高兼容性文本数据预处理模块

系统对于数据的读取准确度有着极高的要求，因此我们设计了此模块以适应在不同编译环境下正确读取自然语言文本数据，我们使用 CSV 文件来储存输入数据，该文件格式具有易读性，易处理性等优势，为我们数据读取模块的实现提供了基础。下面，我们分为两个部分来分别介绍这个模块中的两个关键函数。模块整体逻辑图如图 3 所示。

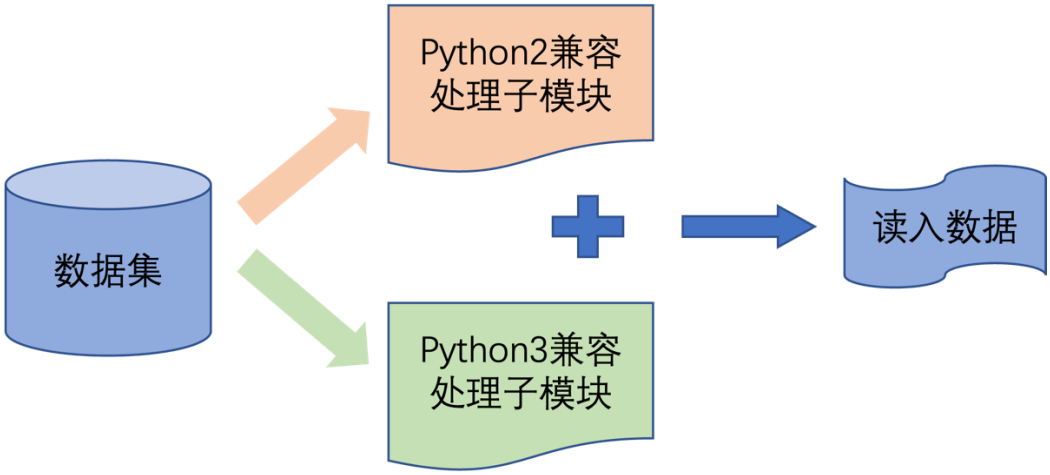

图 3 文本数据预处理模块

一、convert_to_unicode 子模块

- **模块整体框架**

```
def convert_to_unicode(text):
  """Converts `text` to Unicode (if it's not already), assuming utf-8 input."""
  if six.PY3:
    if isinstance(text, str):
      return text
    elif isinstance(text, bytes):
      return text.decode("utf-8", "ignore")
    else:
      raise ValueError("Unsupported string type: %s" % (type(text)))
  elif six.PY2:
    if isinstance(text, str):
      return text.decode("utf-8", "ignore")
    elif isinstance(text, unicode): # Python2 Compatibility errors
      return text
    else:
      raise ValueError("Unsupported string type: %s" % (type(text)))
  else:
```

```
18.     raise ValueError("Not running on Python2 or Python 3?")
```

1. 函数定义及文档字符串

```
1.  def convert_to_unicode(text):
2.    """Converts `text` to Unicode (if it's not already), assuming utf-8 input."""
```

我们定义了一个名为 convert_to_unicode 的函数，该函数的功能是将输入文本转换为 Unicode 格式，以确保后续处理中字符数据的统一编码格式。文档字符串描述了函数的目的，即假设输入为 utf-8 编码，将文本转换为 Unicode。

2. 检查 Python 版本

```
1.  if six.PY3:
```

此处使用 six.PY3 来检查当前运行的 Python 版本是否为 Python 3，这里的 six 库用于在 Python 2 和 3 之间编写兼容代码。

3. 处理 Python3 的不同类型输入

```
1.  if isinstance(text, str):
2.    return text
3.  elif isinstance(text, bytes):
4.    return text.decode("utf-8", "ignore")
5.  else:
6.    raise ValueError("Unsupported string type: %s" % (type(text)))
```

这里我们首先检查输入是否为字符串（'str'类型），如果是，那么直接返回该字符串，因为它已经是 Unicode 格式。之后检查输入是否为字节串（bytes 类型），如果是，那么就使用 utf-8 编码进行解码，并忽略可能引起错误的字符。对于其他类型而言，我们抛出一个异常，指出不支持的字符串类型。

4. 处理 Python2 的不同类型输入

```
1.  elif six.PY2:
2.    if isinstance(text, str):
3.      return text.decode("utf-8", "ignore")
4.    elif isinstance(text, unicode):  # Python 2 Compatibility errors
5.      return text
6.    else:
7.      raise ValueError("Unsupported string type: %s" % (type(text)))
```

在这一部分代码中，我们检查输入是否为字符串（'str'类型），如果是，那么使用 utf-8 编码将其解码为 Unicode。如果输入已经是 Unicode，那么就直接返回。对于其他类型而言，我们同样抛出不支持类型的异常。

5. 冗余化模块设计（处理未知 Python 版本）

```
1.  else:
2.    raise ValueError("Not running on Python2 or Python 3?")
```

为了对不同环境做到更好的适配，我们考虑到如果检测到的不是 Python 2 或 Python 3，则抛出一个异常，指出不支持的 Python 版本。

在该系统检测任务中，处理多种编码格式的文本数据是常见的场景。这些数据可能因为来源不同而存在编码差异。convert_to_unicode 函数确保所有文本数据被统一转换为 Unicode 格式，方便后续的文本处理和分析。这对于准确识别各种文本中的隐写信息非常关键。

二、generate_sequence_data 子模块

- **模块整体框架**

```
def generate_sequence_data(MAX_SEQUENCE_LENGTH, input_file, tokenizer, unlabeled=False, do_pairwise=False):

    X1 = []
    X2 = []
    y = []

    label_count = defaultdict(int)
    with tf.io.gfile.GFile(input_file, "r") as f:
      reader=csv.DictReader(f)
      for line in reader:

        if len(line) == 0:
          continue
        X1.append(convert_to_unicode(line["sentence"]))
        if do_pairwise:
          X2.append(convert_to_unicode(line[1]))
        if not unlabeled:
            if do_pairwise:
              label = int(convert_to_unicode(line[2]))
            else:
              label = int(convert_to_unicode(line["label"]))
            y.append(label)
            label_count[label] += 1
        else:
            y.append(-1)

    if do_pairwise:
      X =  tokenizer(X1, X2, padding=True, truncation=True, max_length = MAX_SEQUENCE_LENGTH)
    else:
```

```
30.       X =  tokenizer(X1, padding=True, truncation=True, max_length = MAX_SEQUENCE_LEN
    GTH)
31.
32.     for key in label_count.keys():
33.         logger.info ("Count of instances with label {} is {}".format(key, label_count
    [key]))
34.
35.     if "token_type_ids" not in X:
36.         token_type_ids = np.zeros((len(X["input_ids"]), MAX_SEQUENCE_LENGTH))
37.         logger.info ("token_type_ids  is {}".format(token_type_ids.shape))
38.     else:
39.         token_type_ids = np.array(X["token_type_ids"])
40.         logger.info ("token_type_ids  is {}".format(token_type_ids.shape))
41.
42.     return {"input_ids": np.array(X["input_ids"]), "token_type_ids": token_type_ids,
    "attention_mask": np.array(X["attention_mask"])}, np.array(y)
```

1. 函数定义

```
1.  def generate_sequence_data(MAX_SEQUENCE_LENGTH, input_file, tokenizer, unlabeled=Fals
    e, do_pairwise=False):
```

在这一子模块中，我们定义了一个名为 generate_sequence_data 的函数，用于生成序列数据。参数详细如下：

- **MAX_SEQUENCE_LENGTH**：每个输入序列的最大长度。
- **input_file**：包含输入数据的文件路径。
- **tokenizer**：用于将文本转换为模型输入格式的分词器。
- **unlabeled**：指示输入数据是否未标注。
- **do_pairwise**：指示是否处理成对的输入数据。

2. 初始化存储数据的列表

```
1.  X1 = []
2.  X2 = []
3.  y = []
```

此处初始化了三个空列表：

- **X1**：存储主要的输入句子。
- **X2**：存储成对输入的第二个句子（如果处理成对输入）。
- **y**：存储分类标签。

3. 初始化标签计数的字典

```
1.  label_count = defaultdict(int)
```

使用 defaultdict 初始化一个字典，用于统计每个标签的出现次数。

4. 读取输入文件

```
with tf.io.gfile.GFile(input_file, "r") as f:
  reader = csv.DictReader(f)
  for line in reader:
```

使用 tf.io.gfile.GFile 打开文件以供读取，确保兼容多种文件系统。csv.DictReader 用于读取 CSV 文件，并将每行数据解析为字典形式。

```
if len(line) == 0:
  continue
X1.append(convert_to_unicode(line["sentence"]))
if do_pairwise:
  X2.append(convert_to_unicode(line[1]))
if not unlabeled:
    if do_pairwise:
      label = int(convert_to_unicode(line[2]))
    else:
      label = int(convert_to_unicode(line["label"]))
    y.append(label)
    label_count[label] += 1
else:
    y.append(-1)
```

此处，我们逐行处理文件中的数据：

- **忽略空行**
- **将主要输入句子转换为 Unicode 并存储到 X1 中**
- **如果处理成对输入：**
- 将第二个句子转换为 Unicode 并存储到 X2 中。
- **如果数据有标注：**
- **如果处理成对输入**，读取对应标签并存储到 y 中，更新标签计数。
- **否则**，读取单独的标签，存储到 y 中，更新标签计数。
- **如果数据无标注：**
- 在 y 中存储-1 表示未标注的数据。

对于上述处理，在我们的系统检测过程中，输入数据可能包含单句或成对句子的文本，以及对应的标签。处理这类数据并将它们转换为模型输入格式，是模型训练前的重要准备工作，能够确保数据在统一格式下被正确读取、处理，并准备用于后续的模型输入。接下来我们处理序列数据。

5. Tokenize 输入数据

```
if do_pairwise:
    X = tokenizer(X1, X2, padding=True, truncation=True, max_length=MAX_SEQUENCE_LENGTH)
else:
    X = tokenizer(X1, padding=True, truncation=True, max_length=MAX_SEQUENCE_LENGTH)
```

使用分词器将文本数据转换为模型所需的输入格式：

- **如果处理成对输入**，将 X1 和 X2 作为输入。
- **否则**，仅处理 X1。

在这一步骤中，分词器返回的对象包含 input_ids、attention_mask 及 token_type_ids，并自动处理填充和截断，确保每个输入序列的长度一致。文本数据此后便从自然语言转换为模型可处理的数值格式。分词器处理后返回的信息使得模型能够正确读取并处理文本输入，帮助识别待检测载体中的隐写信息。

6. 统计并记录标签分布

```
for key in label_count.keys():
    logger.info("Count of instances with label {} is {}".format(key, label_count[key]))
```

通过遍历并记录每个标签的统计信息，我们可以及时观察不同标签的数据量分布是否均衡，并在必要情况下及时调整数据集样本组成结构。

7. 处理 token_type_ids

```
if "token_type_ids" not in X:
    token_type_ids = np.zeros((len(X["input_ids"]), MAX_SEQUENCE_LENGTH))
    logger.info("token_type_ids is {}".format(token_type_ids.shape))
else:
    token_type_ids = np.array(X["token_type_ids"])
    logger.info("token_type_ids is {}".format(token_type_ids.shape))
```

为了适配具体的检测任务，我们需要对 Token 作进一步细化处理，也就是检查分词器输出是否包含 token_type_ids：

- **如果不包含**，创建一个与输入 ID 数量和最大序列长度匹配的全零数组。
- **如果包含**，将其转换为 NumPy 数组。

因为在处理成对句子输入时，token_type_ids 用于区分不同的文本片段，而这对模型理解文本对的关系是必要的，在我们的恶意隐写载体检测中，并不需要使用成对句子输入，因此我们在输入阶段及时将 Token 标准化，以供后续模型能够正确地处理数据。

8. 返回处理后数据

```
return {"input_ids": np.array(X["input_ids"]), "token_type_ids": token_type_ids, "attention_mask": np.array(X["attention_mask"])}, np.array(y)
```

我们将处理后的输入数据和标签作为该子模块的输出，用作模型的标准输入：

- **input_ids**：词元 ID 的 NumPy 数组。
- **token_type_ids**：区分文本片段的 NumPy 数组。
- **attention_mask**：注意力掩码的 NumPy 数组。
- **y**：标签的 NumPy 数组。

返回的数据直接输入到模型中用于后续的采样、训练及预测等阶段。

通过实现 convert_to_unicode 和 generate_sequence_data 两个子模块，我们最终实现了高兼容性的文本数据预处理模块。在恶意隐写载体检测任务中，该模块能够确保文本数据被正确处理和格式化，从而提高模型的训练效果和检测准确性。