

题目 3：基于 PyTorch 的新闻文本分类任务（此题 100 分）

一. 题目概述

现提供一个新闻文本分类的数据集，存储在.csv 的文件中，包含两个变量：

text: 新闻文本内容

category: 新闻文本类别

其中训练集（Training Set）存储在./data/news_train.csv 中，测试集（Testing Set）存储在./data/news_test.csv 中。

请基于以上的数据集，训练自然语言处理的模型，实现新闻文本分类任务。

二. 提交要求

请基于数据集完成以下 5 道小题，按照要求把答案复制到答题卡的指定位置，并提交整个作答过程中的 Notebook 文档（会影响评分）。一共需要提交以下两个文件，请按照命名规则命名。

1. jupyter notebook 运行文件（包括运行结果的运行文件），命名规则为 NOAI3_学生编号.ipynb

（1）不提交 jupyter notebook 文件的，此题计 0 分。

（2）在作答过程中如使用了多个 jupyter notebook，请提供清晰的命名，例如用来做哪道题的。

2. NOAI3 答题卡的.docx 文档，命名规则为 NOAI3_学生编号.docx

三. 参考用库

1. 可以将下表用库直接复制到 notebook 中

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import nltk
from nltk.tokenize import word_tokenize
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from collections import Counter
from tqdm import tqdm
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_fscore_support
# nltk.download("punkt") #如果系统提示没有 punkt 词库，请向工作人员举手示意
```

2. 在整个答题过程中，为了避免卡在代码细节，你可以参考组委会提供的 pandas、numpy、matplotlib、scikit_learn、PyTorch 参考代码，可以根据自己需要进行复制和修改。

四. 题目主体

一共有 5 道小题，每道 20 分。

1. 代码阅读与任务解读（20 分）

以下是数据读取部分的代码，请阅读代码并执行后，[自己撰写一些计算程序，回答以下问题](#)。在阅读代码的过程中，可能会遇到 nltk 库不会使用的情况，这时候可以参考附录 1 和附录 2 的有关内容。

```
# 加载新闻数据集
train_df = pd.read_csv("./data/train_news.csv")
test_df = pd.read_csv("./data/test_news.csv")

def preprocess(df, word2idx=None, label2idx=None, max_len=500):#当一段话单词数（包括标点）超过 500 的时候，
截断该句；当单词数不足 500 时，在句子后面都补上统一的数字
    texts = df["text"].values
    labels = df["category"].values

    # 标签编码
    if label2idx is None:
        unique_labels = set(labels)
        label2idx = {label: idx for idx, label in enumerate(unique_labels)}
    labels = [label2idx[label] for label in labels]

    # 分词
    tokenized_texts = [word_tokenize(text.lower()) for text in texts]

    if word2idx is None:
        # 构建词汇表
        all_tokens = [token for text in tokenized_texts for token in text]
        vocab = Counter(all_tokens)
        vocab_size = 25000 #最多不会超过 25000 个词汇
        vocab = vocab.most_common(vocab_size - 2)
        word2idx = {word: idx + 2 for idx, (word, _) in enumerate(vocab)}
        word2idx["<unk>"] = 0
        word2idx["<pad>"] = 1

    # 数值化
    def encode_text(text):
        return [word2idx.get(word, word2idx["<unk>"]) for word in text]

    encoded_texts = [encode_text(text) for text in tokenized_texts]

    # 填充或截断
    padded_texts = [
        (
            text[:max_len]
            if len(text) > max_len
            else text + [word2idx["<pad>"]] * (max_len - len(text))
        )
    ]
```

```

        for text in encoded_texts
    ]

    return padded_texts, labels, word2idx, label2idx

# 划分训练集和测试集
X_train, y_train, word2idx, label2idx = preprocess(train_df)
X_test, y_test, _, _ = preprocess(test_df, word2idx, label2idx)

# 自定义 Dataset 类
class NewsDataset(Dataset):
    def __init__(self, texts, labels):
        self.texts = torch.tensor(texts, dtype=torch.long)
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        return self.texts[idx], self.labels[idx]

train_dataset = NewsDataset(X_train, y_train)
test_dataset = NewsDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True) #训练集的 dataloader 创建结束
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False) #测试集的 dataloader 创建结束

```

(1) 新闻数据集中包含的 label 种类数为_____，分别_____，它们对应的 label 的取值分别为_____。

(2) 导入到 train_loader 训练集的 text 文本中，包含多少种不同的单词（大小写相同算一个词，不要纠结标点和特殊字符，数值上下浮动 10 不影响评分）？

(3) 导入到 test_loader 测试集的 text 文本中，出现多少个在 train_loader 中没有出现过的单词（大小写相同算一个词，不要纠结标点和特殊字符，数值上下浮动 10 不影响评分）？注意重复出现的不识别的单词要重复计数，例如出现两次 train_loader 中没有出现过的单词“xxx”，则需要统计两次。

(4) 导入到 test_loader 测试集的 text 文本中出现多少**种**在 train_loader 中出现过的单词（大小写相同算一个词，不要纠结标点和特殊字符，数值上下浮动 10 不影响评分）？注意重复出现的可以识别的单词只进行一次计数。

提示：（2）（3）（4）问不作答，不影响后面的程序编写。

2. 实现一个单向 LSTM 的训练和测试过程（20 分）

请根据附录 3 提供参考代码，实现一个单向 LSTM 的模型。

具体要求如下：

- ① model: 单向 LSTM，建议与附录 3 相同
- ② optimizer: 优化器为 Adam，Learning_Rate 为默认
- ③ criterion: 损失函数 nn.CrossEntropyLoss()
- ④ epoch: 10 轮

请将每轮的训练集（Training Set）的损失（Loss）和准确率（Accuracy）都打印出来，并最终打印测试集的测试准确率。样例如下（具体准确率数值没有参考价值）。你在打印时，请将打印日志中“Xiao Ai”替换成你的名字的全拼，例如张三，请替换成“Zhang San”。

备注：请不要假装执行了过程，直接修改文本，如发现直接修改文本提交的，整个题目计 0 分！

```
Epoch 1/10, Xiao Ai Train Loss: 1.4010, Xiao Ai Train Accuracy: 0.3001
Epoch 2/10, Xiao Ai Train Loss: 1.4009, Xiao Ai Train Accuracy: 0.3002
Epoch 3/10, Xiao Ai Train Loss: 1.4008, Xiao Ai Train Accuracy: 0.3003
Epoch 4/10, Xiao Ai Train Loss: 1.4007, Xiao Ai Train Accuracy: 0.3004
Epoch 5/10, Xiao Ai Train Loss: 1.4006, Xiao Ai Train Accuracy: 0.3005
Epoch 6/10, Xiao Ai Train Loss: 1.4005, Xiao Ai Train Accuracy: 0.3006
Epoch 7/10, Xiao Ai Train Loss: 1.4004, Xiao Ai Train Accuracy: 0.3007
Epoch 8/10, Xiao Ai Train Loss: 1.4003, Xiao Ai Train Accuracy: 0.3008
Epoch 9/10, Xiao Ai Train Loss: 1.4002, Xiao Ai Train Accuracy: 0.3009
Epoch 10/10, Xiao Ai Train Loss: 1.4001, Xiao Ai Train Accuracy: 0.3010
Xiao Ai Test Accuracy: 0.3003
```

请在答题卡相应位置提交模型的 class 和上述打印结果。

```
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embd
        super().__init__()
```

3. 实现一个双向 LSTM 的训练和测试过程（20 分）

小爱同学发现单向 LSTM 的准确率太低，为了提升准确率，他将 LSTM 模型改为双向 LSTM，但是由于对代码不熟悉，他求助了大语言模型，如何将一个单向 LSTM 改为双向 LSTM。请你根据附录 4 和附录 5 的提示，将单向 LSTM 改为双向 LSTM。

具体要求如下：

- ① model: 双向 LSTM
- ② optimizer: 优化器为 Adam, Learning_Rate 为默认
- ③ criterion: 损失函数 nn.CrossEntropyLoss()
- ④ epoch: 10 轮

请将每轮的训练集（Training Set）的损失（Loss）和准确率（Accuracy）都打印出来，并最终打印测试集的测试准确率。样例如下（具体准确率数值没有参考价值）。你在打印时，请将打印日志中“Xiao Ai”替换成你的名字的全拼，例如张三，请替换成“Zhang San”。

备注：请不要假装执行了过程，直接修改文本，如发现直接修改文本提交的，整个题目计 0 分！

```
Epoch 1/10, Xiao Ai Train Loss: 1.0010, Xiao Ai Train Accuracy: 0.5001
Epoch 2/10, Xiao Ai Train Loss: 1.0009, Xiao Ai Train Accuracy: 0.5002
Epoch 3/10, Xiao Ai Train Loss: 1.0008, Xiao Ai Train Accuracy: 0.5003
Epoch 4/10, Xiao Ai Train Loss: 1.0007, Xiao Ai Train Accuracy: 0.5004
Epoch 5/10, Xiao Ai Train Loss: 1.0006, Xiao Ai Train Accuracy: 0.5005
Epoch 6/10, Xiao Ai Train Loss: 1.0005, Xiao Ai Train Accuracy: 0.5006
Epoch 7/10, Xiao Ai Train Loss: 1.0004, Xiao Ai Train Accuracy: 0.5007
Epoch 8/10, Xiao Ai Train Loss: 1.0003, Xiao Ai Train Accuracy: 0.5008
Epoch 9/10, Xiao Ai Train Loss: 1.0002, Xiao Ai Train Accuracy: 0.5009
Epoch 10/10, Xiao Ai Train Loss: 1.0001, Xiao Ai Train Accuracy: 0.5010
Xiao Ai Test Accuracy: 0.5003
```

请在答题卡相应位置提交模型的 class 和上述打印结果。

```
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, emb):
        super().__init__()
```

4. 绘制双向 LSTM 训练集 (Training Set) 的损失 (Loss) 和准确率 (Accuracy) 的变化图 (20 分)

请绘制 10 个 Epoch 的训练集的 Loss 和 Accuracy 变化图，具体要求如下：Loss 用蓝线表示，Accuracy 用红线表示。颜色错误不给分。下图是一个示例（注意：示例图颜色和要求颜色不一样），将图绘制在同一张图上或者两张图上均可。



请将你生成的上图（或者两张图）提交到答题卡指定位置。

5. 在测试集 (Testing Set) 中评估双向 LSTM 模型的性能 (20 分)

请在测试集评估模型的性能, 打印每个类别的精确率 (Precision)、召回率 (Recall)、F1 分数 (F1-Score)。由于小爱同学已经忘记了这三个参数的具体计算方法, 所以请教了大语言模型, 大语言模型的回答见附录 6。

你的任务在测试集上打印双向 LSTM 的准确率 (Accuracy), 并打印各类别的精确率 (Precision)、召回率 (Recall)、F1 分数 (F1-Score)。

参考样例如下, Category 一共有 n 行, n 是第 1 题中计算出来的 label 种类。你在打印时, 请将打印日志中 “Xiao Ai” 替换成你的名字的全拼, 例如张三, 请替换成 “Zhang San”。

备注: 请不要假装执行了过程, 直接修改文本, 如发现直接修改文本提交的, 整个题目计 0 分!

```
Xiao Ai Test Accuracy: 0.5001
Xiao Ai Category 0: Precision: 0.5001, Recall: 0.5001, F1: 0.5001
...
Xiao Ai Category n-1: Precision: 0.5001, Recall: 0.5001, F1: 0.5001
```

请将你用于计算评估指标的函数或者关键代码, 与上面的评估指标提交到答题卡指定位置。

```
def precision_recall_f1(preds, labels, num_classes):
```

其中 preds 为预测值, labels 为真实值, num_classes 为 label 种类

附录 1:

小爱在阅读代码的过程中，看不懂下面这句话，于是他求助了大语言模型

```
tokenized_texts = [word_tokenize(text.lower()) for text in texts]
```

某大语言模型对这行代码的解读如下：

这行代码使用Python列表推导式来对一系列文本进行分词和转换为小写。

具体步骤如下：

1. `texts` 是一个包含原始文本数据的列表或数组，其中每个元素都是一个字符串，代表一个文本（比如一篇文章或一条推文）。
2. `word_tokenize(text.lower())` 对每个文本进行处理：
 - `text.lower()`：将文本中的每个字符转换成小写。这是为了确保文本处理不区分大小写，即"Hello"和"hello"被视为相同的词。
 - `word_tokenize(...)`：使用 `nltk` 库中的 `word_tokenize` 函数对转换成小写的文本进行分词。这个函数会根据空格、标点符号等将文本分割成单词的列表。
3. 列表推导式 `[...] for text in texts` 对 `texts` 中的每个元素应用上述的处理步骤，并将结果收集到一个新的列表 `tokenized_texts` 中。

最终，`tokenized_texts` 是一个列表，其中每个元素都是一个单词列表，这些单词是原始文本分词并转换为小写后的结果。例如，如果 `texts` 中的第一个元素是字符串 `"Hello, world!"`，那么 `tokenized_texts` 的第一个元素将是 `["hello", ",", "world", "!"]`。

附录 2:

小爱同学在阅读代码的过程中，看不懂下面这段话，于是他求助了大语言模型

```
if word2idx is None:
    # 构建词汇表
    all_tokens = [token for text in tokenized_texts for token in text]
    vocab = Counter(all_tokens)
    vocab_size = 25000
    vocab = vocab.most_common(vocab_size - 2)
    word2idx = {word: idx + 2 for idx, (word, _) in enumerate(vocab)}
    word2idx["<unk>"] = 0
    word2idx["<pad>"] = 1
```

某大语言模型对这行代码的解读如下：

这段代码用于构建词汇表 (vocabulary)，将单词映射到唯一的整数索引。这是文本数据预处理的一个重要步骤，因为神经网络模型需要处理数值数据。以下是代码的详细解释：

1. 检查 word2idx 是否为 None：

- 这个条件判断用于确定是否需要构建词汇表。如果 word2idx 已经存在，那么它将包含单词到索引的映射，不需要重新构建。

2. 收集所有标记：

- all_tokens 是一个列表推导式，它遍历 tokenized_texts 中的每个文本，再遍历每个文本中的每个标记 (token)，并将所有标记收集到一个列表中。

3. 构建词汇频率表：

- Counter(all_tokens) 使用 collections 模块中的 Counter 类来计数 all_tokens 中每个唯一标记的出现次数。

4. 限制词汇表大小：

- vocab_size 被设置为25000，这意味着词汇表将只包含最常出现的25000个单词。
- vocab.most_common(vocab_size - 2) 返回词汇表中出现次数最多的前 vocab_size - 2 个单词的列表。这里减去2是为了为特殊标记留出空间。

5. 创建单词到索引的映射：

- word2idx 是一个字典推导式，它遍历 vocab 中的每个单词和其计数，并为每个单词分配一个索引。索引从2开始，因为0和1被保留用于特殊标记。
- {word: idx + 2 for idx, (word, _) in enumerate(vocab)} 这样做的目的是确保最常见的单词有一个较小的索引，这有助于在模型训练过程中减少计算量。

6. 添加特殊标记：

- word2idx["<unk>"] = 0：为未知单词（即词汇表中未出现的单词）分配索引0。
- word2idx["<pad>"] = 1：为填充标记分配索引1，这在将所有文本序列填充到相同长度时使用。

最终，word2idx 是一个字典，它将每个单词映射到一个唯一的整数索引，这个映射将用于将文本数据转换为模型可以处理的数值格式。

附录 3:

为了能够实现一个 LSTM 模型，完成整个训练和测试过程，我们求助了大语言模型，我们的问题如下：LSTM 实现文本分类

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import numpy as np

# 假设你已经完成了数据预处理，并得到了以下数据
# X_train, y_train = ... # 训练数据和标签
# X_val, y_val = ... # 验证数据和标签
# X_test, y_test = ... # 测试数据和标签
# vocab_size = ... # 词汇表大小
# max_len = ... # 序列最大长度
# num_classes = ... # 类别数量

# 定义 LSTM 模型
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_classes, pad_idx):
        super(LSTMClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_idx)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_output, (hidden, cell) = self.lstm(embedded)
        # 我们只需要最后一个时间步的输出
        last_hidden = hidden[-1]
        logits = self.fc(last_hidden)
        return logits

# 模型参数
embedding_dim = 128
hidden_dim = 64
vocab_size = len(word2idx) # 这个值取值跟第一问有关，即词汇表大小
num_classes = n # 此处 n 为你在第一问中计算出来的 label 数目
pad_idx = word2idx["<pad>"] # 这个值取值跟第一问有关

# 实例化模型
model = LSTMClassifier(vocab_size, embedding_dim, hidden_dim, num_classes, pad_idx)

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())
```

将数据转换为 PyTorch 张量

```
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)
X_test = torch.tensor(X_test, dtype=torch.long)
y_test = torch.tensor(y_test, dtype=torch.long)
```

创建数据加载器

```
train_loader = DataLoader(X_train, y_train, batch_size=64, shuffle=True) #训练集 (Training Set)
val_loader = DataLoader(X_val, y_val, batch_size=64, shuffle=False) #验证集 (Validation Set) 集 (注意: 本案例没有验证集)
test_loader = DataLoader(X_test, y_test, batch_size=64, shuffle=False) #测试集 (Testing Set)
```

训练模型

```
def categorical_accuracy(preds, y): #用于计算准确率的函数
    max_preds = preds.argmax(dim=1, keepdim=True)
    correct = max_preds.squeeze(1).eq(y)
    return correct.sum() / y.shape[0]
num_epochs = 10 #训练 10 轮
for epoch in range(num_epochs):
    model.train()
    for batch in train_loader:
        optimizer.zero_grad()
        inputs, targets = batch
        logits = model(inputs)
        loss = criterion(logits, targets) #训练集 loss
        acc = categorical_accuracy(logits, targets) #训练集准确率
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()
    print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {epoch_loss/len(train_loader):.4f}, Train Accuracy: {epoch_acc/len(train_loader):.4f}') #打印训练集的 loss 和准确率
    # 在每个 epoch 结束时使用 validation 评估模型
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for batch in val_loader:
            inputs, targets = batch
            logits = model(inputs)
            _, predicted = torch.max(logits, 1)
            total += targets.size(0)
            correct += (predicted == targets).sum().item()

    print(f'Epoch {epoch+1}/{num_epochs}, Accuracy: {correct/total:.4f}')
```

```
# 在测试数据上评估模型
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for batch in test_loader:
        inputs, targets = batch
        logits = model(inputs)
        _, predicted = torch.max(logits, 1)
        total += targets.size(0)
        correct += (predicted == targets).sum().item()

print(f'Test Accuracy: {correct/total:.4f}') #打印测试集的准确率
```

附录 4:

为了让模型的训练结果变好，我们将 LSTM 中的，`bidirectional=True`，并询问大语言模型，能够带来怎么样的效果。

我们的问题如下：

```
self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True, bidirectional = True)
```

大语言模型的回答截图如下：

在LSTM模型中，设置 `bidirectional = True` 意味着你正在使用双向LSTM (BiLSTM)。双向LSTM的特点是它有两个LSTM层，每个层都在不同的方向上处理输入序列：一个从开始到结束，另一个从结束到开始。这样做的好处是模型可以同时考虑输入序列的前后上下文，这通常可以提高模型在序列任务（如文本分类、语音识别等）上的性能。

在你的代码示例中，`self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True, bidirectional = True)` 这行代码创建了一个双向LSTM层，其中 `embedding_dim` 是嵌入层的输出维度，`hidden_dim` 是LSTM层的隐藏状态维度。`batch_first=True` 意味着数据的维度顺序是 `(batch_size, sequence_length, feature_dim)`，这是PyTorch的默认设置。

双向LSTM的输出将是两个方向LSTM层输出的拼接，因此每个时间步的输出维度将是 `2 * hidden_dim`。在定义后续的全连接层时，你需要考虑到这一点，因为它的输入维度将是 `2 * hidden_dim`。

例如，如果你在BiLSTM后面有一个全连接层：

python

复制

```
1 self.fc = nn.Linear(2 * hidden_dim, num_classes)
```

这里，全连接层的输入维度是 `2 * hidden_dim`，因为BiLSTM的输出在最后一个时间步的维度是 `2 * hidden_dim`。`num_classes` 是输出层的维度，对应于分类任务的类别数量。

在使用BiLSTM时，你需要确保你的数据集和任务可以从额外的模型复杂性中受益。对于一些任务，BiLSTM可以提高性能，但对于其他任务，标准的单向LSTM可能就足够了。

附录 5:

我们发现单纯地将 LSTM 中的 `bidirectional = True`，代码会有 bug，于是询问大语言模型帮忙 debug。

我们输入到大语言模型的问题如下：

请假下面代码debug

```
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_classes, pad_idx):
        super(LSTMClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_idx)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True, bidirectional = True)
        self.fc = nn.Linear(2 * hidden_dim, num_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_output, (hidden, cell) = self.lstm(embedded)
        # 我们只需要最后一个时间步的输出
        last_hidden = hidden[-1]
        logits = self.fc(last_hidden)
        return logits
```

大语言模型的回答如下：

代码本身看起来是正确的，但是有一个小错误在获取最后一个时间步的输出。在双向LSTM中，`hidden` 是一个包含两个方向的隐藏状态的元组，每个方向的大小是 `hidden_dim`。因此，你需要拼接这两个方向的最后一个时间步的隐藏状态，而不是只使用一个方向的。

这里是修正后的 `forward` 方法：

```
python
1  def forward(self, x):
2      embedded = self.embedding(x)
3      lstm_output, (hidden, cell) = self.lstm(embedded)
4      # 我们需要拼接双向LSTM的最后一个时间步的输出
5      last_hidden = torch.cat((hidden[-2], hidden[-1]), dim=1)
6      logits = self.fc(last_hidden)
7      return logits
```

在这个修正中，`hidden[-2]` 和 `hidden[-1]` 分别代表了最后一个时间步的向前和向后方向的隐藏状态。使用 `torch.cat` 函数将这些状态在第二个维度 (`dim=1`) 上拼接起来，这样全连接层的输入维度就是 `2 * hidden_dim`，与全连接层的预期输入尺寸相匹配。

现在，`forward` 方法应该可以正确地处理双向LSTM的输出，并且模型应该可以正常训练和预测。

附录 6: Precision、Recall 和 F1-Score 的解读

大语言模型的回答:

在多分类问题中，为了计算每个类别的精确率 (Precision)、召回率 (Recall) 和 F1 分数 (F1 Score)，你需要遍历每个类别，并计算该类别相关的指标。每个类别的指标计算如下：

1. **精确率 (Precision)**：精确率是指在所有被预测为正的样本中，实际为正确的比例。

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

其中，TP (True Positive) 是模型正确预测的正样本数量，FP (False Positive) 是模型错误预测为正的负样本数量。

2. **召回率 (Recall)**：召回率是指实际为正的样本中，模型正确预测为正确的比例。

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

其中，TP (True Positive) 是模型正确预测的正样本数量，FN (False Negative) 是模型错误预测为负的正样本数量。

3. **F1 分数 (F1 Score)**：F1 分数是精确率和召回率的调和平均数，用于综合评估模型的精确性和鲁棒性。

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

进一步解读：对于多分类问题中的第 3 类样本，TP 实际和预测均为 3 的样本数量；FP 为实际不是 3 却预测为 3 的样本数量；FN 为实际为 3 却预测不为 3 的样本数量。