

含Emoji的垃圾文本检测系统

1 算法设计思路




1.0 结合emoji的网络垃圾文本检测系统应用背景

随着互联网的普及和用户规模的增长，emoji表情作为一种情感和语义的表达方式，能够丰富文本的含义，其在网络文本中的使用也越来越广泛。但就和具有特殊语义的字符一样，emoji也可能被用于传播敏感或不当内容。因此，将 emoji 与中文敏感文本检测相结合，不仅能够更全面地捕捉文本的语义信息，还能更准确地识别和过滤包含敏感内容的文本。通过建立一个结合中文文本和 emoji 的检测系统，可以有效减少垃圾信息和敏感内容的传播，保护用户的合法权益，维护网络空间的秩序和安全。

1.1 emoji 表征向量训练模型

我们需要训练一个模型，能够将 emoji 的 ID 映射到与其中文词意相匹配的向量表示。这样我们就可以检测含有emoji的一场文本。

1.1.1 数据收集与预处理

1. 首先是数据收集，我们发现没有办法直接获取对应的含有大量emojiURL编码及其文字组合释义的数据集，就思考能否编写一个爬虫代码，直接从网页上爬取原始数据，来实现数据加载和处理。并且考虑到有些emoji是会保留肤色变体，在后序的数据处理中，需要对这类复合emoji进行编码转换。由于爬虫获取的无效信息较多，我们在投入模型训练前需要进行数据清洗。
2. 为了让模型能够识别emoji中文含义的谐音字/变体，我们专门寻找形如{"":"辣鸡"}需要进行联想的数据集进行训练。考虑到同一个中文意思可以由不同的数个emoji来组合表示，为了能处理这类复杂的复合形式，最后选择采用了Skip-Gram风格的Emoji2Vec模型，这是一种用于训练词嵌入（word embeddings）的模型，属于自然语言处理（NLP）领域中的词向量（word vectors）学习方法，可以将单词映射到低维向量空间中，使得语义相似的单词在向量空间中彼此接近。因此我们需要构造 {"":"辣鸡"} {"":"辣鸡"}的数据对。

1.1.2 模型设计

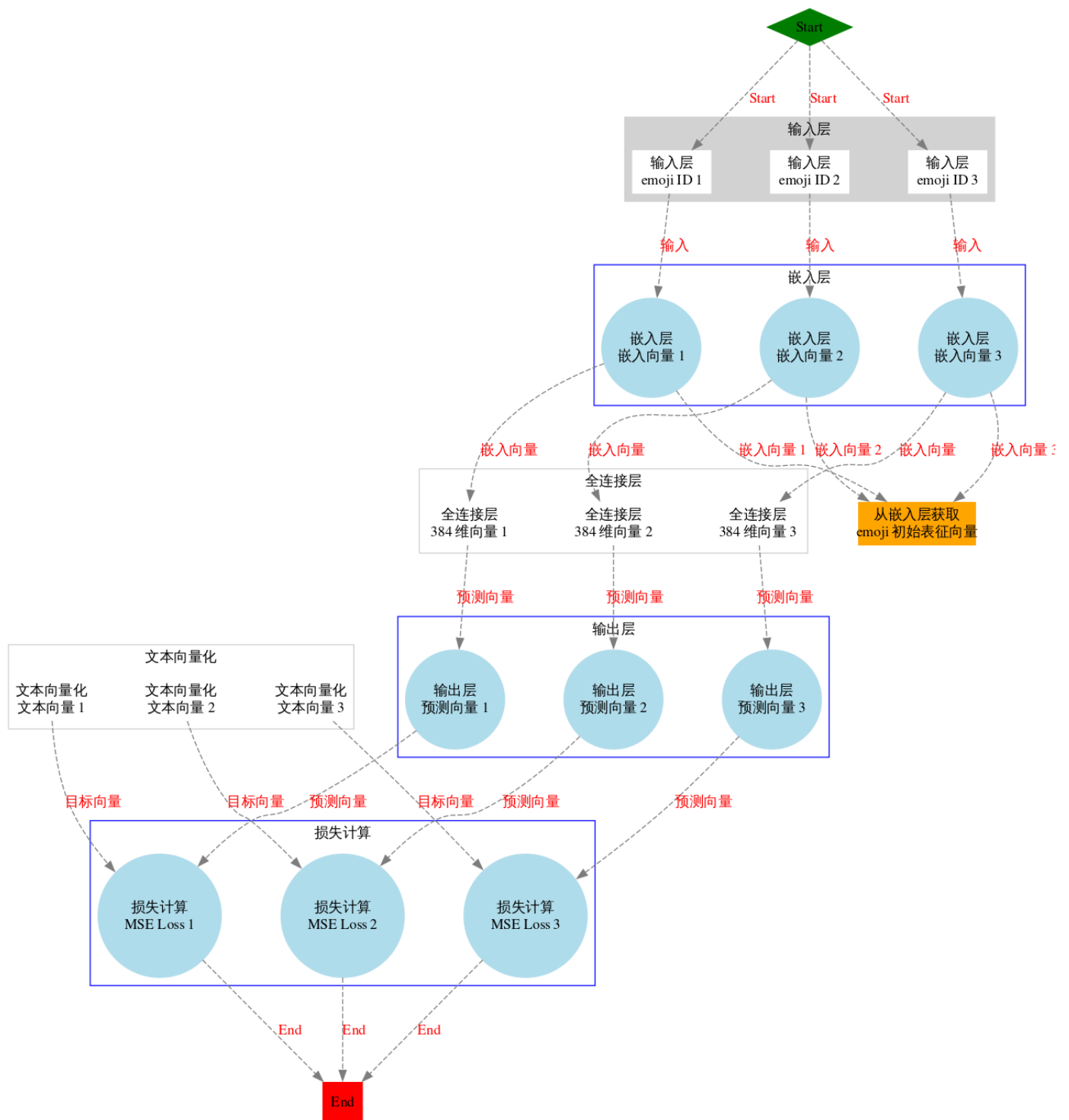
该神经网络模型主要有四层结构：除了全连接层和输出层基本可以看做同一层以外，嵌入层需要能够将输出层的emoji ID 映射到低维的稠密向量空间，保证每个emoji可以学习一个对应的表征向量。由于需要将emoji ID的表征向量及其对应的中文组合词意表征向量进行匹配，我们还需要对前者在全连接层进行升维。

1.1.3 总结

根据以上思路，我们可以把流程梳理如下：

1. 在数据收集与预处理部分：根据爬虫代码从 HTML 文件中提取 emoji 相关信息，包括 emoji 字符、标题、Unicode 代码和中文组合词。这些信息存储在 CSV 文件中，方便后续处理。
2. 接着进行清洗数据，去除无效信息，确保数据的准确性和一致性。
3. 为模型的输入层构建一个词汇表，从所有 emoji 中构建一个唯一的词汇表，并为每个 emoji 分配一个唯一的 ID。
4. 创建训练集，每个训练对包含一个emoji ID和对应的中文组合词意。
5. 模型设计上在嵌入层使用 `nn.Embedding` 将 emoji 的 ID 映射到低维的稠密向量空间；全连接层则将嵌入向量转换为与文本向量相同的维度（384 维）
6. 使用预训练的文本向量模型将中文组合词转换为文本表征向量。
7. 最后定义损失计算与反向传播，进行训练即可。
8. 训练完成后，从嵌入层提取所有 emoji 的嵌入向量，并保存到文件中。这些向量是模型的输出，可以直接用于后续的应用。

可以得到emoji表征向量的训练模型结构图如下：



1.2 特征工程

垃圾文本有很多明显且有力的特征可以用于鉴别：比如；

- 假设有一个敏感词词库，如果文本中含有敏感词，则该文本大概率是垃圾文本
- 如果文本中有长串的数字/字母串（电话号码/网址/微信），则该文本大概率是垃圾文本
- 某些 Unicode 字符（如 零宽字符 U+200B）不占视觉宽度，可能被滥用于逃避检测。
"正⁰常⁰文⁰本⁰"（⁰ = U+200B，视觉不可见）如果这个文本大量使用零宽字符，这个文本大概率是垃圾文本

在本实验中，我们实现长串数字/字母串的识别，并作为一个特征接入分类器。

1.3 中文敏感文本检测系统全流程设计思路

垃圾文本检测是一种自然语言处理领域的任务，主要用于识别和过滤文本中的垃圾或其他不需要的信息。基于数据科学技术开展垃圾文本检测模型的研究或工程应用，是通过应用已有的垃圾文本检测模型或研究改进垃圾文本检测模型，实现在文本数据中识别出符合“异常文本”“垃圾文本”“涉密文本”等特征的数据，形成整体的文本检测结果。

一个基础的流程图可以如下显示：

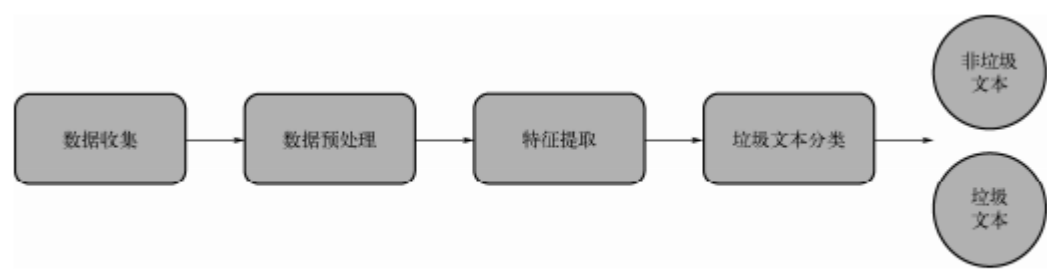


图 4-2 文本检测处理流程图

1.3.1 数据收集和预处理

- 在数据收集阶段，由于文本数据往往包含着各种噪声和无用信息，对模型的分析 and 应用产生干扰，因此对文本数据进行清洗是非常重要的步骤；
- 中文文本中不存在数值缺失或异常，但可能存在的情况包括空白文本、错误编码或意外字符等，我们也需要对上述缺失值和异常值进行处理；我们还需要移除停用词，即文本中频繁出现但通常没有实际意义的词语(如“的”“和”)以提高模型效率。
- 正样本（垃圾文本）在数据总量上可能会比较少，因此我们需要使用 randomsample 保证数据集平衡

1.3.2 汉字编码和特征提取

接着，我们需要定义汉字的特征来进行提取和分类，这步可以从汉字独有的字音和字形出发。

汉字的字形通常包括以下三个要素：汉字结构、汉字形状和汉字笔画数。其中汉字笔画数很好理解，汉字结构也即主要的7种类别：上下结构、左右结构、独体字等，我们后序用一个序号来进行汉字结构的区分。但是汉字形状应该如何表示呢，指导书告诉我们可以用四角编码来描述。四角编码是一种用于描述汉字形状和笔画的编码系统，它将汉字

的形态按照笔画的先后顺序划分为若干部首，并为每个部首分配一个特定的四角编号，用最多 5 个阿拉伯数字来对汉字进行归类。

以汉字“辉”为例，它的 5 位四角编码可能表示为“97256”：

- 1. 左上角的笔画形状编号
- 2. 右上角的笔画形状编号
- 3. 左下角的笔画形状编号
- 4. 右下角的笔画形状编号
- 5. 附角的笔画形状编号



图 4-5 “辉”字的四角编码示意图

汉字字音可以直接用四个要素概括：声母、韵母、补码以及声调。前两者很好理解，声调也可以用数字1-4表示，主要是补码，这个一般作用于于声母和韵母之间还有一个辅音的情况（如 guang中的u就是补码）。

将上述两种编码组合起来，就可以得到汉字的完整编码。

接着我们需要计算两个汉字之间的相似度，由此类比用户对变体汉字的联想过程。

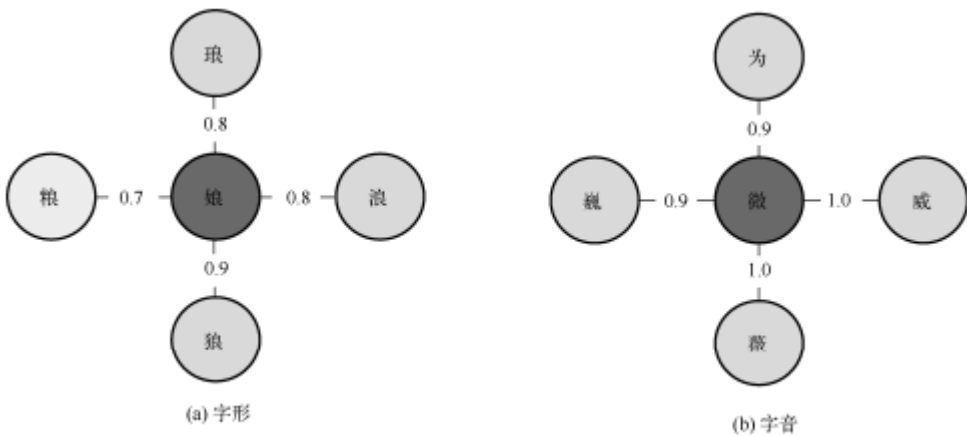


图 4-6 具有相似字形和字音的字符示例

在处理字形相似性和字音相似性时，选择 max 聚合，而不是平均聚合或加权聚合。这是因为在实际应用中，大部分变体都只简单地考虑字形或字音的其中一个角度，并不会采取两者的结合。换句话说，变体不需要在字形和字音上都与正确的字符完全相似，只要在两者中的任一方面相似即可。

1.3.3 分类器选取

二分类问题可以使用逻辑回归、SVM、MLP等模型

1.3.4 总结

结合emoji的特征向量和汉字的相似性网络构造，我们可以得到扩展emoji检测后的垃圾文本检测算法设计流程图如下：

数据准备阶段

数据爬取

BeautifulSoup解析HTML

提取emoji字符和对应文本

保存为emoji_list.csv

数据预处理

读取emoji_list.csv

解析emoji_chars字段

构建emoji词汇表

模型训练

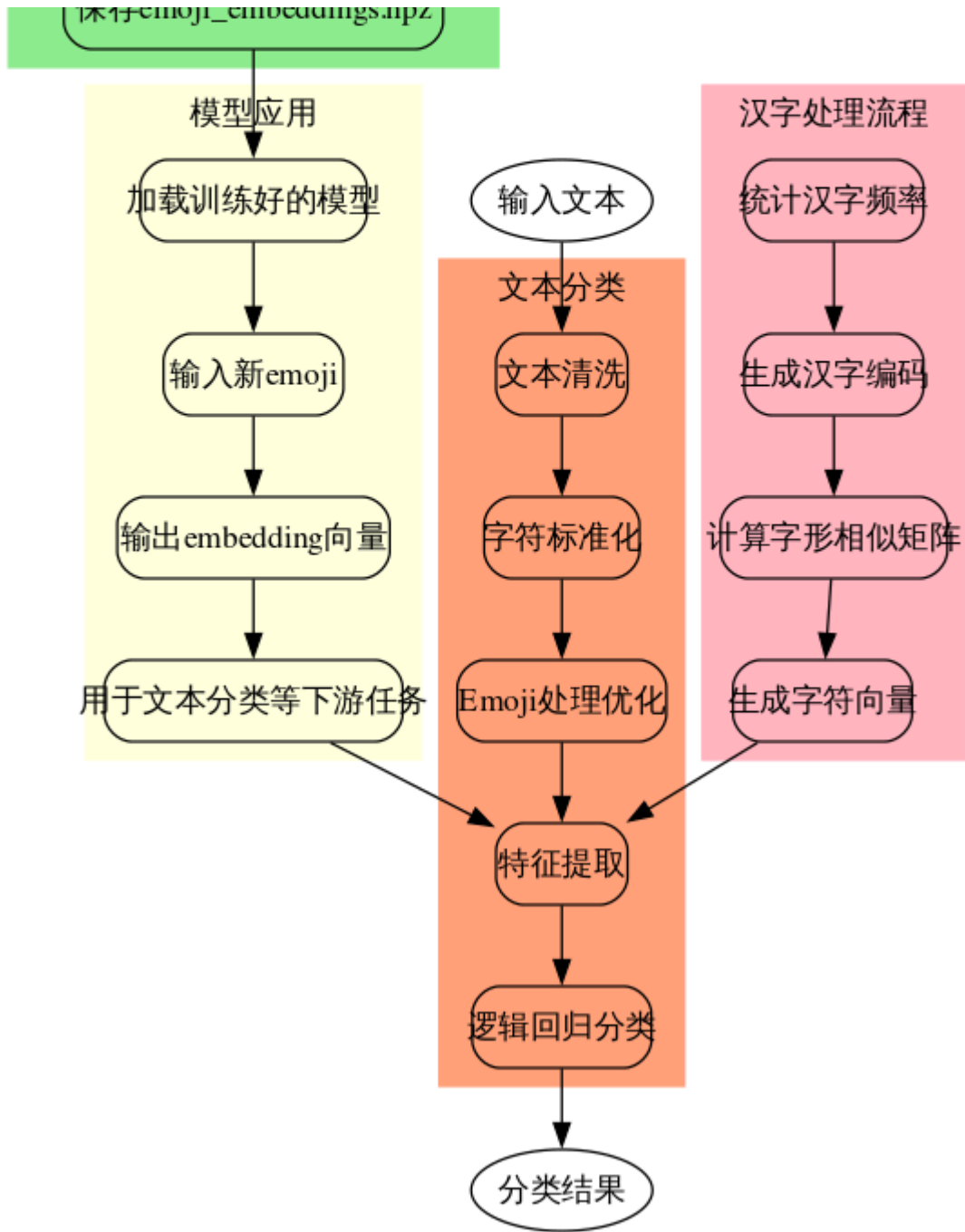
定义Emoji2Vec模型

准备训练数据对

使用SentenceTransformer编码文本

训练模型优化embedding

保存emoji_embeddings.npz



2 算法任务

构建中文敏感文本检测系统，融合汉字字音字形相似度embedding和emoji语义向量，实现识别变形文本和符号化垃圾内容。

1. 通过爬虫等方式获得结构化的含emoji的文本数据集和emoji-中文文本释义对
2. 使用不同的编码模型，比较其效果

做法	原版本	版本1	版本2
字向量的表达	word2vec	word2vec 基于字符共现频率，捕获字形/上下文关联	sentence transformer 基于文本描述语义
emoji向量的表达	无	使用sentence transformer 进行训练，得到中间层输出	使用sentence transformer 进行训练，得到中间层输出

做法	原版本	版本1	版本2
		作为向量	作为向量
<ul style="list-style-type: none"> • Emoji2Vec 的中间层输出是经过汉字释义（如“笑脸”）监督微调 • 由于版本1的emoji向量和字向量语义空间不统一，向量空间不对齐 <ul style="list-style-type: none"> ◦ 若两者量级差异大（如 emoji 向量范数远大于汉字向量），会导致：相似度计算（如余弦相似度）被 emoji 向量主导。 ◦ 模型注意力机制（如你的 generate_sentence_vectors 中的 alpha 矩阵）出现偏差。 ◦ 因此我们在合并两个表之前要进行归一化：将所有向量归一化为单位长度 			

3 实验结果

1.课本原始结果：

混淆矩阵：

```
[[2167 331]
 [ 161 5345]]
```

分类报告：

	precision	recall	f1-score	support
0	0.93	0.87	0.90	2498
1	0.94	0.97	0.96	5506
accuracy			0.94	8004
macro avg	0.94	0.92	0.93	8004
weighted avg	0.94	0.94	0.94	8004

2.字向量编码采用SentenceTransformer且采用SVM模型分类结果：

16282

16282

混淆矩阵：

```
[[2372 208]
 [ 214 5347]]
```

分类报告：

	precision	recall	f1-score	support
0	0.92	0.92	0.92	2580
1	0.96	0.96	0.96	5561
accuracy			0.95	8141
macro avg	0.94	0.94	0.94	8141
weighted avg	0.95	0.95	0.95	8141

3.字向量编码采用且采用SVM模型分类结果

```
16282
16282
混淆矩阵：
[[2265  315]
 [ 401 5160]]
分类报告：
              precision    recall  f1-score   support

      0       0.85        0.88        0.86       2580
      1       0.94        0.93        0.94       5561

 accuracy          0.91
 macro avg          0.90
weighted avg          0.91
```

```
PS D:\邓博高\大数据原理\垃圾文本分类\sensitive-text-detection-master>
```

4 结果分析

4.1 整体性能对比

第一版（Word2Vec向量 + SVM）：

- 准确率（accuracy）：0.95
- 加权平均 F1-score：0.95
- 宏平均 F1-score：0.94

总结：整体性能最优，平衡性好，无明显短板。

第二版（字符级SentenceTransformer向量 + SVM）：

- 准确率：0.91
- 加权平均 F1-score：0.91
- 宏平均 F1-score：0.90

总结：性能明显低于第一版，是所有版本中最弱的，尤其在少数类（类别0）上表现较差。

原始版本（Word2Vec向量 + LR）：

- 准确率：0.94
- 加权平均 F1-score：0.94
- 宏平均 F1-score：0.93

总结：性能接近第一版，但存在类别不平衡问题（类别1召回高，类别0召回低）。

关键点：

- 第一版 vs 第二版：Word2Vec词级向量（第一版）显著优于字符级 SentenceTransformer向量（第二版），准确率相差4个百分点（0.95 vs 0.91），加权F1相差4个百分点（0.95 vs 0.91）。这凸显了向量生成方法对模型性能的关键影响。
- 第一版 vs 原始版：第一版（SVM）略优于原始版（LR），尤其在类别0的召回率上（0.92 vs 0.87），说明SVM结合Word2Vec向量更鲁棒。

4.2 类别性能分析

4.2.1 类别0（少数类，非垃圾文本）

支持样本数：第一版和第二版均为2580，原始版为2498（约占总样本的30-31%）。

性能对比：

版本	Precision	Recall	F1-score
第一版 (SVM + W2V)	0.92	0.92	0.92
第二版 (SVM + Char)	0.85	0.88	0.86
原始版 (LR + W2V)	0.93	0.87	0.90

分析：

- 第一版最佳：所有指标均最高（F1=0.92），表明Word2Vec向量能更好地区分非垃圾文本。
- 第二版最弱：Precision最低（0.85），说明字符级向量导致大量“误报”（将非垃圾预测为垃圾）。Recall略高（0.88）可能是因为字符级特征捕捉了部分通用模式，但噪声较大。
- 原始版问题：Recall最低（0.87），表明LR模型漏检较多非垃圾文本（假阴性高）。

4.2.2 类别1（多数类，垃圾文本）

支持样本数：第一版和第二版均为5561，原始版为5506（约占总样本的69-70%）。

性能对比：

版本	Precision	Recall	F1-score
第一版 (SVM + W2V)	0.96	0.96	0.96

版本	Precision	Recall	F1-score
第二版 (SVM + Char)	0.94	0.93	0.94
原始版 (LR + W2V)	0.94	0.97	0.96

分析：

- **第一版最优：**所有指标最高（F1=0.96），Word2Vec向量精准捕捉垃圾文本特征。
- **第二版稍弱：**所有指标略低于第一版，字符级向量对垃圾文本的区分能力不足。
- **原始版特点：**Recall极高（0.97），但Precision较低（0.94），表明LR倾向于将更多样本预测为垃圾（假阳性高），可能因类别不平衡导致。

4.2.3 类别不平衡问题

所有版本中，类别1（垃圾）样本数约为类别0（非垃圾）的2.2倍（5561 vs 2580），存在明显不平衡。

模型表现：

- **第一版：**对少数类（类别0）的F1高达0.92，表明Word2Vec + SVM有效缓解了不平衡问题。
- **第二版：**对类别0的F1仅0.86，字符级向量放大了不平衡的影响。
- **原始版：**对类别0的Recall仅0.87，LR模型因不平衡而偏向多数类（类别1 Recall=0.97）。

加权 vs 宏平均：

- 加权平均（weighted avg）更反映实际性能（因考虑样本权重）
 - 宏平均（macro avg）更强调类别平等
- 第一版宏平均F1=0.94，加权F1=0.95，说明其平衡性极佳；第二版宏平均F1=0.90，加权F1=0.91，表明少数类拖累整体。

4.3 字向量生成方法的影响

Word2Vec向量（第一版）的优势：

- 上下文感知：Word2Vec基于窗口内上下文训练char向量，对字符级任务更敏感；skipgram风格使得即使是对字符进行编码也能把握上下文信息。

字符级SentenceTransformer向量（第二版）的劣势：

- 预训练模型局限：paraphrase-multilingual-MiniLM-L12-v2是为句子级语义设计的，将其用于单个字符时：
 - 字符缺乏独立语义（尤其中文），导致向量包含噪声。
 - 预训练模型未针对字符级任务优化，泛化能力差。

4.4 分类器选择的影响

SVM vs LR:

- 第一版（SVM）：高维向量空间中，SVM擅长找到最优分离超平面，适合Word2Vec生成的稠密向量（d=100）。
- 原始版（LR）：LR更简单高效，但对不平衡数据敏感（类别0 Recall仅0.87），且可能受特征质量影响更大。

分类器与向量交互：

- Word2Vec向量 + SVM 组合最佳（F1=0.95）
- 字符级向量 + SVM 表现差，说明向量质量是瓶颈，分类器无法弥补

5 算法代码及代码说明

5.0 代码包框架说明

```
Sensitive Text Detection:.  
| data_crawl.py //爬取emoji组合数据用于训练  
| emoji_embedding.py //训练emoji表征向量代码  
| emoji_embedding2.py //emoji训练测试代码  
| emoji_skipgram.model //训练得到的emoji分类模型  
| main.py //主流程模拟代码  
| README.md  
| requirements.txt //参数设置  
| ssc_similarity.py //计算相似性的工具函数模块  
| utils.py //字音字形编码和工具函数模块  
|  
├─.idea  
|  
├─data //数据存放  
|   char_vectors.json //字符向量缓存文件  
|   chinese_unicode_table.txt //汉字笔画对照文件  
|   dataset.txt //原始数据集  
|   emoji_embeddings.npz //emoji表征向量缓存文件  
|   emoji_list.csv //清洗后的emoji数据集  
|   emoji.txt //从zh.xml上复制下的所有emoji的中文释义
```

```
hanzi.txt //汉字编码
hanzijiegou_2w.txt //汉字结构对照文件
hit_stopwords.txt //停用词处理文件
sentence_vectors.npy //句子向量缓存文件
similarity_matrix.pkl //计算得到的字符相似性网络

image //报告附录图

four_corner_method //四角编码相关
```

5.1 utils.py 字音字形编码和工具函数模块

主要实现以下功能：

- 提供汉字的多维度编码系统（字音+字形）
- 构建字符相似性网络矩阵
- 支持动态更新新字符的编码和相似性
- 为后续文本向量化和分类任务提供基础支持

5.1.1 字音字形编码类

在 `utils.py` 中我们实现了 `ChineseCharacterCoder` 类，其主要功能是从汉字的字音字形两个方面来定义多维度的特征编码，并且提供支持动态扩展新字符的接口。

其中有以下三个字典成员：

```
self.structure_dict
self.strokes_dict
self.chinese_char_map
```

其分别存储以下数据：

- `self.structure_dict` 汉字的结构映射：加载 `hanzijiegou_2w.txt` 文件中的汉字结构（如“中”字是“独体字”结构的映射）
- `self.strokes_dict` 笔画数编码：其中1-9笔画直接使用数字1-9表示，10-35笔画使用大写字母A-Z表示，36-51笔画使用小写字母a-p表示
- `self.chinese_char_map` 汉字到笔画数映射字典：加载 `chinese_unicode_table.txt` 文件，通过 `strokes_dict` 字典将笔画数字符串转换为单字符编码，处理后的数组第一个是文字，第7个是笔画数量

该类中的成员函数如下：

```
def split_pinyin(self, chinese_character):
def generate_pronunciation_code(self, hanzi):
def generate_glyph_code(self, hanzi):
def generate_character_code(self, hanzi):
```

```
def split_pinyin(self, chinese_character):
    # 将汉字转换为拼音（带声调）
    pinyin_result = pinyin(chinese_character, style=Style.TONE3, heteronym=True)

    # 多音字的话，选择第一个拼音
    if pinyin_result:
        py = pinyin_result[0][0]

        initials = "" # 声母
        finals = "" # 韵母
        codas = "" # 补码
        tone = "" # 声调

        # 声母列表
        initials_list = ["b", "p", "m", "f", "d", "t", "n", "l", "g", "k", "h",
                        "j", "q", "x", "zh", "ch", "sh", "r", "z", "c", "s", "y", "w"]

        # 韵母列表
        finals_list = ["a", "o", "e", "i", "u", "v", "ai", "ei", "ui", "ao", "ou",
                      "iu", "ie", "ve", "er", "an", "en", "in", "un", "vn", "ang", "eng", "ing", "ong"]

        # 获取声调
        if py[-1].isdigit():
            tone = py[-1]
            py = py[:-1]

        # 获取声母
        for initial in initials_list:
            if py.startswith(initial):
                initials = initial
                py = py[len(initial):]
                break

        # 获取韵母
        for final in finals_list:
            if py.endswith(final):
                finals = final
                py = py[:-len(final)]
                break

        # 获取补码
        codas = py

        return initials, finals, codas, tone
```



```
return None
```

函数 `def split_pinyin(self, chinese_character)` 主要将汉字转换为拼音，并拆分为声母/韵母/补码/声调，以汉字"中"拆分为四元组 ("zh", "ong", "", "1") 为例，说明其实现逻辑和功能如下：

1. 使用 pypinyin 将汉字转换为拼音并加上声调（如"中"字转换为"zhong1"），若为多音字则选择第一个拼音（故不选择“zhong4”）
2. 分离声调数字（如"1"）
3. 分别从预定义声母列表匹配起始部分（如"zh"）和从预定义韵母列表匹配结尾部分（如"ong"）
4. 剩余部分作为补码，如无则为空
5. 返回四元组 `initials, finals, codas, tone` 分别对应（声母，韵母，补码，声调）

```
def generate_pronunciation_code(self, hanzi):
    initial, final, coda, tone = self.split_pinyin(hanzi)

    # 轻声字，例如'了'
    if tone == '':
        tone = '0'

    # 声母映射
    initials_mapping = {'b': '1', 'p': '2', 'm': '3', 'f': '4', 'd': '5', 't': '6',
                        'n': '7', 'l': '8',
                        'g': '9', 'k': 'a', 'h': 'b', 'j': 'c', 'q': 'd', 'x': 'e',
                        'zh': 'f', 'ch': 'g',
                        'sh': 'h', 'r': 'i', 'z': 'j', 'c': 'k', 's': 'l', 'y':
                        'm', 'w': 'n'}

    # 韵母映射
    finals_mapping = {'a': '1', 'o': '2', 'e': '3', 'i': '4', 'u': '5', 'v': '6',
                      'ai': '7', 'ei': '8',
                      'ui': '9', 'ao': 'a', 'ou': 'b', 'iu': 'c', 'ie': 'd',
                      've': 'e', 'er': 'f',
                      'an': 'g', 'en': 'h', 'in': 'i', 'un': 'j', 'vn': 'k',
                      'ang': 'l', 'eng': 'm',
                      'ing': 'n', 'ong': 'o'}

    # 补码映射
    coda_mapping = {'': '0', 'u': '1', 'i': '1'}

    # 获取映射值
    initial_code = initials_mapping.get(initial, '0')
    final_code = finals_mapping.get(final, '0')
    coda_code = coda_mapping.get(coda, '0')

    # 组合生成四位数的字音编码
```

```
pronunciation_code = initial_code + final_code + coda_code + tone
```

```
return pronunciation_code
```

函数 `def generate_pronunciation_code(self, hanzi)` 主要生成4位字音编码，其中：声母(1)+韵母(1)+补码(1)+声调(1)，以汉字"中"生成字音编码为"fo01"为例，说明其实现逻辑和功能如下：

1. 通过 `split_pinyin` 获取声母/韵母/补码/声调
2. 异常处理：对轻声字(如"了")设置默认声调"0"
3. 编码转换：
 - 声母→1位编码（如"zh"→"f"）
 - 韵母→1位编码（如"ong"→"o"）
 - 补码→1位编码（如"i"→"0"）
 - 声调→1位数字（如"1"保持不变）
4. 组合输出：拼接成4位固定长度字音编码 `pronunciation_code`

```
def generate_glyph_code(self, hanzi):
    # 获取汉字的结构
    structure_code = self.structure_dict[hanzi]

    # 获取汉字的四角编码
    fcc = FourCornerMethod().query(hanzi)

    # 获取汉字的笔画数
    stroke = self.chinese_char_map[hanzi]

    # 组合生成的字形编码
    glyph_code = structure_code + fcc + stroke

    return glyph_code
```

函数 `def generate_pronunciation_code(self, hanzi)` 主要生成多位字形编码，其中：结构(1)+四角编码+笔画(1)，以汉字"中"生成字形编码为"6500064"为例，说明其实现逻辑和功能如下：

1. 获取结构编码：从 `self.structure_dict` 中获取（如"中"是独体字，编码为"6"）
2. 使用 `FourCornerMethod` 类查询四角编码（如"50006"）
3. 获取笔画编码：从 `self.chinese_char_map` 获取笔画数（如"4笔"编码为"4"）
4. 将上述结构编码、四角编码、笔画编码组合得到字形编码 `glyph_code`（如"6500064"）

```
def generate_character_code(self, hanzi):
    return self.generate_pronunciation_code(hanzi) +
self.generate_glyph_code(hanzi)
```

函数 `def generate_character_code(self, hanzi)` 将上述两个函数得到的4位固定长度字音编码 `pronunciation_code` 和字形编码 `glyph_code` 组合得到完整编码（如汉字"中"的完整编码为 "fo016500064"）

5.1.2 其余工具函数

在 `utils.py` 中我们定义了其他的工具函数，结合字音字形编码类，共同构成了一个完整的汉字处理工具集，支持从原始文本到汉字编码、相似度计算的完整流程，并考虑了性能优化和动态扩展能力。

其他工具函数如下：

```
def count_chinese_characters(content, output_file_path):
def load_chinese_characters(filename):
def compute_sim_mat(chinese_characters, chinese_characters_code):
def load_sim_mat(filename):
def update_sim_mat(new_characters, chinese_characters_code, sim_mat):
```

```
# 统计数据集中的所有汉字以及对应的出现次数，并对其进行编码
def count_chinese_characters(content, output_file_path):
    chinese_characters = []
    chinese_characters_count = {}
    chinese_characters_code = {}

    if os.path.exists(output_file_path):
        print(f'File exists: {output_file_path}, loading...')
        return load_chinese_characters(output_file_path)

    for line in tqdm(content, desc='Counting characters', unit='line'):
        for char in line:
            if '\u4e00' <= char <= '\u9fff': # 判断是否为汉字
                chinese_characters_count[char] = chinese_characters_count.get(char,
0) + 1

    with open(output_file_path, 'w', encoding='utf-8') as output_file:
        for char, count in tqdm(chinese_characters_count.items(), desc='Computing
Character Code', unit='char'):
            character_code = ChineseCharacterCoder().generate_character_code(char)
            chinese_characters_code[char] = character_code
            output_file.write(f'{char}\t{character_code}\t{count}\n')
            chinese_characters.append(char)
```

```
print(f'Results saved to {output_file_path}')

return chinese_characters, chinese_characters_count, chinese_characters_code
```

函数 `count_chinese_characters(content, output_file_path)` 统计待处理的文本内容列表 `content` 中的汉字及其出现频率，并生成编码：

1. 遍历文本内容，统计每个汉字出现次数
2. 使用 `ChineseCharacterCoder` 生成每个汉字的编码
3. 将结果写入指定输出文件 `output_file_path`
4. 返回三个字典：`chinese_characters` 汉字列表、`chinese_characters_count`: 汉字出现次数统计、`chinese_characters_code`: 汉字编码字典

```
# 加载已有的汉字库
def load_chinese_characters(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        line = f.readlines()
        chinese_characters = []
        chinese_characters_count = {}
        chinese_characters_code = {}

        for row in line:
            char, code, count = row.strip().split('\t')
            chinese_characters.append(char)
            chinese_characters_code[char] = code
            chinese_characters_count[char] = count

    return chinese_characters, chinese_characters_count, chinese_characters_code
```

函数 `load_chinese_characters(filename)` 从文件 `filename` 中加载已统计的汉字数据，同样返回三个字典：`chinese_characters` 汉字列表、`chinese_characters_count`: 汉字出现次数统计、`chinese_characters_code`: 汉字编码字典。

```
# 构建字符相似性网络（用矩阵形式表示）
def compute_sim_mat(chinese_characters, chinese_characters_code):
    # 将结果保存到pkl文件
    output_file = 'data/similarity_matrix.pkl'

    if os.path.exists(output_file):
        print(f'File exists: {output_file}, loading...')
        return load_sim_mat(output_file)

    sim_mat = [[0] * len(chinese_characters) for _ in
range(len(chinese_characters))]
    for i in tqdm(range(len(chinese_characters)), desc='Constructing Similarity
```

```

Matrix', unit='i'):
    for j in range(i, len(chinese_characters)):
        similarity =
computeSSCSimilarity(chinese_characters_code[chinese_characters[i]],
chinese_characters_code[chinese_characters[j]])
        sim_mat[i][j] = similarity
        sim_mat[j][i] = similarity

    with open(output_file, 'wb') as f:
        pickle.dump(sim_mat, f)

    return sim_mat

```

函数 `compute_sim_mat(chinese_characters, chinese_characters_code)` 从 `chinese_characters` 汉字列表和 `chinese_characters_code`: 汉字编码字典中计算汉字相似度矩阵:

1. 使用 `ssc_similarity.py` 中的 `computeSSCSimilarity` 计算每对汉字之间的相似度
2. 将结果保存为 pkl 文件以便复用
3. 如果已有缓存文件则直接加载

```

# 从pkl文件中加载相似性矩阵
def load_sim_mat(filename):
    with open(filename, 'rb') as f:
        sim_mat = pickle.load(f)

    return sim_mat

```

函数 `load_sim_mat(filename)` 实现从 pkl 文件加载相似度矩阵 `sim_mat`

```

# 更新相似性矩阵
def update_sim_mat(new_characters, chinese_characters_code, sim_mat):
    for char in new_characters:
        # 计算新汉字与现有汉字之间的相似性
        new_code = chinese_characters_code[char]
        similarities = [computeSSCSimilarity(new_code, code) for code in
chinese_characters_code.values()]

        # 更新相似性矩阵
        new_row = np.array(similarities)
        sim_mat = np.vstack([sim_mat, new_row])
        sim_mat = np.hstack([sim_mat, new_row.reshape(-1, 1)])

    return sim_mat

```

函数 `update_sim_mat(new_characters, chinese_characters_code, sim_mat)` 可以动态更新相似度矩阵，并添加新汉字 `new_characters`：

1. 使用 `ssc_similarity.py` 中的 `computeSSCSimilarity` 函数计算新汉字与现有汉字的相似度 `similarities`
2. 更新相似性矩阵并返回 `sim_mat`，主要扩展相似度矩阵的维数并填充新增的行和列。

5.2 ssc_similarity.py 相似度计算模块

`ssc_similarity.py` 主要实现了汉字相似度计算的相关函数，为敏感文本检测系统提供基础相似度计算能力，其中的三个函数：

`computeSoundCodeSimilarity(soundCode1, soundCode2)` 计算字音编码相似性，
`computeShapeCodeSimilarity(shapeCode1, shapeCode2)` 计算字形编码相似性，
`computeSSCSimilarity(ssc1, ssc2)` 组合字音字形相似性，计算两个字符的相似性。

因为计算过程本质上是对字音字形的编码进行不同权重的加权计算，故对于一些参数的设定不再详细解释，仅展示计算字符相似性的函数的输入和返回格式：

```
# 计算字符相似性的函数
def computeSSCSimilarity(ssc1, ssc2):
    # 组合字音和字形的相似性，根据权重计算
    shapeSimi=computeShapeCodeSimilarity(ssc1[4:], ssc2[4:])
    soundSimi=computeSoundCodeSimilarity(ssc1[:4], ssc2[:4])
    return max(soundSimi, shapeSimi)
```

5.3 emoji-embedding向量训练模块

这部分模块主要涉及以下两部分的代码：

- `data_crawl.py`：负责从 HTML 文件中提取 emoji 相关信息，并将其保存为 CSV 文件，为后续的emoji向量训练提供数据基础。
- `emoji_embedding.py`：负责加载 CSV 文件中的数据，构建词汇表，定义和训练神经网络模型，生成 emoji 的词向量，并将结果保存为 NPZ 文件，为后续的应用做准备。

5.3.1 data_crawl.py emoji训练集爬取模块

本模块主要实现从本地HTML文件中爬取emoji组合数据，提取信息并保存为CSV文件提供给 `source_data_emoji` 训练。

主要核心处理模块如下：

```
for file_path in file_list:
    with open(file_path, 'r', encoding=' -8') as f:
        html_content = f.read()

    # 使用 BeautifulSoup 解析 HTML
    soup = BeautifulSoup(html_content, 'html.parser')

    for li in soup.find_all("li", class_="row"):
        emoji_div = li.find("div", class_="emoji")
        if not emoji_div:
            continue

        emojis = []
        titles = []
        unicode_codes = []

        for a in emoji_div.find_all("a", class_="emoji_link"):
            emoji_char = a.get_text(strip=True)
            title = a.get("title", "")
            href = a.get("href", "")
            unicode_part = href.split('/')[ -1] if href else ''
            emojis.append(emoji_char)
            titles.append(title)
            unicode_codes.append(unicode_part)

        # 提取中文组合词
        name_span = li.find("span", class_="col small text-center")
        combined_name = name_span.get_text(strip=True) if name_span else ''
        combined_emoji = ''.join(emojis)

        results.append({
            "emoji_chars": emojis,
            "titles": titles,
            "unicode_codes": unicode_codes,
            "combined_name": combined_name,
            "combined_emoji": combined_emoji
        })
```

1. 文件扫描：自动识别 `source_data_emoji` 目录下所有 `combine_list*.devtools` 文件，通过 `os.listdir` 和 `os.path.join`，筛选出所有符合条件的文件，并存储在 `file_list` 中
2. 使用BeautifulSoup解析HTML，查找所有li元素，每个li对应一个emoji组合项
3. 提取单个emoji信息，包括以下内容：单个emoji字符/对应的标题说明/url链接转为Unicode编码/中文组合词即emoji的释义

4. 数据整合：将上一步中每个emoji组合的信息存入对应的字典：单个emoji字符列表/对应的标题列表/Unicode编码列表/组合中文释义列表/组合后的完整emoji字符串
5. 将上述所有字典组合的DataFrame 保存为 CSV 文件，以便于进行下一步训练

5.3.2 emoji_embedding.py emoji向量训练模块

主要通过PyTorch实现Skip-Gram模型构造emoji的词向量（Word2Vec），通过得到emoji表征向量和对应的文本表征向量进行监督学习，得到训练模型以便于通过中心emoji预测周围上下文emoji。

```
# 加载数据
# 从TXT文件处理数据（Unicode注释）
if from_txt:
    with open(file_path, 'r', encoding='utf-8') as file:
        content = file.readlines()

    emoji_to_meanings = defaultdict(set)
    pattern = r'<annotation cp="([^\"]+)"[^>]*>([^\<]+)</annotation>'

    for line in content:
        match = re.search(pattern, line)
        if match:
            emoji = match.group(1)
            meanings = match.group(2).split('|')
            # 过滤中文字符
            for meaning in meanings:
                if chinese_pattern.fullmatch(cleaned):
                    emoji_to_meanings[emoji].add(meaning.strip())

    # 构建DataFrame
    data = []
    for emoji, meanings in emoji_to_meanings.items():
        data.append({'emoji': emoji, 'meanings': list(meanings)})
    df = pd.DataFrame(data)

# 从CSV文件处理数据
else:
    df = pd.read_csv(file_path)
    df['emoji_chars'] = df['emoji_chars'].apply(ast.literal_eval)
    df['titles'] = df['titles'].apply(ast.literal_eval)

    data = []
    for _, row in df.iterrows():
        for emoji in row['emoji_chars']:
            # 添加标题释义
            data.append({'emoji': emoji, 'meanings': row['titles']})
            # 添加组合名称释义
```



```
data.append({'emoji': emoji, 'meanings': [row['combined_name']]})
df = pd.DataFrame(data)
```

TXT模式:

解析XML格式的Unicode注释文件
使用正则提取emoji和释义
过滤非中文字符
构建emoji→[释义列表]的映射

CSV模式:

解析包含emoji组合的CSV文件
将字符串列转换为列表
为每个emoji添加两类释义:
标题列表 (titles)
组合名称 (combined_name)

解析字段

```
df['emoji_chars'] = df['emoji_chars'].apply(ast.literal_eval)
df['titles'] = df['titles'].apply(ast.literal_eval)
df['unicode_codes'] = df['unicode_codes'].apply(ast.literal_eval)
```

构建emoji词汇表

```
all_emojis = sorted(set(emoji for sublist in df['emoji_chars'] for emoji in
sublist))
emoji_to_id = {e: i for i, e in enumerate(all_emojis)}
id_to_emoji = {i: e for e, i in emoji_to_id.items()}
```

在加载了CSV数据后，首先需要对数据进行一定形式的提取，也即创建一个词汇表。

使用 `ast.literal_eval` 将字符串形式的列表字段（即 `data_crawl.py` 爬取并整理的：单个emoji字符列表/对应的标题列表/Unicode编码列表/组合中文释义列表/组合后的完整emoji字符串）解析为实际的列表。

接着从所有 emoji 中构建一个唯一的词汇表，并为每个 emoji 分配一个唯一的 ID。

定义模型

```
class Emoji2Vec(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, 384)

    def forward(self, x):
        x = self.embedding(x)
        return self.linear(x)
```

初始化模型

```
model = Emoji2Vec(len(emoji_to_id), embedding_dim)
optimizer = torch.optim.Adam(model.parameters(), lr)
loss_fn = nn.MSELoss()
```

文本向量模型

```

text_model = SentenceTransformer("paraphrase-multilingual-MiniLM-L12-v2")

# 训练数据构造
pairs = []
for _, row in df.iterrows():
    for emoji in row['emoji_chars']:
        pairs.append((emoji, row['combined_name'])) # emoji和对应的文本标签，注意这里是skip_gram风格，同一个combined_name对应不同的emoji

# 向量化文本标签
text_lookup = {}
for _, text in df['combined_name'].items():
    if text not in text_lookup:
        text_lookup[text] = text_model.encode(text)

```

接着定义一个简单的神经网络模型 **Emoji2Vec**，其主要功能是从emoji的ID得到其对应的emoji表征向量，除了输入层、输出层外还包含一个嵌入层 (**nn.Embedding**) 和一个全连接层 (**nn.Linear**):

- 输入层：输入 emoji 的 ID
- 嵌入层：将 emoji 的 ID 映射到低维的稠密向量空间，使用PyTorch 的 **nn.Embedding** 层模型实现，输出一个更低维的嵌入向量来表示对应的emoji
- 全连接层：将嵌入向量转换为与文本向量相同的维度（384 维），以便与文本向量进行比较
- 输出层：输出emoji的表征向量

构造训练数据 (**emoji, text**)，每个训练对包含一个 emoji 的ID **emoji**和其对应的中文组词 **text**，注意skip_gram风格下同一个中文组词可以对应不同的emoji。

接着使用 **pytorch**的 **SentenceTransformer** 预训练模型 (**paraphrase-multilingual-MiniLM-L12-v2**) 将中文组词转换为文本表征向量并存储在 **text_lookup** 中。

```

# 训练模型
total_steps = epoches * len(pairs)
progress_bar = tqdm(total=total_steps, desc="Training Emoji2Vec")

for epoch in range(epoches):
    total_loss = 0
    for emoji, text in pairs:
        emoji_id = torch.tensor([emoji_to_id[emoji]])
        emoji_vec = model(emoji_id)

        target_vec = torch.tensor(text_lookup[text],
dtype=torch.float32).unsqueeze(0)
        loss = loss_fn(emoji_vec, target_vec)

    optimizer.zero_grad()

```

```

        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        progress_bar.update(1) # ✅ 每训练一步，进度条前进一步

    progress_bar.set_postfix(epoch=epoch, loss=total_loss)

# 获取所有emoji向量（只用embedding层）
emoji_embeddings = {}
for emoji, idx in emoji_to_id.items():
    with torch.no_grad():
        vec = model.embedding(torch.tensor([idx])).squeeze(0).numpy()
        emoji_embeddings[emoji] = vec
# 组合为两个数组：emoji 列表 和 向量矩阵
emoji_list = list(emoji_embeddings.keys())
embedding_matrix = np.stack([emoji_embeddings[e] for e in emoji_list])

```

每轮训练迭代遍历时，对每个训练对 (`emoji`, `text`) 中emoji的ID转换为一个 PyTorch 张量，输入训练模型中的到对应的emoji表征向量，将其与对应的中文组词构建的文本表征向量进行监督学习，每轮迭代进行一次损失函数的计算并通过反向传播更新参数。

最终从嵌入层（embedding层）中获取所有emoji的初始表征向量并得到向量矩阵，组合为两个数组 `emoji_list` 和 `embedding_matrix`。

5.4 main.py 主函数整合

主要整合其他的代码模块，通过 `utils.py` 模块和 `emoji_embedding.py` 模块分别实现汉字字音字形的多维特征编码和emoji表征向量；基本流程为：

1. 数据预处理：读取并预处理数据，包括清洗文本、去除停用词和生成汉字及emoji的词向量
2. 创建字符向量的特征工程：基于汉字相似度矩阵生成字符向量，并构建句子嵌入向量，生成emoji表征向量
3. 模型训练与评估：使用逻辑回归模型进行训练和预测，评估后输出混淆矩阵和分类报告

5.4.1 数据预处理

这个部分主要包含以下函数，实现读取文本并清洗，通过分词和停用词处理后保证得到干净的分词结果。

```
def read_data(filename):
def clean_text(dataset):
def tokenize_and_remove_stopwords(dataset):
```

```
def read_data(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        text_data = f.readlines() # 读取原始数据文件
        dataset = [s.strip().split('\t', 1) for s in text_data] # 按制表符分割标签和文本
        dataset = [data for data in dataset if len(data) == 2 and data[1].strip()] #
过滤无效数据
        tag = [data[0] for data in dataset] # 提取标签列
        text = [data[1] for data in dataset] # 提取文本列
        return tag, text
```

主要实现堆原始数据文件的读取，使用 `strip` 方法去除每行首尾的空白字符，使用列表推导式过滤掉不符合要求的数据，从中提取并返回标签 `tag` 和文本 `text` 列。

```
def clean_text(dataset):
    cleaned_text = []
    for text in tqdm(dataset, desc='Cleaning text'):
        clean = re.sub(r'^[\u4e00-\u9fa5a-zA-Z0-9\s]', '', text) # 只保留中文字符、
字母、数字和空格
        cleaned_text.append(clean.strip()) # 去除首尾空白
    return cleaned_text
```

该函数主要实现将非汉字、非字母数字的字符去除，即清洗文本数据，首先遍历每个文本，使用正则表达式 `re.sub` 去除不符合条件的字符。保留汉字（`\u4e00-\u9fa5`）、字母（`a-zA-Z`）、数字（`0-9`）和空白字符（`\s`）。最后将清洗后的文本添加到 `cleaned_text` 列表中返回。

```
def tokenize_and_remove_stopwords(dataset):
    stopwords_file = 'data/hit_stopwords.txt'
    with open(stopwords_file, 'r', encoding='utf-8') as file:
        stopwords = {line.strip() for line in file}

    tokenized_text = []
    for text in tqdm(dataset, desc='Tokenizing and removing stopwords'):
        cleaned_text = ''.join([char for char in text if char not in stopwords and
re.search("[\u4e00-\u9fa5]", char)])
        tokenized_text.append(cleaned_text)

    return tokenized_text
```

该函数主要用以去除停用词将文本分割为字符列表。首先从文件 `hit_stopwords.txt` 中读取停用词，存储为集合 `stopwords`。接着遍历每个文本去除停用词，保留汉字字符，将处理后的文本添加到 `tokenized_text` 列表中返回。

5.4.2 构造特征工程

这部分主要实现从文本数据中提取有用的特征，生成用于模型训练的嵌入向量。需要统计汉字并计算字形相似矩阵生成Word2Vec初始字符向量，结合字形相似性生成增强字符向量，通过emoji_embedding.py得到并合并emoji向量，综上使用注意力机制生成句子向量。

主要包括以下几个函数：

```
def generate_w2v_vectors(tokenized_text, d=100):
def update(w2v_vectors, text, character, d=100):
def save_char_vectors(filename, vectors):
def load_char_vectors(filename):
def generate_char_vectors(chinese_characters, w2v_vectors, sim_mat, text,
chinese_characters_count, threshold=0.6, force_recompute: bool = False):
def generate_sentence_vectors(texts, char_vectors, d=100, force_recompute: bool =
False):
```

```
def generate_w2v_vectors(tokenized_text, d=100):
    model = Word2Vec(sentences=tokenized_text, vector_size=d, window=5,
min_count=1, sg=0)
    word_vectors = model.wv

    w2v_vectors = {}
    for tokens in tqdm(tokenized_text, desc='Generating word vectors'):
        for word in tokens:
            if word not in w2v_vectors.keys():
                w2v_vectors[word] = word_vectors[word]

    return w2v_vectors
```

该函数主要使用 `Word2Vec` 模型为每个汉字生成初始表征向量，请注意，这步和 `emoji_embedding.py` 中对emoji文字组合词的向量化操作不同，在本函数中使用了相似性加权平均和注意力机制，需要处理的是整个文本数据集，包括汉字和 emoji，要考虑汉字之间的相似性并生成句子级别的嵌入向量。后者的目标是训练 emoji 的词向量，使 emoji 的 ID 和其中文词意进行匹配。因此，它需要处理的是 emoji 和对应的中文词意，生成 emoji 的嵌入向量。

```
def update(w2v_vectors, text, character, d=100):
    model = Word2Vec(sentences=text, vector_size=d, window=5, min_count=1, sg=0)
    word_vectors = model.wv
    w2v_vectors[character] = word_vectors[character]

    return w2v_vectors
```

该函数是语料库的动态更新函数，可以为语料库中不存在的汉字生成字符向量，实现逻辑与上一个函数 `generate_w2v_vectors` 相似。

```
# 自定义保存/加载 numpy 向量为 json-friendly 格式
def save_char_vectors(filename, vectors):
    with open(filename, 'w', encoding='utf-8') as f:
        json.dump({k: v.tolist() for k, v in vectors.items()}, f,
        ensure_ascii=False)

def load_char_vectors(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        return {k: np.array(v, dtype=np.float32) for k, v in json.load(f).items()}
```

上述两个函数分别实现字符向量字典 `char_vectors` 的保存和加载，因为其实现的逻辑较为简单故不再赘述。

```
def generate_char_vectors(chinese_characters, w2v_vectors, sim_mat, text,
chinese_characters_count, threshold=0.6, force_recompute: bool = False):
    cache_file = 'data/char_vectors.json'
    if os.path.exists(cache_file) and not force_recompute:
        print(f"[INFO] Found cached file: {cache_file}. Loading char vectors...")
        return load_char_vectors(cache_file)

    char_vectors = {}
    for i in tqdm(range(len(chinese_characters)), desc='Generating char vectors'):
        character = chinese_characters[i]
        similar_group = []
        for j in range(len(sim_mat[i])):
            if sim_mat[i][j] >= threshold:
                similar_group.append(chinese_characters[j])
        sum_count = 0
        emb = np.zeros_like(w2v_vectors[list(w2v_vectors.keys())[0]]) # 初始化一个
全零向量
        for c in similar_group:
            if c not in w2v_vectors.keys():
                update(w2v_vectors, text, c)
            emb += int(chinese_characters_count[c]) * w2v_vectors[c] #注意
chinese_characters_count[c]是str类型
            sum_count += int(chinese_characters_count[c])
        emb /= sum_count if sum_count else 1 # 避免除以0
        char_vectors[character] = emb
```

```
save_char_vectors(cache_file, char_vectors)
return char_vectors
```

该函数是特征工程的关键环节，将汉字的初始特征向量与相似性信息结合起来，生成更准确的字符向量，这些向量将用于后续生成句子嵌入向量。这些向量不仅包含每个汉字的初始特征向量，还考虑了汉字之间的相似性，从而能够更好地反映汉字的语义信息。主要实现逻辑如下：

1. 检查是否存在缓存文件 `char_vectors.json`。如果存在且不需要重新计算（`force_recompute=False`），则直接加载缓存文件中的字符向量并返回。这样可以避免重复计算，提高效率，也便于我们成员间进行对接和代码试运行。
2. 初始化用于存储最终的字符向量字典 `char_vectors` 和进度条。对于当前汉字 `character`，使用汉字之间的相似度矩阵 `sim_mat` 找到与其相似度高于阈值 `threshold` 的其他汉字，存储在 `similar_group` 列表中。
3. 遍历 `similar_group` 中的每个相似汉字 `c`。如果 `c` 不在 `w2v_vectors` 中，则调用 `update` 函数动态生成该汉字的特征向量并更新字典。将相似汉字的特征向量按其出现次数加权累加到 `emb` 中。更新 `sum_count`，记录相似汉字的总出现次数。
4. 最后将加权平均嵌入向量 `emb` 归一化，需要考虑避免除以0，将结果存储到 `char_vectors` 字典中。

```
# 根据字嵌入向量生成句子嵌入向量（相似度是靠算出来的，并没有用自注意力机制）
def generate_sentence_vectors(texts, char_vectors, d=100, force_recompute: bool = False):
    cache_file = 'data/sentence_vectors.npy'
    if os.path.exists(cache_file) and not force_recompute:
        print(f"[INFO] Found cached file: {cache_file}. Loading sentence vectors...")
        return np.load(cache_file, allow_pickle=True)
    sentence_vectors = []
    for text in tqdm(texts, desc='Generating sentence vectors'):
        alpha = np.zeros((len(text), len(text))) # 存储第 i 个字符和第 j 个字符之间的余弦相似度
        for i in range(len(text)):
            for j in range(len(text)):
                # alpha[i][j] = alpha[j][i] = np.dot(char_vectors[text[i]],
                char_vectors[text[j]]) / np.sqrt(d)
                char_i = text[i]
                char_j = text[j]
                if char_i in char_vectors and char_j in char_vectors:
                    sim = np.dot(char_vectors[char_i], char_vectors[char_j]) /
                    np.sqrt(d)
                else:
                    sim = 0.2 # 或者用 np.nan，取决于你后续是否需要忽略缺失值
                alpha[i][j] = alpha[j][i] = sim

        # 注意力权重矩阵 alpha_hat 经过 Softmax将值映射到[0,1]
        alpha_hat = np.zeros_like(alpha)
```



```

for i in range(len(text)):
    for j in range(len(text)):
        alpha_hat[i][j] = alpha_hat[j][i] = np.exp(alpha[i][j]) /
np.sum(alpha[i])

#m是句子的embedding,是多个token部分求和
m = np.zeros((d,)) # 初始化一个全零向量
for i in range(len(text)):
    mi = np.zeros((d,))
    for j in range(len(text)):
        if text[j] not in char_vectors: #处理不在表中的emoji
            continue
        else:
            mi += alpha_hat[i][j] * char_vectors[text[j]]
    m += mi
sentence_vectors.append(m / d)

sentence_vectors = np.array(sentence_vectors)
np.save(cache_file, sentence_vectors)
return sentence_vectors

```

本函数的主要功能是根据 `generate_char_vectors` 得到的字符向量通过注意力机制生成句子嵌入向量，这些句子嵌入向量能够捕捉句子的整体语义信息，用于后续模型训练和预测任务。

其中注意力机制是本函数的关键，它能够让模型在处理输入数据时，可以分配不同的权重给输入数据的不同部分。通过计算字符之间的相似度，使用 Softmax 函数归一化这些相似度为注意力权重，然后根据这些权重和字符向量进行加权求和，注意力机制使得生成的句子嵌入向量能够根据相似度权重捕捉到句子的上下文信息，通过加权求和的方式更好地表示句子的语义信息，综上注意力机制能够提高句子表示的准确性和上下文敏感性。

我们主要实现的是伪注意力机制，即只是静态基于相似度的权重平均，不是可学习的 Transformer 式自注意力机制。我们通过计算简单的逆序位置来实现加权，即字符位置越靠前，权重越高，还需要归一化处理保证权重总和为1，这样可以突出句首重要字符（敏感词常出现在句首），且可以避免尾部字符权重过低，缓解长文本稀释效应。归一化也保证了在后续的分类器处理中的线性可加性。

代码整体逻辑如下：

1. 同样的，检查缓存文件如果存在且不需要重新计算（`force_recompute=False`），加载缓存文件中的句子嵌入向量并返回
2. 初始化用于存储最终的字符向量字典 `sentence_vectors` 和进度条。接着遍历文本中的每个字符，计算其与其他字符的余弦相似度，并存储在 `alpha` 矩阵中
3. 使用 Softmax 函数将相似度矩阵 `alpha` 归一化为注意力权重矩阵 `alpha_hat`

4. 重新遍历文本中的每个字符，根据注意力权重和字符向量，计算句子的嵌入向量。
将计算得结果归一化，并存储到 `sentence_vectors` 列表中

5.4.3 模型训练与评估流程

这是main.py代码中的最后一个模块，主要包括 `spam_classification` 和 `evaluation` 两个函数，分别实现模型分类和模型评估的作用。

```
# 垃圾文本分类
def spam_classification(train_tags, train_word_vectors, test_word_vectors):
    #oversampler = RandomOverSampler(sampling_strategy='auto', random_state=42)
    #train_word_vectors, train_tags = oversampler.fit_resample(train_word_vectors,
    train_tags)

    logistic_repression = LogisticRegression()
    logistic_repression.fit(np.array(train_word_vectors), np.array(train_tags))

    predictions = logistic_repression.predict(test_word_vectors)

    return predictions
```

该函数的主要功能是使用逻辑回归模型对垃圾文本进行二分类，它接收训练集的标签和向量，以及测试集的向量，训练模型并进行预测。

```
def evaluation(test_tags, predictions):
    # 输出混淆矩阵和分类报告
    cm = confusion_matrix(np.array(test_tags), np.array(predictions))
    print("混淆矩阵:")
    print(cm)

    report = classification_report(np.array(test_tags), np.array(predictions))
    print("分类报告:")
    print(report)
```

该函数的主要功能是评估模型的性能，输出混淆矩阵和分类报告。这些评估指标帮助我们了解模型在测试集上的表现。

其中的混淆矩阵是一个二维数组，表示模型预测结果与真实标签之间的关系。它可以帮助我们直观地了解模型的分类效果，包括真正例、假正例、真负例和假负例。

分类报告则包括精确率、召回率、F1 分数和每个类别的支持度。

```

if __name__ == "__main__":
    tag, text = read_data('data/dataset.txt')

    chinese_characters, chinese_characters_count, chinese_characters_code =
count_chinese_characters(text, 'data/hanzi.txt')
    sim_mat = compute_sim_mat(chinese_characters, chinese_characters_code)

    #text_train, text_test, tag_train, tag_test = divide_dataset(tag, text)

    cleaned_text = clean_text(text)
    tokenized_text = tokenize_and_remove_stopwords(cleaned_text)
    w2v_vectors = generate_w2v_vectors(tokenized_text)

    #new: emoji
    emoji_vectors = generate_emoji_vectors(force_recompute=False)
    char_vectors = generate_char_vectors(chinese_characters, w2v_vectors, sim_mat,
text, chinese_characters_count,force_recompute=False)
    char_vectors.update(emoji_vectors)# 合并 emoji 向量到字符向量字典中

    #对两种不同训练方法的向量进行归一化
    char_vectors = normalize_vectors(char_vectors)
    emoji_vectors = normalize_vectors(emoji_vectors)

    sentence_vectors = generate_sentence_vectors(tokenized_text,
char_vectors,force_recompute=False)

    train_vectors, test_vectors, train_tag, test_tag = divide_dataset(tag,
sentence_vectors)

    predictions = spam_classification(train_tag, train_vectors, test_vectors)

    evaluation(test_tag, predictions)

```

我们可以将main函数的核心流程梳理如下，可以参考在前文的流程图理解：

1. 从文件 `dataset.txt` 中读取原始数据集，统计文本中出现的汉字及其频率，得到汉字编码，并计算汉字之间的相似度矩阵。
2. 进行文本清洗，停用词处理和文本分割后使用 Word2Vec 为每个汉字生成初始特征向量。
3. 调用 `emoji_embedding.py` 中的函数生成 emoji 的词向量。
4. 根据汉字之间的相似度矩阵，生成最终的字符向量，并将上一步中的 emoji 向量合并到字符向量字典中。
5. 通过注意力机制，根据字符向量生成句子嵌入向量，划分为训练集和测试集。
6. 使用逻辑回归模型进行垃圾文本分类，评估模型性能，输出混淆矩阵和分类报告。

6 创新性说明

- 特征工程，通过人为设计特征提高分类准确率
- 基于大模型的emoji编码，利用emoji的语义信息和预训练大语言模型paraphrase-multilingual-MiniLM-L12-v2 对emoji进行编码有效识别使用emoji表达垃圾信息的文本，并比较不同编码方法的效果

7 优化空间

- 原项目着重于字符相似性网络的使用，因此是对每一个汉字进行word2vec编码，缺乏了对语义的捕捉。（字符级向量忽略词序和组合语义（如“彩票” vs “票彩”），而垃圾文本常依赖特定词组合）因此可以引入多头注意力机制，具体思路如下
 - 对文本进行分词操作
 - 其中一个注意力头计算词和词之间中的汉字编码的相似度（已经实现）。之所以直接点积是因为这样可以强化“语义上重要字符”对句子表示的贡献——因为它们在相似度加权中会显得更突出
 - 引入可训练的QKV矩阵，对词进行QKV得到q,k,v后再计算对齐分数，这样可以捕捉深层的词的语义关联
 - 综合多个注意力头得到句子表征
- 知识增强，利用敏感词词库进行检索；利用贝叶斯概率公式/先验概率提高准确性