Deepfake Voice

王博文 522031910633

1. 任务目标

本次实验的目标是利用传统机器学习/深度学习技术对音频数据进行分类，区分真实和伪造的声音。将使用传统机器学习方法（支持向量机 `SVM` 和逻辑回归 `Logistic Regression` ）以及深度学习方法（卷积神经网络 `CNN` ）来实现这一目标。实验将包括数据预处理、特征提取、模型训练和评估等步骤。

2. 传统机器学习（SVM，Logistic Regression）

2.1 `train1.py` 的实现

2.1.1 数据集加载 & 特征提取、标准化

首先分析特征提取函数 `extract_features` ，该函数从音频文件中提取mel/mfcc/cqt特征，并进行标准化处理。

对于Deepfake语音检测任务，目标是区分真实语音和伪造语音。伪造语音可能在频谱细节、音色自然度、韵律等方面存在不自然的痕迹。

查阅资料得知，MFCC, Mel Spectrogram, 和 CQT 都是从音频信号中提取特征的常用方法：

- MFCC (Mel-Frequency Cepstral Coefficients - 梅尔频率倒谱系数)
  - 原理：模拟人耳对声音频率的感知特性（梅尔标度），通过傅里叶变换得到频谱，然后通过梅尔滤波器组滤波，取对数能量，最后进行离散余弦变换（DCT）得到倒谱系数。
  - 特点：
    - 能够有效地表示语音的静态特征，特别是音色。
    - 系数之间相关性较低，维度相对较小。
- Mel Spectrogram (梅尔频谱图)
  - 原理：与 MFCC 的初始步骤类似，将音频信号的频谱映射到梅尔标度上，得到一个表示能量在不同梅尔频率和时间上分布的图谱。它通常是计算 MFCC 的中间产物（在 DCT 之前）。
  - 特点：
    - 保留了更多的频谱细节信息，相比于 MFCC，Mel Spectrogram 更能反映音频信号的动态变化。
    - 可以更好地捕捉到伪造过程中可能引入的细微频谱失真或不自然之处。
- CQT (Constant-Q Transform - 恒定Q变换)

    - 原理：一种时频分析方法，其频率轴是对数分布的，且频率分辨率在低频段更高，时间分辨率在高频段更高。这使得它在分析音乐信号（音高、和弦）时特别有效，因为音乐的音高是对数感知的。
    - 特点：
        - 在音高分析上非常强大，能够捕捉到音频信号的谐波结构。
        - 但对于区分真假语音的细微纹理和频谱伪影，其优势可能不如梅尔频谱图明显，除非伪造主要体现在音高或谐波结构的不自然。

特征提取函数 `extract_features` 的具体代码如下：

```
def extract_feature(file_path, feature_type='mel', n_mfcc=20):
    sr, y = scipy.io.wavfile.read(file_path)
    y = y.astype(np.float32)
    # 归一化到[-1, 1]，防止溢出
    if y.dtype == np.int16:
        y = y / 32768.0
    elif y.dtype == np.int32:
        y = y / 2147483648.0
    if y.ndim > 1:
        y = y.mean(axis=1)
    if feature_type == 'cqt':
        feat = librosa.cqt(y, sr=sr)
        feat = np.abs(feat)
        return np.mean(feat, axis=1)
    elif feature_type == 'mel':
        feat = librosa.feature.melspectrogram(y=y, sr=sr)
        feat = np.log1p(feat)
        return np.mean(feat, axis=1)
    else:  # mfcc
        feat = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n_mfcc)
        return np.mean(feat, axis=1)
```

该函数的主要步骤如下：

- i. 读取音频文件，获取采样率和音频数据。
- ii. 将音频数据转换为浮点数格式，并进行归一化处理。
- iii. 根据指定的特征类型（ `mel` 、 `mfcc` 或 `cqt` ）提取相应的特征。
- iv. 对提取的特征进行平均处理，得到一个一维特征向量并返回。

在读取音频文件时，使用 `scipy.io.wavfile.read` 函数读取WAV格式的音频文件。同样也可以使用 `sf.read` 函数读取。二者在效果上区别很小，具体结果在 `2.3 结果分析` 部分会进行比较。

依据上述特征提取函数，我们可以实现完整的数据加载过程：

```
def load_data(data_dir, feature_type='mel'):
    X, y = [], []
    for label_name, label in [('real', 1), ('fake', 0)]:
        # 此处省略部分代码
        for fname in os.listdir(folder):
                try:
                    feat = extract_feature(fpath, feature_type=feature_type)
                    X.append(feat)
                    y.append(label)
                # 此处省略部分代码
    return np.array(X), np.array(y)
```

在 `load_data` 函数中，首先定义了一个空列表 `X` 和 `y` ，用于存储特征和标签。然后遍历数据目录中的每个子目录（ `real` 和 `fake` ），读取音频文件并提取特征，最后将特征和标签添加到列表中。

对于数据的标准化处理，使用 `StandardScaler` 进行特征的标准化，使得每个特征的均值为0，方差为1。标准化可以提高模型的收敛速度和性能。

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

2.1.2 模型训练

接下来，分析模型训练部分。使用 `scikit-learn` 库中的支持向量机（SVM）和逻辑回归（Logistic Regression）模型进行训练。

具体而言，使用了以下模型：

- Logistic Regression (逻辑回归)
  - 原理：
    - 逻辑回归是一种广义线性模型。它通过一个 logistic 函数（或称 sigmoid 函数）将线性回归的输出映射到 (0, 1) 区间，从而得到样本属于某个类别的概率。
    - 决策边界是线性的（在原始特征空间或经过变换的特征空间中）。
  - 优点：
    - 简单快速：模型简单，训练速度快，计算开销小。
    - 可解释性强：可以很容易地理解各个特征对最终结果的影响（通过查看系数）。
    - 对于线性可分或近似线性可分的数据集表现良好。
  - 缺点：
    - 线性限制：它假设特征与结果的对数几率之间是线性关系，因此难以捕捉复杂的非线性模式。如果真实决策边界是非线性的，逻辑回归可能表现不佳。
    - 对特征敏感：容易受到多重共线性的影响。
- SVM with Linear Kernel (线性核SVM)
  - 原理：

        - 支持向量机 (SVM) 的目标是找到一个最优的超平面，使得不同类别之间的间隔 (margin) 最大化。
        - 线性核意味着这个超平面在原始特征空间中是线性的。
        - 它主要关注那些离决策边界最近的样本点（即支持向量）。
    - 优点：
        - 在高维空间中表现良好：即使特征维度高于样本数量，依然有效。
        - 对于线性可分数据效果好：能找到最优的线性分界。
        - 相对RBF核，训练更快，参数更少（主要是正则化参数C）。
    - 缺点：
        - 对非线性数据不佳：如果数据不能被线性超平面很好地分开，线性SVM的效果会受限。
        - 对于大规模数据集，训练时间可能较长（尽管通常比带复杂核的SVM快）。
- SVM with RBF Kernel (RBF核SVM)
    - 原理：
        - RBF (Radial Basis Function) 核是一种常用的非线性核函数。
        - 它能将原始特征空间映射到一个更高维甚至无限维的空间，在这个高维空间中，原本线性不可分的数据可能变得线性可分。
        - 决策边界可以是高度非线性的，能够拟合复杂的数据分布。
    - 优点：
        - 处理非线性关系：能够有效地处理特征和类别之间复杂的非线性关系，这是其相对于线性模型的主要优势。
        - 灵活性高：通过调整参数 C (正则化参数) 和 gamma (核系数)，可以控制模型的复杂度和拟合能力。
    - 缺点：
        - 计算成本高：训练时间和预测时间通常比线性SVM和逻辑回归更长，尤其是在大数据集上。
        - 参数调优敏感：C 和 gamma 参数的选择对模型性能影响很大，需要仔细调优（例如通过网格搜索和交叉验证），否则容易过拟合或欠拟合。
        - 可解释性差：由于数据被映射到高维空间，模型的决策过程不如线性模型直观。

模型训练的代码如下：

- SVM with Linear Kernel

```
print("训练SVM模型(Linear)...")
clf = SVC(kernel='linear', probability=True)
clf.fit(X, y)
```

- SVM with RBF Kernel

```
print("训练SVM模型(RBF)...")
clf = SVC(kernel='rbf', probability=True)
clf.fit(X, y)
```

- Logistic Regression

```
print("训练Logistic Regression模型...")
clf = LogisticRegression(max_iter=2000)
clf.fit(X, y)
```

最后，将训练好的模型保存到文件中，以便后续测试：

```
print(f"保存模型到 {model_path}")
joblib.dump({'model': clf, 'scaler': scaler}, model_path)
```

2.2 `test1.py` 的实现

2.2.1 数据集加载 & 特征提取、标准化

这里和 `train1.py` 中的数据加载和特征提取过程相同，不做赘述。

2.2.2 模型加载

```
print("加载模型...")
model_dict = joblib.load(model_path)
clf = model_dict['model']
scaler = model_dict['scaler']
print(f"模型类型: {type(clf)}")
```

在这里，我们使用 `joblib.load` 函数加载之前保存的模型和标准化器（ `model_dict` 是一个包含了模型和标准化器的字典）。

2.2.3 模型预测 & 指标计算

```
y_pred = clf.predict(X_test)
y_prob = clf.predict_proba(X_test)[:, 1] if hasattr(clf, "predict_proba") else y_pred

acc = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
try:
    auc = roc_auc_score(y_test, y_prob)
except Exception:
    auc = 0.0

print(f"Model: {model_path}")
print(f"Accuracy {acc*100:.2f}%. F1 {f1*100:.2f}%. AUC {auc:.4f}")
```

使用 `predict` 方法进行预测，并使用 `predict_proba` 方法获取预测概率。最后计算准确率（Accuracy）、F1 值和 AUC 值，并打印结果。

2.3 结果分析

使用 `mfcc` 特征进行初步实验的结果如下：

模型	Accuracy	F1 值	AUC 值
svm_linear_sf_mfcc.joblib	85.12%	79.59%	0.8886
svm_linear_scipy_mfcc.joblib	85.25%	79.73%	0.8902
svm_rbf_mfcc.joblib	85.12%	78.48%	0.8728
logreg_mfcc.joblib	82.00%	75.34%	0.8947

首先是 `scipy.io.wavfile.read` 和 `sf.read` 的比较。可以看到， `scipy.io.wavfile.read` 的结果略好于 `sf.read` ，但差异很小。因此，在后续实验中统一使用 `scipy.io.wavfile.read` 。

接下来是不同模型的比较。三者的结果都相对较低，尤其是 `svm_linear_sf_mfcc.joblib` 和 `svm_rbf_mfcc.joblib` 的准确率和F1值都在85%左右，而逻辑回归模型的表现略差。

尝试使用 `mel` 特征进行训练和测试。

此时Logistic Regression出现了 `ConvergenceWarning` ，提示模型未收敛。因此增加 `max_iter` 参数的值，设置为2000，同时使用 `StandardScaler` 对特征进行标准化处理。

改用 `mel` 特征并标准化处理后，模型的表现有了显著提升：

模型	Accuracy	F1 值	AUC 值
svm_linear_mel.joblib	99.88%	99.83%	1.0000
svm_rbf_mel.joblib	99.88%	99.83%	1.0000
logreg_mel.joblib	99.62%	99.50%	1.0000

可以看到，使用 `mel` 特征并标准化后，所有模型的准确率和F1值都达到了99%以上，AUC值也达到了1.0000，说明模型在区分真实和伪造语音方面表现非常出色。

3. 深度学习模型（CNN）

3.1 `train2.py` 的实现

3.1.1 数据集加载 & 特征提取

这里和 `train1.py` 中的数据加载和特征提取过程非常类似，只是改用类 `AudioDataset(Dataset)` 来封装数据集：

```
class AudioDataset(Dataset):
    def __init__(self, data_dir, feature_type='mel', n_mels=128):
        self.X = []
        self.y = []
        for label_name, label in [('real', 1), ('fake', 0)]:
            folder = os.path.join(data_dir, label_name)
            for fname in os.listdir(folder):
                fpath = os.path.join(folder, fname)
                try:
                    feat = extract_feature(fpath, feature_type=feature_type, n_mels=n_mels)
                    self.X.append(feat)
                    self.y.append(label)
    # 此处省略部分代码
    def __getitem__(self, idx):
        return torch.tensor(self.X[idx]), torch.tensor(self.y[idx], dtype=torch.long)
```

该类继承自 `torch.utils.data.Dataset` ，用于加载音频数据并提取特征。 `__init__` 方法中，遍历数据目录，读取音频文件并提取特征，最后将特征和标签存储在列表中。在特征提取时，使用 `extract_feature` 函数提取 `mel` 频谱特征，并将其转换为PyTorch张量。

3.1.2 模型训练

使用卷积神经网络（CNN）进行训练。CNN在处理图像和音频数据时表现出色，能够自动提取特征并进行分类。

模型的定义如下：

```
class SimpleCNN(nn.Module):
    def __init__(self, flatten_dim):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(flatten_dim, 64)
        self.fc2 = nn.Linear(64, 2)

    def forward(self, x):
        x = x.unsqueeze(1)
        x = self.relu(self.conv1(x))
        x = self.pool(x)
        x = self.relu(self.conv2(x))
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

该模型包含两个卷积层和两个全连接层。卷积层用于提取特征，全连接层用于分类。使用ReLU激活函数和最大池化层来增加非线性和减少特征维度。

随后进行模型训练：

```
n_mels = 128
batch_size = 16
epochs = 10
lr = 1e-3
# 此处省略部分代码
model = SimpleCNN(flatten_dim=flatten_dim).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)

print("训练中...")
for epoch in range(epochs):
    model.train()
    # 此处省略部分代码
```

在训练过程中，使用交叉熵损失函数（`CrossEntropyLoss`）和Adam优化器（`optim.Adam`）。每个epoch中，先将模型设置为训练模式，然后进行前向传播、计算损失、反向传播和参数更新。

为了防止过拟合，使用了验证集来监控模型的性能，并在每个epoch结束时打印训练和验证的损失和准确率：

```
train_idx, val_idx = train_test_split(indices, test_size=0.2, random_state=42, stratify=full_da

class SubsetDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __len__(self):
        return len(self.y)
    def __getitem__(self, idx):
        return torch.tensor(self.X[idx]), torch.tensor(self.y[idx], dtype=torch.long)
```

```
print(f"Epoch {epoch+1}/{epochs} "
        f"Train Loss: {train_loss:.4f} Acc: {train_acc:.2f}% | "
        f"Val Loss: {val_loss:.4f} Acc: {val_acc:.2f}%")
```

3.1.3 模型保存

训练完成后，将模型保存到文件中，以便后续测试：

```
print(f"保存模型到 {model_path}")
torch.save(model.state_dict(), model_path)
```

3.2 `test2.py` 的实现

3.2.1 数据集加载 & 特征提取

这里和 `train2.py` 中的数据加载和特征提取过程相同，不做赘述。

3.2.2 模型加载

```
model = SimpleCNN(flatten_dim=flatten_dim).to(device)
model.load_state_dict(torch.load(model_path, map_location=device))
```

首先，构造与 `train2.py` 中相同的模型和数据处理流程。然后使用 `load_state_dict(torch.load())` 加载模型参数。

3.2.3 模型评估 & 指标计算

```
model.eval()

y_true = []
y_pred = []
y_prob = []

with torch.no_grad():
    for X_batch, y_batch in dataloader:
        X_batch = X_batch.to(device)
        outputs = model(X_batch)
        probs = torch.softmax(outputs, dim=1)[:, 1].cpu().numpy()
        preds = outputs.argmax(dim=1).cpu().numpy()
        y_true.extend(y_batch.numpy())
        y_pred.extend(preds)
        y_prob.extend(probs)
```

这里使用 `model.eval()` 方法进行预测。

最后再计算准确率（Accuracy）、F1 值和 AUC 值，并打印结果：

```
acc = accuracy_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)
try:
    auc = roc_auc_score(y_true, y_prob)
except Exception:
    auc = 0.0

print(f"Model: {model_path}")
print(f"Accuracy {acc*100:.2f}%. F1 {f1*100:.2f}%. AUC {auc:.4f}")
```

3.3 结果分析

首先分析特征提取时， `n_mels` 参数对结果的影响：

模型	Accuracy	F1 值	AUC 值
dl_mels40_epochs10.pth	96.12%	94.55%	0.9999
dl_mels128_epochs10.pth	99.88%	99.83%	1.0000

可以看到，使用默认值 `n_mels=128` 时，模型的性能得到了显著提升，准确率达到了99.88%.

随后，观察 `dl_mels128_epochs10.pth` 的训练过程：

```
Epoch 1/10 Train Loss: 0.1702 Acc: 93.53% | Val Loss: 0.0819 Acc: 97.00%
Epoch 2/10 Train Loss: 0.0722 Acc: 97.39% | Val Loss: 0.3704 Acc: 96.94%
Epoch 3/10 Train Loss: 0.1038 Acc: 95.67% | Val Loss: 0.0404 Acc: 98.67%
Epoch 4/10 Train Loss: 0.0275 Acc: 98.86% | Val Loss: 0.0118 Acc: 99.83%
Epoch 5/10 Train Loss: 0.0016 Acc: 99.97% | Val Loss: 0.0130 Acc: 99.83%
Epoch 6/10 Train Loss: 0.0012 Acc: 99.99% | Val Loss: 0.0110 Acc: 99.78%
Epoch 7/10 Train Loss: 0.0003 Acc: 99.99% | Val Loss: 0.0111 Acc: 99.78%
Epoch 8/10 Train Loss: 0.0002 Acc: 99.99% | Val Loss: 0.0117 Acc: 99.67%
Epoch 9/10 Train Loss: 0.0001 Acc: 99.99% | Val Loss: 0.0128 Acc: 99.78%
Epoch 10/10 Train Loss: 0.0001 Acc: 99.99% | Val Loss: 0.0127 Acc: 99.78%
```

发现验证集的损失和准确率在第6个epoch以后没有明显下降，可能存在过拟合现象。

尝试改变 `num_epochs` ，即减少训练轮数。

模型	Accuracy	F1 值	AUC 值
dl_mels128_epochs10.pth	99.88%	99.83%	1.0000
dl_mels128_epochs9.pth	99.12%	98.84%	0.9999
dl_mels128_epochs6.pth	99.25%	98.99%	1.0000
dl_mels128_epochs5.pth	98.12%	97.56%	0.9999

可以看到，在 `epoch=6` 之后，模型的效果始终很好。可见，只要在合适范围内，不论是否有些许的过拟合， `num_epochs` 对模型性能的影响不大。

在 `epoch=6` 之后，选定 `n_mels=128` ，模型的准确率和F1值都达到了99%以上，AUC值也很接近了1.0000，说明深度学习模型在区分真实和伪造语音方面的表现同样非常出色。

4. 总结

4.1 结果总结

传统机器学习和深度学习模型的最佳表现分别如下：

传统机器学习/深度学习	模型	Accuracy	F1 值	AUC 值
传统机器学习	svm_linear_mel.joblib	99.88%	99.83%	1.0000
传统机器学习	svm_rbf_mel.joblib	99.88%	99.83%	1.0000
深度学习	dl_mels128_epochs10.pth	99.88%	99.83%	1.0000

总而言之，此次实验非常成功。

4.2 收获&心得

本次实验让我对音频处理、特征提取以及机器学习/深度学习在语音分类任务中的应用有了更深入的理解和实践经验。

- 特征选择的关键性：
  - 实验结果清晰地表明，选择合适的音频特征对于模型性能至关重要。从最初使用 `mfcc` 特征效果不甚理想，到切换为 `mel` 频谱特征后，无论是传统机器学习模型还是深度学习模型，性能都得到了质的飞跃。这让我深刻体会到领域知识和特征工程在机器学习项目中的重要性。 `mel` 频谱图保留了更多频谱细节，更适合捕捉伪造语音可能存在的细微失真。
- 数据预处理的必要性：
  - 对提取的特征进行标准化处理，特别是对于逻辑回归这类对特征尺度敏感的模型，能够有效改善模型的收敛速度和最终性能。实验中，标准化处理帮助解决了逻辑回归模型的收敛问题，并提升了所有模型的表现。
- 模型选择与对比：
  - 传统机器学习模型（如SVM和逻辑回归）在选择了合适的特征 `mel` 并进行适当预处理后，依然能够达到非常高的准确率，甚至与深度学习模型媲美。这说明在某些任务中，简单有效的传统方法不容忽视。
  - 深度学习模型（CNN）展现了其强大的自动特征学习能力。通过调整网络结构和超参数（如 `n_mels` 和训练轮数 `epochs` ），CNN也取得了顶尖的性能。实验中对 `n_mels` 和 `epochs` 参数的调整，以及对训练过程中过拟合现象的观察和调整，都加深了我对深度学习模型训练调优的理解。
- 实践与问题解决能力：
  - 在实验过程中，遇到了诸如逻辑回归不收敛的问题，通过查阅文档和分析原因（未进行特征标准化、迭代次数不足），成功解决了问题。
  - 对比了不同的音频读取库（ `scipy` 与 `soundfile` ），虽然最终性能差异不大，但也了解了不同工具的细微差别。
  - 学习了如何使用 `joblib` 和 `torch.save/load_state_dict` 来保存和加载不同类型的模型，这对于模型的部署和复用非常重要。
  - 实验流程的规范性和完整性：从数据加载、特征提取、模型训练、验证到测试评估，遵循了完整的机器学习实验流程。通过设置验证集来监控模型训练过程，有助于及时发现并调整潜在的过拟合等问题。

总的来说，这次实验不仅让我掌握了使用不同技术手段进行语音真伪检测的方法，更重要的是锻炼了分析问题、选择合适技术方案以及动手解决实际问题的能力。

4.3 遇到的问题 & 解决方法

- 逻辑回归模型未收敛：切换为 `mel` 频谱特征后，Logistic Regression模型出现了 `ConvergenceWarning` ，提示模型未收敛。通过查阅文档，发现是由于未进行特征标准化和迭代

次数不足导致的。解决方法是对特征进行标准化处理，并适当增加迭代次数。
- 深度学习模型可能过拟合：在训练过程中发现验证集的损失和准确率在第6个epoch以后没有明显下降，可能存在过拟合现象。解决方法是减少训练轮数（ `num_epochs` ），并观察模型在不同轮数下的表现。
- 通过所给脚本安装的环境存在问题：在一开始运行 `train1.py` 时，发现使用脚本安装的部分库版本不兼容（如 `numpy` ），导致无法正常运行。解决方法是手动更新相关库的版本，
  即 `pip install --upgrade librosa soundfile numpy` 。

5. 课程建议

本次课程内容充实，实验设计合理，让我对机器学习、深度学习及其应用有了更直观和深入的认识。

对于本课程，我有以下几点建议：

- 深化深度学习调参技巧的讲解：
  - 在完成深度学习小作业时，我往往会在调参上比较迷茫，花费较多时间，结果还不太理想。感觉深度学习的调参更多的是经验主义，因此希望老师能够更系统、更深入地讲解深度学习模型调参的策略和技巧。例如，结合实践案例分享，展示调参的完整过程和思路，包括如何设定初始参数、如何迭代调整以及如何评估调参效果。而不是仅仅从理论的角度介绍。
- 实验环境配置的说明：
  - 建议在实验指导中提供统一的、经过测试的依赖包版本列表（如提供 `requirements.txt` 文件），以减少学生在环境配置上花费的时间。虽然本次实验提供了配置脚本，但还存在些许问题，需要通过自行更新库版本解决。更明确的 `requirements.txt` 使用起来会更方便。

总体而言，这是一门非常有价值的课程，感谢老师和助教的辛勤付出。希望课程能够越办越好！