# 实验一：文本分类

**10215501433 仲韦萱 实验报告**

## 实验目的：

本实验旨在进行文本分类任务，通过训练数据的划分、特征提取（**TF-IDF/Word2vec**）和分类器的训练，探索不同分类器（**SVM/MLP/DTC/Logistic**）在10分类任务中的性能表现，并分析实验结果。

## 实验方法：

- 数据集划分：将已有的数据集自行划分为训练集和验证集，确保两者之间没有重叠的样本，以充分评估模型的泛化能力。
- 特征提取：采用TF-IDF方法将文本映射为向量表示，将文本的词频与逆文档频率相乘作为特征值，以捕捉每个词对于文本整体的重要程度。
- 分类器训练：分别选择SVM、MLP、DTC和Logistic回归等经典分类器进行训练，并使用网格搜索和交叉验证在训练集进行模型拟合，得到最终的分类器。

## 实验环境：

在**VSCode**中编写和运行Python代码

## 实验过程：

### 1. 读取数据集，预处理

读取给定的train_data.txt中的数据，自行划分其中的特征和标签

```
#打开txt文件并读取内容
with open('train_data.txt', 'r') as f:
    train_data = [json.loads(line.strip()) for line in f]

df = pd.DataFrame(train_data)
#划分特征和标签
X = df["raw"]
y = df["label"]
```

### 2. 划分数据集

按照一定比例将数据集划分为训练集和验证集（训练集：验证集 = 2：8）

```
#划分训练集和验证集
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
```

## 3. 文本映射成向量

使用向量器将文本数据转换为对应的向量表示。（TF-IDF/Word2vec两种方法，后文会比较两种方法对模型准确率的影响）
首先是TF-IDF向量器

```
#实例化TF-IDF向量化器，（其中首先去掉英文中的停用词，max_features参数表示特征维度，后续调参过程中不同的模型数值不同）
vectorizer = TfidfVectorizer(stop_words='english',max_features=3000)
X_train_vec = vectorizer.fit_transform(X_train)
X_val_vec = vectorizer.transform(X_val)
```

然后是WordVec2方法

```
# 定义分词函数
def tokenize(text):
    return nltk.word_tokenize(text.lower())
# 定义Word2Vec模型训练函数
def train_word2vec(sentences):
    model = gensim.models.Word2Vec(sentences, size=100, window=5, min_count=1,
workers=4)
    return model
# 定义文本向量化函数
def encode_text(text, model):
    words = tokenize(text)
    vector = []
    for word in words:
        if word in model.wv.vocab:
            vector.append(model.wv[word])
    if not vector:
        return [0] * 100
    return np.mean(vector, axis=0)
# 训练Word2Vec模型
sentences = [tokenize(d['raw']) for d in train_data]
model = train_word2vec(sentences)
# 将每个文本向量化
train_features = [encode_text(text, model) for text in X_train]
test_features = [encode_text(text, model) for text in X_val]
```

## 4. 分类器训练，调参以获取最优准确率

本次分别选取了：**逻辑回归/SVM/MLP/决策树**四种方法作为分类器训练模型，接下来将分别详细讲述四种方法的实现过程以及最后使用网格搜索+交叉验证进行的调参实现。

**逻辑回归**
logistic回归是一种广义线性回归，因此与多重线性回归分析有很多相同之处。它们的模型形式基本上相同，都具有 w‘x+b，其中w和b是待求参数，其区别在于他们的因变量不同，多重线性回归直接将w‘x+b作为因变量，即y =w‘x+b，而logistic回归则通过函数L将w‘x+b对应一个隐状态p，p =L(w‘x+b),然后根据p 与1-p的大小决定因变量的值。如果L是logistic函数，就是logistic回归，如果L是多项式函数就是多项式回归。

**为什么使用逻辑回归：**

- 逻辑回归模型是一种简单而有效的分类算法：逻辑回归是线性分类器的一种扩展，它在文本分类等任务中表现良好。它基于对数几率函数（logistic function）进行建模，能够输出样本属于某个类别的概率。并且逻辑回归模型具有较高的可解释性。
- 逻辑回归模型计算速度快：相对于其他复杂的模型（如支持向量机、神经网络），逻辑回归模型的训练和预测速度较快，特别适用于处理大规模的文本数据集。
- 过拟合的控制能力强：逻辑回归模型可以通过正则化参数来控制过拟合的程度，防止模型在训练集上过于匹配，从而提高模型的泛化能力。

**如何使用逻辑回归**

- 定义参数和参数范围：指定逻辑回归分类器的参数（这里使用了正则化参数：C，惩罚项penalty和）和参数范围，通过网格搜索来寻找最佳参数组合。
- 创建逻辑回归分类器：创建一个逻辑回归分类器对象，并设置最大迭代次数为2000（防止出现收敛警告）。
- 创建GridSearchCV对象：创建一个GridSearchCV对象，用于在给定参数范围内执行交叉验证，寻找最佳参数组合。
- 拟合训练数据：通过调用 `fit` 方法，使用训练数据拟合模型，并进行参数搜索。

```
# 定义参数网格
param_grid = {
    'vectorizer__ngram_range': [(1, 1), (1, 2)],  # 考虑不同的n-gram范围
    'classifier__C': [0.1, 1.0, 10.0,20.0]  # 尝试不同的正则化参数C
}

# 创建Pipeline，使用TF-IDF向量化器和逻辑回归分类器
pipeline = Pipeline([
    ('vectorizer', TfidfVectorizer()),  # 使用TF-IDF向量化器
    ('classifier', LogisticRegression())  # 使用逻辑回归分类器
])
# 使用网格搜索选择最佳参数组合
grid_search = GridSearchCV(pipeline, param_grid, cv=5)  # 创建GridSearchCV对象
grid_search.fit(X_train, y_train)  # 在训练集上拟合GridSearchCV对象
```

**决策树**

决策树沿着特征做切分，具有如面对“是否、好坏”等二分类任务的二叉树结构。在得出最终决策过程中，对所有数据中各特征a子决策判断的积累，使求解范围不断缩小。

**为什么使用决策树**

- 对于文本数据的分类问题，决策树模型表现良好，可以处理非线性关系以及高维稀疏性等特点。由于文本数据通常是高度稀疏的，因此采用 TF-IDF 向量化器将文本数据向量化，然后使用决策树分类器建模是一种基本但是有效的做法。
- 决策树是一种自上而下的方法，适用于具有明确结构和层次关系的数据集。对于文本数据，通常是由单词/词语组成，这些单词/词语之间有明显的关系，因此决策树能够捕获重要的词汇关联。
- 决策树需要的计算资源相对较少，训练速度快，适用于大规模的文本分类问题。并且，决策树模型不需要对特征进行缩放，也不受异常值的影响，所以可以在处理文本数据时保持较好的性能。
- 决策树在处理非线性问题时表现良好，并且可以处理不平衡的数据集。在文本分类中，同类别和异类别样本的比例经常存在不平衡情况，使用决策树可以有效地解决这一问题。
- 决策树还可以通过剪枝等方法避免过拟合问题

#### 决策树如何使用

- 将上文中已经转换好的文本向量带入决策树分类器中，使用网格搜索寻找最佳参数，对于决策树模型我选取的参数有划分标准criterion、树的最大深度max_depth、分割内部节点所需要的最小样本数min_samples_split、叶子节点上的最小样本数min_samples_leaf
- 通过交叉验证（cv=3），网格搜索评估每个超参数组合的性能，并返回最佳超参数组合。然后，使用最佳超参数组合重新构建一个新的决策树分类器，并对训练集进行拟合。

```
# 网格搜索的参数列表
param_grid = {
    'max_depth': [None, 5000],
    'min_samples_leaf': [2],
    'min_samples_split':[200],
    'criterion': ['gini', 'entropy'],
}
# 网格搜索
grid_search = GridSearchCV(DecisionTreeClassifier(), param_grid, cv=3)
grid_search.fit(X_train_vec, y_train)

# 输出最佳超参数
print("Best Parameters: ", grid_search.best_params_)

# 在验证集上评估模型性能
classifier = DecisionTreeClassifier(**grid_search.best_params_)
classifier.fit(X_train_vec, y_train)
```

### 支持向量机（SVM）

SVM的核心思想是找到不同类别之间的分界线，使得两类样本尽量落再面的两边，找到合理划分数据的超平面是该算法基本思想。

#### 为什么使用SVM

- 高维数据处理：SVM在高维度的特征空间中表现出色。对于文本分类等问题，特征往往是高维的，SVM能够处理这种情况，并通过间隔最大化来寻找最优的分类超平面。
- 处理非线性问题：通过引入核函数，SVM可以处理非线性问题。核函数将样本映射到更高维度的特征空间，使得原本线性不可分的样本在新的特征空间中线性可分，从而实现了非线性分类。
- 泛化能力强：SVM通过最大化间隔来构建分类超平面，这使得它具有较好的泛化能力。它对于训练集之外的数据也能较好地进行分类，避免了过拟合问题。
- 少样本学习：SVM适用于少样本学习，即在训练数据较少的情况下依然能够取得良好的效果。这是由于SVM的决策边界只与少量的支持向量相关，降低了对训练样本数量的依赖。
- 对噪声数据具有强鲁棒性：SVM在优化过程中通过支持向量筛选掉了其他样本的影响，因此对噪声数据具有一定的鲁棒性。

#### SVM如何使用

- 向量化后的数据上，我们使用GridSearchCV来搜索SVM模型的最佳超参数。在这里，我们指定了C（惩罚项系数）和kernel（核函数类型）的候选值，并通过交叉验证方法找到最佳的组合。
- 在找到最佳参数后，我们使用这些参数初始化SVC分类器，并在整个数据集上进行训练。

```
params = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid']
}

grid_search = GridSearchCV(SVC(), params, cv=3, n_jobs=-1)
grid_search.fit(X_train_vec, y_train)
print('Best parameters:', grid_search.best_params_)

clf = SVC(C=grid_search.best_params_['C'],
kernel=grid_search.best_params_['kernel'])
clf.fit(X_vec, y)
```

**多层感知机（MLP）**

**为什么使用MLP**

- 模型参数相对较少。相比于其他深度学习模型，如卷积神经网络（CNN）和循环神经网络（RNN），MLP模型的参数数量较少，训练速度更快。
- 具有较强的表达能力和灵活性，适用于处理复杂的非线性关系。MLP模型可以通过增加隐藏层的数量和节点数来增强模型的表达能力，从而更好地拟合训练数据。

**如何使用MLP**

- 首先定义一个名为MLP的类，该类继承自nn.Module。MLP类包含两个全连接层（self.fc1和self.fc2），使用ReLU激活函数（self.relu）对隐藏层进行非线性变换。
- 在模型训练过程中，使用了交叉熵损失函数（criterion = nn.CrossEntropyLoss()）作为模型的目标函数，优化器采用Adam优化算法（optimizer = optim.Adam(model.parameters(), lr=learning_rate)）。
- 模型训练过程中，通过循环迭代的方式进行多轮训练。每轮训练中，首先将模型参数的梯度清零（optimizer.zero_grad()），然后将训练集输入模型（outputs = model(X_train_tensor)），计算模型输出与真实标签之间的交叉熵损失（loss = criterion(outputs, y_train_tensor)），然后进行反向传播优化模型参数（loss.backward()），最后根据优化算法更新模型参数（optimizer.step()）。
- 在训练过程中，打印每轮训练的损失值。训练完成后，通过torch.max函数找到验证集上模型输出的最大值对应的类别索引（_, predicted = torch.max(predictions, 1)），并计算验证集的准确率。

```
# 转换为torch tensor
X_train_tensor = torch.tensor(X_train_vec.toarray(), dtype=torch.float32)
X_val_tensor = torch.tensor(X_val_vec.toarray(), dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.int64)

# 设置超参数
input_dim = X_train_tensor.shape[1]
hidden_dim = 100
output_dim = len(set(y_train))

learning_rate = 0.05
epochs = 20

# 定义MLP模型
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLP, self).__init__()
```

```
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 实例化模型、损失和优化器
model = MLP(input_dim, hidden_dim, output_dim)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)


# 训练模型
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}")
```

## 5. 模型评估

使用验证集评估模型在10个类别上的分类性能，计算准确率指标。并将最后的验证结果保存在results.txt文件中。
DTC、SVM、Logistic代码如下：

```
y_pred = classifier.predict(X_val_vec)
accuracy = accuracy_score(y_val, y_pred)

# 输出验证集准确率
print("Validation Accuracy: ", accuracy)
```

MLP代码如下：

```
with torch.no_grad():
    X_val_tensor = torch.tensor(X_val_vec.toarray(), dtype=torch.float32)
    y_val_tensor = torch.tensor(y_val.values, dtype=torch.int64)
    predictions = model(X_val_tensor)
    _, predicted = torch.max(predictions, 1)
    accuracy = (predicted == y_val_tensor).sum().item() / y_val_tensor.size(0)
    print("Validation accuracy:", accuracy)
```

## 6. 保存模型并进行预测

(以MLP模型的保存和预测为例)

```
with open('classifier_MLP.pkl', 'wb') as file:
    pickle.dump(classifier, file)

with open('vectorizer_MLP.pkl', 'wb') as file:
    pickle.dump(vectorizer, file)
```

```
# 打开txt文件并读取内容
with open('test.txt', 'r') as f:
    lines = f.readlines()
texts = [line.strip().split(", ")[1] for line in lines[1:]]

# 加载保存的TF-IDF向量化器,将文本数据转换为特征向量
with open('vectorizer_MLP.pkl', 'rb') as file:
    vectorizer = pickle.load(file)
X_test_vec = vectorizer.transform(texts)

# 加载保存的MLP模型
with open('classifier_MLP.pkl', 'rb') as file:
    saved_classifier = pickle.load(file)

# 预测测试集样本标签
y_pred = saved_classifier.predict(X_test_vec)

# 将预测结果写入文件
with open('predict_MLP.txt', 'w') as fw:
    fw.write("id, pred\n")
    for i, pred in enumerate(y_pred):
        fw.write("{}, {}\n".format(i, pred))
```

### 7. 实验结果

最后选取较高的预测准确率(选取SVM)的模型对测试集进行预测，结果（部分）如下图：

```
id, pred
0, 6
1, 0
2, 4
3, 1
4, 6
5, 7
6, 5
7, 4
8, 9
9, 7
10, 6
11, 2
12, 3
13, 0
14, 6
15, 9
16, 1
```

## 讨论：

在这里主要对两个文本向量化器、四个模型的调参过程以及本次实验过程中遇到的问题进行讨论：

## 文本向量化器的选择

实验过程中尝试了TF-IDF和WordVec2两种向量化器

- TF-IDF：一种常用的用于文本特征向量化的方法。它通过计算词项在文档中的频率和在整个语料库中的逆文档频率来反映词项的重要性。TF-IDF可以有效地表示文档的主题特征，并且在文本分类任务中表现出良好的性能。它是一种简单而有效的特征表示方法。
- Word2Vec：一种基于神经网络的词嵌入模型，可以将词语映射到低维度的稠密向量空间中。Word2Vec模型通过训练一个神经网络来预测词语的上下文或者预测某个词在一定窗口内的出现概率。Word2Vec能够捕捉到词语之间的语义和语法关系，能够更好地表达词语的含义和上下文信息。
  通过查询可知如果任务主要侧重于词语的重要性和特征频率，如文本分类、关键词提取等，TF-IDF更具优势。然后通过将两种方法分别带入并计算最后文本分类任务的准确率可以看到确实TF-IDF更具优势。
- 以SVM模型为例看两种向量化器对最终准确率的影响如下图，TF-IDF更优。

  TF-IDF：

```
Train set size: 6400
Validation set size: 1600
TF-IDF Validation Accuracy: 0.941875
```

  WordVec2：

```
Word2Vec Test Accuracy: 0.6225
```

## 调参分析

**逻辑回归**

涉及到的参数：

- 惩罚项$penalty$：用于控制模型的复杂度，防止过拟合。
- 正则化参数$C$：正则化项的权重，用于平衡模型的分类准确率和复杂度。C越大，惩罚项在损失函数中所占比重就越大，模型的分类准确率也会相应提高；但同时模型可能会过拟合。C越小，惩罚项的作用就越弱，模型复杂度越高，容易欠拟合。
- 逻辑回归模型求解器$solve$：优化算法，用于求解逻辑回归模型的参数。常见的求解器有：
  'liblinear'：适用于小数据集和二分类问题。它是一个基于坐标轴下降法的优化算法。
  'saga'：适用于大规模数据集和多类别问题（最新版本sklearn中也支持二分类）。支持L1和L2正则化。

在这里，未调参之前的准确率为0.939，通过网格搜索确定了相对准确率较高的参数组合,'C': 10, 'penalty': 'l2', 'solver': 'liblinear' (交叉验证折数)'cv':3, 该参数组合得到的准确率：0.95

```
Train set size: 6400
Validation set size: 1600
Best parameters:  {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
Best accuracy:  0.9473439427112923
Accuracy on validation set:  0.95
```

**决策树**

涉及到的参数：

- 划分标准$\mathrm{criterion}$：决策树的划分标准有多种，比如基尼系数和信息熵等。基尼系数在计算时更为简单，运行速度相对更快，而信息熵可以更好地考虑样本的权重以及类别不平衡的情况。选择合适的划分标准可以有效提高决策树模型的分类性能。
- 树的最大深度$\mathrm{max_depth}$：控制了决策树的复杂度，避免过拟合。

- 分割内部节点所需要的最小样本数min_samples_split：控制了决策树分割内部节点时所需的最小样本数。如果一个内部节点的样本数小于该参数，这个节点不会被分裂，也就是停止分支过程，成为叶子节点。增加这个参数的值可以使得决策树对噪声更加鲁棒。
- 叶子节点上的最小样本数min_samples_leaf：如果一个叶子节点的样本数小于该参数，这个叶子节点会和与其相邻的某些叶子节点合并。增加这个参数的值可以减小决策树的复杂度，避免过拟合。

在这里，未调参之前的准确率为0.766，通过网格搜索确定了相对准确率较高的参数组合:'criterion': 'gini', 'max_depth': 2000, 'min_samples_leaf': 2, 'min_samples_split': 100, (交叉验证折数)'cv':3该参数组合得到的准确率：0.778

```
Best Parameters:  {'criterion': 'gini', 'max_depth': 2000, 'min_samples_leaf': 2, 'min_samples_split': 100}
Validation Accuracy:  0.778125
```

**支持向量机**

涉及到的参数：

- C（惩罚项系数）：C是SVM中的正则化参数，控制了分类器误分类样本和决策边界之间的权衡。
- kernel（核函数类型）的候选值：常见的核函数类型包括线性核、多项式核和高斯核等。

在这里，未调参之前的准确率为0.9418，通过网格搜索确定了相对准确率较高的参数组合:'C': 10 , 'kernel': 'linear' , (交叉验证折数)'cv':3该参数组合得到的准确率：0.955

```
Train set size: 6400
Validation set size: 1600
Best parameters: {'C': 10, 'kernel': 'linear'}
Validation Accuracy: 0.955
```

**多层感知机**

涉及到的参数：

- 隐藏层的神经元数量hidden_dim：增加隐藏层的神经元数量可以使得模型更加复杂，提高拟合能力，但也可能导致过拟合。
- 输出层的神经元数量output_dim：输出层的神经元数量应该与类别数相同。因此，output_dim的值需要根据数据集中的类别数进行设置。
- 学习率learning_rate：控制了每次迭代更新权重的步长大小。学习率过高会导致模型在参数空间上震荡，难以收敛；学习率过低会导致模型训练缓慢，需要更多的迭代次数才能达到最优解。
- 模型的训练轮数epochs：对于复杂的模型和大规模的数据集，通常需要更多的训练轮数才能使得模型收敛，达到最优解。但是，过多的训练轮数可能会导致过拟合.

在这里，未调参之前的准确率为0.92，通过确定了相对准确率较高的参数组合:input_dim = X_train_tensor.shape[1]，hidden_dim = 110，output_dim = len(set(y_train))，learning_rate = 0.05，epochs = 10, 该参数组合得到的准确率：0.948

```
Train set size: 6400
Validation set size: 1600
Epoch 1/10, Loss: 2.3037
Epoch 2/10, Loss: 1.9905
Epoch 3/10, Loss: 1.0764
Epoch 4/10, Loss: 0.3074
Epoch 5/10, Loss: 0.0673
Epoch 6/10, Loss: 0.0148
Epoch 7/10, Loss: 0.0049
Epoch 8/10, Loss: 0.0023
Epoch 9/10, Loss: 0.0012
Epoch 10/10, Loss: 0.0013
Validation accuracy: 0.948125
```

注：MLP 模型参数较多：MLP 模型通常具有多个隐藏层和大量的参数，这增加了模型的容量，并可以更好地拟合训练数据。由于参数数量较多，使用交叉验证可能会导致计算成本显著增加。所以MLP的训练中没有使用交叉验证方法。

## 实验过程中遇到的问题&解决

- 网格搜索过程中对参数范围的选择：刚开始确定范围比较盲目，导致准确率忽高忽低。应该需要先大致确定较优参数范围（代入几个参数值比较），再设立参数网格进行搜索。
- 逻辑回归模型中参数调整报错：

1. ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
   "the coef_ did not converge", ConvergenceWarning)

   表示迭代次数过少，因此设置迭代次数为2000以解除警告

2. got %s penalty." % (solver, penalty))
   ValueError: Solver lbfgs supports only 'l2' or 'none' penalties, got l1 penalty.
   FitFailedWarning)

   表示使用了不兼容的正则化参数和求解器组合。在这种情况下，逻辑回归模型的求解器是lbfgs，而正则化参数是l1，这是不被允许的。因为在lbfgs求解器中，只支持l2或none正则化参数。需要修改代码以使用适当组合的求解器和正则化参数。例如，对于具有l1正则化的逻辑回归模型，应使用saga或liblinear求解器。