首先，先读取IMDB的影评数据，将训练集和测试集的影评分开存储

In [1]:

```
import numpy as np
import json

path = "../dataset/imdb.npz"
dict_path = "../dataset/imdb_word_index.json"

with np.load(path, allow_pickle=True) as f:
    x_train, labels_train = f["x_train"], f["y_train"]
    x_test, labels_test = f["x_test"], f["y_test"]

with open(dict_path) as f:
    word_index = json.load(f)

reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_review = [" ".join([reverse_word_index.get(i, "?") for i in line]) for line in x_train]
decoded_review_test = [" ".join([reverse_word_index.get(i, "?") for i in line]) for line in x_test]
```

之后，再对每段影评进行分词处理，分别保存在train_data和test_data中

In [2]:

```
train_data = []
word_list = []
for line in decoded_review:
    line_spl = list(line.split(" "))
    train_data.append(line_spl)
    word_list += line_spl
```

In [3]:

```
test_data = []
for line in decoded_review_test:
    line_spl = list(line.split(" "))
    test_data.append(line_spl)
```

使用训练集数据进行训练，通过word2vec得到每个词的100维embedding

In [4]:

```
from gensim.models import Word2Vec
# 训练Word2Vec模型，size为词向量的维度，词频小于min_count的词将不被考虑
model = Word2Vec(train_data, vector_size=100, min_count=1)
```

汇总影评中出现的词，以他们出现的频数降序排列

In [5]:

```
import collections
import pandas as pd

tab = collections.Counter(word_list)
df = pd.DataFrame.from_dict(tab, orient="index").reset_index()
df = df.rename(columns = {'index':'word', 0:'count'})
df = df.sort_values(by='count', ascending=False)

df
```

Out[5]:

	word	count
9	the	336148
61	and	164097
3	a	163040
53	of	145847
29	to	135708
...	...	...
58914	dalek's	1
58915	orators	1

	word	count
58917	miscalculations	1
58920	walkways	1
88583	shite'	1

88584 rows × 2 columns

根据上面得到的单词汇总数据框，生成一个重新编码的字典

In [6]:
```
word_length = len(df.word)
top_word = list(df.word)
mydict = dict(zip(top_word, list(range(word_length))))
```

根据新字典中对每个词的编码，分别生成训练集和测试集每段影评的one-hot编码矩阵，再将这个矩阵与word2vec编码下的矩阵拼接在一起

In [7]:
```
from operator import itemgetter

train_comment_num = len(decoded_review)
train_mat1 = np.zeros([train_comment_num, word_length + 1])
for i in range(train_comment_num):
    column_id = itemgetter(*train_data[i])(mydict)
    train_mat1[i, column_id] = 1
    train_mat1[i, word_length] = len(train_data[i])

train_mat2 = np.zeros([train_comment_num, 100])
for i in range(train_comment_num):
    vec = model.wv[train_data[i]]
    train_mat2[i,:] = np.mean(vec, 0)

train_mat1 = np.hstack((train_mat1, train_mat2))
train_mat1.shape
```

Out[7]: (25000, 88685)

In [8]:
```
test_comment_num = len(decoded_review_test)
test_mat1 = np.zeros([test_comment_num, word_length + 1])
for i in range(test_comment_num):
    column_id = itemgetter(*test_data[i])(mydict)
    test_mat1[i, column_id] = 1
    test_mat1[i, word_length] = len(test_data[i])

test_mat2 = np.zeros([test_comment_num, 100])
for i in range(test_comment_num):
    vec = model.wv[test_data[i]]
    test_mat2[i,:] = np.mean(vec, 0)

test_mat1 = np.hstack((test_mat1, test_mat2))
test_mat1.shape
```

Out[8]: (25000, 88685)

基于这两个矩阵，进行基础的逻辑回归

In [9]:
```
# 逻辑回归
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

scaler = StandardScaler()
train_scaled = scaler.fit_transform(train_mat1)
test_scaled = scaler.fit_transform(test_mat1)

logit_model = LogisticRegression()

# 在训练集上训练模型
```

```
logit_model.fit(train_scaled, labels_train)

# 在测试集上进行预测
y_pred = logit_model.predict(test_scaled)

# 计算准确率
accuracy = accuracy_score(labels_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.85084
```

第二个模型，基于神经网络结构对影评分类

In [10]:

```
# 神经网络
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

class network(nn.Module):
    def __init__(self):
        super(network, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(100, 64),
            nn.ReLU(),
            nn.Linear(64, 2)
        )
    def forward(self, x):
        x = self.fc(x)
        x = F.log_softmax(x, dim = 1)
        return x

nn_model = network().cuda()
```

In [11]:

```
train_set = torch.tensor(train_mat2, dtype=torch.float).cuda()
labels_train = torch.tensor(labels_train, dtype=torch.float).cuda()
test_set = torch.tensor(test_mat2, dtype=torch.float).cuda()
labels_test = torch.tensor(labels_test, dtype=torch.float).cuda()
```

In [12]:

```
lr = 1e-3
optimizer = optim.Adam(nn_model.parameters(), lr = lr)
```

In [13]:

```
def accuracy(outputs, labels):
    preds = torch.max(outputs, dim = 1)[1]
    acc = torch.sum(preds == labels).item() / len(labels)
    return acc
```

In [14]:

```
def test(model, test_set, labels_test):
    model.eval()
    outputs = model(test_set)
    labels_test = labels_test.long()
    loss = F.cross_entropy(outputs, labels_test)
    acc = accuracy(outputs, labels_test)
    return loss.item(), acc

def printlog(epoch, epochs, loss, acc, test_loss, test_acc):
    print(f"Epoch [{epoch}/{epochs}], loss: {loss:.4f}, acc: {acc:.4f}, test_loss: {test_loss:.4f}, t

def train(model, optimizer, train_set, test_set, labels_train, labels_test, epochs=1000):
    model.train()
    for i in range(epochs):
        optimizer.zero_grad()
        outputs = model(train_set)
        labels_train = labels_train.long()
        loss = F.cross_entropy(outputs, labels_train)
        acc = accuracy(outputs, labels_train)
        loss.backward()
        optimizer.step()
```

```python
        test_loss, test_acc = test(model, test_set, labels_test)
        if (i+1) % 100 == 0:
            printlog(i+1, epochs, loss.item(), acc, test_loss, test_acc)
```

In [16]:

```python
train(nn_model, optimizer, train_set, test_set, labels_train, labels_test, epochs=3000)
```

```
Epoch [100/3000], loss: 0.3674, acc: 0.8377, test_loss: 0.4060, test_acc: 0.8135
Epoch [200/3000], loss: 0.3627, acc: 0.8406, test_loss: 0.4069, test_acc: 0.8129
Epoch [300/3000], loss: 0.3577, acc: 0.8430, test_loss: 0.4076, test_acc: 0.8127
Epoch [400/3000], loss: 0.3533, acc: 0.8455, test_loss: 0.4089, test_acc: 0.8124
Epoch [500/3000], loss: 0.3495, acc: 0.8468, test_loss: 0.4108, test_acc: 0.8124
Epoch [600/3000], loss: 0.3457, acc: 0.8495, test_loss: 0.4125, test_acc: 0.8119
Epoch [700/3000], loss: 0.3422, acc: 0.8501, test_loss: 0.4143, test_acc: 0.8120
Epoch [800/3000], loss: 0.3389, acc: 0.8514, test_loss: 0.4161, test_acc: 0.8111
Epoch [900/3000], loss: 0.3357, acc: 0.8534, test_loss: 0.4178, test_acc: 0.8113
Epoch [1000/3000], loss: 0.3328, acc: 0.8543, test_loss: 0.4196, test_acc: 0.8109
Epoch [1100/3000], loss: 0.3300, acc: 0.8564, test_loss: 0.4215, test_acc: 0.8102
Epoch [1200/3000], loss: 0.3276, acc: 0.8574, test_loss: 0.4235, test_acc: 0.8087
Epoch [1300/3000], loss: 0.3249, acc: 0.8582, test_loss: 0.4252, test_acc: 0.8087
Epoch [1400/3000], loss: 0.3226, acc: 0.8598, test_loss: 0.4270, test_acc: 0.8081
Epoch [1500/3000], loss: 0.3203, acc: 0.8612, test_loss: 0.4288, test_acc: 0.8072
Epoch [1600/3000], loss: 0.3180, acc: 0.8622, test_loss: 0.4305, test_acc: 0.8069
Epoch [1700/3000], loss: 0.3160, acc: 0.8628, test_loss: 0.4325, test_acc: 0.8075
Epoch [1800/3000], loss: 0.3139, acc: 0.8638, test_loss: 0.4342, test_acc: 0.8053
Epoch [1900/3000], loss: 0.3121, acc: 0.8648, test_loss: 0.4360, test_acc: 0.8055
Epoch [2000/3000], loss: 0.3106, acc: 0.8646, test_loss: 0.4379, test_acc: 0.8044
Epoch [2100/3000], loss: 0.3086, acc: 0.8666, test_loss: 0.4401, test_acc: 0.8046
Epoch [2200/3000], loss: 0.3070, acc: 0.8675, test_loss: 0.4416, test_acc: 0.8040
Epoch [2300/3000], loss: 0.3053, acc: 0.8686, test_loss: 0.4430, test_acc: 0.8029
Epoch [2400/3000], loss: 0.3041, acc: 0.8694, test_loss: 0.4454, test_acc: 0.8036
Epoch [2500/3000], loss: 0.3030, acc: 0.8704, test_loss: 0.4465, test_acc: 0.8033
Epoch [2600/3000], loss: 0.3009, acc: 0.8709, test_loss: 0.4478, test_acc: 0.8029
Epoch [2700/3000], loss: 0.3001, acc: 0.8716, test_loss: 0.4497, test_acc: 0.8030
Epoch [2800/3000], loss: 0.2983, acc: 0.8720, test_loss: 0.4512, test_acc: 0.8021
Epoch [2900/3000], loss: 0.2970, acc: 0.8738, test_loss: 0.4524, test_acc: 0.8022
Epoch [3000/3000], loss: 0.2959, acc: 0.8742, test_loss: 0.4539, test_acc: 0.8024
```

可以发现，基础的逻辑回归在测试集的预测精度上优于神经网络模型，可能的原因是：1.基础的逻辑回归学习的是one-hot和word2vec两种编码拼接在一起的数据，而神经网络模型只用了word2vec编码的数据，基础逻辑回归模型对文字特征的掌握更细；2.神经网络模型更复杂，参数更多，在训练样本不多的情况下很容易达到过拟合，可以发现，随着训练的进行，训练集上的损失一直在降低，精度一直在提升，而测试集上的损失和精度一直无显著变化，