当代人工智能实验一——文本分类

--10215501435 杨茜雅

实验流程&实验目的：

●该任务是一个 10 分类任务

●将训练数据自行划分训练集和验证集（固定划分或多折交叉验证），本次实验中我采用**固定划分的手段**，（使用 sklearn.model_selection 里面的 train_test_split 对训练集分割成训练集和验证集）。

●将文本映射成向量（TF-IDF/Word2vec/Fasttext/Glove/Bert/XLNet）、，本次实验中我采用**TF-IDF 方法。**

●通过训练集训练分类器（SVM/MLP/TextCNN/TextRNN/Bert...），作业要求实现逻辑回/SVM/MLP/决策树四种。**此外我还额外实现了 Bert 模型。**

●模型评估：使用验证集评估训练好的模型的性能。本次实验中的评估指标包括**准确率、精确率、召回率和 F1 分数。**

●模型优化：根据评估结果，对模型进行优化。在本次实验中**采用 GridSearchCV 进行最有超参数搜索**，这是一个自动化的过程，比手动调参可解释性高并且普遍适用。

●结果输出：模型经过优化后，将最终表现最优异的模型输出存为 results.txt 文件。最后的结果来看，**Bert 模型**表现最优。

数据预处理

Step 1：

exp_1data 里有 train_data.txt 和 test.txt，分别用于分割成训练集、验证集和作为测试集。train_data.txt 是一个字典类型的文件，而 test.txt 是以每一行的第一个逗号和换行符进行分隔的文件。这里我先使用两个函数将其转化为 csv 文件。也就是代码文件里面的 txt_to_csv_train 和 txt_to_csv_test 函数。

```python
def txt_to_csv_train(filePathSrc, filePathDst):
    list_data = []
    with open(filePathSrc, 'r', encoding='utf-8') as input_file, open(filePathDst, 'w', newline='') as output_file:
        for data in input_file:
            data = eval(data)
            list_data.append(data)
        keys = list_data[0].keys()
        dict_writer = csv.DictWriter(output_file, keys)
        dict_writer.writeheader()
        dict_writer.writerows(list_data)
```

```python
def txt_to_csv_test(filePathSrc, filePathDst):
    with open(filePathSrc, 'r', encoding='utf-8') as input_file, open(filePathDst, 'w', newline='') as output_file:
        stripped = (line.strip('\n') for line in input_file)
        # 这里只需要用第一个逗号进行分割，因为一个句子中可能有很多个逗号
        lines = (line.split(', ', 1) for line in stripped if line)
        writer = csv.writer(output_file)
        writer.writerows(lines)
```

Step 2:

X 存有文本值，而 y 存有标签值。通过 X = X.values.tolist()让 pd DataFrame 对象转化为一个列表，发现它一个二维列表，所以就使用循环迭代的方式将其转化为一维列表。

```
if __name__ == '__main__':
    txt_to_csv_train('train_data.txt', 'train_data.csv')
    txt_to_csv_test('test.txt', 'test.csv')

    dataset_train = pd.read_csv('train_data.csv')
    X = dataset_train.drop('label', axis=1)
    y = dataset_train['label']
    X = X.values.tolist()
    list1 = []
    for index_i in range(len(X)):
        for index_j in range(len(X[index_i])):
            list1.append(X[index_i][index_j])
```

Step 3:

本次实验中我采用 TF-IDF 方法将文本映射成向量。

在众多方法中 TF-IDF/Word2vec/Fasttext/Glove/Bert/XLNet 选择 TF-IDF 的**原因**如下：

●TF-IDF 不考虑词汇的顺序，它只关注每个词汇在文档中的重要性。而 Word2Vec、FastText、GloVe、BERT 和 XLNet：这些都考虑了词汇的顺序信息，因此更适合处理需要考虑文本序列的任务，如情感分析或机器翻译。

●TF-IDF：计算简单，不需要大量计算资源。而 Word2Vec、FastText、GloVe、BERT 和 XLNet：这些方法通常需要更多的计算资源，特别是在大规模数据集上训练或微调时。

●TF-IDF：对标注数据需求较小，适用于小规模数据集。而 Word2Vec、FastText、GloVe、BERT 和 XLNet：这些方法通常需要大量标注数据来进行训练，尤其是在深度学习方法中。

综上所述，TF-IDF 在处理小规模数据集、对计算资源有限的环境时具有优势。所以我选择它作为本课程实验的文本数据预处理方法。

创建一个 TfidfVectorizer 对象，“stop_words='english'“ 表示在向量化的过程中会忽略英文的常用停用词，这有助于减少特征的数量并提高计算效率。，并且将其值赋给 tv，用于将文本数据转化为 TF-IDF 特征。

用刚刚得到的一维列表喂给这个对象，使用 tv.fit_transform(list1) 对文本数据 list1 进行了向量化处理，将每条文本数据转换为一个 TF-IDF 特征向量。这一步骤计算了每个单词在文本数据中的重要性，生成了一个 TF-IDF 特征矩阵。

.toarray() 将生成的 TF-IDF 特征矩阵由稀疏矩阵转换为密集矩阵，这样可以更容易地在后续的机器学习模型中使用。

我们可以看到打印了生成的 TF-IDF 特征矩阵的形状，即行数（文本数据的数量）和列数（特征的数量），为(8000，29697)。

```
for index_i in range(len(X)):
    for index_j in range(len(X[index_i])):
        list1.append(X[index_i][index_j])
tv = TfidfVectorizer(stop_words='english')
X_fit = tv.fit_transform(list1).toarray()
print(X_fit.shape)
```

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\LR.py"
(8000, 29697)
this model is running
```

训练数据划分：训练集和验证集（固定划分）

test_size=0.2 表示将数据集的 20%作为测试集，剩余的 80%用于训练。

random_state=42 是一个常用的数值。我发现在机器学习的过程中，常常遇到 random_state 这个参数，并且好多时候都是 random_state=42，所以我也取了 42。random_state 是一个随机种子，是在任意带有随机性的类或函数里作为参数来控制随机模式。当 random_state 取某一个值时，也就确定了一种规则。

```
# 数据集划分，进行模型的训练以及预测
X_train, X_val, y_train, y_val = train_test_split(X_fit, y, test_size=0.2, random_state=42)
```

模型选择：

1、 向量机 SVM

支持向量机是一个分类模型，学习算法是求解凸二次规划的最优化算法。线性支持向量机的学习问题是如下的凸二次规划问题（原始问题）：

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i$$
$$s.t. \quad y_i(w \cdot x_i + b) \geq 1 - \xi_i$$
$$\xi_i \geq 0$$

其中分离超平面是

$$w^* \cdot x + b^* = 0$$

分类决策函数为

$$f(x) = sign(w^* \cdot x + b^*)$$

其中的 **C** 为惩罚参数，一般由应用问题决定，**C** 值大的时候对误分类的惩罚增大，小的时候对误分类的惩罚减小。我们需要使 $\frac{1}{2}\|w\|^2$ 尽量小即间隔尽量大，同时让误分类点的个数尽量小，**C** 是调和两者的系数。我们应该让 **C** 值尽量小一些，允许容错，让模型的泛化能力更强一些。

参数优化：

对于 SVM，设置 kernel 值为线性核，然后参数搜索网格设置惩罚参数五个值，分别为 [0.5, 1, 2, 3,4]. 并且设置四折交叉验证。**设置线性核的原因**是因为线性核的运行速度相对较快，而本身这个数据集的特征量庞大，所以选择运行效率相对较高的线性核。参数搜索完成之后，我们发现 C=2 是最优的值，代回到模型中进行训练。

```
In [13]: grid = GridSearchCV(SVC(kernel='linear'), param_grid={"C":[0.5, 1, 2, 3, 4]}, cv=4)
         print('start selecting...')
         grid.fit(X_fit, y)
         print("The best parameters are %s with a score of %0.2f" %(grid.best_params_, grid.best_score_))

         start selecting...
         The best parameters are {'C': 2} with a score of 0.89
```

调参前：

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\SVM.py"
(8000, 29697)
this model is running
this model finishes running, running time: 529 seconds.
Model score: 0.956875
Model Accuracy: 0.956875
Model Precision: 0.9575550006366083
Model Recall: 0.9570216130446209
Model F1 Score: 0.9569559049780606
```

调参后：

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\SVM.py"
(8000, 29697)
this model is running
this model finishes running, running time: 540 seconds.
Model score: 0.958125
Model Accuracy: 0.958125
Model Precision: 0.9585739807937529
Model Recall: 0.9581346895437
Model F1 Score: 0.9580610294828048
```

2、 逻辑回归

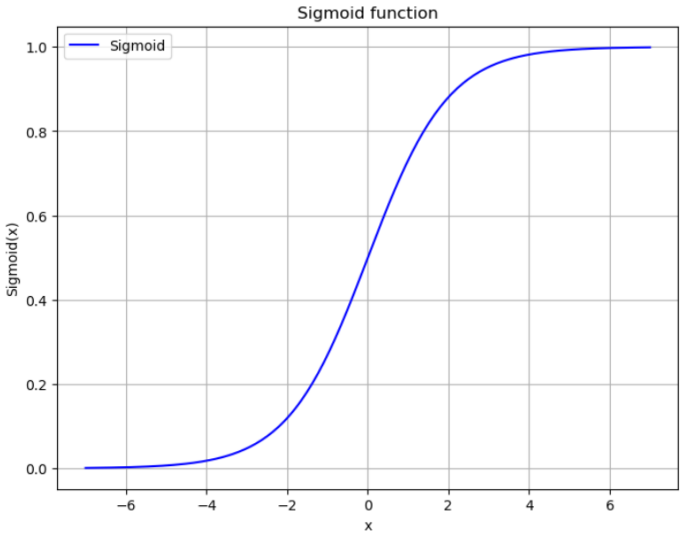
逻辑回归是一种广义的线性回归分析模型，是一种预测分析。虽然它名字里带回归，但实际上逻辑回归是一种分类学习方法。它不是仅预测出“类别”， 而是可以得到近似概率预测，这对于许多需要利用概率辅助决策的任务很有用。普遍应用于预测一个实例是否属于一个特定类别的概率，比如一封 email 是垃圾邮件的概率是多少。 因变量可以是二分类的，也可以是多分类的。因为结果是概率的，除了分类外还可以做 ranking model。LR 的应用场景很多，如点击率预测（CTR）、天气预测、一些电商的购物搭配推荐、一些电商的搜索排序基线等。

逻辑斯谛分布的分布函数是一种“Sigmoid”函数，呈现 S 型曲线，它将 ~z 值转化为一个接近 0 或 1 的 ~y 值。 对数几率回归公式如下：

$$\sim y = g(z) = \frac{1}{1+e^{-z}}, \sim z = w^T x + b$$

其中，$\sim y = \frac{1}{1+e^{-z}}$，被称做 Sigmoid 函数。3 辑回归算法是将线性函数的结果映射到 Sigmoid 函数中。

下图绘制了 Sigmoid 函数形状，如图所示，sigmoid 函数输出值范围在（0，1）之间，即代表了数据属于某一类别的概率，0.5 是作为判别的临界值。



使用 Sigmoid 函数的好处在于它能够将线性回归的结果转换为概率值，更直观地表示样本属于某一类的概率。这样，逻辑回归就可以解决分类问题，而不仅仅是回归问题。

参数优化：

与支持向量机一样，我们使用 GridSearchCV 进行参数选择。我们发现最优的超参数是 **C=10**。

```
In [6]: grid = GridSearchCV(LogisticRegression(max_iter=300), param_grid={"C":[1, 2, 5, 10], "penalty":['l2']}, cv=4)
        print('start selecting...')
        grid.fit(X_fit, y)
        print("The best parameters are %s with a score of %0.2f" %(grid.best_params_, grid.best_score_))

        start selecting...
        The best parameters are {'C': 10, 'penalty': 'l2'} with a score of 0.90
```

调参前：

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\LR.py"
(8000, 29697)
this model is running
this model finishes running, running time: 32 seconds.
Model score: 0.95125
Model Accuracy: 0.95125
Model Precision: 0.9516721781732581
Model Recall: 0.9506536394663371
Model F1 Score: 0.9508598430311445
```

调参后：

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\LR.py"
(8000, 29697)
this model is running
this model finishes running, running time: 51 seconds.
Model score: 0.9525
Model Accuracy: 0.9525
Model Precision: 0.9529427321230886
Model Recall: 0.9520112050291765
Model F1 Score: 0.9521316032102863
```

观察输出结果我们可以发现，逻辑回归模型的运行时间远小于 SVM 模型，但是两者的分数相差不大，SVM 略高。

对比两者的输出内容有什么差异：

```
使用 SVM 和逻辑斯谛回归训练出来的数据有： 104 行的不同。
320, 0

128, 0

622, 9

406, 6
```

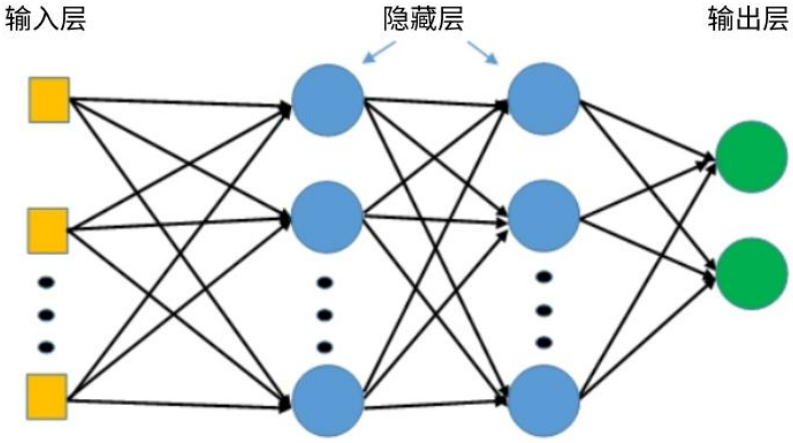
两个模型输出的差异有 104 行，占整个测试集的 5%。

如果比较一下两个机器学习方法的差异的话，从原理角度来讲，逻辑回归是把线性回归所得到的值通过 Sigmoid 函数映射到 [0, 1] 这个区间中，然后使用概率来进行判别。但是由于我们的数据在通过 TfidfVectorizer 向量化之后，有 29697 个特征，接近三万个。而这三万个特征如果全部选取，容易出现过拟合的问题，因为其中有严重的多重共线性。**所以这也是逻辑回归在模型所得的分数上面较低的一个原因**。而且在 Sigmoid 函数映射之后，值无限接近 0 或者 1 的时候，参数更新的梯度会明显下降，导致学习的效率变得很缓慢。不过 L2 正则化让我们降低了模型的特征数，减少了过拟合的程度。

而支持向量机在高维空间的表现相对较好，不仅是**因为感知器权重向量的更新会用到所有的数据，它的分类决策函数也没有逻辑回归那样梯度下降得那么快但是后期乏力的问题，核函数也可以将高维的数据映射到低维空间中**。所以支持向量机的模型分数较高。

3、 MLP 模型

多层感知机除了输入输出层，它中间可以有多个隐层，最简单的 MLP 只含一个隐层，即三层的结构，如下图:



左边层是输入层，由神经元集合{xi | xl,x2,...,xm},代表输入特征，隐藏层的每个神经元将前一层的的值通过线性加权求和的方式表示，即 wlxl+w2x2+...+wmxm,其次是一个非线性激活函数 g(.):R->R,比如双曲函数，输出层接受从最后一个隐藏层输出的值并将他们转换成值。

这个模块包含公共属性 coefs_和 intercepts_。Coefs_是权重矩阵列表，下标为 i 的权重矩阵代表 i 和 i+1 层的权重。intercepts 是一个偏差向量列表，其中第 i 个偏差向量代表加到 i+1 层上的偏差值。

MLP 的优点是:

可以学习非线性横型并且可以使用 partial_fit 实时学习

MLP 的缺点是:

有隐藏层的 MLP 包含一个非凸性损失函数，存在超过一个最小值，所以不同的随机初始权重可能导致不同验证精确度；MLP 要求调整一系列超参数，比如隐藏神经元，隐藏层的个数以及迭代的次数；MLP 对特征缩放比较敏感

参数优化：

```
In [20]: from sklearn.neural_network import MLPClassifier
         from sklearn.model_selection import GridSearchCV

         # 定义MLP模型
         model = MLPClassifier(max_iter=200, random_state=42)

         # 定义要搜索的参数范围
         param_grid = {
             'hidden_layer_sizes': [(50,), (100,), (100, 50), (200, 100)],
             'activation': ['relu', 'tanh'],
             'alpha': [0.0001, 0.001, 0.01],
         }

         # 创建GridSearchCV对象
         grid = GridSearchCV(model, param_grid, cv=4)

         print('start selecting...')
         grid.fit(X_fit, y)
         print("The best parameters are %s with a score of %0.2f" % (grid.best_params_, grid.best_score_))

         start selecting...
         The best parameters are {'activation': 'tanh', 'alpha': 0.001, 'hidden_layer_sizes': (100, 50)} with a score of 0.90
```

hidden_layer_sizes=(100, 50)：这指定了模型的隐藏层结构，包括两个隐藏层，第一个隐藏层有 100 个神经元，第二个隐藏层有 50 个神经元；activation='tanh'：这定义了激活函数，使用双曲正切函数 "tanh" 作为激活函数；alpha=0.001：这是正则化参数，用于控制模型的复杂度。较小的 alpha 值表示较弱的正则化。max_iter=200：这是最大迭代次数，指定模型在训练过程中允许的最大迭代次数。

调参前：

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\MLP.py"
(8000, 29697)
this model is running
this model finishes running
this model finishes running, running time: 885 seconds.
Model score: 0.95375
Model Accuracy: 0.95375
Model Precision: 0.9551122702627077
Model Recall: 0.9531067422544555
Model F1 Score: 0.9531447100469602
```

调参后：

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\MLP.py"
(8000, 29697)
this model is running
this model finishes running
this model finishes running, running time: 472 seconds.
Model score: 0.96
Model Accuracy: 0.96
Model Precision: 0.9600650697065098
Model Recall: 0.9599052636232621
Model F1 Score: 0.9597841418415791
```

4、 决策树模型

决策树是基于已知各种情况（特征取值）的基础上，通过构建树型决策结构来进行分析的一种方式，是常用的有监督的分类算法。它是一种树形结构，其中每个内部结点表示一个属性的测试；每个分支表示一个测试输出；每个叶结点代表一种类别。决策树模型核心有:结点和有向边组成；结点有内部结点和叶结点两种类型；内部结点表示一个特征，叶结点表示一个类。

决策树的决策过程就是从根结点开始，测试待分类项中对应的特征属性，并按照其值选择输出分支，直到叶子结点，将叶子结点的存放的类别作为决策结果。简单说来，决策树的总体流程是自根至叶的递归过程，在每个中间结点寻找一个「划分」(split or test) 属性。

参数优化：

```
# 数据集划分，进行模型的训练以及预测
X_train, X_val, y_train, y_val = train_test_split(X_fit, y, test_size=0.2, random_state=42)
# 创建决策树模型
decision_tree = DecisionTreeClassifier()
# 定义要优化的参数范围
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [None, 'sqrt', 'log2']
}
# 创建GridSearchCV对象
grid_search = GridSearchCV(decision_tree, param_grid, cv=4)
print('start selecting...')
# 执行网格搜索
grid_search.fit(X_fit, y)
# 输出最佳参数和对应的评分
print("The best parameters are %s with a score of %0.2f" %(grid_search.best_params_, grid_search.best_score_))
```

```
(8000, 29697)
start selecting...
The best parameters are {'criterion': 'gini', 'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2} with a score of
0.74
```

调参前：

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\tree.py"
(8000, 29697)
this model is running
this model finishes running
this model finishes running, running time: 2 seconds.
Model score: 0.166875
Model Accuracy: 0.166875
Model Precision: 0.6858069930731554
Model Recall: 0.16775159177189677
Model F1 Score: 0.14556952700434164
(2000, 29697)
```

调参后：

```
PS F:\文档\大三上\当代人工智能\MY 项目一\项目一> python -u "f:\文档\大三上\当代人工智能\MY 项目一\项目一\tree.py"
(8000, 29697)
this model is running
this model finishes running
this model finishes running, running time: 140 seconds.
Model score: 0.780625
Model Accuracy: 0.780625
Model Precision: 0.7826955506116499
Model Recall: 0.7793952552804013
Model F1 Score: 0.7802164818141339
(2000, 29697)
```

观察输出结果可以发现，MLP 模型的运行时间远大于决策树模型，分数也高很多。

对比两者的输出内容有什么差异：

```
使用 MLP 和决策树训练出来的数据有： 1124 行的不同。
320, 0

1019, 8

975, 7
```

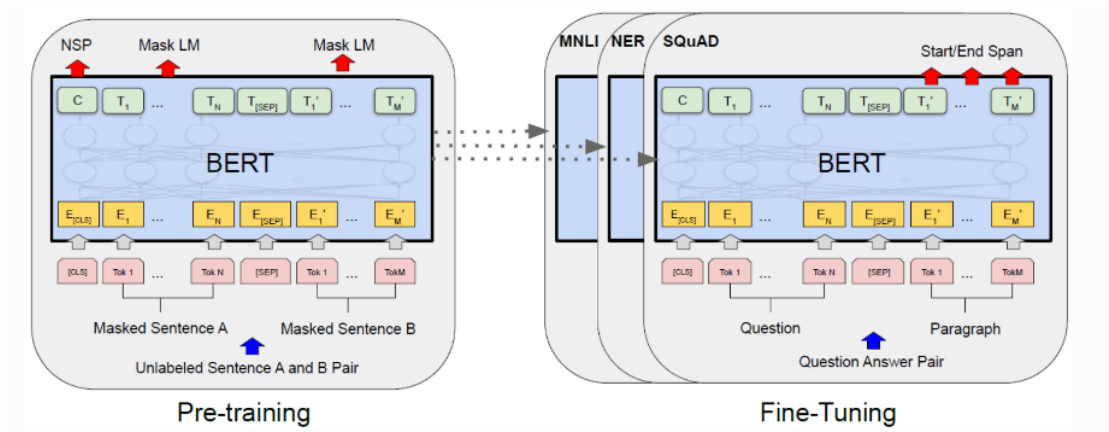
两个模型输出的差异有 1124 行，占整个测试集的一大半。

高维数据集通常具有大量特征，如在我们的情况中，有 29697 个特征。从原理上来说，这使得决策树模型难以捕捉特征之间的复杂关系，因为它需要在每个节点上选择一个特征进行分割。容易生成深度较大的树，**尤其在高维数据中，这可能导致过拟合，因为树变得复杂，适应训练数据但泛化性能较差**。MLP 模型使用多层神经网络结构，每一层可以自动学习数据中的特征表示。高维数据中可能存在各种复杂的特征和模式，MLP 通过深度结构能够更好地捕捉这些复杂性。**高维数据中，MLP 的神经元能够适应更多种类的特征，这增加了模型的**

表达能力。高维数据可能在高维空间中更加分散，这对于决策树来说增加了**构建明确决策边界的困难**。决策树需要在每个维度上找到最佳分割点，这在高维情况下更加复杂。MLP 模型在学习高维数据的非线性决策边界方面更具优势。它可以通过多层非线性激活函数适应更复杂的数据分布，**并且经过我的自动化超参数调参，原本 MLP 模型中默认使用的激活函数从 Relu——修正线性单元激活函数变换成了 tanh——双曲正切函数。**此外，MLP 模型可以通过正则化技巧，如 Dropout 和 L2 正则化，有效地控制过拟合问题。在高维数据中，过拟合风险更大，而正则化有助于提高模型的泛化性能。

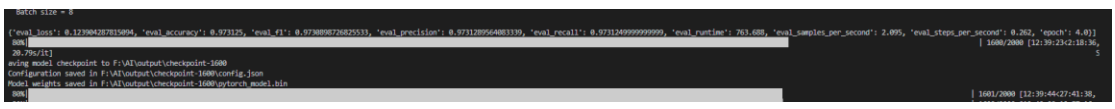
5、 Bert 模型

在这里就不详细介绍 Bert 模型的工作原理了，模型比前四个都要复杂，三言两语介绍不清楚，我们来看看 coding 和结果。



我的 Bert 模型代码主要由三部分组成：train_model.py use_model.py best_model

train_model: 初始化：首先，初始化一个名为`BertTrain`的类。在此类的`__init__`方法中，它加载了预训练的 BERT 模型，并为分类任务设置了 10 个类别。此外，它还加载了一个 BERT 分词器。数据加载：在`load_data`方法中，该脚本从给定的路径加载数据，将数据集拆分为训练集和验证集，并使用 BERT 分词器对数据进行编码。评估指标：在`compute_metrics`方法中，该脚本定义了如何计算分类任务的各种评估指标，包括精确度、召回率、F1 得分和准确率。训练：`train`方法首先加载数据，然后设置训练参数，并使用`transformers`库中的`Trainer`类进行模型的训练。一个 epoch 意味着模型在训练过程中**已经看过了整个训练数据集一次**。在一个 epoch 中，模型的每个参数都会根据它的梯度（和设置的学习率）进行一次更新。训练模型的过程通常包括多个 epoch，以便模型有足够的机会学习训练数据集中的规律。**我一共设置了 5 轮训练，即 num_train_epochs=5，这个参数设置在机器学习训练中指定了模型应该在整个训练数据集上迭代的次数。这里的数字 5 表示模型将遍历整个训练数据集五次。每次遍历整个训练数据集的过程称为一个“epoch”。**训练过程中每个参数将进行 5 次更新。每一次训练结束都会输出这个样的分数。



（有点小，可以放大看，见谅）

最后，它保存训练好的模型到指定路径。执行：脚本的最后部分定义了当该脚本作为主程序

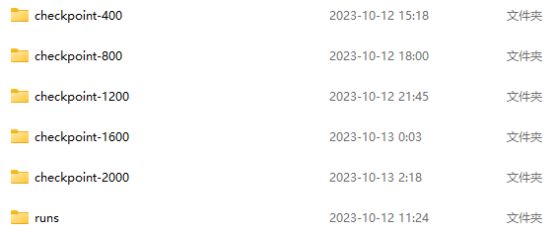
运行时应该做什么。它创建了`BertTrain`的一个实例，并开始模型的训练。

use_model.py：定义了一个 BertPredict 类，首先从 best_model 加载一个训练好的 BERT 模型和 test.txt 测试数据集。然后，它使用该模型对测试数据集进行预测，并将预测结果保存为 results.txt 文件。读入测试数据时，需要使用 BERT 分词器将其编码，并使用 datasets 库的 Dataset 类将编码后的数据格式化。

```python
# 读入测试数据
def load_data(self):
    with open(self.data_path, 'r', encoding='utf-8') as f:
        data = []
        for line in f:
            data.append(line.replace('\n', '').split(', ', 1))
        dataset = pd.DataFrame(data[1:][:], columns=data[0])

    encodings = self.tokenizer(  # tokenize
        list(dataset['text']),
        truncation=True,
        padding='max_length',
        max_length=256
    )
    # 拼接成 Dataset 并且格式化为 torch 规范并能被 BERT 接受的数据集
    test_set = Dataset.from_dict(encodings)
    test_set.set_format(
        type='torch',
        columns=['input_ids', 'token_type_ids', 'attention_mask'])
    return test_set
```

best_model 文件夹：**是我手动创建的**，里面由已经训练好的模型和参数，包括 config.json、pytorch_model.bin、training_args.bin（因为现场训练实在是太慢了！）。等 train_model.py 运行结束以后，会出现一个 output 文件夹，里面又会有五个文件夹，分别是 **checkpoint-400、checkpoint-800、checkpoint-1200、checkpoint-1600 和 checkpoint-2000。是五轮训练后的输出结果。**



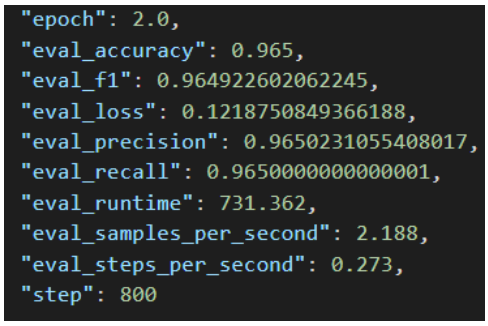
同时后面的文件也可以在 log_history 里**查看之前几轮的历史分数，我们可以发现模型的得分一直在优化。**

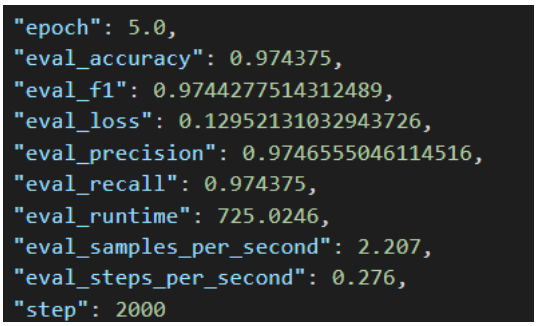
我们来看看训练结果：

结果告诉我，checkpoint-800 的模型是最好的，它有最棒的 best_metric。

```
"best_metric": 0.1218750849366188,
"best_model_checkpoint": "F:\\AI\\output\\checkpoint-800",
```

有趣的是当我查看准确率、精确率、召回和 F1 分数时，我发现 checkpoint-2000 的模型才是得分最高的，但是代码却告诉我 checkpoint-800 的模型才是最优秀的。





（左为 checkpoint-800，右为 checkpoint-2000）

对此我观察了一下各个分数，**我发现选择一个好的模型可能并不能只看 accuracy、precision、recall 和 F1 这些指标，而这里的最佳模型，指的是有最低的验证损失。**在机器学习中，损失函数度量模型的预测值与真实值之间的差异。损失值越低，模型的预测越接近真实值。不同的任务和模型架构可能使用不同的损失函数。例如，分类任务可能使用交叉熵损失，而回归任务可能使用均方误差损失。**同时，对于 epoch 我也有一些疑问和思考，按理来说，不是遍历整个训练数据集的次数越多，模型越成熟吗？**这个观点很符合我的基础认知，类似于**“见得多了就熟了”。**

但其实并不然，比如结果就告诉我 **epoch=2 时比 epoch=5 的情况更好**。对此通过查阅资料，我的解释是：

在某些情况下，**过多的 epoch 可能会导致模型过拟合，这意味着模型在训练数据上表现得太好，但在未见过的验证/测试数据上表现不佳。**因此，选择合适的 epoch 数量通常需要基于验证集的性能来进行调整。在许多情况下，我们会在训练过程中监控验证集的性能，并在性能不再提高时提前停止训练，这种策略称为提前停止（early stopping）。在提供的代码片段中，通过 load_best_model_at_end=True 参数实现了在每个 epoch 后加载在验证集上表现最好的模型，所以我需要相信代码告诉我的结果。正好这学期统计方法与机器学习的课上也刚刚讲过欠拟合和过拟合的知识，这对我在这个疑问上的解答和思考也有一定帮助，**原来模型“见到”训练数据集的次数不是越多越好。**

同时我们可以发现五轮的“相见”，Bert 模型表现比之前的四个模型都要好。

最后再来思考一下为什么 Bert 模型这么优越：

BERT 是一个深度双向变换器模型，它通过在大规模文本数据上的预训练，获得了丰富的词汇和语言表示，能够抓住词汇的微妙语义以及上下文中的语言规律，能够在特定任务（例如文本分类）上进行微调，**利用在预训练阶段学到的语言知识提高模型性能。BERT 能够理解单词的双向上下文**（即考虑到一个单词左侧和右侧的所有单词），而传统机器学习模型往往忽略了词语在文本中的上下文信息或者只能考虑到局部上下文。由于 BERT 能理解单词的上下文，它能较好地**处理多义词（同一单词在不同上下文中的不同含义）**，这是传统方法难以做到的。BERT 能理解**文本中的长距离依赖关系**（即文本中距离较远的词汇之间的关系），这通常对于理解文本含义至关重要。BERT 是一个**端到端的模型**，直接从原始文本输入到分类输出，无需复杂的特征工程。而传统机器学习模型通常依赖于手工或者基于统计的特征抽取技术。

遇到的问题和思考：

1、首先是调参的方法选取，我选择的是 GridSearchCV，这是一种自动化的网格搜索调参方法。但是它的运行时间非常长（至少相对参数固定时，模型的运行时间而言）。尤其是对 MLP 模型进行调参时，该 cell 几乎跑了十几个小时。对此我的解释是：**数据集维度高**，当数据集的特征数量非常高时，参数搜索空间也随之变得巨大。GridSearchCV 需要遍历所有可能的参数组合，这将导致组合数量呈指数级增长，从而增加搜索时间。同时数据集包含大量样本，交叉验证的次数也会增多，导致搜索时间变长。并且 MLP 模型相比于其他模型，更为复杂，计算每个参数组合的代价高，那么搜索时间会增加。所以逻辑回归这种线性模型的调参时间就比 MLP 这种神经网络模型快得多。**广泛的参数搜索范围，**如果搜索的参数空间非常广泛，即每个参数具有大范围的可能取值，GridSearchCV 需要尝试更多的组合，这会导致运行时间较长。比如逻辑回归模型中只需要找到最优的正规化参数 C，而 MLP 模型需要找到最优的隐藏层结构、激活函数、正规化参数和最大迭代次数。**所以，为了减少调参时间，可以采取的方法有降维、特征选择、并行化、缩小参数空间、分步搜索等。**另外，也可以考虑使用 RandomizedSearchCV 方法，它在参数空间中随机采样参数组合，通常比 GridSearchCV 更高效。

2、模型的训练时间有点长，每次运行的代码中我都会记录模型训练需要的时间，其实最多也没有超过 1000 seconds。但这只是一个本科课程实验，如果放到业界的实际运用中来看，数据集会大得多，模型也会复杂很多，如果每次预测都需要训练一次模型，那基本上就一直在训练了。**对此我想到的方法是，可以将训练好的模型保存在本地，预测模型的时候直接用**

```
# 保存该 model, save the model as a pickle object in Python
with open('text_classifier', 'wb') as picklefile:
    pickle.dump(model, picklefile)
```

```
# 从本地加载训练好的模型
# with open('text_classifier', 'rb') as training_model:
#     model = pickle.load(training_model)
```

3、预测Precision、Recall和F1分数的时候，报这样的错

```
Traceback (most recent call last):
  File "f:\文档\大三上\当代人工智能\MY 项目一\项目一\LR.py", line 74, in <module>
    precision = precision_score(y_val, model.predict(X_val))
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\86138\anaconda3\Lib\site-packages\sklearn\utils\_param_validation.py", line 211, in wrapper
    return func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\86138\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py", line 2127, in precision_score
    p, _, _, _ = precision_recall_fscore_support(
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\86138\anaconda3\Lib\site-packages\sklearn\utils\_param_validation.py", line 184, in wrapper
    return func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\86138\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py", line 1721, in precision_recall_fscore_support
    labels = _check_set_wise_labels(y_true, y_pred, average, labels, pos_label)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\86138\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py", line 1516, in _check_set_wise_labels
    raise ValueError(
ValueError: Target is multiclass but average='binary'. Please choose another average setting, one of [None, 'micro', 'macro', 'weighted'].
```

因为精确率、召回和 F1 分数通常用于二进制分类问题（即两个类别的分类）。但题目是一个十分类的问题，**所以需要制定使用"average"参数来指定如何计算这些指标。**

4、 在使用 GridSearchCV 搜索最佳超参数时，运行结果告诉我 LR 模型的最优超参数是 C=10

```
grid = GridSearchCV(LogisticRegression(max_iter=300), param_grid={"C":[1, 2, 5, 10], "penalty":['l2']}, cv=4)
print('start selecting...')
grid.fit(X_fit, y)
print("The best parameters are %s with a score of %0.2f" %(grid.best_params_, grid.best_score_))
```

```
start selecting...
The best parameters are {'C': 10, 'penalty': 'l2'} with a score of 0.90
```

但我心血来潮改成了 C=8，却发现 C=8 时的准确率要比 C=10 时来得更高！

左为 C=10、右为 C=8

```
(8000, 29697)
this model is running
this model finishes running, running time: 30 seconds.
Model score: 0.95125
Model Accuracy: 0.95125
Model Precision: 0.9516721781732581
Model Recall: 0.9506536394663371
Model F1 Score: 0.9508598430311445
```

```
(8000, 29697)
this model is running
this model finishes running, running time: 38 seconds.
Model score: 0.951875
Model Accuracy: 0.951875
Model Precision: 0.9523261007604263
Model Recall: 0.9513900870167541
Model F1 Score: 0.951525351469105
```

我尝试解释的原因有：

GridSearchCV 使用交叉验证方法来评估每种参数组合的性能。尽管交叉验证可以提供对模型性能的不错估计，但它并不保证在所有可能的数据集上都是最优的。**训练/测试数据的分布：**如果训练数据和测试数据的分布存在差异，模型在训练数据上的最佳参数可能不适用于测试数据。**过拟合：选择的最佳参数可能导致模型在训练数据上过拟合，但在未见过的数据上性能不佳。**参数网格的密度：我提供的参数网格是[1, 2, 5, 10]，而实际的最佳参数 C=8 并没有包含在内。**这意味着我可能需要更密集的网格**，或考虑使用如 RandomizedSearchCV 这样的方法，它会随机选择参数值。**随机性**：有些模型（尤其是那些使用随机初始化的模型，如神经网络）在不同的训练周期可能会有不同的性能。因此，有时候稍微不同的参数可能会产生相似的交叉验证得分。对此，我想到的解决方案有：更密集的参数网格：考虑使用更密集的参数网格，或者使用随机搜索；使用不同的交叉验证策略：例如，考虑使用分层的交叉验证；验证数据：除了训练和测试数据，还可以使用一个验证数据集来调整参数。

5、 在运行 bert 模型的 train_model.py 时报错

```
ValueError: --load_best_model_at_end requires the save and eval strategy to match, but
found
- Evaluation strategy: IntervalStrategy.NO
- Save strategy: IntervalStrategy.STEPS
```

这是由于在 TrainingArguments 中的评估策略与保存策略不匹配导致的。当设置 load_best_model_at_end=True 时，需要确保评估策略和保存策略是相同的。但在我的代码中，由于设置了 do_eval=False（不进行评估），所以默认的评估策略是 IntervalStrategy.NO。而默认的保存策略是 IntervalStrategy.STEPS。**为了解决这个问题：**我设置评估策略为每个 epoch 结束时评估，即 evaluation_strategy='epoch'。将 save_strategy 也设置为'epoch'。

```
per_device_eval_batch_size=8,
do_train=True,
do_eval=True,  # 开启评估
evaluation_strategy='epoch',  # 每个epoch结束时评估
save_strategy='epoch',  # 每个epoch结束时保存
no_cuda=False,
load_best_model_at_end=True,
learning_rate=3e-5,
output_dir='F:\AI\output',
overwrite_output_dir=True,
```

实验总结：

本次实验是我大学本科期间的第一个小型的机器学习项目，有很多不成熟和不严谨的地方，也收获了很多，**特别是打破了我对机器学习模型“唯 accuracy”论的印象**。

本次实验中，就分数而言，表现最优益的是 Bert 模型，但是逻辑回归模型也不错，并且训练时间很短，综合考虑的话两者都有优点。过程中我首先都对五个模型的原理稍作了解，并且在训练的分数出来之前有一个大致的预判，虽然了解模型的函数和原理不是实验的主要目的，但是稍微有一点背景知识让我在原因分析的时候更加有针对性。Bert 模型训练了很久，也是我唯一一个保存下来的已经训练完毕的模型，不愧是成为 NLP 发展史上的里程碑式的模型成就，但是没有对 best_model 进行超参数调参是比较遗憾的事情。本次实验中我将逻辑回归模型和 SVM 模型分为一组，MLP 模型和决策树模型分为一组，原因是**逻辑回归和 SVM 都有较强的数学基础，并且都是优化问题。**逻辑回归的目标是最小化逻辑损失函数，而 SVM 的目标是找到最大间隔的超平面。它们在优化和数学理论上有一些共通之处，例如都可以使用拉格朗日乘子法进行优化。**MLP（多层感知器）和决策树的数学基础相对较为复杂。**MLP 基于神经网络，其优化通常涉及到反向传播算法和梯度下降等复杂的数学概念。决策树是基于树结构的非参数模型，其构建过程涉及信息增益或基尼系数等决策规则，数学上也较为复杂。**逻辑回归和 SVM 都是线性模型**，通常用于处理线性或接近线性可分的问题。它们在处理具有清晰边界的数据时表现良好，例如二元分类问题。**MLP 和决策树更适合处理非线性关系和复杂的数据**。MLP 可以通过深度学习来学习非线性映射，而决策树可以捕捉特征之间的非线性关系。逻辑回归和 SVM 通常在小到中等规模的数据集上表现良好，而且对于高维数据也比较适用。MLP 和决策树在大规模数据和高维数据上也能够表现良好，因为它们可以适应更复杂的模式。以后可以尝试很多的分组类型或者更多模型。