再对错误窗口进行纠正预测并替换原字符, 输出最终纠错结果。

说实话，基于 ml 的方法很难做好，因为其实是需要有时候去看伪标签的，这个就很麻烦：

```
===== Chinese Text Correction Evaluation Results =====
Sample-level Evaluation:
Accuracy: 0.2778
Character Accuracy: 0.8843

Detection Evaluation:
Precision: 0.0509
Recall: 0.0842
F1 Score: 0.0635
F0.5 Score: 0.0553

Correction Evaluation:
Precision: 0.0435
Recall: 0.0729
F1 Score: 0.0545
F0.5 Score: 0.0473

Final Score:
F0.5 Score: 0.0473
=====
```

Figure 4: ml

1.2.3 方法 3:ensemble 模型

对于集成学习模型，由于每个单独的学习器本身都是弱学习器，因此整体性能的提升依赖于合理的集成策略。我们采用规则模型与统计模型的串联训练方式：先由规则模型进行初步纠错，然后将结果作为统计模型的训练输入。

在此基础上，为了在推理时做出更精确的决策，我们引入编辑距离和模型权重来综合评估多个学习器的输出。决策策略如下：

- 如果有目标文本 (target):
 - 选择编辑距离最小的模型输出；
 - 若多个模型的编辑距离相同，则根据这些模型的权重进行加权随机选择。
- 如果没有目标文本 (target):
 - 无法计算编辑距离，直接根据模型权重进行加权随机选择。

那这个就不讲道理了，因为我从理论上说，可以集成一百个或者任意多个学习器，但是确实是因为弱学习器的原因，导致效果不好。代码里面集成 3 个 corrector，理论上说，你可以集成无数个。至于为什么这里集成 3 个，因为 3 个效果好，问就是其他情况我都试过。而且可以有无数种顺序。

```
===== Chinese Text Correction Evaluation Results =====
Sample-level Evaluation:
Accuracy: 0.2999
Character Accuracy: 0.8864

Detection Evaluation:
Precision: 0.0531
Recall: 0.0858
F1 Score: 0.0656
F0.5 Score: 0.0574

Correction Evaluation:
Precision: 0.0464
Recall: 0.0759
F1 Score: 0.0576
F0.5 Score: 0.0503

Final Score:
F0.5 Score: 0.0503
=====
```

Figure 5: ensemble

1.3 其余方法: 深度学习

(1) 从零开始的 bert 训练学习：根据 bert 的原文来看，要训练两个东西，一个是 mask 的，一个是 nextseq。所以我们尝试去不用预训练的去弄一个 bert 架构的。同时引入 3 种噪声，insert, delete, replace。但是有一个问题是，训练集太小了，bert 完全训练不出来，最后结果就是一个都不对。(尝试过数据增强，但是由于本身数据集是稀疏的，然后就会导致一个问题，就是数据增强的时候，很多结果本身带来了噪声，导致更不准确了)

```
100% | 2311/2311 [01:48:00:00, 21.21it/s]
===== Chinese Text Correction Evaluation Results =====
Sample-level Evaluation:
Accuracy: 0.0013
Character Accuracy: 0.3528

Detection Evaluation:
Precision: 0.0045
Recall: 0.0942
F1 Score: 0.0085
F0.5 Score: 0.0055

Correction Evaluation:
Precision: 0.0000
Recall: 0.0000
F1 Score: 0.0001
F0.5 Score: 0.0000

Final Score:
F0.5 Score: 0.0000
=====
```

Figure 6: nn

(2) 用了预训练 bert 的学习：主要干了几件事：在初始化时，模型加载了 BERT tokenizer 和一个掩码语言模型 (BertForMaskedLM)，并使用 Adam 优化器和交叉熵损失函数进行训练。训练过程中，输入和目标文本会被转换为 token IDs，并确保它们的长度一致，模型通过前向传播计算损失并更新参数。在纠错阶段，correct 方法接收一个文本输入，使用 BERT 模型进行预测，输出修正后的文本。通过选择每个位置的最大概率 token 来生成纠错文本，并返回去掉空格后的修正结果。

但其实根据要求好像挺作弊的，就给出一个实现吧，大概结果如下：

```
===== Chinese Text Correction Evaluation Results =====
Sample-level Evaluation:
Accuracy: 0.5599
Character Accuracy: 0.8855

Detection Evaluation:
Precision: 0.5178
Recall: 0.2672
F1 Score: 0.3525
F0.5 Score: 0.4360

Correction Evaluation:
Precision: 0.4866
Recall: 0.2553
F1 Score: 0.3349
F0.5 Score: 0.4120

Final Score:
F0.5 Score: 0.4120
=====
```

Figure 7: nnpre

2 如何复现结果和代码环境依赖问题：sh run.sh

直接运行 run.sh. 指令如下：sh run.sh

注意，数据集要放在目录下运行当然，因为 run 只是帮你运行的更快，如果你要调参或者看数据格式的话：解析如下：

- --train_file: 训练数据路径（默认为 data/train.jsonl）
- --test_file: 测试数据路径（默认为 data/test.jsonl）
- --method: 选择纠错方法，可选项如下：
 - rule: 基于规则的纠错
 - statistical: 统计模型纠错
 - ensemble: 融合多个模型
 - nn: 神经网络模型
 - ol: 在线集成学习
 - olnc: 不作弊在线集成学习
 - nnpre: 带预训练 bert 神经网络模型
- --analyze: 是否进行数据分析（0 表示否，1 表示是）
- --statistical_method: 统计方法选择：
 - ml: 使用机器学习模型
 - ngram: 使用 N-gram 模型
- --statistical_optparam: 是否进行超参数网格搜索（0 表示否，1 表示是）
- Rule-based 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
```

- **Ensemble Learning 复现指令:**

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

- **Ensemble Online Learning 复现指令:**

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

- **Ensemble Online Learning(true)(不测试集学习版) 复现指令:**

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

- **Neural Network (NN) 复现指令:**

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

- **Neural Network Pretrained Bert(NN) 复现指令:**

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

环境依赖问题: 见 requirements.txt. 运行 run.sh 会帮你自己安装。

3 不同实验方法的对比结果

Ps:ml 模型没有固定种子, 所以可能运行出来有所略微区别。

模型	rule 模型	统计 ngram 模型	统计 ml 模型	集成模型	深度学习模型	预训练 bert
Final F0.5	0.0774	0.0452	0.0473	0.0503	0.0000	0.4120

Table 1: 方法性能对比

4 一些思考: 在线学习方法

通过研究我发现, 似乎对于后面的预测是一个一个进行的, 前面的学习也是一样, 那能不能通过**在线学习 (online learning)** 的方法来提高准确率呢, 好消息, 我觉得是可以的。

鉴于之前集成学习方法表现的实在是太垃圾了, 因为如果两个学习器都很弱, 集成了肯定更弱, 总之不会好。

那有没有方法可以让他变好呢, 有的兄弟, 有的。

我的想法就是在喂了测试集合之后, 先预测, 再学习, 类似于在线学习的方法来做。做一个在线集成学习, 这样也可以有效的平衡权重。根据在线集成学习的公式:

$$w_t = w_{t-1} \exp(-\lambda x)$$

这个公式里面, λ 就是学习率, 然后 x 我们需要一个合理的衡量标准来衡量两个学习器的贡献, 根据之前学过的算法知识, 编辑距离是一个不错的选择。

从 train 开始为了不改变结构, 前面先学, 学了之后后面来预测准确率, 发现准确率高的离谱。由于涉及不同的 seed, 我们做一个 0-10seed 的平均调参。

给出最新的对比模型:

模型	rule 模型	统计 ngram 模型	统计 ml 模型	集成模型	深度学习模型	预训练 bert	在线学习模型
F0.5	0.0774	0.0452	0.0473	0.0503	0.0000	0.4120	0.1812 ± 0.0055

Table 2: 方法性能对比

5 另一个思路：在线学习，但预测的时候不学习（所以并非动态更新）

听起来可能有点怪，不在 test 做学习，也就是说，我们的就用训练集分布去 fit 测试集分布。但这样对于这个任务（非在线的）其实好像是不泄漏的。所以我也尝试这么去做一下。

简单的讲两者的区别就是：

- 第一个把 train 和 test 都作为学习集了。
- 第二个 train 拿来学习，然后把学到的在线模型拿到 test 评估。

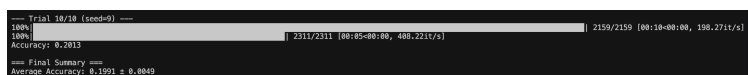


Figure 8: olnc

但是，我是真没想到这样竟然做的比把测试集拿来在线学习更好。这也是一个很神奇的现象。

模型	在线学习模型	在线学习模型 2
F0.5	0.1812 ± 0.0055	0.1991 ± 0.0049

Table 3: 方法性能对比 2，在线之间，亦有差距。

所以综上，不考虑作弊的 pretrain 和第四个在线学习模型，我们的对比如下：

模型	rule 模型	统计 ngram 模型	统计 ml 模型	集成模型	深度学习模型	在线学习模型
F0.5	0.0774	0.0452	0.0473	0.0503	0.0000	0.1991 ± 0.0049

Table 4: 方法性能对比