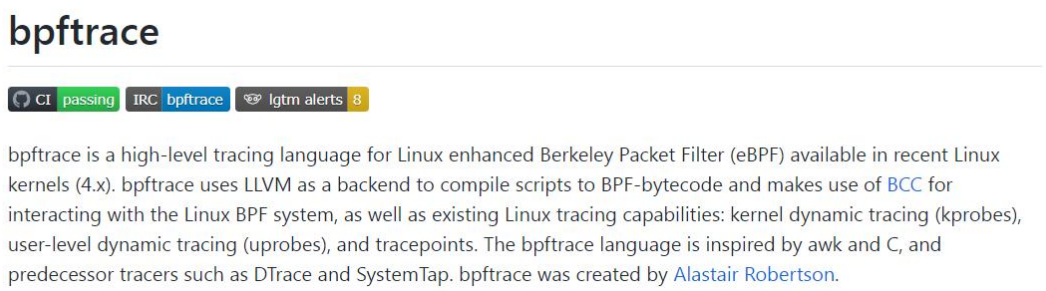
一、bpftrace-Linux 性能工具

（一）bpftrace 简介

bpftrace 是基于 ebpf 内核 vm 扩展出来的 trace 工具。

该工具基于 eBPF 和 BBC 实现了通过探针机制采集内核和程序运行的信息，然后用图表等方式将信息展示出来，帮助开发者找到隐藏较深的 Bug、安全问题和性能瓶颈。git hub 主页介绍如下：



bpftrace 是一种基于 Linux 的 eBPF 高级跟踪语言，可用于最新的 Linux 内核 (4.x)。bpftrace 使用 LLVM 作为后端将脚本编译为 BPF 字节码，并利用 BCC 与 Linux BPF 系统进行交互，以及现有的 Linux 跟踪功能：内核动态跟踪（kprobes）、用户级动态跟踪（uprobes）、和跟踪点。bpftrace 语言的灵感来自 awk 和 C，以及 DTrace 和 SystemTap 等前身跟踪器。

（二）bpftrace 使用

下面是一个使用 bpftrace 的示例：

示例	说明
bpftrace -e ‘syscall:sys_exit_read { printf(“%d %s\n”, pid, comm); }’	该命令会跟踪进程的 read 系统调用，并输出进程的 PID 和名称。
bpftrace -e ‘kprobe:do_sys_open /comm==”cat”/ { printf(“%s(%s)\n”, comm, str(arg1)); }’	该命令会在调用 do_sys_open 函数时，检查进程名称是否为“cat”，如果是则输出进程名称和打开的文件名。

（三）bpftrace 的应用场景

应用场景	功能和优势
系统性能调优与故障	通过跟踪系统调用、函数调用和硬件事件等，帮助用户分析系统性

排查	能问题和故障。
网络分析与安全监控	通过捕获网络数据包和分析网络协议，帮助用户进行网络流量分析和安全监控。
应用程序性能分析	通过跟踪应用程序的系统调用和函数调用，帮助用户分析应用程序的性能瓶颈和优化方向。

（四）选择 bpftrace 作为数据采集工具的原因

1、无侵入性：bpftrace 允许在不修改应用程序代码的情况下进行监控，这意味着您可以监控正在运行的系统或应用程序，而无需担心引入额外的复杂性或潜在的错误。

2、高性能：由于 bpftrace 在内核层面执行，它减少了对应用性能的影响。这使得 bpftrace 成为在生产环境中进行实时监控和分析的理想工具，因为它不会因为监控活动而显著降低系统性能。

3、细粒度监控：bpftrace 能够精确到单个系统调用或内核函数的监控，这为深入分析系统行为提供了可能。无论是性能调优、故障诊断还是安全审计，bpftrace 都能提供必要的详细信息。

4、实时监控能力：bpftrace 能够实时监控系统调用、磁盘 I/O、网络活动等，帮助开发者快速定位性能瓶颈和故障点。

5、简洁而强大的脚本语言：bpftrace 的脚本语言设计简洁而强大，支持复杂的追踪逻辑和数据聚合，使得编写和运行跟踪脚本变得非常容易。

6、丰富的内置功能：bpftrace 内置了许多实用的功能，如定时采样、直方图统计、聚合计数、调用栈跟踪等，可以轻松处理各种常见的跟踪需求。

7、可扩展性：bpftrace 支持借助 eBPF 技术进行动态追踪和事件捕获，可以根据需要自定义和开发更高级的跟踪功能。

基于以上诸多优势，因此本项目选择 bpftrace 作为数据采集工具。

二、项目结构

本项目分为两部分，分别为数据采集部分与结果预测部分。其中数据采集部分使用 bpftrace 编写(runqlen.bt/oepnsnoop.bt/biosnoop.bt)。结果预测部分由 python 部分编写，其中 runqlen.py 使用逻辑回归进行预测，opensnoop.py 使用了向量化以及随机森林的方法。

三、具体实现细节

（一）数据收集部分

1、runqlen.bt

主要功能是监控和统计当前系统的 CFS 运行队列长度。具体步骤如下：

（1）初始化提示：在脚本开始时，打印一条消息告知用户采样频率和终止方式。

（2）定时采样：以 50Hz 的频率定期采样。

（3）获取运行队列长度：

- 获取当前任务的 task_struct。
- 从 task_struct 中提取所属的 CFS 运行队列。
- 获取运行队列中正在运行的任务数量，并减去自身任务，得到实际等待运行的任务数。

（4）记录数据：将运行队列长度记录到直方图 @runqlen 中，以便后续分析和可视化。

代码实现如下：

```
#ifndef BPFTRACE_HAVE_BTF
#include <linux/sched.h>

struct cfs_rq {
  struct load_weight load;
  unsigned int nr_running;
  unsigned int h_nr_running;
};
#endif

BEGIN
{
  printf("Sampling run queue length at 99 Hertz... Hit Ctrl-C to end.\n");
}

profile:hz:50
{
  $task = (struct task_struct *)curtask;
  $my_q = (struct cfs_rq *)$task->se.cfs_rq;
  $len = (uint64)$my_q->nr_running;
  $len = $len > 0 ? $len - 1 : 0;  // subtract currently running task
  @runqlen = lhist($len, 0, 100, 1);
```

```
}
```

2、oepnsnoop.bt

主要功能是追踪系统调用 open 和 openat 的执行情况，并输出每次调用的详细信息，包括进程 ID、进程名、文件描述符、错误码和打开的文件路径。具体步骤如下：

（1）打印提示信息和表头，用于显示追踪结果。

（2）追踪 open 和 openat 系统调用进入事件。

（3）追踪 open 和 openat 系统调用退出事件。

（4）清除关联数组 @filename，释放资源。

代码实现如下：

```
BEGIN
{
  printf("Tracing open syscalls... Hit Ctrl-C to end.\n");
  printf("%-6s %-16s %4s %3s %s\n", "PID", "COMM", "FD", "ERR", "PATH");
}

tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat
{
  @filename[tid] = args->filename;
}

tracepoint:syscalls:sys_exit_open,
tracepoint:syscalls:sys_exit_openat
/@filename[tid]/
{
  $ret = args->ret;
  $fd = $ret >= 0 ? $ret : -1;
  $errno = $ret >= 0 ? 0 : - $ret;

  printf("%-6d %-16s %4d %3d %s\n", pid, comm, $fd, $errno,
    str(@filename[tid]));
  delete(@filename[tid]);
}

END
```

```
{
    clear(@filename);
}
```

3、biosnoop.bt

主要功能是追踪块设备的 I/O 操作，记录每个 I/O 请求的开始时间和结束时间，并计算其延迟（以毫秒为单位）。具体步骤如下：

（1）打印表头，用于显示追踪结果的列标题。

（2）追踪开始：当 I/O 操作开始时，记录当前时间、进程 ID、进程名和相关磁盘名称。

（3）追踪结束：当 I/O 操作结束时，计算延迟，打印详细信息，并清理记录的数据。

（4）结束清理：在脚本结束时，清除所有关联数组。

代码实现如下：

```
#ifndef BPFTRACE_HAVE_BTF
#include <linux/blkdev.h>
#include <linux/blk-mq.h>
#endif

BEGIN
{
    printf("%-12s %-16s %-7s %-6s %7s\n", "TIME(ms)", "COMM", "DISK", "PID", "LAT(ms)");
}

kprobe:blk_account_io_start,
kprobe:__blk_account_io_start
{
    @start[arg0] = nsecs;
    @iopid[arg0] = pid;
    @iocomm[arg0] = comm;
    @disk[arg0] = ((struct request *)arg0)->q->disk->disk_name;
}

kprobe:blk_account_io_done,
kprobe:__blk_account_io_done
/@start[arg0] != 0 && @iopid[arg0] != 0 && @iocomm[arg0] != ""/

{
```

```
        $now = nsecs;
        printf("%-12u %-16s %-7s %-6d %7d\n",
            elapsed / 1000000,@iocomm[arg0], @disk[arg0], @iopid[arg0],
            ($now - @start[arg0]) / 1000000);

        delete(@start[arg0]);
        delete(@iopid[arg0]);
        delete(@iocomm[arg0]);
        delete(@disk[arg0]);
    }

    END
    {
        clear(@start);
        clear(@iopid);
        clear(@iocomm);
        clear(@disk);
    }
```

（二）结果预测部分

1、runqlen.py

主要功能是从文本中提取特征，训练一个逻辑回归模型，并使用该模型进行预测。

（1）extract_features 函数

- 功能：从输入的文本中提取特征。
- 方法：使用正则表达式查找所有匹配的模式（两个数字之间由空格分隔），这些模式可能表示某种计数（例如，runqlen 的计数）。
- 输出：一个长度为 100 的列表，其中每个索引代表一个特征（可能是 runqlen 的值），列表中的值代表该特征的计数。

具体实现：

```
def extract_features(text):
    pattern = r'(\d+)\s+:\s+(\d+)'
    matches = re.findall(pattern, text)
    features = [0] * 100  # 假设 runqlen 最大为 100
    for match in matches:
        index, count = int(match[0]), int(match[1])
        if index < len(features):
```

```
            features[index] = count
        return features
```

（2）read_file 函数

- 功能

该函数的功能是读取指定文件的内容，并将其作为字符串返回。

- 方法
  - 打开文件：使用 with open(file_path, 'r') as file: 语句以只读模式（'r'）打开文件。with 语句确保文件在操作完成后自动关闭。
  - 读取文件内容：使用 file.read() 方法读取文件的全部内容，并将其存储在变量 content 中。
  - 返回文件内容：将读取到的文件内容作为字符串返回。
- 输出
  - 返回值：文件内容的字符串表示。
  - 异常处理（在增强版本中）：
    - 如果文件不存在，打印错误信息 "文件 {file_path} 未找到。"
    - 如果没有权限读取文件，打印错误信息 "没有权限读取文件 {file_path}。"
    - 对于其他异常，打印错误信息 "读取文件 {file_path} 时发生错误：{e}"

具体实现：

```
def read_file(file_path):
    with open(file_path, 'r') as file:
        content = file.read()
    return content
```

（3）convert_to_formatted_string 函数

- 功能

该函数的功能是将读取的文件内容转换成一种特定的、格式化的字符串格式，便于后续的处理或展示。

- 方法
  - 正则表达式提取：
    - 使用 re.compile 编译一个正则表达式模式 r'\[(\d+), (\d+)\)\s+(\d+)\s+\|(.*?)\|'，用于匹配和提取文件内容中的特定部分。
    - 使用 pattern.findall(content) 查找所有匹配的部分，返回一个匹配结果的列表。
  - 构建格式化字符串：
    - 初始化一个空列表 formatted_lines 用于存储每一行的格式化结果。
    - 遍历匹配结果 matches，对于每一个匹配项：
      - 提取 start, end, count, distribution 四个部分。
      - 计算 runqlen 为当前匹配项的索引 i。
      - 将 count 转换为整数。
      - 去除 distribution 两端的空白字符。

- 使用格式化字符串 f"{runqlen:>8} : {count:>8} |{distribution}|" 构建每一行的格式化结果，并添加到 formatted_lines 列表中。

- 拼接所有行：
  - 使用 " ".join(formatted_lines) 将所有格式化后的行拼接成一个完整的字符串。
  - 在拼接的字符串前添加一行标题 " runqlen : count distribution "。
- 返回格式化字符串：
  - 返回最终拼接好的格式化字符串。

- 输出
  - 返回值：一个格式化后的字符串，包含以下内容：
    - 一行标题 " runqlen : count distribution "。
    - 多行数据，每行数据的格式为 "{runqlen:>8} : {count:>8} |{distribution}|"，其中：
      - runqlen 是从 0 开始的索引。
      - count 是提取的计数值。
      - distribution 是提取的分布信息，去除了两端的空白字符。

具体实现：

```
def convert_to_formatted_string(content):
    # 使用正则表达式提取所需的部分
    pattern = re.compile(r'\[(\d+), (\d+)\)\s+(\d+)\s+\|(.*?)\|')
    matches = pattern.findall(content)

    # 构建格式化后的字符串
    formatted_lines = []
    for i, (start, end, count, distribution) in enumerate(matches):
        runqlen = i
        count = int(count)
        distribution = distribution.strip()
        formatted_line = f"{runqlen:>8}      : {count:>8}    |{distribution}|"
        formatted_lines.append(formatted_line)

    # 拼接所有行
    formatted_string = "    runqlen     : count    distribution\n" + "\n".join(formatted_lines)
    return formatted_string
```

（4）train_model 函数

- 功能

该函数的功能是训练一个逻辑回归模型，用于对输入的文本数据进行分类。

- 方法

- 特征提取：

使用列表推导式 [extract_features(text) for text in texts] 对每个训练文本调用 extract_features 函数，提取文本的特征。

假设 extract_features 函数已经定义，并且能够将文本转换为适合模型训练的特征向量。

- 准备训练数据：

X_train 是提取的特征向量列表，作为模型的输入特征。

y_train 是标签列表，作为模型的目标输出。

- 构建模型管道：

使用 make_pipeline 函数创建一个包含两个步骤的管道：

StandardScaler：标准化特征向量，使其具有零均值和单位方差。

LogisticRegression：逻辑回归分类器，用于进行二分类或多分类任务。

- 训练模型：

使用 model.fit(X_train, y_train) 方法训练管道中的模型，传入特征向量和对应的标签。

- 返回模型：

返回训练好的模型，以便后续进行预测或评估。

◆ 输出

返回值：训练好的逻辑回归模型。

具体实现：

```
def train_model(texts, labels):
    X_train = [extract_features(text) for text in texts]
    y_train = labels
    model = make_pipeline(StandardScaler(), LogisticRegression())
    model.fit(X_train, y_train)
    return model
```

（5）predict 函数

◆ 功能：使用训练好的模型对新的文本数据进行预测。

◆ 方法：

- 提取文本特征。
- 准备预测数据。
- 使用模型进行预测。

◆ 输出：预测结果列表，包含对每个输入文本的预测结果。

具体实现：

```
def predict(model, texts):
    X_new = [extract_features(text) for text in texts]
    predictions = model.predict(X_new)
```

```
4.      return predictions
```

（6）evaluate_model 函数

- 功能：评估一个已经训练好的模型在给定训练数据集上的准确率。
- 方法：
  - 提取文本特征。
  - 准备评估数据。
  - 使用模型进行预测。
  - 计算预测结果的准确率。
- 输出：模型在训练集上的准确率。

具体实现：

```
1.  def evaluate_model(model, texts, labels):
2.      X_train = [extract_features(text) for text in texts]
3.      y_train = labels
4.      predictions = model.predict(X_train)
5.      accuracy = accuracy_score(y_train, predictions)
6.      return accuracy
```

2、opensnoop.py

主要功能是构建一个机器学习模型，用于预测进程行为的异常或正常。

（1）load_data 函数

- 功能：初始化并返回一个具有指定列结构的 Pandas DataFrame。
- 方法：通过创建一个列表并使用 Pandas 的 DataFrame 构造函数来生成 DataFrame。
- 输出：一个 Pandas DataFrame，包含 opensnoop 和 labels 两列。

（2）vectorize_data 函数

- 功能：将文本数据转换为 TF-IDF 特征向量，并提取目标变量。
- 方法：
  - 初始化 TfidfVectorizer 对象。
  - 使用 fit_transform 方法将文本数据转换为 TF-IDF 特征矩阵。
  - 提取目标变量。
- 输出：
  - TF-IDF 特征矩阵 X。
  - 目标变量 y。
  - TfidfVectorizer 对象 vectorizer。

具体实现：

```
1.  def vectorize_data(df):
2.      vectorizer = TfidfVectorizer()
```

```
    X = vectorizer.fit_transform(df['opensnoop'])
    y = df['labels']
    return X, y, vectorizer
```

（3）train_model 函数

- 功能：使用提供的训练数据训练一个随机森林分类器模型。
- 方法：
  - 初始化一个 RandomForestClassifier 对象，设置决策树的数量和随机种子。
  - 使用 fit 方法在训练数据上训练模型。
- 输出：返回训练好的 RandomForestClassifier 模型对象。

具体实现：

```
def train_model(X_train, y_train):
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    return model
```

（4）evaluate_model 函数

- 功能：评估模型在测试数据集上的性能，计算并打印准确率，并返回预测结果。
- 方法：
  - 使用模型对测试数据进行预测。
  - 计算预测结果与真实标签之间的准确率。
  - 打印准确率。
  - 返回预测结果。
- 输出：
  - 打印输出：模型的准确率。
  - 返回值：模型在测试数据集上的预测结果。

具体实现：

```
def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Model Accuracy: {accuracy * 100:.2f}%")
    return y_pred
```

（5）predict_behavior 函数

- 功能：使用预先训练好的模型和向量化器对新的 opensnoop 文本数据进行行为预测，并返回预测结果。
- 方法：
  - 使用 vectorizer 将文本数据转换为特征向量。
  - 使用 model 对特征向量进行预测。
  - 返回预测结果的第一个元素。

- 输出：模型对输入文本数据的预测结果。

具体实现：

```
def predict_behavior(model, vectorizer, opensnoop_text):
    opensnoop_vec = vectorizer.transform([opensnoop_text])
    prediction = model.predict(opensnoop_vec)
    return prediction[0]
```

（6）read_file 函数

- 功能：读取文件内容并返回字符串。
- 方法：
  - 使用 with open 语句以只读模式打开文件。
  - 使用 file.read() 方法读取文件的全部内容。
  - 返回读取到的文件内容。
- 输出：文件内容的字符串表示。

具体实现：

```
def read_file(file_path):
    with open(file_path, 'r') as file:
        content = file.read()
    return content
```

四、使用说明

本项目利用 bpftrace 采集容器运行时的数据。具体使用方法为：

（一） 打开一个已安装好 bpftrace 的容器

```
cooler@ubuntu:~/Desktop/bpf_tools$ sudo docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS     NAMES
0a15fb516985   ebpf-for-mac   "/bin/sh -c 'mount -…"   43 hours ago   Up 23 hours             ebpf-for-mac
cooler@ubuntu:~/Desktop/bpf_tools$ sudo docker exec -it 0a15fb516985 /bin/bash
```

其中，ebpf-for-mac 是已安装好 bpftrace 的容器，启动该容器，进入到含有 bpftrace 程序运行脚本的文件夹中。也可和主机共享文件夹，通过 docker run-v 在容器中设置一个挂载点，并将主机上含有 bpftrace 的文件目录关联到该挂载点下。例如运行以下命令：

docker run -itd \

--name ebpf-for-mac \

--privileged \

-v /lib/modules:/lib/modules:ro \

-v /etc/localtime:/etc/localtime:ro \

-v /home/cooler/Desktop/bpf_tools:/root/bpf_tools \

--pid=host \

ebpf-for-mac

（二） 运行执行脚本 run_bpftrace.sh

```
root@0a15fb516985:~/bpf_tools# ./run_bpftrace.sh
bpftrace has been terminated and output saved to bt.log
```

（三） 输出日志，使用 python 利用容器输出的日志文件 bt.log，在主机中进行预测。

判定 CPU 异常行为的指标为 runqlen，相应文件为 runqlen.py

```
cooler@ubuntu:~/Desktop/bpf_tools$ python3 runqlen.py
Model accuracy on training set: 0.94
Predictions for new samples: [0]
```

```
Model accuracy on training set: 0.94
Predictions for new samples: [1]
```

其中 0 为正常，1 为异常。

判定 C 磁盘异常行为的指标为 opensnoop 与 biosnoop。由于 dataset 中 biosnoop 原始数据为七列，bpftrace 输出 log 中数据为五列，无法进行精准预测，故只采用 opensnoop 一个指标，相应文件为 opensnoop.py

```
cooler@ubuntu:~/Desktop/bpf_tools$ python3 opensnoop.py
Model Accuracy: 92.31%
Predicted Behavior for Example Process: Normal
```

其中 normal 为正常，abnormal 为异常。

五、可改进的地方

使用容器以及主机的 cron 服务来进行定时检测。

六、组内分工

本次大作业由肖晗溪、董旭、台胜雨三人合作完成，具体分工如下：肖晗溪负责代码部分，董旭和台胜雨负责论文调研、查找资料，以及最后的技术文档和 PPT 的制作。

七、项目链接

https://github.com/Star-Striker/bpf-homework