中国矿业大学计算机学院

2019 级本科生课程作业

课程名称	机器学习及优化——实验三
报告时间	2023 年 4 月 25 日
学生姓名	许万鹏
学　　号	05191643
班　　级	信息安全 2019-01 班
任课教师	李政伟

摘要

情感分析是对文本中所表达情感进行分析和识别的过程，我们可以使用机器学习技术更好地分析自然语言所表达的情感。在本次实验中，本人首先使用机器学习库 Scikit-Learn 实现了一种简洁且有效的预测方式。为了更深入地理解各机器学习算法的原理，将理论知识工程化，本人基于目前最常用的深度学习框架 PyTorch，从零实现了本次任务中用到的所有机器学习算法，如 SGD、Adam 等，并将它们组合成了一个完整的框架——XuTorch，最后调用该框架完成了中文的情感分析任务。

目　录

1 概述

1.1 数据集介绍

情感分析作为 NLP 领域内比较热门的方向，其数据集规模很大。出于理解基本机器学习技术而不是对 SOTA 方法改进的目的，我们不使用大规模数据集，而是从大众点评网站上爬取 6000 条左右的好评/差评信息（如下图）做二分类任务。当然对于多分类任务也是一样的，只需要更改网络的输出维度即可。这种中文的情感二分类任务也是有实际用途的，例如我国的 AI 辅助监管系统，就可以通过用户评论的正面/负面来标记用户是否经常发送违背社区规则的言论。

spot	content	label
黄果树	还记得小时候，常常守在电视机前，等候《西游记》的播出。“你挑着担,我牵着马。翻山涉水两肩双滑……”熟悉的歌曲	好评
黄果树	飞流直下三千尺 疑是银河落九天！	好评
黄果树	黄果树大瀑布景区，位置坐落在贵州省安顺市关岭县黄果树风景区。这里是贵州省最著名的景区。也是中国最美的瀑布，	好评
黄果树	国家重点风景名胜区和首批国家A级旅游景区 今天黄果树景区时而艳阳高照，时而瓢泼大雨；当地人说两个多月没有这么	好评
黄果树	黄果树瀑布群绝对是值得一看的。景区非常大，不止黄果树瀑布，天星桥景区绝对不要错过。那个天生桥下方的珍珠链瀑	好评
黄果树	到了贵州，必须要到的就是黄果树瀑布！以黄果树瀑布为中心，周边分布着十几条瀑布，黄果树瀑布是黄果树瀑布群中最	好评
黄果树	状观的黄果树瀑布 晴天隐隐响春雷， 绝顶奔来云深处。	好评
黄果树	带父母带贵州玩，那一定要来贵州最有名的黄果树瀑布看看。父母年纪大了，所以选择的跟团游比较安全。 提前做好的攻	好评
黄果树	去年的时候来玩的，当时一直想着来到新看到的景色如何。没想到来到这之后，感觉跟之前书上描述的一模一样。真是大	好评
黄果树	进门以后乘坐景区大巴前往第一站陡坡塘瀑布，据说这里是拍摄西游记的地方，陡坡塘瀑布是三个瀑布里面最小的，也不	好评

通过 jieba 库进行简单的分词预处理，可以得到如下图所示的数据。

content	label
记得 小时候 守 电视机 前 等候 西游记 播出 挑 担 牵 马 翻山 涉水 两肩 双滑 ″ 熟悉 歌曲 耳边 响起 时 歌词 中 水	0
飞流直下三千尺 疑是 银河 落九天	0
黄果树 瀑布 景区 位置 坐落 贵州省 安顺市 关岭 县 黄果树 风景区 贵州省 著名 景区 中国 最美 瀑布 享有 中华 第一 瀑	0
国家 重点 风景 名胜区 首批 国家 A 级 旅游 景区 黄果树 景区 时而 艳阳高照 时而 瓢泼大雨 当地人 说 多月 晴朗 天	0
黄果树瀑布 群 值得一看 景区 黄果树瀑布 天星桥 景区 错过 天生桥 下方 珍珠 链 瀑布 值得一看 瀑布 黄果树 观景 距离	0
贵州 黄果树瀑布 黄果树瀑布 中心 周边 分布 十几条 瀑布 黄果树瀑布 黄果树瀑布 群中 一级 瀑布 因该 区域 生长 黄果树	0
状观 黄果树瀑布 晴天 隐隐 响 春雷 奔来 云 深处	0
带 父母 带 贵州 玩 贵州 有名 黄果树瀑布 父母 年纪 选择 团游 提前 做好 攻略 黄果树 核心 景点 西游记 取景 慕名而	0
去年 玩 想着 来到 新 景色 没想到 来到 书上 描述 一模一样 大自然 魅力四射 人间仙境 想到 西游记 情景 美丽 瀑布 无	0
进门 乘坐 景区 大巴 前往 第一站 陡坡 塘 瀑布 拍摄 西游记 地方 陡坡 塘 瀑布 三个 瀑布 最小 太 壮观 个人感觉 四十	0

1.2 基于特征的自然语言情感分析

这里介绍了基于特征的自然语言情感分析与传统机器学习任务的区别和联系，以及其中出现问题的解决方法。

NLP 领域作为人工智能的热门子领域，已经有多种针对自然语言进行情感预测的方法，如 Transform 和 Bert。但我们在这里用的是一种基于特征的方法，这种方法属于机器学习方法，大致思路是将句子分词为单词，将单词作为特征，然后像传统机器学习方法一样通过特征和标签进行模型学习。这种方法的好处是简单，快速，模型小；但也有不好的地方，就是没有语法和前后文关系，这些需要使用循环神经网络，如 LSTM，或更高级的 Transform 或 Bert 实现。

“将句子分为数个词汇作为特征，然后进行传统机器学习。”这个地方听起来简单，但

是自然语言中的词汇全部都是离散的，且数量庞大，其中不乏有许多没有情绪意义的词/数字/符号，如“而且”、“一个”、“一般”、“211”、“（）”等。

对于离散的特征，我们有解决方法，就是将其 one-hot 化，一个特征对应一个位置，如[0, 0, 0, 1]。在 NLP 中有一种类似的方法——Bag-of-Words Model，即词袋模型，同样是将离散特征（单词）数值化，但不是一个单词对应一个位置，而是一个单词对应一个数，如3。这次任务我们会使用词袋模型。

对于数量庞大的词汇们，我们也有一种简单有效的处理办法——去除低频单词。将单词数值化放入词袋的同时，我们会统计它们的频率，因此我们可以通过设置最低频率阈值 min_df 来对低频单词进行去除。当然，这是中文的情况，因为我们主要处理中文词汇，所以对于其中夹杂的英文词汇，我们可以直接通过其长度进行去除，毕竟大部分英文词汇长度越短，其自身所包含的信息量越少。

对于没有意义的词，我们可以将其标记为停用词，保存在字典里面，当处理时碰到位于停用词字典里的词，忽略它即可。

即便这样，处理后的词汇数量仍然达到了 5000 个以上！这和前两个任务中的 13 个、30 个特征并不在一个数量级，要知道我们仅仅爬取了约 6000 条数据！

而这样又带来了两个新的问题。

一：从每条句子中提取的特征数量都不同，同一个 batch 内甚至无法将输入进行堆叠，更无法将变长的输入放入模型。

二：从一条句子中提取的特征远小于从句子全集中提取的特征，这是因为自然语言的词汇本就是离散的。例如一条句子可能仅有 20 个特征，要怎么放到一个宽度为 5000 条特征的网络呢？

首先，对于第一个问题，我们可以强制规定句子的特征数量 num_sentence_word，多裁少补，将规定长度外的特征去除，或用某个固定的数（如<PAD>: 0）来补足特征。这样一般不会带来问题，因为超长的特征带来的细微情感变化对于我们预测大体情感是多余的，另外添加的 0 也会被深度神经网络学习到它并没有帮助分析情感的作用。

然后，对于第二个问题，对于小模型，我们可以不设置全体特征数为层的尺寸，毕竟宽度为 5000 的网络不那么容易训练，我们将特征数设置得比 num_sentence_word 大即可，这样的网络更方便训练，对理解 NLP 任务的帮助更大，至于为什么可以呢？我们可以认为神经网络将部分类似情感的词汇进行了合并，完成了训练。

另外，如果想要训练更精确的模型，也可以设置全体特征数为层的尺寸，我们可以将输入层设置为 Embedding 层，这个层会自动将低维度的输入映射到高维度上。

2 基于 SkLearn 的简洁实现

首先，导入 sklearn 中已实现的函数。

```
import numpy as np
import pandas as pd
from scipy.sparse import coo_matrix
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
```

然后，按照机器学习的一般流程：准备数据、数据分析、特征工程、模型选择及训练、模型评估，逐个调用函数。这里与一般的机器学习任务有些不同，但大致上是一样的。

```
# 加载黄果山好评数据集
df = pd.read_csv('huangguo_mountain.csv', encoding='UTF-8')
df = df.dropna()

contents = df['content']
labels = df['label']

#将文本中的词语转换为词频矩阵 矩阵元素a[i][j]表示j 词在i 类文本下的词频
vectorizer = CountVectorizer(min_df=5)

#该类会统计每个词语的tf-idf 权值
transformer = TfidfTransformer()

#第一个fit_transform 是计算tf-idf 第二个fit_transform 是将文本转为词频矩阵
tfidf = transformer.fit_transform(vectorizer.fit_transform(contents))

# 获取词袋模型中的所有词语
word = vectorizer.get_feature_names_out()
print("词汇数量:", len(word))

#将tf-idf 矩阵抽取出来，元素w[i][j]表示j 词在i 类文本中的tf-idf 权重
X = coo_matrix(tfidf, dtype=np.float32).toarray()          #稀疏矩阵 注意float

# 随机划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.3,
random_state=1)

# 创建MLP
classifier = MLPClassifier()

# 训练
classifier.fit(X_train, y_train)
```

```
print('模型的准确度: ', classifier.score(X_test, y_test))

y_preds = classifier.predict(X_test)
print(classification_report(y_test, y_preds))
```

得到输出如下。

```
词汇数量: 4923
模型的准确度:  0.9890230515916575
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      1527
           1       0.94      1.00      0.97       295

    accuracy                           0.99      1822
   macro avg       0.97      0.99      0.98      1822
weighted avg       0.99      0.99      0.99      1822
```

可以看出，基于 sklearn 的代码可以很简洁地（不超过 30 行）得到较好的结果（精确率 Acc 0.989）。

3 基于 PyTorch 的从零实现

3.1 调用 XuTorch 的实现

导入 PyTroch 和 XuTroch 库

```
import torch
import xutorch
```

设置参数

```
args = {
    'min_df': 5,
    'num_sentence_word': 100,
    'lr': 0.0005,
    'num_epochs': 200,
    'batch_size': 8,
    'early_stop': 50,
    'save_path': 'weights/hw3_model.pth'
}
```

创建数据加载器

```
train_loader, train_dataset = xutorch.dataset.prep_dataloader(
    dataset_name='Sentiment',
    download=False,
    train=True,
```

```
    batch_size=args['batch_size'],
    test_ratio=0.25,
    min_df=args['min_df'],
    num_sentence_word = args['num_sentence_word']
)
val_loader, val_dataset = xutorch.dataset.prep_dataloader(
    dataset_name='Sentiment',
    download=False,
    train=False,
    batch_size=args['batch_size'],
    test_ratio=0.25,
    min_df=args['min_df'],
    num_sentence_word = args['num_sentence_word']
)
```

输出如下。

```
Finished reading the train set of Text Dataset (4573 samples found), each dim
= 5044
Finished reading the val set of Text Dataset (1500 samples found), each dim =
5044
```

创建模型、优化器、损失函数

```
xutorch.misc.set_seeds(42)

device = xutorch.misc.get_device()

# model = xutorch.model.EmbeddingMLP(input_dim=train_dataset.dim,
hidden_dim=8192, output_dim=2).to(device)
model = xutorch.model.MLP(input_dim = args['num_sentence_word'],
hidden_dim1=128, hidden_dim2=256, output_dim=2).to(device)

# 可读取模型权重接续训练
# state_dict = torch.load(args['save_path'])
# model.load_state_dict(state_dict)

# 创建一个CE 损失函数
criterion = xutorch.loss.CrossEntropy()

# 创建一个Adam 优化器
optimizer = xutorch.optim.SimpleAdam(model.parameters(), lr=args['lr'])
```

训练模型

```
loss_record = {
    'train': [],
```

```
    'val': []
}

best_val_acc = 0.0
best_val_acc_epoch = 0.0

early_stop_cnt = 0

for epoch in range(args['num_epochs']):
    train_acc = 0.0
    train_loss = 0.0

    model.train()
    for batch_idx, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        outputs = model(inputs)
        train_pred = torch.sign(outputs)

        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # print(train_pred.cpu(), labels.cpu())
        train_pred_labels = torch.argmax(train_pred, dim=1)
        train_acc += (train_pred_labels.cpu() == labels.cpu()).sum().item()
        train_loss += loss.item()
    loss_record['train'].append(train_loss / len(train_dataset))

    val_acc = 0.0
    val_loss = 0.0

    model.eval()
    for batch_idx, (inputs, labels) in enumerate(val_loader):
        inputs, labels = inputs.to(device), labels.to(device)
        with torch.no_grad():
            outputs = model(inputs)
            val_pred = torch.sign(outputs)

            loss = criterion(outputs, labels)

        val_pred_labels = torch.argmax(val_pred, dim=1)
```

```
        val_acc += (val_pred_labels.cpu() == labels.cpu()).sum().item()
        val_loss += loss.item()
    loss_record['val'].append(val_loss / len(val_dataset))

    if epoch == 0 or val_acc > best_val_acc:
        best_val_acc = val_acc
        best_val_acc_epoch = epoch
        print(f'Saving model (epoch = {epoch + 1 : 4d}, acc = {best_val_acc /
len(val_dataset) : .4f})')
        torch.save(model.state_dict(), args['save_path'])
        early_stop_cnt = 0
    else:
        early_stop_cnt += 1

    if early_stop_cnt > args['early_stop']:
        print('EARLY STOP')
        break

    # if epoch % 10 == 9:
    print('[{:03d}/{:03d}] Train Acc: {:3.6f} Loss: {:3.6f} | Val Acc: {:3.6f}
loss: {:3.6f}'.format(
        epoch + 1, args['num_epochs'],
        train_acc / len(train_dataset), train_loss / len(train_dataset),
        val_acc / len(val_dataset), val_loss / len(val_dataset)))
```

输出如下。

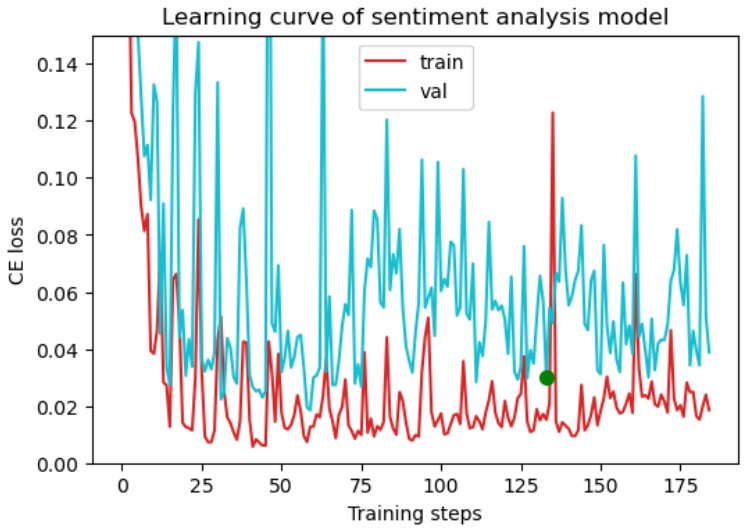
```
Saving model (epoch =    1, acc =  0.8320)
Saving model (epoch =    2, acc =  0.8547)
Saving model (epoch =    3, acc =  0.8733)
Saving model (epoch =    4, acc =  0.8853)
Saving model (epoch =    6, acc =  0.8900)
Saving model (epoch =   10, acc =  0.8947)
[010/200] Train Acc: 0.895692 Loss: 0.039401 | Val Acc: 0.894667 loss:
0.092232
Saving model (epoch =   11, acc =  0.9093)
Saving model (epoch =   16, acc =  0.9120)
Saving model (epoch =   20, acc =  0.9187)
[020/200] Train Acc: 0.918872 Loss: 0.014198 | Val Acc: 0.918667 loss:
0.053652
Saving model (epoch =   22, acc =  0.9220)
[030/200] Train Acc: 0.912311 Loss: 0.011373 | Val Acc: 0.911333 loss:
0.038753
Saving model (epoch =   35, acc =  0.9240)
```

```
[040/200] Train Acc: 0.908157 Loss: 0.042239 | Val Acc: 0.910000 loss:
0.063794
[050/200] Train Acc: 0.916029 Loss: 0.038269 | Val Acc: 0.908667 loss:
0.069276
Saving model (epoch =   51, acc =  0.9267)
[060/200] Train Acc: 0.920621 Loss: 0.012680 | Val Acc: 0.922667 loss:
0.018538
[070/200] Train Acc: 0.915592 Loss: 0.019676 | Val Acc: 0.910667 loss:
0.048084
[080/200] Train Acc: 0.919965 Loss: 0.009358 | Val Acc: 0.914667 loss:
0.088475
Saving model (epoch =   86, acc =  0.9300)
[090/200] Train Acc: 0.915810 Loss: 0.014332 | Val Acc: 0.917333 loss:
0.041101
[100/200] Train Acc: 0.919965 Loss: 0.014983 | Val Acc: 0.909333 loss:
0.105510
[110/200] Train Acc: 0.915154 Loss: 0.012200 | Val Acc: 0.912667 loss:
0.050361
[120/200] Train Acc: 0.917122 Loss: 0.012642 | Val Acc: 0.916667 loss:
0.055190
[130/200] Train Acc: 0.919309 Loss: 0.011532 | Val Acc: 0.906000 loss:
0.034873
Saving model (epoch =  134, acc =  0.9313)
[140/200] Train Acc: 0.920840 Loss: 0.013111 | Val Acc: 0.914000 loss:
0.067818
[150/200] Train Acc: 0.913623 Loss: 0.013278 | Val Acc: 0.912000 loss:
0.032541
[160/200] Train Acc: 0.898972 Loss: 0.024381 | Val Acc: 0.896667 loss:
0.048209
[170/200] Train Acc: 0.888038 Loss: 0.023985 | Val Acc: 0.891333 loss:
0.043248
[180/200] Train Acc: 0.900503 Loss: 0.024974 | Val Acc: 0.902000 loss:
0.046386
**EARLY STOP**
```

通过最后保存的 Acc 可以看出，其略低于同随机种子下的基于 sklearn 的实现。因为数据和模型部分进行了许多取舍，但是 0.931 的准确率也十分高了。

绘制学习曲线

```
xutorch.plot.plot_learning_curve(loss_record, title='sentiment analysis
model', loss_name='CE', bottom=0.0, top=0.15, min_loss_x=best_val_acc_epoch,
min_loss_y=loss_record['val'][best_val_acc_epoch])
```



3.2 基于 Pytorch 实现 XuTorch

这里介绍**本次任务所用到的** XuTorch 中的机器学习算法的实现过程。

```
XuTorch
|   __init__.py
|
├──dataset
|   |   boston.py
|   |   breast_cancer.py
|   |   prep_dataloder.py
|   |   sentiment.py
|   |   __init__.py
|   |
|   └──data
|           boston.txt
|           huangguo_mountain.csv
|           wdbc.data
|
├──loss
|       cross_entropy.py
|       hinge.py
|       mean_squared_error.py
|       __init__.py
```

```
    |
    ├──misc
    |       get_device.py
    |       r2_score.py
    |       set_seeds.py
    |       __init__.py
    |
    ├──model
    |       embedding_mlp.py
    |       linear.py
    |       mlp.py
    |       svm.py
    |       __init__.py
    |
    ├──optim
    |       adam.py
    |       sgd.py
    |       __init__.py
    |
    └──plot
            plot.py
            __init__.py
```

3.2.1 xutorch.dataset.prep_dataloader

这里定义了一个数据加载函数 prep_dataloader，其目的是根据数据集名称生成相应数据集对象并将其放入数据加载器中。

```
from xutorch.dataset.boston import BOSTON
from xutorch.dataset.breast_cancer import BreastCancer
from xutorch.dataset.sentiment import Sentiment
from torch.utils.data import DataLoader


def prep_dataloader(dataset_name, download, train, batch_size, num_workers=0,
test_ratio=0.25, **kwargs):
    dataset = None
    if dataset_name == 'BOSTON':
```

```
        dataset = BOSTON(download=download, train=train, test_ratio=test_ratio,
**kwargs)
    elif dataset_name == 'BreastCancer':
        dataset = BreastCancer(download=download, train=train,
test_ratio=test_ratio, **kwargs)
    elif dataset_name == 'Sentiment':
        dataset = Sentiment(download=download, train=train,
test_ratio=test_ratio, **kwargs)

    dataloader = DataLoader(dataset, batch_size, shuffle=train,
drop_last=False, num_workers=num_workers, pin_memory=True)
    return dataloader, dataset
```

3.2.2 xutorch.dataset.Sentiment

这里实现了一个基于特征的文本处理类 Sentiment，用于处理文本数据进行情感分析任务。

__init__方法的参数含义如下：

download：是否下载数据集，默认为 False；

path：数据集的路径，默认为 data/huangguo_mountain.csv；

train：是否用于训练集，默认为 True；

test_ratio：测试集占总数据集的比例，默认为 0.25；

min_df：单词至少出现的次数，默认为 5；

num_sentence_word：每个句子包含的最大单词数量，默认为 100。

在__init__方法中，首先读取数据集，并删除其中的空值。然后将数据集的内容和标签分别存储到 data 和 labels 中。接下来，读取停用词表 stopwords.txt，并将数据集中的单词进行统计，去除停用词后，对单词进行统计，并存储到 word_counts 字典中。

然后根据 min_df 参数筛选出单词数大于等于 min_df 的单词，并将其与对应的索引存储到 word_to_idx 字典中。最后在字典中加入了一个<PAD>单词，其对应的索引为 0。

根据 train 和 test_ratio 参数，选出相应的数据集作为当前数据集，并将其长度和字典大小存储到 dim 中。最后输出数据集的基本信息，如数据集类型、数据集大小和字典大小。

因为 NLP 任务的数据量大，最好不要一次性地对全集进行预处理，而是在通过__getitem__取出时对每一个样本单独进行预处理。

在__getitem__方法中，根据给定索引获取相应的文本内容和标签，并将文本内容转换成 PyTorch 中的 FloatTensor 类型返回。在转换过程中，先根据单词索引进行转换，然后进行多裁少补，若单词数量大于 num_sentence_word，则只取前 num_sentence_word 个单词，若单词数量小于 num_sentence_word，则在末尾加上若干个 0，使其长度达到 num_sentence_word。

```
import os
import torch
from torch.utils.data import Dataset
import pandas as pd


# 基于特征的文本处理方法
class Sentiment(Dataset):
    data_url = None
    current_dir = os.path.dirname(os.path.abspath(__file__))

    def __init__(self, download=False, path=os.path.join(current_dir, 'data',
'huangguo_mountain.csv'), train=True, test_ratio=0.25, min_df=5,
num_sentence_word = 100, stopword_path=os.path.join(current_dir, 'data',
'stopwords.txt')):
        if download:
            path = self.data_url

        df = pd.read_csv(path, encoding='UTF-8')
        df = df.dropna()

        data = df['content']
        labels = df['label']

        with open(stopword_path, 'r', encoding='utf-8') as f:
            stop_words = [stop_word.strip() for stop_word in f.readlines()]

        self.word_counts = {}
        for sentence in data:
            words = sentence.split()
            for word in words:
                if word not in stop_words and len(word) > 1:
                    self.word_counts[word.lower()] =
self.word_counts.get(word.lower(), 0) + 1

        self.num_sentence_word = num_sentence_word

        self.word_to_idx = {word: idx + 1 for idx, (word, count) in
enumerate(self.word_counts.items()) if count >= min_df}
        self.word_to_idx['<PAD>'] = 0

        pivot = int(100 * (1 - test_ratio))
        if train:
```

```
            indices = [i for i in range(len(data)) if i % 100 < pivot]
        else:
            indices = [i for i in range(len(data)) if i % 100 >= pivot]

        self.data = data.iloc[indices].values
        self.labels= labels.iloc[indices].values

        self.dim = len(self.word_to_idx)

        print('Finished reading the {} set of Text Dataset ({} samples found),
each dim = {}'.format(
            'train' if train else 'val', len(self.data), self.dim))

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

    # NLP 任务数据量大，取出时再进行处理
    def __getitem__(self, index):
        content = self.data[index]
        label = int(self.labels[index])
        return self.convert_to_tensor(content), torch.tensor(label)

    def convert_to_tensor(self, sentence):
        content_idx = [self.word_to_idx[word.lower()] for word in
sentence.split() if word.lower() in self.word_to_idx]
        if len(content_idx) >= self.num_sentence_word:
            content_idx = content_idx[:self.num_sentence_word]
        else:
            content_idx[len(content_idx):self.num_sentence_word] = [0] *
(self.num_sentence_word - len(content_idx))
        return torch.FloatTensor(content_idx)
        # return torch.LongTensor(content_idx)      # EmbeddingMLP
```

3.2.3 xutorch.misc.set_seeds

固定随机种子以复现相同结果。

```
import os
import torch
import random
import numpy as np


def set_seeds(seed):
    torch.manual_seed(seed)
```

```
    os.environ['PYTHONHASHSEED'] = str(seed)
    random.seed(seed)
    np.random.seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
    # if torch.backends.cudnn.is_available():
    #     torch.backends.cudnn.benchmark = False
    #     torch.backends.cudnn.deterministic = True
```

3.2.4 xutorch.misc.get_device

检查计算设备：是否有可用的 GPU，如果有则使用 GPU，否则使用 CPU。

```
import torch


def get_device():
    return torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

3.2.5 xutorch.model.MLP

这里定义了 MLP 类，与线性模型并没有太大区别，只是在层间加入了 ReLU 作为激活函数。

```
import torch.nn as nn


class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim1)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_dim2, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        return x.squeeze(1)
```

3.2.6 xutorch.model.EmbeddingMLP

这里定义了 EmbeddingMLP 类，其本质是一个将输入层从 Linear 换为了 Embedding 的单隐层网络。

```
import torch.nn as nn


class EmbeddingMLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(EmbeddingMLP, self).__init__()
        self.fc1 = nn.Embedding(input_dim, hidden_dim)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        return x.squeeze(1)
```

3.2.7 xutorch.optim.SimpleAdam

这里实现了简单的随机梯度下降（SGD）算法。为了省略一些不重要的操作（如 zero_grad 和另存参数），我令该类继承自 PyTorch 的 Optimizer 类。

构造函数__init__中，接收五个参数：params 表示要更新的参数组成的列表，lr 是学习率（默认为 0.01），betas 是指数衰减率（默认为(0.9, 0.999)），eps 是小常数（默认为 1e-8），weight_decay 是正则化系数（默认为 0）。

类中的 step 方法是该优化器的核心方法，用于更新参数，**参考自课程 PPT 中的 Chap3 P68 Adam**。

(5) Adam(Adaptive Moment Estimation)

参数：学习率η，矩估计的指数衰减速率，一阶动量衰减系数ρ_1和 ρ_2 在 [0,1)内（默认0.9 和 0.999），用于数值稳定的小常数 δ（默认：10^{-8}），初始参数 $\boldsymbol{\theta}$

初始化：一阶和二阶矩变量 $s = 0, r = 0$，时间步 $t = 0$

while 没有达到停止准则 **do**

从训练数据中抽取m个样本$\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$

$t \leftarrow t + 1$

更新有偏一阶矩估计：$\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$

更新有偏二阶矩估计：$\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$

修正一阶矩的偏差：$\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}}{1-\rho_1^t}$

修正二阶矩的偏差：$\hat{\boldsymbol{r}} = \frac{\boldsymbol{r}}{1-\rho_2^t}$

计算更新：$\Delta\boldsymbol{\theta} = -\eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$（逐元素应用操作）

end while

在这个方法中，对于每个参数 param，如果其梯度为 None，则跳过此参数。否则，根据当前的学习率、动量参数 beta1 和 beta2 以及梯度 grad，更新其对应的状态 state，包括步数 step、一次动量的指数平均值 exp_avg 和二次动量的指数平均值 exp_avg_sq。然后，利用偏差修正项对指数平均值进行修正，计算步长 step_size，并更新参数的值。如果设置了 weight_decay，则还需要对参数进行权值衰减。注意，如果不在 step 内部清空梯度（p.data=None），则一定要在代码外部显式地书写 optimizer.zero_grad()

需要注意的是，对于原地方法 buf.mul_和 p.data.add_中的 alpha 参数，它们的含义分别是乘数和加数的系数，用于对乘积或加和进行缩放。

```
import torch
import torch.optim as optim


# extend from torch.optim.Optimizer, 'zero_grad' method can be omitted
class SimpleAdam(optim.Optimizer):
    def __init__(self, params, lr=0.001, betas=(0.9, 0.999), eps=1e-8,
weight_decay=0):
        defaults = dict(lr=lr, betas=betas, eps=eps, weight_decay=weight_decay)
        super(SimpleAdam, self).__init__(params, defaults)

        for group in self.param_groups:
            for param in group['params']:
                state = self.state[param]
                state['step'] = 0
                state['exp_avg'] = torch.zeros_like(param.data)
```

```
            state['exp_avg_sq'] = torch.zeros_like(param.data)

    def step(self):
        for group in self.param_groups:
            for param in group['params']:  # 这个params其实是weights而不是模型的超参数
                if param.grad is None:
                    continue
                grad = param.grad.data

                state = self.state[param]
                state['step'] += 1

                beta1, beta2 = group['betas']
                exp_avg = state['exp_avg']
                exp_avg_sq = state['exp_avg_sq']

                exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
                exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1 - beta2)

                bias_correction1 = 1 - beta1 ** state['step']
                bias_correction2 = 1 - beta2 ** state['step']

                step_size = group['lr'] * (bias_correction2 ** 0.5) / bias_correction1

                if group['weight_decay'] != 0:
                    param.data.add_(-group['lr'] * group['weight_decay'], param.data)

                param.data.add_(exp_avg / (exp_avg_sq.sqrt() + group['eps']), alpha=-step_size)
```

3.2.8 xutorch.loss.CE

该类用于计算交叉熵（CrossEntropy）损失函数。

forward 方法实现了接受两个 tensor 作为输入参数 outputs 和 targets，首先使用 nn.LogSoftmax 对网络的输出结果进行 log softmax 操作，接着使用 nn.NLLLoss 计算 log softmax 的负对数似然损失，并将其返回。

```
import torch.nn as nn


class CrossEntropy(nn.Module):
```

```
    def __init__(self):
        super(CrossEntropy, self).__init__()

    def forward(self, outputs, targets):
        log_softmax = nn.LogSoftmax(dim=1)
        log_probs = log_softmax(outputs)
        loss = nn.NLLLoss()(log_probs, targets)
        return loss
```

3.2.9 xutorch.plot.*

这里包含了三个用于绘图的函数，其中：

plot_gt_and_pred 用于绘制预测值与真实值的散点图（预测-真实值散点图）；

plot_gt_vs_pred 用于绘制真实值与预测值的曲线图（拟合曲线）；

plot_learning_curve 用于绘制训练曲线。

```
import torch
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure


def plot_gt_and_pred(targets=None, preds=None, bottom=0.0, top=35.0):
    import os
    os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"

    preds = torch.cat(preds, dim=0).numpy()
    targets = torch.cat(targets, dim=0).numpy()

    figure(figsize=(5, 5))
    plt.scatter(targets, preds, c='r', alpha=0.5)
    plt.plot([bottom, top], [bottom, top], c='b')
    plt.xlim(bottom, top)
    plt.ylim(bottom, top)
    plt.xlabel('ground truth value')
    plt.ylabel('predicted value')
    plt.title('Ground Truth v.s. Prediction')
    plt.show()


def plot_gt_vs_pred(y_true, y_pred):
    y_pred = torch.cat(y_pred, dim=0).numpy()
    y_true = torch.cat(y_true, dim=0).numpy()
    plt.plot(y_true, label="gt")
    plt.plot(y_pred, label="predict")
```

```
    plt.xlabel("index")
    plt.ylabel("price")
    plt.legend(loc="best")
    plt.title("gt v.s. predict")
    plt.show()


def plot_learning_curve(loss_record, loss_name='', title='', bottom=0.0,
top=100.0, min_loss_x=None, min_loss_y=None):
    x_1 = range(len(loss_record['train']))
    x_2 = x_1[::len(loss_record['train']) // len(loss_record['val'])]
    figure(figsize=(6, 4))
    plt.plot(x_1, loss_record['train'], c='tab:red', label='train')
    plt.plot(x_2, loss_record['val'], c='tab:cyan', label='val')
    if min_loss_x and min_loss_y:
        plt.scatter(min_loss_x, min_loss_y, c='g', alpha=1, marker='o', s=50,
zorder=10)
    plt.ylim(bottom, top)
    plt.xlabel('Training steps')
    plt.ylabel('{} loss'.format(loss_name))
    plt.title('Learning curve of {}'.format(title))
    plt.legend()
    plt.show()
```