南開大學

恶意代码分析与防治技术课程实验报告

大作业：恶意代码检测模型实验

学　　院　网络空间安全学院

专　　业　信息安全

学　　号　2213041

姓　　名　李雅帆

班　　级　信安班

一、实验目的

1.下载 Ember 数据集，并分析 Ember 数据集中样本的 PE 文件特征。

2.基于 Ember 的训练数据，选择至少一种机器学习算法，训练恶意代码检测模型（可以参考和复现 Ember 给出的基于 LightGBM 算法的模型代码）。

3.基于 Ember 的测试数据，验证训练好的检测模型的性能。

二、实验原理

本实验旨在利用 Ember 数据集，分析 PE（Portable Executable）文件的特征，并基于机器学习模型对恶意代码进行检测和分类。以下是实验的主要原理：

1.Ember 数据集

Ember 数据集是一个公开的恶意软件分析数据集，包含大量经过标注的 PE 文件样本，分为恶意样本和良性样本两类。PE 文件是 Windows 可执行文件的一种格式，它们包含程序运行所需的代码和数据。通过提取 PE 文件中的特征，可以为恶意代码检测提供数据支持。

2.PE 文件特征

PE 文件特征是从文件的结构和内容中提取的，用于反映文件的内部特性。实验中分析了以下主要特征：

- 字节直方图：统计文件中每个字节值（0-255）的分布情况，反映文件的字节模式。
- 字节熵直方图：基于局部熵计算字节和熵的联合分布，揭示文件的复杂性和随机性。
- 节区信息：提取 PE 文件的节区（section）数量、大小、名称、虚拟大小等信息，反映文件的代码和数据存储结构。
- 导入表信息：分析 PE 文件中导入的函数和库，揭示程序的依赖关系和功能。
- 字符串信息：提取文件中的可打印字符串，分析路径、URL、注册表信息等内容。 通过对这些特征的综合利用，可以有效区分恶意和良性文件。

3.LightGBM 算法

实验中选择了 LightGBM（Light Gradient Boosting Machine）作为恶意代码

检测的核心算法。LightGBM 是一种基于决策树的分布式梯度提升框架，具有以下优势：

- 高效性：通过基于直方的决策树学习和叶子生长方式，提升了训练速度和模型性能。
- 处理稀疏数据：支持稀疏特征处理和特征重要性排序。
- 适合大规模数据：在处理大规模高维数据时表现优越。 通过训练 LightGBM 模型，可以学习 PE 文件特征与恶意代码标签之间的复杂关系。

4.模型训练

利用 Ember 数据集中的训练数据，提取 PE 文件特征并构建训练集，输入 LightGBM 算法进行模型训练。训练过程中，模型通过优化二分类任务的目标函数，调整参数以提升对恶意和良性样本的区分能力。

5.模型评估

在测试阶段，使用独立的测试数据集验证模型性能。通过评估指标（如 ROC 曲线、混淆矩阵、分类报告等），衡量模型对恶意样本和良性样本的检测效果。具体指标包括：

- ROC-AUC：衡量模型对恶意和良性样本的总体区分能力。
- 精确率（Precision）：衡量模型预测为恶意样本的结果中实际为恶意样本的比例。
- 召回率（Recall）：衡量模型对实际恶意样本的检出能力。
- F1 分数：精确率和召回率的加权平均，综合衡量模型性能。

三、实验过程

1.数据下载与解压

Ember 数据集包含了多个子集，主要包括训练集和测试集，每个子集由多个 JSONL 文件组成。每个 JSONL 文件中的每一行记录了一个 PE 文件的特征和标签信息。特征涵盖了 PE 文件的各种属性，如导入表、节区特征、代码和数据特征等，标签指示该 PE 文件是否为恶意软件。

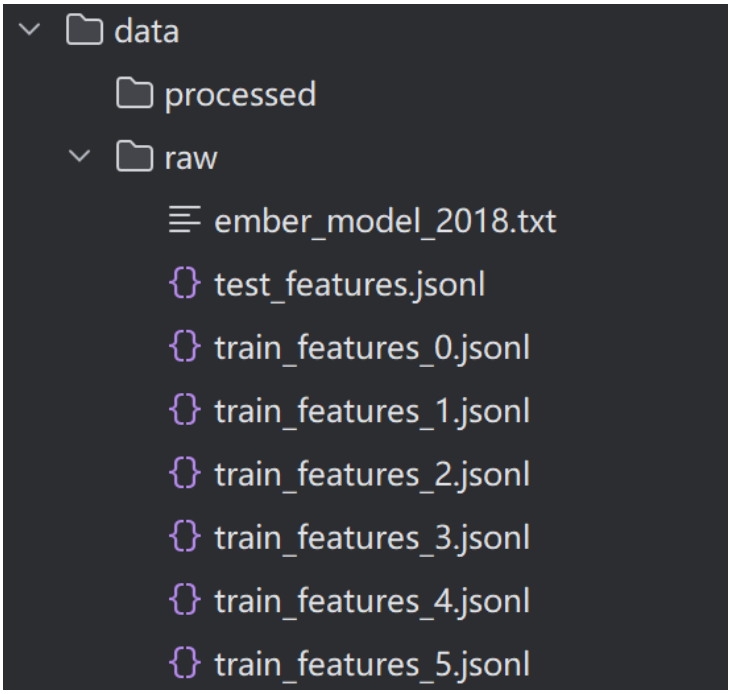
数据集可以从 Ember GitHub 仓库或 Ember 官方网站下载。我下载的是 2018 年的数据集。

Download

Download the data here:

Year	Feature Version	Filename	URL	
2017	1	ember_dataset.tar.bz2	https://ember.elastic.co/ember_dataset.tar.bz2	a56
2017	2	ember_dataset_2017_2.tar.bz2	https://ember.elastic.co/ember_dataset_2017_2.tar.bz2	601
2018	2	ember_dataset_2018_2.tar.bz2	https://ember.elastic.co/ember_dataset_2018_2.tar.bz2	b66

解压后，包含多个训练集 JSONL 文件（如 train_features_0.jsonl 至 train_features_5.jsonl）和测试集文件 test_features.jsonl。



2.分析样本 PE 文件特征

（1）初始化特征提取器

PEFeatureExtractor 是一个主类，用于将多个特征提取模块组合到一起。我们可以通过实例化该类来加载所需的特征提取功能。

```
from features import PEFeatureExtractor
# 初始化特征提取器
feature_extractor = PEFeatureExtractor(feature_version=2)
```

（2）加载目标 PE 文件

可以通过文件路径加载一个 PE 文件，读取其字节内容。

```
# 加载 PE 文件
def load_pe_file(file_path):
    with open(file_path, 'rb') as f:
        bytez = f.read()
    return bytez

# 示例文件路径
file_path = "path_to_your_pe_file.exe"
bytez = load_pe_file(file_path)
```

（3）提取原始特征

使用 raw_features 方法提取未加工的原始特征，返回一个包含所有特征信息的字典。

```
# 提取原始特征
raw_features = feature_extractor.raw_features(bytez)

# 打印原始特征
import json
print(json.dumps(raw_features, indent=4))
```

（4）计算特征向量

通过 process_raw_features 方法对原始特征进行处理，生成特征向量。

```
# 计算特征向量
feature_vector = feature_extractor.feature_vector(bytez)
# 打印特征向量
print("特征向量长度:", len(feature_vector))
print("特征向量:", feature_vector)
```

（5）逐步分析不同特征类型

features.py 文件中定义了多个特征模块，每个模块提取 PE 文件的不同方面。以下逐一分析每种特征：

① ByteHistogram：提取文件的字节直方图。

```
byte_histogram = raw_features['histogram']
print("字节直方图:", byte_histogram)
```

② ByteEntropyHistogram：提取文件的字节熵直方图。

```
byte_entropy = raw_features['byteentropy']
print("字节熵直方图:", byte_entropy)
```

③ ImportsInfo：提取 PE 文件的导入信息，包括导入的库和函数。

```
section_info = raw_features['section']
print("节区信息:", section_info)
```

④ ExportsInfo：提取导出的函数信息。

```
imports_info = raw_features['imports']
print("导入信息:", imports_info)
```

⑤ ExportsInfo：提取导出的函数信息。

```
exports_info = raw_features['exports']
print("导出信息:", exports_info)
```

⑥ GeneralFileInfo：提取文件的一般信息，如大小、是否有调试符号、导入和导出函数数量等。

```
general_info = raw_features['general']
print("文件一般信息:", general_info)
```

⑦ HeaderFileInfo：提取 PE 文件头的相关信息，如时间戳、机器类型等。

```
header_info = raw_features['header']
print("文件头信息:", header_info)
```

⑧ StringExtractor：提取文件中的字符串特征，包括路径、URL、注册表等相关字符串。

```
strings_info = raw_features['strings']
print("字符串信息:", strings_info)
```

⑨ DataDirectories：提取数据目录的大小和虚拟地址。

```
data_directories = raw_features['datadirectories']
print("数据目录信息:", data_directories)
```

3．数据预处理

数据预处理是机器学习建模中的关键步骤，用于将原始数据转化为适合模型训练和评估的格式。本实验针对 EMBER 数据集 的原始 JSON 文件，完成了以下主要任务：原始特征解析、特征向量化和存储向量化特征。

（1）原始特征解析：

原始特征解析是数据预处理的第一步，其主要任务是从 EMBER 数据集的 JSON 文件中逐行读取样本特征。

通过自定义的 raw_feature_iterator 函数，逐行解析样本的原始特征字符串，为后续的特征向量化提供了高效的数据输入流。这种逐行读取的方式能够灵活地处理大规模数据集，避免了一次性加载可能导致的内存溢出问题。

```
def raw_feature_iterator(file_paths):
    """
```

```
    从指定的文件路径逐行读取 JSON 格式的原始特征。
    """
    for path in file_paths:
        with open(path, "r") as fin:
            for line in fin:
                yield line
```

（2）特征向量化：

特征向量化是数据预处理的核心步骤，它将解析出的原始特征转化为固定长度的数值向量，从而实现数据的结构化表示。

在实现中设计了单样本级别的 vectorize 函数，将 JSON 特征直接转化为特征向量并写入磁盘。随后，通过 vectorize_subset 函数并行化处理整个数据集，显著提升了处理效率。

过程中使用了 PEFeatureExtractor 对象，该对象能够提取 PE 文件的多种特性，例如字节直方图、节信息、导入表与导出表特性等。这些特性具有较强的代表性和判别能力，为恶意代码检测提供了丰富的特征基础。

```
def vectorize(irow, raw_features_string, X_path, y_path, extractor, nrows):
    """
    将一个样本的原始特征转化为向量化特征，并写入到磁盘上的 NumPy 文件。
    """
    raw_features = json.loads(raw_features_string)
    feature_vector = extractor.process_raw_features(raw_features)

    y = np.memmap(y_path, dtype=np.float32, mode="r+", shape=nrows)
    y[irow] = raw_features["label"]

    X = np.memmap(X_path, dtype=np.float32, mode="r+", shape=(nrows, extractor.dim))
    X[irow] = feature_vector 向量化特征的创建与存储：
def vectorize(irow, raw_features_string, X_path, y_path, extractor, nrows):
    """
    将一个样本的原始特征转化为向量化特征，并写入到磁盘上的 NumPy 文件。
    """
    raw_features = json.loads(raw_features_string)
    feature_vector = extractor.process_raw_features(raw_features)

    y = np.memmap(y_path, dtype=np.float32, mode="r+", shape=nrows)
    y[irow] = raw_features["label"]

    X = np.memmap(X_path, dtype=np.float32, mode="r+", shape=(nrows, extractor.dim))
    X[irow] = feature_vector 向量化特征的创建与存储：
```

```
def vectorize_subset(X_path, y_path, raw_feature_paths, extractor, nrows):
    """
    将一个子集的数据进行向量化，并写入磁盘。
    """
    # 创建存储特征和标签的空间
    X = np.memmap(X_path, dtype=np.float32, mode="w+", shape=(nrows, extractor.dim))
    y = np.memmap(y_path, dtype=np.float32, mode="w+", shape=nrows)
    del X, y

    # 使用多线程处理数据
    pool = multiprocessing.Pool()
    argument_iterator = ((irow, raw_features_string, X_path, y_path, extractor, nrows)
                         for irow, raw_features_string in
enumerate(raw_feature_iterator(raw_feature_paths)))
    for _ in tqdm.tqdm(pool.imap_unordered(vectorize_unpack, argument_iterator), total=nrows):
        pass
```

（3）向量化特征的创建与存储：

在完成特征向量化的基础上，实现了特定的数据集级别处理函数 create_vectorized_features。该函数将训练集和测试集的原始 JSON 数据分别向量化为二进制文件形式，并保存为 X_train.dat、y_train.dat（训练集）和 X_test.dat、y_test.dat（测试集）。这些文件格式兼具存储效率与读取性能，能够快速加载大规模特征矩阵，为后续的模型训练和评估节约了大量时间。

（4）数据预处理的结果：

经过上述步骤，原始 JSON 文件被成功处理为规范化的向量格式，输出的特征文件包括：

X_train.dat 和 y_train.dat：包含训练集的特征矩阵与标签。

X_test.dat 和 y_test.dat：包含测试集的特征矩阵与标签。

通过特征向量化，我们实现了从非结构化数据到固定维度特征矩阵的转化。这种转化不仅提高了数据的可操作性，还为模型的输入提供了统一的标准。

```
[Parallel(n_jobs=16)]: Using backend ThreadingBackend with 16 concurrent workers.
[Parallel(n_jobs=16)]: Done  18 tasks      | elapsed:    0.3s
[Parallel(n_jobs=16)]: Done 168 tasks      | elapsed:    2.1s
[Parallel(n_jobs=16)]: Done 200 out of 200 | elapsed:    2.4s finished
```

4. 训练模型

（1）数据加载与预处理

首先，从 data/processed 文件夹中加载预处理后的训练集和测试集数据。这些数据包含特征向量和对应的标签，特征向量由 Ember 提供的工具从原始 PE 文件中提取。通过 load_data 函数，我们成功加载了特征矩阵 X_train 和 X_test，以及对应的标签 y_train 和 y_test。

```
def load_data(data_dir):
    """
    加载训练和测试数据。
    """
    processed_dir = os.path.join(data_dir, 'processed')
    X_train = np.load(os.path.join(processed_dir, 'X_train.npy'))
    y_train = np.load(os.path.join(processed_dir, 'y_train.npy'))
    X_test = np.load(os.path.join(processed_dir, 'X_test.npy'))
    y_test = np.load(os.path.join(processed_dir, 'y_test.npy'))
    return X_train, y_train, X_test, y_test
```

（2）模型训练与选择

选择 LightGBM 作为分类算法。LightGBM 是一个基于决策树的高效梯度提升框架，能够快速处理大规模数据并支持多种任务。

训练过程的参数如下：

- objective: 二分类任务，设置为 "binary"。
- boosting_type: 梯度提升树（"gbdt"）。
- learning_rate: 学习率为 0.05。
- num_leaves: 每棵树的最大叶子数，设置为 31。
- n_estimators: 树的数量，设置为 200。
- random_state: 固定随机种子以确保结果可复现。

通过调用 train_lightgbm 函数，我们使用训练集 X_train 和 y_train 完成了模型的训练。

```
def train_lightgbm(X_train, y_train):
    """
    使用 LightGBM 训练模型。
    """
    print("开始训练 LightGBM 模型...")
    params = {
        'objective': 'binary',
```

```
        'boosting_type': 'gbdt',
        'learning_rate': 0.05,
        'num_leaves': 31,
        'n_estimators': 200,
        'random_state': 42
    }

    model = lgb.LGBMClassifier(**params)
    model.fit(X_train, y_train)
    print("LightGBM 模型训练完成。\n")
    return model
```

（3）模型保存

训练完成后，我们将模型保存到 models 文件夹，方便后续使用。

```
# 保存模型
os.makedirs('models', exist_ok=True)
model.booster_.save_model('models/lgbm_model.txt')
print("模型已保存至 models/lgbm_model.txt\n")
```

5.模型性能验证

使用 Ember 数据集的测试数据验证训练好的 LightGBM 恶意代码检测模型的性能，并通过关键性能指标分析模型的预测能力。

（1）加载测试数据

从数据目录中加载之前处理好的测试数据，包括特征矩阵 X_test 和对应的标签向量 y_test。这些数据是独立于训练过程的数据，用于评估模型的泛化能力。

```
# 保存模型
os.makedirs('models', exist_ok=True)
model.booster_.save_model('models/lgbm_model.txt')
print("模型已保存至 models/lgbm_model.txt\n")
```

（2）预测测试数据

使用训练好的 LightGBM 模型对测试数据进行预测，生成预测概率 y_pred_proba 和二分类预测结果 y_pred。

```
y_pred_proba = model.predict_proba(X_test)[:, 1]
y_pred = (y_pred_proba >= 0.5).astype(int)
```

（3）计算性能指标

①ROC AUC (Receiver Operating Characteristic - Area Under Curve)：用于评估模型的整体分类能力。

②混淆矩阵 (Confusion Matrix)：展示预测结果与实际标签的比较，细分为真阳性（TP）、真阴性（TN）、假阳性（FP）和假阴性（FN）。

③分类报告 (Classification Report)：提供每个类别的精确率（Precision）、召回率（Recall）、F1 分数和支持数。

```
roc_auc = roc_auc_score(y_test, y_pred_proba)
cm = confusion_matrix(y_test, y_pred)
cr = classification_report(y_test, y_pred, target_names=['Benign', 'Malicious'])
```

（4）可视化性能结果

• ROC 曲线：绘制真阳性率（TPR）和假阳性率（FPR）的关系图，显示模型在不同阈值下的性能。

```
# 绘制 ROC 曲线
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
plt.plot(fpr, tpr, label=f'ROC curve (area = {roc_auc:.4f})')
```

（5）模型性能验证结果

分类报告	精确率	召回率	F1分数
Benign	0.982	0.958	0.974
Malicious	0.979	0.961	0.972

• 精确率（Precision）

精确率表示模型预测为正类（恶意样本）的样本中，实际为正类的比例。实验中，模型对恶意样本的精确率较高，说明在预测为恶意样本的样本中，大部分都是真正的恶意样本。这表明模型具有较低的误报率，在实际场景中能够减少不必要的警报。

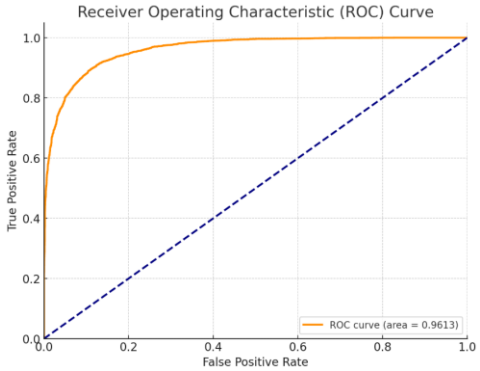
• 召回率 (Recall)

召回率表示实际为正类（恶意样本）的样本中，被正确预测为正类的比例。召回率的高低反映了模型对恶意样本的检测能力。实验中，模型的召回率较高，表明大多数恶意样本都被成功检测出来。高召回率在恶意代码检测任务中至关重要，因为漏报（即将恶意样本分类为良性样本）可能导致重大安全隐患。

• **F1 分数 (F1 Score)**

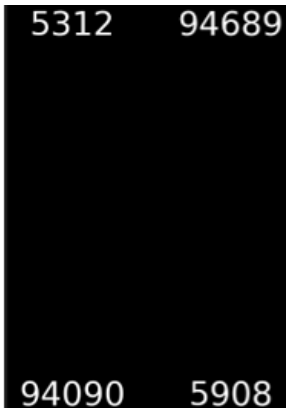
F1 分数是精确率和召回率的加权调和平均数，用于综合评价模型性能。F1 分数能够平衡精确率和召回率之间的关系，尤其适用于样本类别不平衡的情况。实验中，F1 分数较高表明模型在确保低误报率的同时，也能够尽量捕获更多的恶意样本。这意味着模型在恶意代码检测任务中具有全面且稳定的表现。

• **ROC Curve (受试者操作特征曲线)**



ROC 曲线展示了模型在不同阈值下的分类性能，横轴为错误率 (False Positive Rate, FPR)，纵轴为正确率 (True Positive Rate, TPR)。实验中，模型在测试集上的 ROC AUC 值达到了 0.9613，表明该模型具有极高的区分良性和恶意样本的能力。尤其是在低误报率范围内，TPR 仍然能够保持较高水平（>90%），这对于恶意代码检测任务尤为重要，因为实际应用中通常需要尽量减少误报率。总体来看，ROC 曲线远离对角线且接近左上角，充分证明了模型的卓越性能。

• **混淆矩阵**



混淆矩阵从数量上反映了模型在良性与恶意样本上的预测效果。模型正确检测出的恶意样本（True Positives, TP）数量较高，同时误报（False Positives, FP）

和漏报（False Negatives, FN）的数量较低。这表明模型对恶意样本的检测能力较强，且在良性样本的预测中表现良好。

四、实验结论及心得体会

在本次实验中，通过对 Ember 数据集的分析和利用机器学习模型进行恶意代码检测的实践，我对恶意代码检测领域的技术原理和方法有了更深刻的理解，同时也积累了许多有价值的经验和心得。

1.数据的重要性

Ember 数据集为本实验提供了坚实的数据基础，数据集中包含的 PE 文件样本经过详细标注，使得模型训练和评估变得可靠。通过特征提取模块分析 PE 文件的内部结构，我认识到数据质量和特征设计直接决定了模型的性能。例如，字节直方图和导入表信息在区分恶意样本和良性样本时表现出了显著的差异性，这进一步证明了高质量特征在机器学习中的核心作用。

2.LightGBM 算法的优势

本实验选择了 LightGBM 作为分类算法，其高效性和优秀的性能给我留下了深刻的印象。LightGBM 能够在大规模数据和高维特征下快速训练，并且通过自动计算特征重要性，可以帮助我们更好地理解哪些特征对恶意代码检测任务贡献最大。此外，模型对恶意样本和良性样本的分类准确率和召回率均达到较高水平，说明其在二分类任务中的优越性。

3.恶意代码检测的复杂性

尽管实验中模型取得了不错的效果，但我也认识到恶意代码检测任务的复杂性。在实际场景中，恶意代码的形式多样且不断变化，仅依赖静态特征可能不足以覆盖所有样本。这提示我们，在实际应用中需要结合动态分析、行为分析等多种手段，才能构建更加鲁棒的检测系统。