Наука о данных о вредоносных программах: обнаружение и атрибуция атак

Русский перевод практического руководства Джошуа Сакса и Хиллари Сандерс по анализу вредоносных программ методами науки о данных, машинного и глубокого обучения, обнаружению угроз и атрибуции атак.

Больше крутых материалов в моём телеграм-канале @grogrigs

Глава 1. Базовый статический анализ вредоносных программ

В этой главе мы рассмотрим основы статического анализа вредоносных программ. Статический анализ выполняют, исследуя дизассемблированный код программного файла, графические изображения, печатные строки и другие ресурсы, хранящиеся на диске. Под этим термином понимают обратную разработку без фактического запуска программы. Хотя у методов статического анализа есть недостатки, они помогают разобраться в самых разных вредоносных программах.

Тщательная обратная разработка позволит вам лучше понять как те преимущества, которые вредоносные двоичные файлы дают злоумышленникам после захвата целевой системы, так и способы, которыми злоумышленники могут скрываться и продолжать атаки на заражённом компьютере. Как вы увидите, в этой главе описания сочетаются с примерами. В каждом разделе сначала представлен метод статического анализа, а затем показано его применение при анализе реальных вредоносных программ.

Я начну главу с описания формата переносимых исполняемых файлов Portable Executable (PE), который используется большинством программ Windows, а затем покажу, как при помощи популярной библиотеки Python `pefile` исследовать реальный двоичный файл вредоносной программы. После этого я опишу такие методы, как анализ импортируемых функций, графических изображений и строк. Во всех случаях я покажу, как применять соответствующий метод к реальной вредоносной программе с помощью инструментов с открытым исходным кодом. Наконец, в конце главы я расскажу о том, как вредоносные программы могут усложнять жизнь аналитикам, и рассмотрю некоторые способы ослабить эти трудности.

Используемый в примерах этой главы образец вредоносной программы находится среди данных книги в каталоге `/ch1`. Для демонстрации рассматриваемых методов мы будем использовать `ircbot.exe` - экспериментального бота Internet Relay Chat (IRC), представляющего тип вредоносных программ, часто встречающихся на практике. Эта программа спроектирована так, чтобы оставаться в памяти целевого компьютера, сохраняя подключение к IRC-серверу. После того как `ircbot.exe` захватит целевую систему, злоумышленники смогут управлять компьютером через IRC: например, включать веб-камеру, чтобы записывать и незаметно извлекать видеопоток с изображением места, где физически находится цель, делать снимки рабочего стола, извлекать файлы с целевого компьютера и выполнять другие действия. На протяжении всей главы я буду показывать, как возможности этой вредоносной программы раскрываются с помощью статического анализа.

Формат переносимых исполняемых файлов Microsoft Windows

Для статического анализа вредоносных программ необходимо понимать формат Windows PE. Он описывает структуру современных программных файлов Windows, включая файлы `.exe`, `.dll` и `.sys`, и определяет способ хранения данных в них. PE-файлы содержат инструкции x86, данные, например изображения и текст, а также метаданные, необходимые для работы программы.

Изначально формат PE был создан для решения следующих задач:

• **Сообщать Windows, как загрузить программу в память.** Формат PE описывает, какие фрагменты файла и в какие области памяти следует загрузить. Он также указывает, с какого места в программном коде Windows должна начать выполнение программы и какие библиотеки динамически связанного кода необходимо загрузить в память.

• **Предоставлять мультимедийные данные (или ресурсы), которые работающая программа может использовать во время выполнения.** К таким ресурсам относятся символьные строки, например текст диалоговых окон графического интерфейса или консольного вывода, а также изображения и видео.
• **Предоставлять данные безопасности, например цифровые подписи кода.** Windows использует такие данные, чтобы убедиться, что код поступил из доверенного источника.

Все эти задачи формат PE решает при помощи последовательности конструкций, показанных на рис. 1-1.

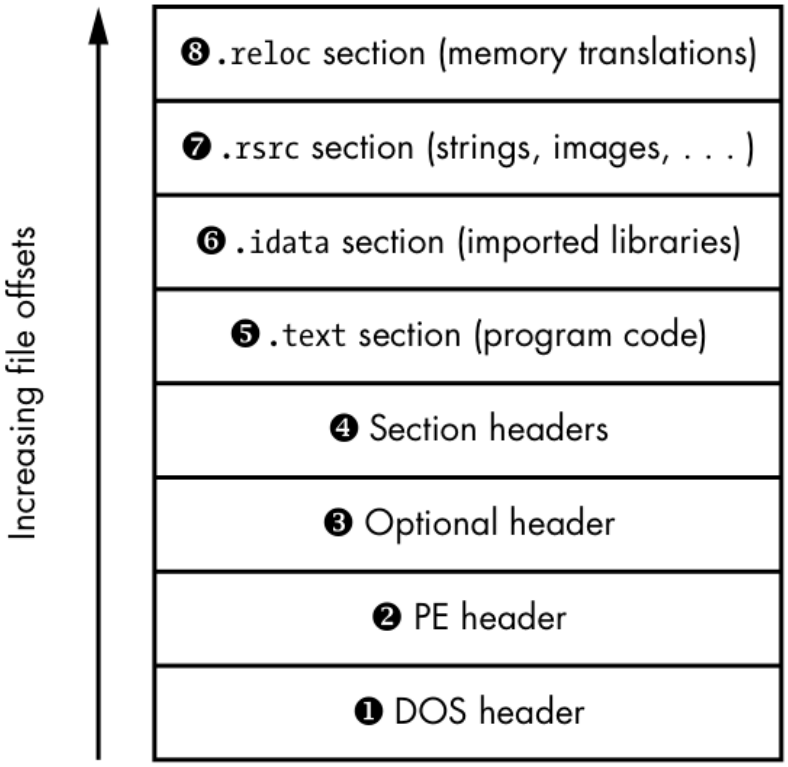

Рисунок 1-1. Формат PE-файла

Как видно из рисунка, формат PE включает ряд заголовков, сообщающих операционной системе, как загрузить программу в память. Кроме того, в него входит ряд секций, содержащих собственно данные программы. Windows загружает секции в память так, чтобы их смещения в памяти соответствовали их расположению на диске. Рассмотрим эту структуру файла подробнее, начав с заголовка PE. Обсуждение заголовка DOS мы пропустим: это пережиток операционной системы Microsoft DOS эпохи 1980-х годов, присутствующий лишь ради совместимости.

Заголовок PE

В нижней части рис. 1-1, над заголовком DOS (1), находится заголовок PE (2). Он определяет общие атрибуты программы, относящиеся к двоичному коду, изображениям, сжатым данным и другим составляющим программы. Кроме того, он сообщает, предназначена ли программа для 32- или 64-разрядных систем. Заголовок PE предоставляет аналитику вредоносных программ основные, но полезные контекстные сведения. Например, в этом заголовке есть поле временной метки, которое может выдать время компиляции файла автором вредоносной программы. Такое происходит, когда автор забывает заменить это поле вымышленным значением, хотя обычно авторы это делают.

Необязательный заголовок

Вопреки названию, необязательный заголовок (3) фактически присутствует повсеместно в современных исполняемых PE-программах. Он определяет расположение точки входа программы в PE-файле, то есть первой инструкции, выполняемой программой после загрузки. В нём также задаются размер данных, которые Windows помещает в память при загрузке PE-файла, подсистема Windows, на которую рассчитана программа (например, графический интерфейс Windows или командная строка Windows), и другие высокоуровневые характеристики программы. Сведения из этого заголовка могут оказаться бесценными для специалистов по обратной разработке, поскольку точка входа программы указывает, откуда следует начинать анализ.

Заголовки секций

Заголовки секций (4) описывают секции данных, содержащиеся в PE-файле. Секция PE-файла - это фрагмент данных, который либо будет отображён в память при загрузке программы операционной системой, либо будет содержать инструкции по загрузке программы в память. Иными словами, секция представляет собой последовательность байтов на диске, которая либо превратится в непрерывную последовательность байтов в памяти, либо сообщит операционной системе о некоторой особенности процесса загрузки.

Заголовки секций также указывают Windows, какие разрешения следует предоставить секциям: например, разрешить ли программе во время выполнения чтение, запись или исполнение их содержимого. Так, секция `.text`, содержащая код x86, обычно помечается как доступная для чтения и исполнения, но не для записи, чтобы программный код случайно не изменил сам себя в процессе работы.

На рис. 1-1 показан ряд секций, в том числе `.text` и `.rsrc`. При исполнении PE-файла они отображаются в память. Другие специальные секции, например `.reloc`, в память не отображаются. Их мы также обсудим. Рассмотрим секции, приведённые на рис. 1-1.

Секция .text

Каждая PE-программа содержит по меньшей мере одну секцию кода x86, которая в заголовке секции помечена как исполняемая; почти всегда такие секции называются `.text` (5). При дизассемблировании и обратной разработке программы в главе 2 мы дизассемблируем данные секции `.text`.

Секция .idata

Секция `.idata` (6), также называемая секцией импорта, содержит таблицу адресов импорта Import Address Table (IAT), в которой перечислены динамически подключаемые библиотеки и их функции. IAT относится к важнейшим структурам PE, которые необходимо исследовать при первоначальном знакомстве с двоичным PE-файлом: она раскрывает вызываемые программой библиотечные функции, а те, в свою очередь, могут выдать высокоуровневые возможности вредоносной программы.

Секции данных

К секциям данных PE-файла могут относиться `.rsrc`, `.data` и `.rdata`, в которых хранятся изображения указателей мыши, варианты оформления кнопок, звук и другие мультимедийные данные, используемые программой. Например, секция `.rsrc` (7) на рис. 1-1 содержит печатные символьные строки, которые программа использует для вывода текста.

Сведения в секции ресурсов `.rsrc` могут иметь для аналитиков вредоносных программ решающее значение: исследуя печатные символьные строки, графические изображения и другие ресурсы PE-файла, они получают важные подсказки о назначении файла. В разделе «Исследование изображений вредоносной программы» на с. 7 вы узнаете, как извлекать графические изображения из секций ресурсов вредоносных двоичных файлов с помощью набора инструментов `icoutils`, включая `icotool` и `wrestool`. Затем, в разделе «Исследование строк вредоносной программы» на с. 8, вы научитесь извлекать из секций ресурсов вредоносных программ печатные строки.

Секция .reloc

Код двоичного PE-файла не является позиционно-независимым: если переместить его из предполагаемой области памяти в другую, он не сможет выполняться правильно. Секция `.reloc` (8) обходит это ограничение, позволяя перемещать код без нарушения его работы. Если код был перемещён, она предписывает операционной системе Windows преобразовать адреса памяти в коде PE-файла, чтобы он продолжал правильно выполняться. Обычно такие преобразования заключаются в прибавлении смещения к адресу памяти или вычитании смещения из него.

Хотя секция `.reloc` PE-файла вполне может содержать сведения, полезные при анализе вредоносной программы, далее в этой книге мы не будем её обсуждать. Наше внимание сосредоточено на применении машинного обучения и анализа данных к вредоносным программам, а не на глубокой обратной разработке, требующей изучения перемещений адресов.

Исследование формата PE с помощью pefile

Модуль Python `pefile`, созданный и сопровождаемый Эро Каррерой, стал фактическим отраслевым стандартом среди библиотек анализа вредоносных программ для исследования PE-файлов. В этом разделе я покажу, как исследовать `ircbot.exe` с помощью `pefile`. Файл `ircbot.exe` находится на виртуальной машине, прилагаемой к книге, в каталоге `~/malware_data_science/ch1/data`. В листинге 1-1 предполагается, что `ircbot.exe` находится в текущем рабочем каталоге.

Введите следующую команду, чтобы установить библиотеку `pefile` и получить возможность импортировать её в Python:

```
$ pip install pefile
```

Теперь выполните команды из листинга 1-1: запустите Python, импортируйте модуль `pefile`, а затем откройте и проанализируйте PE-файл `ircbot.exe` с помощью `pefile`.

```
$ python
>>> import pefile
>>> pe = pefile.PE("ircbot.exe")
```

Листинг 1-1. Загрузка модуля pefile и разбор PE-файла ircbot.exe

Мы создаём экземпляр `pefile.PE` - основного класса, реализованного модулем PE. Этот класс разбирает PE-файлы, чтобы мы могли исследовать их атрибуты. Вызывая конструктор `PE`, мы загружаем и разбираем указанный PE-файл, в данном примере `ircbot.exe`. Теперь, когда файл загружен и разобран, выполните код из листинга 1-2, чтобы получить сведения из полей PE-файла `ircbot.exe`.

```
# на основе примера кода Эро Карреры (автора библиотеки pefile)
for section in pe.sections:
  print (section.Name, hex(section.VirtualAddress),
    hex(section.Misc_VirtualSize), section.SizeOfRawData )
```

Листинг 1-2. Перебор секций PE-файла и вывод сведений о них

В листинге 1-3 показан результат.

```
('.text\x00\x00\x00', (1)'0x1000', (2)'0x32830', (3)207360)
('.rdata\x00\x00', '0x34000', '0x427a', 17408)
('.data\x00\x00\x00', '0x39000', '0x5cff8', 10752)
('.idata\x00\x00', '0x96000', '0xbb0', 3072)
('.reloc\x00\x00', '0x97000', '0x211d', 8704)
```

Листинг 1-3. Получение данных о секциях ircbot.exe с помощью модуля pefile языка Python

Как видно из листинга 1-3, мы получили данные из пяти разных секций PE-файла: `.text`, `.rdata`, `.data`, `.idata` и `.reloc`. Результат представлен пятью кортежами - по одному на каждую извлечённую секцию PE. Первая запись в каждой строке обозначает секцию PE. Последовательность нулевых байтов `\x00` можно не учитывать: это обычные завершающие нулевые символы строк в стиле C. Остальные поля сообщают, сколько памяти будет занимать каждая секция после загрузки и где она окажется в памяти.

Например, `0x1000` (1) - базовый виртуальный адрес памяти, по которому будут загружены эти секции. Его можно считать базовым адресом памяти секции. Значение `0x32830` (2) в поле виртуального размера задаёт объём памяти, необходимый секции после загрузки. Число `207360` (3) в третьем поле обозначает объём данных, который секция займёт внутри этого фрагмента памяти.

Помимо разбора секций программы, `pefile` позволяет получить список загружаемых двоичным файлом DLL и вызываемых им функций из этих DLL. Для этого можно вывести содержимое IAT PE-файла. В листинге 1-4 показано, как получить IAT файла `ircbot.exe` с помощью `pefile`.

```
$ python

pe = pefile.PE("ircbot.exe")
for entry in pe.DIRECTORY_ENTRY_IMPORT:
    print entry.dll
    for function in entry.imports:
        print '\t',function.name
```

Листинг 1-4. Извлечение импортируемых функций из ircbot.exe

Листинг 1-4 должен вывести результат, показанный в листинге 1-5 (для краткости он усечён).

```
KERNEL32.DLL
      GetLocalTime
      ExitThread
      CloseHandle
(1) WriteFile
(2) CreateFileA
      ExitProcess
(3) CreateProcessA
      GetTickCount
      GetModuleFileNameA
--snip--
```

Листинг 1-5. Содержимое IAT файла ircbot.exe с библиотечными функциями, используемыми этой вредоносной программой

Как видно из листинга 1-5, этот вывод ценен для анализа вредоносных программ, поскольку содержит обширный набор функций, которые объявляет и будет вызывать вредоносная программа. Например, первые строки показывают, что она будет записывать данные в файлы при помощи `WriteFile` (1), открывать файлы вызовом `CreateFileA` (2) и создавать новые процессы посредством `CreateProcessA` (3). Это лишь самые общие сведения о вредоносной программе, но с них можно начать более подробное изучение её поведения.

Исследование изображений вредоносной программы

Чтобы понять, каким образом вредоносная программа может быть рассчитана на обман цели, рассмотрим значки в её секции `.rsrc`. Например, вредоносные двоичные файлы нередко маскируются под документы Word, установщики игр, PDF-файлы и другие знакомые объекты, чтобы заставить пользователей щёлкнуть по ним. Во вредоносных программах также встречаются изображения, указывающие на программы, интересующие самих злоумышленников, например на средства сетевых атак и программы удалённого управления скомпрометированными компьютерами. Мне доводилось видеть даже двоичные файлы со значками рабочего стола, изображавшими джихадистов, зловещих персонажей киберпанковских мультфильмов и автоматы Калашникова. В качестве примера анализа изображений рассмотрим вредоносный образец, который компания по информационной безопасности Mandiant связала с поддерживаемой китайским государством хакерской группой. Этот образец находится в каталоге данных главы под именем `fakepdfmalware.exe`. Он использует значок Adobe Acrobat, чтобы пользователь принял его за документ Adobe Acrobat, хотя на самом деле это вредоносный исполняемый PE-файл.

Прежде чем извлекать изображения из двоичного файла `fakepdfmalware.exe` с помощью инструмента командной строки Linux `wrestool`, необходимо создать каталог для извлечённых изображений. В листинге 1-6 показано, как выполнить обе операции.

```
$ mkdir images
$ wrestool -x fakepdfmalware.exe -output=images
$ icotool -x -o images images/*.ico
```

Листинг 1-6. Команды оболочки, извлекающие изображения из образца вредоносной программы

Сначала командой `mkdir images` мы создаём каталог для извлечённых изображений. Затем при помощи `wrestool` извлекаем ресурсы изображений (`-x`) из `fakepdfmalware.exe` в каталог `/images`, после чего с помощью `icotool` извлекаем (`-x`) и преобразуем (`-o`) все ресурсы в формате значков Adobe `.ico` в графические файлы `.png`, которые можно просматривать обычными программами для работы с изображениями. Если в вашей системе не установлен `wrestool`, его можно загрузить по адресу nongnu.org.

После того как вы преобразуете изображения из исследуемого исполняемого файла в формат PNG с помощью `wrestool`, их можно будет открыть в предпочитаемой программе просмотра и увидеть значок Adobe Acrobat в разных разрешениях. Как показывает этот пример, извлекать изображения и значки из PE-файлов относительно просто, а полученные данные позволяют быстро обнаружить интересные и полезные сведения о вредоносных двоичных файлах. Столь же легко можно извлечь из вредоносной программы печатные строки, чтобы получить дополнительную информацию. Этим мы сейчас и займёмся.

Исследование строк вредоносной программы

Строки - это последовательности печатных символов в двоичном файле программы. Аналитики вредоносных программ часто обращаются к строкам исследуемого образца, чтобы быстро составить первоначальное представление о происходящем внутри. В таких строках нередко встречаются команды HTTP и FTP для загрузки веб-страниц и файлов, IP-адреса и имена узлов, указывающие, к каким адресам подключается вредоносная программа, и другие подобные сведения. Иногда даже язык строк может подсказать страну происхождения вредоносного двоичного файла, хотя такую подсказку можно подделать. В строке может встретиться даже текст, который на «языке хакеров» объясняет назначение вредоносного двоичного файла.

Строки способны раскрыть и более технические сведения о двоичном файле. Например, в них можно найти информацию о компиляторе, использованном для его создания, языке программирования, на котором он написан, встроенных сценариях или HTML и многом другом. Авторы вредоносных программ могут обфусцировать, зашифровать и сжать все эти следы, но даже опытные авторы часто оставляют хотя бы некоторые из них открытыми. Поэтому при анализе вредоносных программ особенно важно исследовать дампы строк.

Использование программы strings

Обычный способ просмотреть все строки файла - воспользоваться инструментом командной строки `strings` со следующим синтаксисом:

```
$ strings filepath | less
```

Эта команда печатает в терминале все строки файла, по одной на строку. Добавление `| less` в конце не позволяет строкам просто промелькнуть по экрану терминала. По умолчанию команда `strings` находит все печатные строки длиной не менее 4 байт, однако можно установить другую минимальную длину и изменить ряд других параметров, перечисленных на странице руководства по команде. Я рекомендую оставить стандартную минимальную длину, равную 4, но её можно изменить параметром `-n`. Например, команда `strings -n 10 filepath` извлечёт только строки длиной не менее 10 байт.

Анализ дампа strings

Теперь, когда мы получили дамп печатных строк вредоносной программы, необходимо понять смысл этих строк. Предположим, например, что мы сохраняем строки из файла `ircbot.exe`, который уже исследовали в этой главе с помощью библиотеки `pefile`, в файл `ircbotstring.txt`:

```
$ strings ircbot.exe > ircbotstring.txt
```

Файл `ircbotstring.txt` содержит тысячи строк текста, но некоторые из них должны привлечь внимание. Например, в листинге 1-7 показана группа извлечённых из дампа строк, которые начинаются словом `DOWNLOAD`.

```
[DOWNLOAD]: Bad URL, or DNS Error: %s.
[DOWNLOAD]: Update failed: Error executing file: %s.
[DOWNLOAD]: Downloaded %.1fKB to %s @ %.1fKB/sec. Updating.
[DOWNLOAD]: Opened: %s.
--snip--
[DOWNLOAD]: Downloaded %.1f KB to %s @ %.1f KB/sec.
[DOWNLOAD]: CRC Failed (%d != %d).
[DOWNLOAD]: Filesize is incorrect: (%d != %d).
[DOWNLOAD]: Update: %s (%dKB transferred).
[DOWNLOAD]: File download: %s (%dKB transferred).
```

```
[DOWNLOAD]: Couldn't open file: %s.
```

Листинг 1-7. Вывод strings, свидетельствующий о способности вредоносной программы загружать на целевой компьютер файлы, указанные злоумышленником

Эти строки показывают, что `ircbot.exe` попытается загрузить на целевой компьютер файлы, указанные злоумышленником.

Попробуем проанализировать ещё один пример. Дамп строк в листинге 1-8 указывает, что `ircbot.exe` может выступать в роли веб-сервера, который на целевом компьютере ожидает подключений злоумышленника.

```
(1) GET
(2) HTTP/1.0 200 OK
Server: myBot
Cache-Control: no-cache,no-store,max-age=0
pragma: no-cache
Content-Type: %s
Content-Length: %i
Accept-Ranges: bytes
Date: %s %s GMT
Last-Modified: %s %s GMT
Expires: %s %s GMT
Connection: close
HTTP/1.0 200 OK
(3) Server: myBot
Cache-Control: no-cache,no-store,max-age=0
pragma: no-cache
Content-Type: %s
Accept-Ranges: bytes
Date: %s %s GMT
Last-Modified: %s %s GMT
Expires: %s %s GMT
Connection: close
HH:mm:ss
ddd, dd MMM yyyy
application/octet-stream
text/html
```

Листинг 1-8. Вывод strings, показывающий, что вредоносная программа содержит HTTP-сервер, к которому может подключиться злоумышленник

В листинге 1-8 приведено множество стандартных фрагментов HTTP, с помощью которых `ircbot.exe` реализует HTTP-сервер. Вероятно, этот сервер позволяет злоумышленнику подключаться к целевому компьютеру по HTTP и отдавать команды, например сделать снимок рабочего стола жертвы и отправить его злоумышленнику. Признаки функций HTTP видны на протяжении всего листинга. Например, метод `GET` (1) запрашивает данные интернет-ресурса. Строка `HTTP/1.0 200 OK` (2) возвращает код состояния HTTP 200, который означает успешное выполнение сетевой транзакции HTTP, а `Server: myBot` (3) указывает, что HTTP-сервер называется `myBot`, тем самым выдавая наличие встроенного HTTP-сервера в `ircbot.exe`.

Все эти сведения помогают понять и остановить конкретный вредоносный образец или вредоносную кампанию. Например, зная, что вредоносная программа содержит HTTP-сервер, который выдаёт при подключении определённые строки, можно просканировать сеть и выявить заражённые узлы.

Итоги

В этой главе вы получили общее представление о статическом анализе вредоносных программ, при котором вредоносную программу исследуют, фактически не запуская её. Вы узнали о формате PE, определяющем устройство файлов Windows `.exe` и `.dll`, и научились исследовать реальный вредоносный двоичный файл `ircbot.exe` при помощи библиотеки Python `pefile`. Кроме того, вы применили такие методы статического анализа, как анализ изображений и строк, чтобы извлечь дополнительные сведения из образцов вредоносных программ. В главе 2 обсуждение статического анализа продолжится и будет сосредоточено на анализе кода на языке ассемблера, который можно восстановить из вредоносной программы.

Глава 2. За пределами базового статического анализа: дизассемблирование x86

Чтобы досконально разобраться во вредоносной программе, зачастую требуется выйти за пределы базового статического анализа её секций, строк, импортируемых функций и изображений. Для этого необходимо выполнить обратную разработку программы на уровне кода на языке ассемблера. Действительно, дизассемблирование и обратная разработка лежат в основе глубокого статического анализа образцов вредоносных программ.

Обратная разработка одновременно является искусством, техническим ремеслом и наукой, поэтому её всестороннее рассмотрение выходит за рамки этой главы. Моя цель - познакомить вас с обратной разработкой настолько, чтобы вы могли применять её в анализе данных о вредоносных программах. Понимание этой методологии необходимо для успешного применения машинного обучения и анализа данных к вредоносным программам.

Я начну главу с понятий, необходимых для понимания дизассемблирования x86. Далее я покажу, как авторы вредоносных программ пытаются воспрепятствовать дизассемблированию, и расскажу о способах ослабления этих мер против анализа и обнаружения. Но сначала рассмотрим несколько распространённых методов дизассемблирования и основы языка ассемблера x86.

Методы дизассемблирования

Дизассемблирование - это процесс преобразования двоичного кода вредоносной программы в корректный код на языке ассемблера x86. Обычно авторы пишут вредоносные программы на языке высокого уровня, например C или C++, а затем компилируют исходный текст в двоичный код x86. Язык ассемблера представляет собой удобочитаемое для человека представление этого двоичного кода. Следовательно, чтобы понять работу вредоносной программы на самом низком уровне, её необходимо дизассемблировать.

К сожалению, дизассемблирование - непростая задача, поскольку авторы вредоносных программ регулярно применяют приёмы, призванные помешать потенциальным специалистам по обратной разработке. Более того, идеальное дизассемблирование при наличии преднамеренной обфускации остаётся нерешённой задачей информатики. В настоящее время для дизассемблирования таких программ существуют лишь приближённые методы, подверженные ошибкам.

Рассмотрим, например, самомодифицирующийся код - двоичный код, который изменяет сам себя во время выполнения. Единственный способ правильно его дизассемблировать - понять программную логику, в соответствии с которой код изменяет себя, однако эта логика может быть чрезвычайно сложной.

Поскольку идеальное дизассемблирование пока невозможно, задачу приходится решать несовершенными методами. Мы будем применять линейное дизассемблирование: находить в файле Portable Executable (PE) непрерывную последовательность байтов, соответствующую программному коду x86, а затем декодировать эти байты. Главное ограничение такого подхода состоит в том, что он не учитывает тонкости декодирования инструкций процессором во время выполнения программы. Кроме того, он не принимает во внимание различные способы обфускации, которыми авторы вредоносных программ иногда затрудняют анализ своих разработок.

Другие методы обратной разработки, которые мы здесь рассматривать не будем, представляют собой более сложные способы дизассемблирования, применяемые промышленными дизассемблерами, например IDA Pro. Эти более совершенные методы фактически имитируют или

логически анализируют выполнение программы, чтобы определить, каких инструкций на языке ассемблера программа может достичь в результате последовательности условных переходов.

Хотя такое дизассемблирование бывает точнее линейного, оно требует гораздо больше процессорных ресурсов. Поэтому оно хуже подходит для задач анализа данных, где необходимо дизассемблировать тысячи или даже миллионы программ.

Однако прежде чем приступать к анализу методом линейного дизассемблирования, необходимо вспомнить основные составляющие языка ассемблера.

Основы языка ассемблера x86

Язык ассемблера - это самый низкоуровневый удобочитаемый язык программирования для данной архитектуры, тесно соответствующий формату двоичных инструкций конкретной процессорной архитектуры. Строка на языке ассемблера почти всегда соответствует одной инструкции процессора. Благодаря столь низкому уровню представления код на языке ассемблера часто можно без труда получить из двоичного файла вредоносной программы, воспользовавшись подходящими инструментами.

Приобрести базовый навык чтения дизассемблированного кода вредоносных программ для x86 проще, чем может показаться. Причина в том, что большую часть времени такой код обращается к операционной системе через динамически подключаемые библиотеки Windows (DLL), загружаемые в память программы во время выполнения. Именно посредством DLL вредоносные программы выполняют основную практическую работу: изменяют системный реестр, перемещают и копируют файлы, устанавливают сетевые соединения, обмениваются данными по сетевым протоколам и так далее. Поэтому чтение кода вредоносной программы на языке ассемблера часто сводится к пониманию того, как из ассемблерного кода вызываются функции и что делают различные вызовы DLL. Разумеется, всё может оказаться гораздо сложнее, но даже эти знания способны многое рассказать о вредоносной программе.

В следующих разделах я познакомлю вас с несколькими важными понятиями языка ассемблера. Кроме того, я объясню такие абстрактные понятия, как поток управления и графы потока управления. Наконец, мы дизассемблируем программу `ircbot.exe` и исследуем, какие сведения о её назначении можно получить из ассемблерного кода и потока управления.

У языка ассемблера x86 есть два основных диалекта: Intel и AT&T. В этой книге я использую синтаксис Intel, который поддерживают все основные дизассемблеры и который применяется в официальной документации Intel по процессорам x86.

Начнём с регистров процессора.

Регистры процессора

Регистры - это небольшие хранилища данных, над которыми процессоры x86 выполняют вычисления. Поскольку регистры находятся непосредственно в процессоре, доступ к ним на несколько порядков быстрее доступа к памяти. Именно поэтому основные вычислительные операции, например арифметические инструкции и инструкции проверки условий, работают с регистрами. По той же причине процессор хранит в регистрах сведения о состоянии выполняющихся программ. Опытному программисту на языке ассемблера x86 доступно множество регистров, но здесь мы сосредоточимся лишь на нескольких важных.

Регистры общего назначения

Регистры общего назначения играют для программиста на языке ассемблера роль рабочей области. В 32-разрядной системе каждый такой регистр предоставляет 32, 16 или 8 бит, над которыми можно выполнять арифметические и побитовые операции, операции перестановки порядка байтов и многое другое.

В типичном вычислительном процессе программа перемещает данные из памяти или внешних аппаратных устройств в регистры, выполняет над ними операции, а затем возвращает данные в память для хранения. Например, при сортировке длинного списка программа обычно извлекает элементы массива из памяти, сравнивает их в регистрах, а затем записывает результаты сравнения обратно в память.

Некоторые тонкости модели регистров общего назначения в 32-разрядной архитектуре Intel показаны на рис. 2-1.

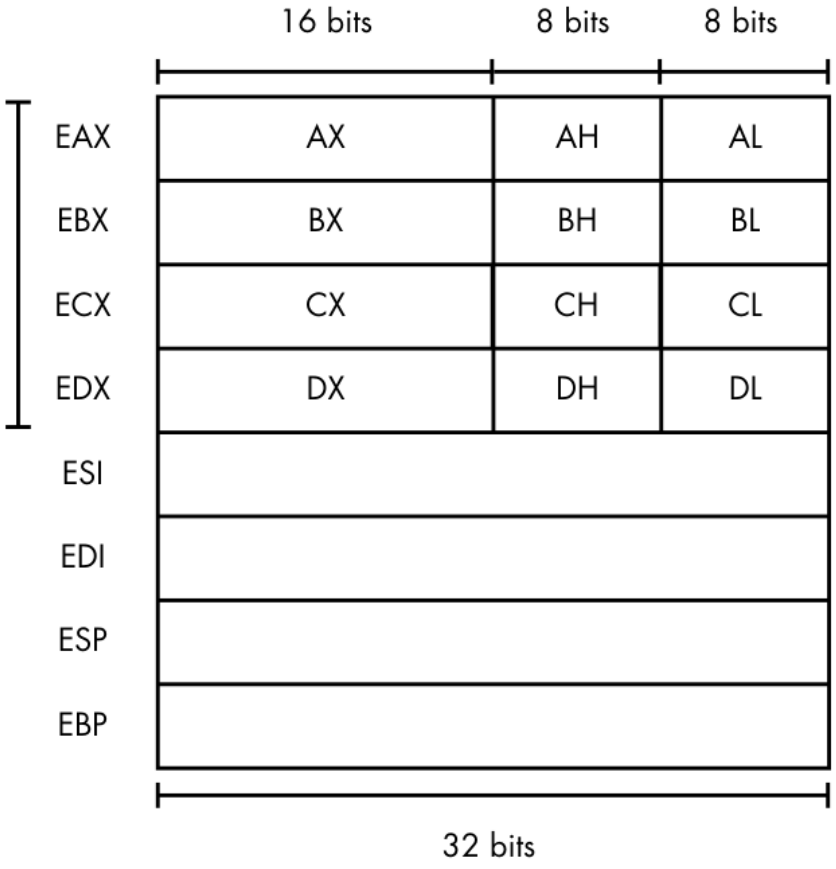

Рисунок 2-1. Регистры архитектуры x86

По вертикальной оси показано расположение регистров общего назначения, а по горизонтальной - способ деления EAX, EBX, ECX и EDX. EAX, EBX, ECX и EDX представляют собой 32-разрядные регистры, внутри которых находятся более короткие 16-разрядные регистры AX, BX, CX и DX. Как видно на рисунке, эти 16-разрядные регистры можно разделить на верхние и нижние 8-разрядные регистры: AH, AL, BH, BL, CH, CL, DH и DL. Хотя иногда бывает полезно обращаться к отдельным частям EAX, EBX, ECX и EDX, чаще всего вам будут встречаться прямые обращения к EAX, EBX, ECX и EDX.

Регистры стека и потока управления

Регистры управления стеком хранят важнейшие сведения о стеке программы, в котором размещаются локальные переменные функций, переданные функциям аргументы и управляющие данные, относящиеся к потоку управления программы. Рассмотрим некоторые из этих регистров.

Если говорить упрощённо, регистр ESP указывает на вершину стека выполняющейся в данный момент функции, а регистр EBP - на основание её стека. Эти сведения имеют решающее значение для современных программ: обращаясь к данным относительно стека, а не по абсолютным адресам, процедурный и объектно-ориентированный код может получать доступ к локальным переменным более изящно и эффективно.

Хотя в ассемблерном коде x86 вы не встретите прямых обращений к регистру EIP, он важен для анализа безопасности, особенно при исследовании уязвимостей и разработке эксплойтов, использующих переполнение буфера. Дело в том, что EIP содержит адрес выполняющейся в данный момент инструкции в памяти. С помощью эксплойта переполнения буфера злоумышленник может косвенно повредить значение регистра EIP и перехватить управление выполнением программы.

Помимо роли в эксплуатации уязвимостей, EIP важен и при анализе вредоносного кода. В любой момент можно исследовать значение EIP с помощью отладчика и тем самым понять, какой код вредоносная программа выполняет именно сейчас.

EFLAGS - это регистр состояния, содержащий флаги процессора, то есть биты со сведениями о состоянии выполняющейся программы. Регистр EFLAGS играет ключевую роль при выполнении условных переходов в программах x86 - изменений потока выполнения, зависящих от результата программной логики вида «если/то». В частности, когда программа на языке ассемблера x86 проверяет, больше или меньше нуля некоторое значение, а затем в зависимости от результата переходит к функции, именно регистр EFLAGS обеспечивает возможность такого поведения. Подробнее это описано в разделе «Базовые блоки и графы потока управления» на с. 19.

Арифметические инструкции

Инструкции выполняют действия над регистрами общего назначения. Простейшие вычисления над ними можно производить с помощью арифметических инструкций. К часто встречающимся при обратной разработке вредоносных программ арифметическим инструкциям относятся, например, `add`, `sub`, `inc`, `dec` и `mul`. В табл. 2-1 приведены примеры основных инструкций и их синтаксис.

Таблица 2-1. Арифметические инструкции

Инструкция	**Описание**
`add ebx, 100`	Прибавляет 100 к значению в EBX и сохраняет результат в EBX
`sub ebx, 100`	Вычитает 100 из значения в EBX и сохраняет результат в EBX
`inc ah`	Увеличивает значение в AH на 1
`dec al`	Уменьшает значение в AL на 1

Инструкция add складывает два целых числа и сохраняет результат в первом указанном операнде, которым может быть область памяти или регистр. Следует помнить, что только один аргумент может обозначать область памяти. Инструкция sub похожа на add, но выполняет вычитание целых чисел. Инструкция inc увеличивает целочисленное значение регистра или области памяти, а dec уменьшает его.

Инструкции перемещения данных

Процессор x86 предоставляет богатый набор инструкций для перемещения данных между регистрами и памятью. Эти инструкции образуют базовые механизмы, позволяющие манипулировать данными. Главная инструкция перемещения данных в памяти - mov. В табл. 2-2 показано, как с её помощью перемещать данные.

Таблица 2-2. Инструкции перемещения данных

Инструкция	Описание
`mov ebx,eax`	Перемещает значение из регистра EAX в регистр EBX
`mov eax, [0x12345678]`	Перемещает данные по адресу памяти 0x12345678 в регистр EAX
`mov edx, 1`	Перемещает значение 1 в регистр EDX
`mov [0x12345678], eax`	Перемещает значение из EAX в область памяти по адресу 0x12345678

С инструкцией mov связана инструкция lea: она загружает указанный абсолютный адрес памяти в регистр, используемый для получения указателя на область памяти. Например, `lea edx, [esp-4]` вычитает 4 из значения ESP и загружает получившееся значение в EDX.

Инструкции работы со стеком

Стек в языке ассемблера x86 - это структура данных, в которую можно помещать значения и извлекать их оттуда. Это напоминает добавление тарелок на вершину стопки и снятие тарелок с неё.

Поток управления в языке ассемблера x86 часто выражается через вызовы функций в стиле C. Такие вызовы используют стек для передачи аргументов, выделения локальных переменных и запоминания части программы, к которой следует вернуться после завершения функции. Поэтому стек и поток управления необходимо рассматривать совместно.

Инструкция push помещает значения в стек программы, когда программисту нужно сохранить в нём значение регистра, а инструкция pop удаляет значения из стека и помещает их в указанный регистр. Инструкция push выполняет свою операцию со следующим синтаксисом:

```
push 1
```

В этом примере программа перемещает указатель стека, то есть регистр ESP, к новому адресу памяти, тем самым освобождая место для значения 1, которое теперь будет храниться на вершине стека. Затем она копирует значение аргумента в только что освобождённую процессором область

памяти на вершине стека.

Сравним это с pop:

```
pop eax
```

Инструкция pop снимает верхнее значение со стека и перемещает его в указанный регистр. В данном примере pop eax снимает верхнее значение со стека и перемещает его в EAX.

У стека программ x86 есть неочевидная, но важная особенность: он растёт вниз по памяти, поэтому самое верхнее значение стека фактически хранится по самому низкому адресу стековой памяти. Об этом особенно важно помнить при анализе ассемблерного кода, который обращается к данным в стеке: не зная устройства стековой памяти, очень легко запутаться.

Поскольку стек x86 растёт вниз по памяти, при выделении инструкцией push места в стеке для нового значения она уменьшает ESP, чтобы тот указывал на более низкий адрес памяти, а затем копирует значение из нужного регистра в эту область памяти, начиная с верхнего адреса стека и продвигаясь вверх. И наоборот, инструкция pop копирует верхнее значение со стека, а затем увеличивает ESP, чтобы он указывал на более высокий адрес памяти.

Инструкции потока управления

Поток управления программы x86 определяет сеть возможных последовательностей исполнения инструкций, которые программа может выполнить в зависимости от полученных данных, взаимодействий с устройствами и других входных воздействий. Поток управления программы задаётся соответствующими инструкциями. Они сложнее инструкций работы со стеком, но всё же достаточно понятны. Поскольку поток управления в языке ассемблера x86 часто выражается через вызовы функций в стиле C, стек и поток управления тесно связаны. Связь объясняется ещё и тем, что вызовы функций используют стек для передачи аргументов, выделения локальных переменных и запоминания той части программы, к которой следует вернуться после завершения функции.

Инструкции потока управления call и ret имеют первостепенное значение для вызова функций программами на языке ассемблера x86 и возврата из функций после завершения их работы.

Инструкция call вызывает функцию. Её можно сопоставить с функцией на языке высокого уровня вроде C: программа должна вернуться к инструкции, следующей за call, после того как вызванная функция завершит выполнение. Инструкция call вызывается со следующим синтаксисом, где address обозначает адрес памяти, с которого начинается код функции:

```
call address
```

Инструкция call выполняет два действия. Сначала она помещает на вершину стека адрес инструкции, которая должна выполниться после возврата из функции, чтобы программа знала, куда вернуться по завершении вызванной функции. Затем call заменяет текущее значение EIP значением операнда address. После этого процессор начинает выполнение с новой области памяти, на которую указывает EIP.

Если call начинает вызов функции, то ret его завершает. Инструкцию ret можно использовать отдельно, без параметров:

```
ret
```

При выполнении ret снимает с вершины стека значение, которое, как предполагается, является сохранённым значением счётчика команд EIP, помещённым в стек инструкцией call при вызове. Затем снятое значение счётчика команд возвращается в EIP, и выполнение возобновляется.

Другая важная конструкция потока управления - инструкция jmp, которая действует проще call. Вместо сохранения EIP инструкция jmp просто предписывает процессору перейти к адресу памяти,

указанному в качестве параметра, и начать выполнение с этого места. Например, `jmp 0x12345678` сообщает процессору, что следующей должна выполняться инструкция программного кода, расположенная в памяти по адресу `0x12345678`.

Возможно, вы задаётесь вопросом, как выполнять инструкции `jmp` и `call` условно, например: «Если программа получила сетевой пакет, выполнить следующую функцию». В языке ассемблера x86 нет высокоуровневых конструкций `if`, `then`, `else`, `else if` и подобных им. Вместо этого для перехода к адресу в программном коде обычно требуются две инструкции: `cmp`, которая сравнивает значение некоторого регистра с проверочным значением и сохраняет результат проверки в EFLAGS, и инструкция условного перехода.

Большинство инструкций условного перехода начинается с `j`, обозначающей возможность перехода программы к адресу памяти, а после неё следуют буквы, обозначающие проверяемое условие. Например, `jge` предписывает программе перейти, если значение больше проверочного или равно ему. Это означает, что значение в проверяемом регистре должно быть больше проверочного значения или равно ему.

Инструкция `cmp` использует следующий синтаксис:

```
cmp register, memory location, or literal, register, memory location, or literal
```

Как уже говорилось, `cmp` сравнивает значение в указанном регистре общего назначения со значением, а затем сохраняет результат сравнения в регистре EFLAGS.

Различные условные варианты `jmp` вызываются следующим образом:

```
j* address
```

Как видите, после префикса `j` можно указать одну из множества инструкций проверки условий. Например, чтобы перейти только в том случае, если проверяемое значение больше значения в регистре или равно ему, используется следующая инструкция:

```
jge address
```

Обратите внимание: в отличие от `call` и `ret`, семейство инструкций `jmp` никогда не обращается к стеку программы. При использовании этих инструкций программа x86 сама отвечает за отслеживание собственного потока выполнения и, возможно, за сохранение или удаление сведений о посещённых адресах и о том, куда ей следует вернуться после выполнения определённой последовательности инструкций.

Базовые блоки и графы потока управления

Когда мы прокручиваем код программы x86 в текстовом редакторе, он выглядит последовательным, но в действительности содержит циклы, условные и безусловные переходы, то есть поток управления. Благодаря им каждая программа x86 обладает сетевой структурой. Рассмотрим её на примере простой учебной программы на языке ассемблера из листинга 2-1.

```
setup: # символ, обозначающий адрес инструкции в следующей строке
(1) mov eax, 10
loopstart: # символ, обозначающий адрес инструкции в следующей строке
(2) sub eax, 1
(3) cmp 0, eax
   jne $loopstart
loopend: # символ, обозначающий адрес инструкции в следующей строке
   mov eax, 1
# здесь находился бы дальнейший код
```

Листинг 2-1. Программа на языке ассемблера для знакомства с графом потока управления

Как видите, программа инициализирует счётчик значением 10, которое сохраняется в регистре EAX (1). Затем выполняется цикл, на каждой итерации уменьшающий значение EAX на 1 (2). Наконец, когда значение EAX достигает 0 (3), программа выходит из цикла.

На языке анализа графов потока управления можно сказать, что эти инструкции образуют три базовых блока. Базовый блок - это последовательность инструкций, о которой известно, что она всегда выполняется непрерывно. Иными словами, базовый блок всегда заканчивается инструкцией перехода либо инструкцией, являющейся целью перехода, а начинается с первой инструкции программы, называемой точкой входа, либо с цели перехода.

В листинге 2-1 видны границы базовых блоков нашей простой программы. Первый блок состоит из инструкции `mov eax, 10` под меткой `setup:`. Во второй входят строки от `sub eax, 1` до `jne $loopstart` под меткой `loopstart:`, а третий начинается с `mov eax, 1` под меткой `loopend:`. Связи между базовыми блоками можно наглядно представить графом на рис. 2-2. Термин *граф* здесь используется как синоним термина *сеть*: в информатике они взаимозаменяемы.

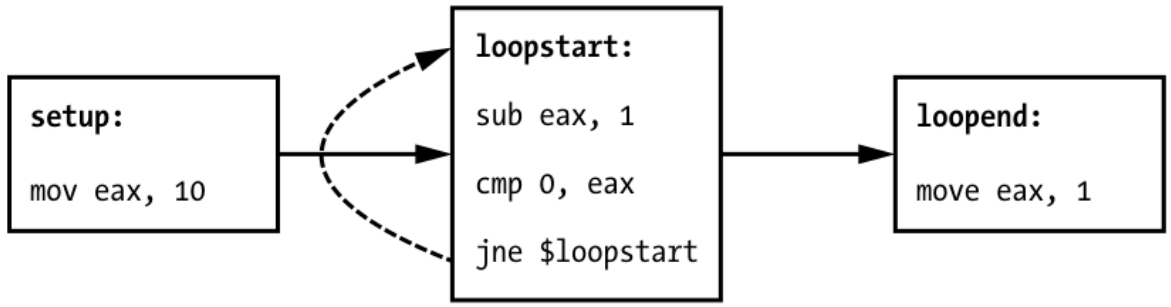

Рисунок 2-2. Наглядное представление графа потока управления простой программы на языке ассемблера

Если поток выполнения хотя бы при некоторых обстоятельствах может перейти из одного базового блока в другой, мы соединяем эти блоки, как показано на рис. 2-2. Из рисунка видно, что блок `setup` ведёт к блоку `loopstart`, который повторяется 10 раз, а затем переходит к блоку `loopend`. Графы потока управления реальных программ устроены подобным образом, но гораздо сложнее: в них могут быть тысячи базовых блоков и тысячи связей между ними.

Дизассемблирование ircbot.exe с помощью pefile и capstone

Теперь, когда вы хорошо представляете основы языка ассемблера, выполним линейное дизассемблирование первых 100 байт кода `ircbot.exe`. Для этого воспользуемся библиотекой Python с открытым исходным кодом `pefile`, представленной в главе 1, и библиотекой дизассемблирования с открытым исходным кодом `capstone`, способной дизассемблировать 32-разрядный двоичный код x86. Обе библиотеки можно установить с помощью `pip` следующими командами:

```
pip install pefile
pip install capstone
```

После установки обеих библиотек код из листинга 2-2 позволит дизассемблировать `ircbot.exe`.

```
#!/usr/bin/python
import pefile
from capstone import *

# загрузить исследуемый PE-файл
pe = pefile.PE("ircbot.exe")
# получить адрес точки входа программы из заголовка программы
```

```
entrypoint = pe.OPTIONAL_HEADER.AddressOfEntryPoint

# вычислить адрес памяти, по которому будет загружен код точки входа
entrypoint_address = entrypoint+pe.OPTIONAL_HEADER.ImageBase

# получить двоичный код из объекта PE-файла
binary_code = pe.get_memory_mapped_image()[entrypoint:entrypoint+100]

# подготовить дизассемблер для 32-разрядного двоичного кода x86
disassembler = Cs(CS_ARCH_X86, CS_MODE_32)

# дизассемблировать код
for instruction in disassembler.disasm(binary_code, entrypoint_address):
    print "%s\t%s" %(instruction.mnemonic, instruction.op_str)
```

Листинг 2-2. Дизассемблирование ircbot.exe

Код должен вывести следующий результат:

```
(1) push    ebp
   mov     ebp, esp
   push    -1
   push    0x437588
   push    0x41982c
(2) mov     eax, dword ptr fs:[0]
   push    eax
   mov     dword ptr fs:[0], esp
(3) add     esp, -0x5c
   push    ebx
   push    esi
   push    edi
   mov     dword ptr [ebp - 0x18], esp
(4) call    dword ptr [0x496308]
   --snip--
```

Не беспокойтесь, если понимаете не все инструкции в дизассемблированном выводе: для этого понадобились бы знания языка ассемблера, выходящие за рамки книги. Тем не менее многие инструкции уже должны быть вам знакомы, и вы должны в общих чертах представлять, что они делают. Например, вредоносная программа помещает в стек значение регистра EBP (1), тем самым сохраняя его. Затем она перемещает значение ESP в EBP и помещает в стек несколько числовых значений. Программа перемещает некоторые данные из памяти в регистр EAX (2) и прибавляет значение `-0x5c` к значению регистра ESP (3). Наконец, инструкцией `call` программа вызывает функцию, хранящуюся по адресу памяти `0x496308` (4).

Поскольку эта книга не посвящена обратной разработке, я не буду подробнее разбирать значение кода. Представленного материала достаточно, чтобы начать знакомство с работой языка ассемблера. Дополнительные сведения можно найти в руководстве Intel для программистов: intel.com.

Факторы, ограничивающие статический анализ

В этой и предыдущей главах вы узнали о различных способах применения статического анализа для выяснения назначения и принципов работы недавно обнаруженного вредоносного двоичного файла. К сожалению, у статического анализа есть ограничения, из-за которых в некоторых обстоятельствах он менее полезен. Например, авторы вредоносных программ могут применять определённые наступательные приёмы, реализовать которые гораздо проще, чем защититься от них. Рассмотрим некоторые из этих приёмов и способы защиты.

Упаковка

Упаковка вредоносной программы - это процесс, при котором автор сжимает, шифрует или иным способом искажает основную часть вредоносной программы, чтобы она стала непонятной аналитику. При запуске вредоносная программа распаковывает себя и приступает к выполнению. Очевидный способ обойти упаковку - фактически запустить программу в безопасной среде, применив метод динамического анализа, который будет рассмотрен в главе 3.

> **Примечание.** Упаковка программ применяется по законным причинам и в установщиках безвредного программного обеспечения. Разработчики безвредных программ упаковывают свой код, поскольку это позволяет сжать ресурсы программы и уменьшить размер загружаемого установщика. Кроме того, упаковка мешает конкурентам выполнять обратную разработку и предоставляет удобный способ объединить множество программных ресурсов в одном файле установщика.

Обфускация ресурсов

Ещё один применяемый авторами вредоносных программ метод против обнаружения и анализа - обфускация ресурсов. Авторы скрывают способ хранения на диске программных ресурсов, например строк и графических изображений, а во время выполнения устраняют обфускацию, чтобы вредоносная программа могла использовать эти ресурсы. Простейший пример обфускации - прибавить 1 ко всем байтам изображений и строк, хранящихся в секции ресурсов PE, а во время выполнения вычесть 1 из всех этих данных. Разумеется, возможны самые разные способы обфускации, и каждый из них осложняет работу аналитиков, пытающихся понять двоичный файл вредоносной программы средствами статического анализа.

Как и в случае упаковки, обфускацию ресурсов можно обойти, запустив вредоносную программу в безопасной среде. Если это невозможно, остаётся лишь выяснить, каким способом программа обфусцировала ресурсы, и вручную устранить обфускацию. Именно так часто и поступают профессиональные аналитики вредоносных программ.

Методы противодействия дизассемблированию

Третью группу применяемых авторами вредоносных программ методов против обнаружения и анализа составляют методы противодействия дизассемблированию. Они используют неустранимые ограничения самых современных способов дизассемблирования, чтобы скрыть код от аналитика либо заставить его поверить, что хранящийся на диске блок кода содержит не те инструкции, которые находятся в нём на самом деле.

Один из примеров противодействия дизассемблированию - переход к такой области памяти, которую дизассемблер аналитика истолкует как другую инструкцию. Тем самым подлинные инструкции вредоносной программы фактически скрываются от специалиста по обратной разработке. У методов противодействия дизассемблированию огромные возможности, а безупречной защиты от них не существует. На практике есть два основных способа защиты: запустить образец вредоносной программы в динамической среде либо вручную выяснить, где в образце проявляются приёмы противодействия дизассемблированию и как их обойти.

Динамически загружаемые данные

Последний класс методов противодействия анализу связан с получением данных и кода из внешних источников. Например, при запуске образец вредоносной программы может динамически загружать код с внешнего сервера. В таком случае статический анализ этого кода будет бесполезен. Подобным образом программа может получать при запуске ключи расшифрования с внешних серверов, а затем использовать их для расшифрования данных или кода, необходимых при выполнении.

Очевидно, что при использовании вредоносной программой промышленного алгоритма шифрования статического анализа будет недостаточно для восстановления зашифрованных данных и кода. Такие методы противодействия анализу и обнаружению весьма эффективны. Обойти их можно лишь каким-либо способом получить код, данные или закрытые ключи с внешних серверов, а затем использовать их при анализе соответствующей вредоносной программы.

Итоги

В этой главе вы познакомились с анализом кода на языке ассемблера x86 и увидели, как выполнять статический анализ `ircbot.exe` на основе дизассемблирования с помощью инструментов Python с открытым исходным кодом. Этот материал не задуман как полное введение в язык ассемблера x86, однако теперь вы должны чувствовать себя достаточно уверенно, чтобы иметь отправную точку для выяснения происходящего в дампе ассемблерного кода вредоносной программы. Наконец, вы узнали, какими способами авторы вредоносных программ защищаются от дизассемблирования и других методов статического анализа и как можно ослабить эти стратегии против анализа и обнаружения. В главе 3 вы научитесь проводить динамический анализ вредоносных программ, компенсирующий многие недостатки статического анализа.

Глава 3. Краткое введение в динамический анализ

В главе 2 вы познакомились с более совершенными методами статического анализа, позволяющими дизассемблировать код на языке ассемблера, полученный из вредоносной программы. Статический анализ различных компонентов вредоносной программы, хранящихся на диске, может быть эффективным способом получения полезных сведений, однако он не позволяет наблюдать за поведением программы.

В этой главе вы узнаете об основах динамического анализа вредоносных программ. В отличие от статического анализа, сосредоточенного на том, как вредоносная программа выглядит в виде файла, динамический анализ заключается в её запуске в безопасной изолированной среде для наблюдения за поведением. Это напоминает помещение опасного штамма бактерий в герметичную среду с целью увидеть его воздействие на другие клетки.

Динамический анализ позволяет обходить распространённые препятствия для статического анализа, например упаковку и обфускацию, а также получать более непосредственное представление о назначении конкретного вредоносного образца. Сначала мы рассмотрим базовые методы динамического анализа, их значение для анализа данных о вредоносных программах и способы применения. Работу динамического анализа мы изучим на примерах с использованием таких инструментов с открытым исходным кодом, как `malwr.com`. Следует учитывать, что это сокращённый обзор темы, не претендующий на исчерпывающую полноту. Более полное введение можно найти в книге *Practical Malware Analysis* (No Starch Press, 2012).

Зачем нужен динамический анализ

Чтобы понять значение динамического анализа, рассмотрим проблему упакованных вредоносных программ. Напомним, что упаковкой называют сжатие или обфускацию ассемблерного кода x86 вредоносной программы с целью скрыть её вредоносную природу. При заражении целевого компьютера упакованный образец распаковывает себя, чтобы код получил возможность выполняться.

Можно попытаться дизассемблировать упакованный или обфусцированный образец с помощью средств статического анализа из главы 2, но это весьма трудоёмкий процесс. Сначала при статическом анализе пришлось бы найти обфусцированный код в файле вредоносной программы. Затем потребовалось бы отыскать подпрограммы, которые устраняют обфускацию кода, позволяя ему выполняться. Найдя эти подпрограммы, пришлось бы разобраться в их работе, чтобы применить ту же процедуру к коду. Только после этого можно было бы приступить к собственно обратной разработке вредоносного кода.

Простая, но остроумная альтернатива - выполнить вредоносную программу в безопасной изолированной среде, называемой песочницей. При запуске в песочнице программа распаковывает себя так же, как при заражении настоящей цели. Просто запустив программу, можно выяснить, к каким серверам подключается конкретный вредоносный двоичный файл, какие параметры конфигурации системы он изменяет и какие операции ввода-вывода устройств пытается выполнять.

Динамический анализ в анализе данных о вредоносных программах

Динамический анализ полезен не только для обратной разработки вредоносных программ, но и для анализа данных о них. Поскольку он раскрывает действия образца, эти действия можно сравнивать с поведением других образцов. Например, динамический анализ показывает, какие файлы вредоносные программы записывают на диск, а эти сведения позволяют связать друг с другом образцы, записывающие файлы с похожими именами. Подобные признаки помогают классифицировать вредоносные программы по общим свойствам. Они способны даже выявить образцы, созданные одними и теми же группами либо относящиеся к одним и тем же кампаниям.

Что особенно важно, динамический анализ полезен при создании детекторов вредоносных программ на основе машинного обучения. Наблюдая за поведением вредоносных и безвредных двоичных файлов во время динамического анализа, можно обучить детектор различать их. Например, проанализировав тысячи журналов динамического анализа вредоносных и безвредных файлов, система машинного обучения может усвоить, что запуск программой `msword.exe` процесса `powershell.exe` является вредоносным действием, а запуск Internet Explorer той же программой, вероятно, безобиден. В главе 8 будет подробнее рассказано о создании детекторов вредоносных программ по данным как статического, так и динамического анализа. Но прежде чем создавать сложные детекторы, рассмотрим несколько основных инструментов динамического анализа.

Основные инструменты динамического анализа

В интернете доступен ряд бесплатных инструментов динамического анализа с открытым исходным кодом. В этом разделе основное внимание уделено `malwr.com` и CuckooBox. Сайт `malwr.com` предоставляет веб-интерфейс, через который можно бесплатно отправлять двоичные файлы на динамический анализ. CuckooBox - программная платформа для создания собственной среды динамического анализа, позволяющей исследовать двоичные файлы локально. Создатели CuckooBox также обслуживают `malwr.com`, а в основе работы `malwr.com` лежит CuckooBox. Поэтому, научившись анализировать результаты на `malwr.com`, вы сможете понимать и результаты CuckooBox.

> **Примечание.** На момент отправки книги в печать интерфейс CuckooBox на `malwr.com` был закрыт на техническое обслуживание. Надеемся, к тому времени, когда вы будете читать этот раздел, сайт снова заработает. Если этого не произойдёт, сведения из главы можно применить к результатам собственного экземпляра CuckooBox, который настраивается по инструкциям на сайте cuckoosandbox.org.

Типичное поведение вредоносных программ

Ниже перечислены основные категории действий, которые образец вредоносной программы может выполнять после запуска:

• **Изменение файловой системы.** Например, запись драйвера устройства на диск, изменение файлов конфигурации системы, добавление новых программ в файловую систему и изменение ключей реестра для автоматического запуска программы.
• **Изменение реестра Windows с целью поменять конфигурацию системы.** Например, изменение настроек межсетевого экрана.

• **Загрузка драйверов устройств.** Например, загрузка драйвера, записывающего нажатия клавиш пользователем.
• **Сетевые действия.** Например, разрешение доменных имён и выполнение HTTP-запросов.

Мы подробнее исследуем это поведение на примере вредоносной программы и анализа её отчёта на `malwr.com`.

Загрузка файла на malwr.com

Чтобы прогнать образец вредоносной программы через `malwr.com`, откройте malwr.com, нажмите кнопку **Submit**, загрузите двоичный файл и отправьте его на анализ. Мы будем использовать двоичный файл, хеш SHA256 которого начинается символами `d676d95`; он находится в каталоге данных, прилагаемом к этой главе. Советую вам отправить этот файл на `malwr.com` и по мере чтения самостоятельно исследовать результаты. Страница отправки показана на рис. 3-1.

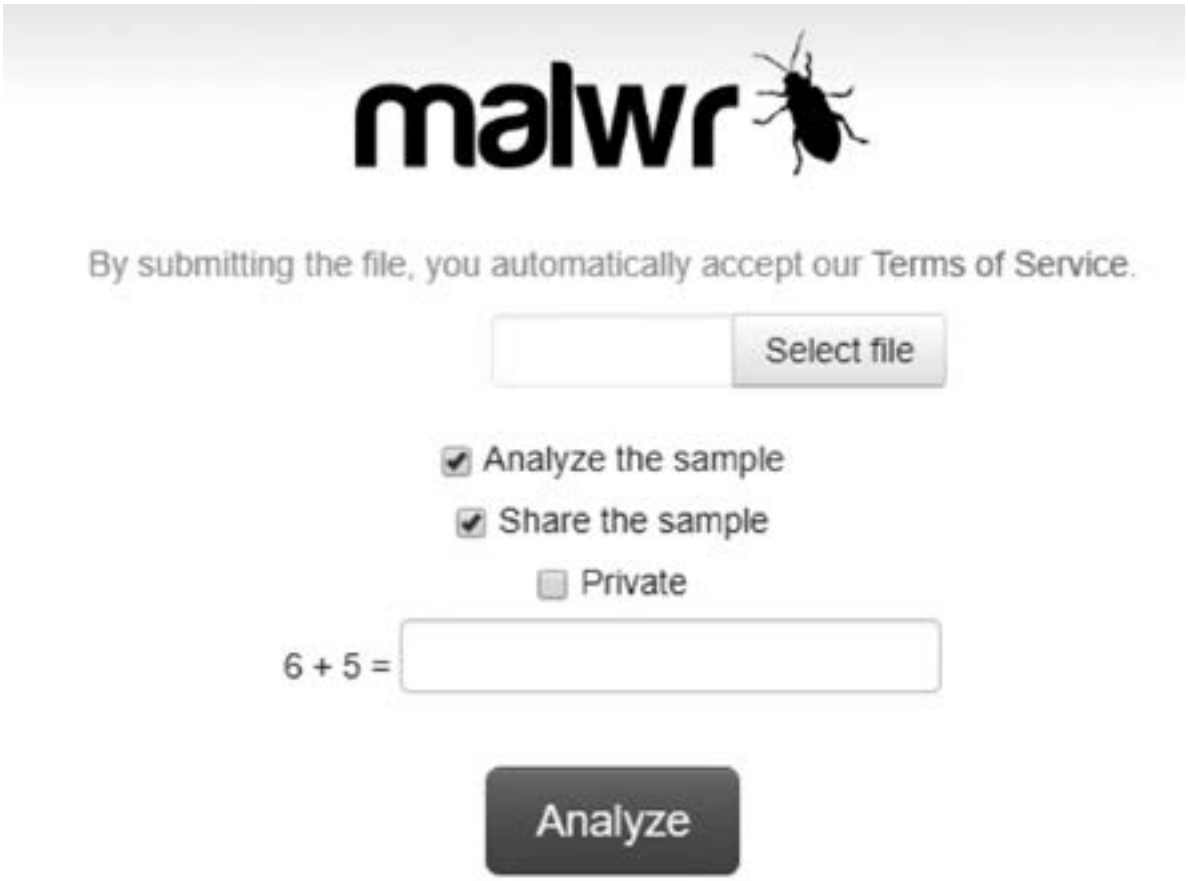

Рисунок 3-1. Страница отправки образца вредоносной программы

После отправки образца через эту форму сайт должен предложить подождать окончания анализа, которое обычно занимает около пяти минут. Когда результаты загрузятся, их можно будет изучить и понять, что делал исполняемый файл во время работы в среде динамического анализа.

Анализ результатов на malwr.com

Страница результатов для нашего образца должна выглядеть примерно так, как показано на рис. 3-2.

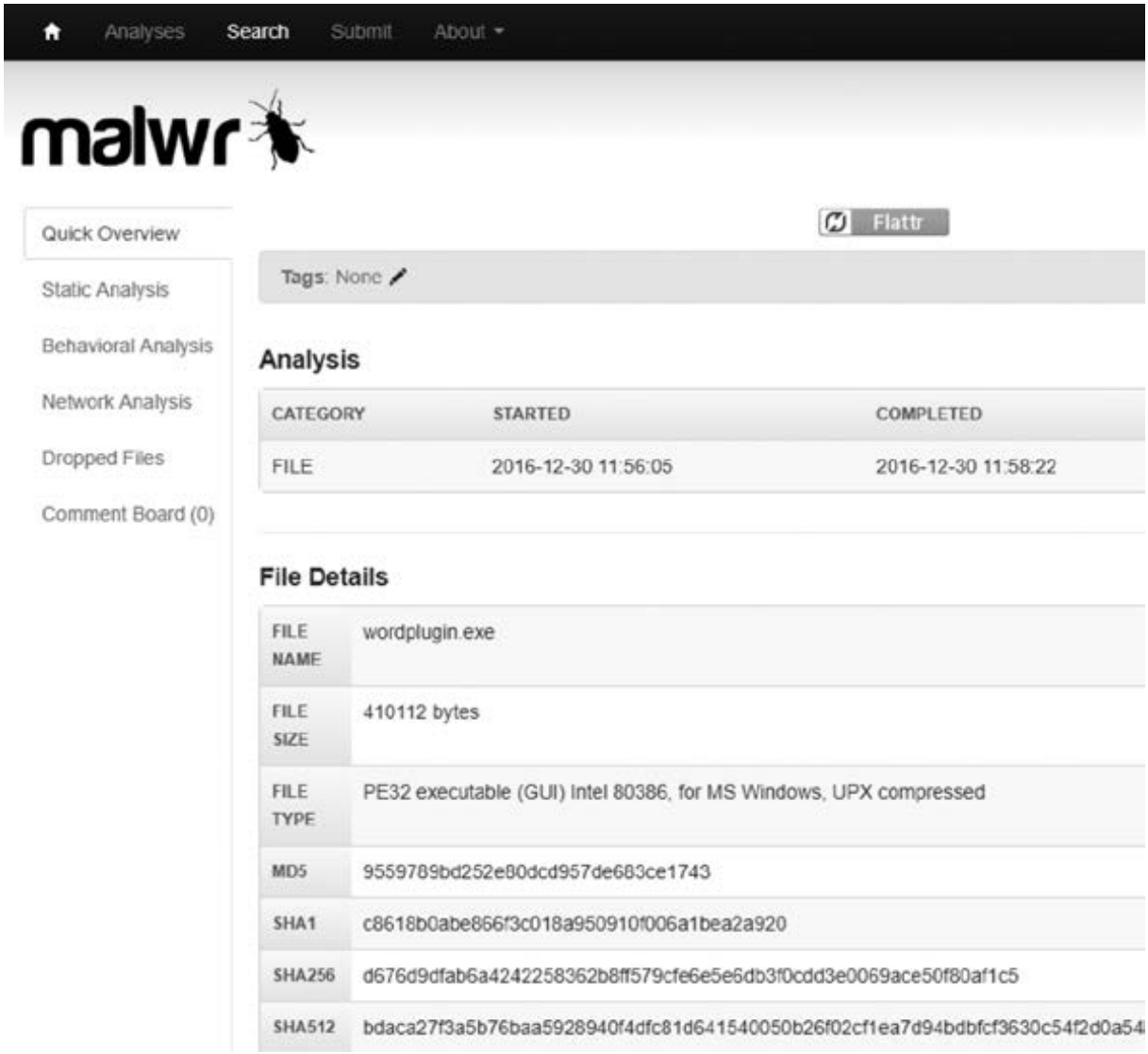

Рисунок 3-2. Верхняя часть страницы результатов анализа образца вредоносной программы на malwr.com

Результаты этого файла демонстрируют несколько важных сторон динамического анализа, которые мы сейчас рассмотрим.

Панель Signatures

Первыми на странице результатов расположены панели **Analysis** и **File Details**. В них указано время запуска файла и другие статические сведения о нём. Здесь нас интересует панель **Signatures**, показанная на рис. 3-3. Она содержит высокоуровневые сведения, полученные как из самого файла, так и из его поведения во время работы в среде динамического анализа. Разберём значение каждой из этих сигнатур.

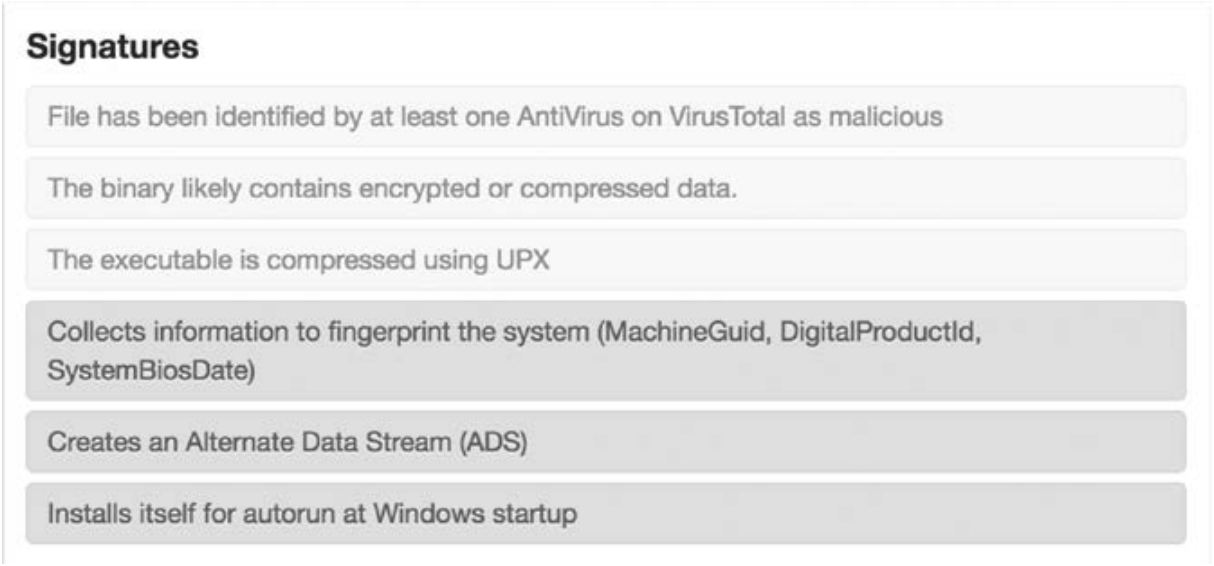

Рисунок 3-3. Сигнатуры malwr.com, соответствующие поведению нашего образца вредоносной программы

Первые три сигнатуры на рисунке получены в результате статического анализа, то есть основаны на свойствах самого файла вредоносной программы, а не на его действиях. Первая лишь сообщает, что ряд антивирусных механизмов на популярном агрегаторе VirusTotal.com пометили файл как вредоносный. Вторая указывает, что двоичный файл содержит сжатые или зашифрованные данные - распространённый признак обфускации. Третья сообщает, что файл был сжат популярным упаковщиком UPX. Сами по себе эти статические признаки не раскрывают действий файла, но позволяют заключить, что он, вероятно, вредоносный. Обратите внимание: цвет обозначает не статическую или динамическую категорию, а серьёзность каждого правила; красный цвет, представленный здесь более тёмным оттенком серого, указывает на более подозрительное поведение, чем жёлтый.

Следующие три сигнатуры получены при динамическом анализе файла. Первая указывает, что программа пытается определить аппаратное обеспечение и операционную систему компьютера. Вторая сообщает об использовании опасной возможности Windows Alternate Data Streams (ADS), которая позволяет вредоносной программе скрывать данные на диске так, что они остаются невидимыми для обычных средств просмотра файловой системы. Третья сигнатура означает, что файл изменяет реестр Windows таким образом, чтобы после перезагрузки системы автоматически выполнялась указанная им программа. Благодаря этому вредоносная программа будет запускаться заново при каждой перезагрузке пользователем системы.

Как видите, уже на уровне автоматически срабатывающих сигнатур динамический анализ существенно расширяет наши знания о предполагаемом поведении файла.

Панель Screenshots

Под панелью **Signatures** расположена панель **Screenshots**. В ней показан снимок рабочего стола среды динамического анализа во время работы вредоносной программы. Пример такого снимка приведён на рис. 3-4.

Рисунок 3-4. Снимок экрана с динамическим поведением образца вредоносной программы

Видно, что перед нами программа-вымогатель - разновидность вредоносной программы, которая шифрует файлы цели и вынуждает её заплатить за возвращение доступа к данным. Мы раскрыли назначение программы, просто запустив её, не прибегая к обратной разработке.

Панель Modified System Objects

Ряд заголовков под панелью **Screenshots** отображает сетевую активность образца. Наш двоичный файл не участвовал в сетевом обмене, но если бы он это делал, здесь были бы показаны узлы, с которыми он связывался. На рис. 3-5 представлена панель **Summary**.

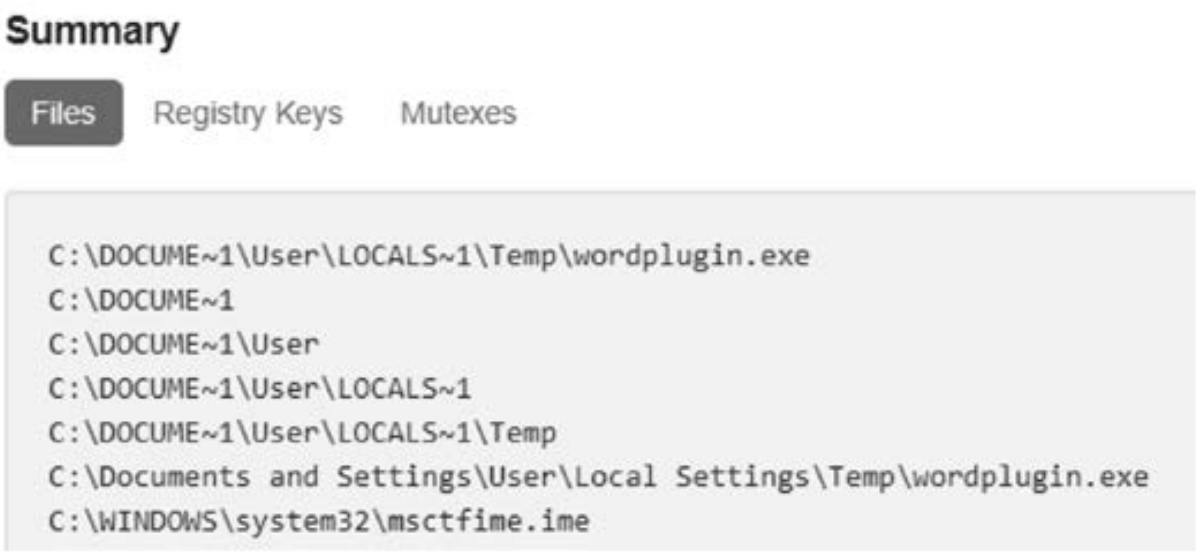

Рисунок 3-5. Вкладка Files панели Summary с файлами, изменёнными нашим образцом вредоносной программы

Здесь показаны изменённые вредоносной программой системные объекты: файлы, ключи реестра и мьютексы.

При взгляде на вкладку **Files** на рис. 3-6 становится ясно, что эта программа-вымогатель действительно зашифровала пользовательские файлы на диске.

```
C:\Perl\win32\cpan.ico
C:\Perl\win32\D3BAFC2EA6A713B05444C65E75689DA2.locked
C:\Perl\win32\onion.ico
C:\Perl\win32\D3CDFF5FA6D613B12F44BD5F75199DD1FAB3.locked
C:\Perl\win32\perldoc.ico
C:\Perl\win32\D0B9FF54A1DD13B22F31C65C756899A5FACECDED14AE.locked
C:\Perl\win32\perlhelp.ico
C:\Perl\win32\D0B9FF54A1DD13B22F46C62D751E9ED2FEB2CD9C14DB4F25.locked
```

Рисунок 3-6. Пути к файлам на вкладке Files панели Summary, указывающие, что наш образец является программой-вымогателем

После каждого пути расположен файл с расширением `.locked`, который, как можно заключить, представляет собой зашифрованную версию заменённого им файла.

Теперь рассмотрим вкладку **Registry Keys**, показанную на рис. 3-7.

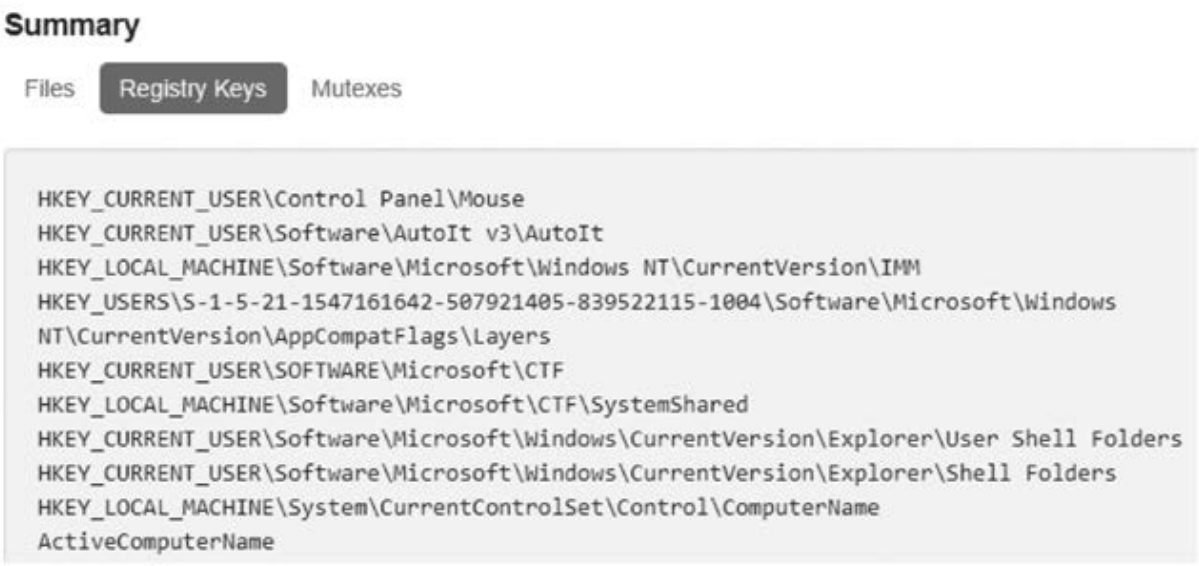

Рисунок 3-7. Вкладка Registry Keys панели Summary с ключами реестра, изменёнными нашим образцом вредоносной программы

Реестр - это база данных, в которой Windows хранит сведения о конфигурации. Параметры конфигурации хранятся в виде ключей реестра, с которыми связаны значения. Подобно путям в файловой системе Windows, компоненты ключей реестра разделяются обратными косыми чертами. `Malwr.com` показывает ключи, изменённые вредоносной программой. На рис. 3-7 этого не видно, но в полном отчёте на `malwr.com` среди примечательных изменённых ключей должен присутствовать `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`. Этот ключ предписывает Windows запускать программы при каждом входе пользователя в систему. Весьма вероятно, что вредоносная программа меняет его, чтобы Windows запускала её заново при каждой загрузке системы и заражение сохранялось после перезагрузок.

Вкладка **Mutexes** отчёта `malwr.com` содержит имена мьютексов, созданных вредоносной программой, как показано на рис. 3-8.

Рисунок 3-8. Вкладка Mutexes панели Summary с мьютексами, созданными нашим образцом вредоносной программы

Мьютексы - это файлы блокировки, которые сигнализируют, что программа завладела некоторым ресурсом. Вредоносные программы часто используют мьютексы, чтобы не заражать одну систему дважды. Известно, что по меньшей мере один из созданных мьютексов (`CTF.TimListCache.FMPDefaultS-1-5-21-1547161642-507921405-839522115-1004MUTEX.DefaultS-1-5-21-1547161642-507921405-839522115-1004 ShimCacheMutex`) сообщество специалистов по безопасности связывает с

вредоносными программами; возможно, здесь он служит именно этой цели.

Анализ вызовов API

Если щёлкнуть вкладку **Behavioral Analysis** на левой панели интерфейса `malwr.com`, как показано на рис. 3-9, откроются подробные сведения о поведении нашего вредоносного двоичного файла.

Эти сведения показывают, какие вызовы API выполнял каждый запущенный вредоносной программой процесс, а также их аргументы и возвращаемые значения. Просмотр такой информации требует много времени и экспертного знания API Windows. Подробное обсуждение анализа вызовов API вредоносной программы выходит за рамки книги, но, если вы хотите узнать больше, можно искать описания отдельных вызовов API и выяснять их последствия.

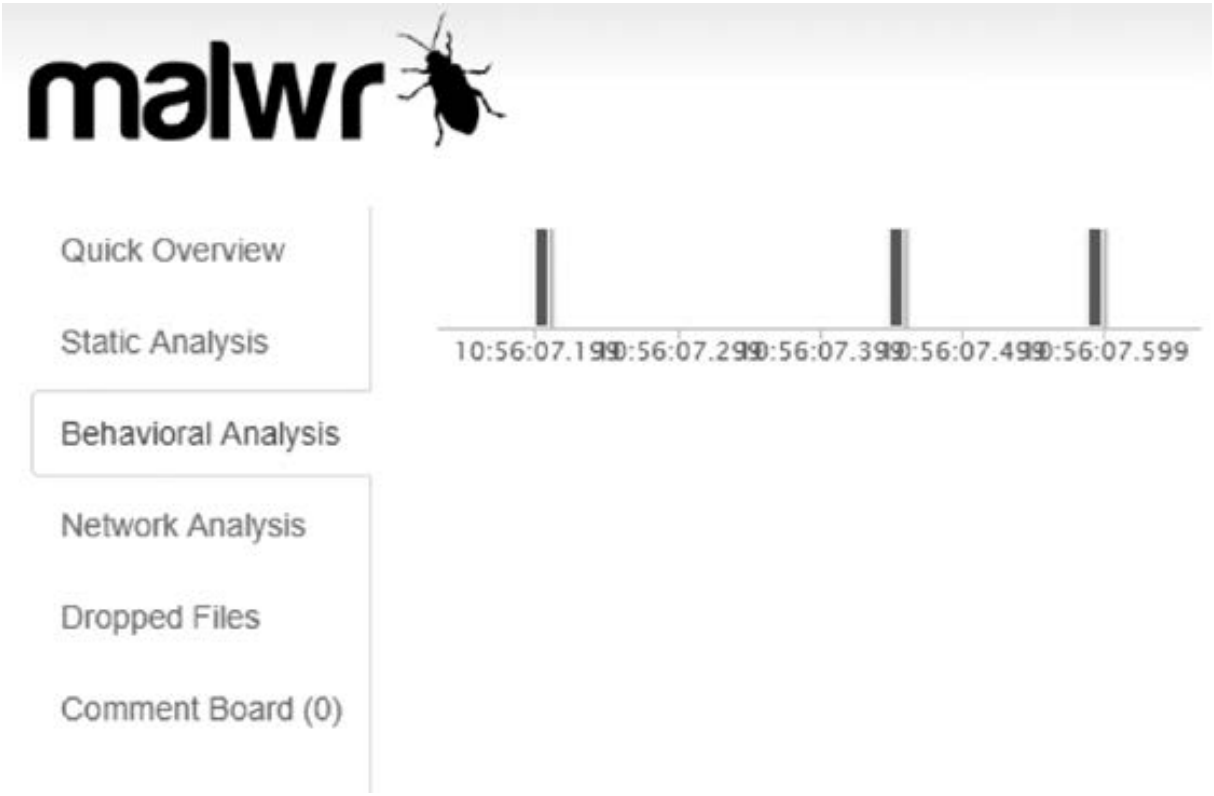

Рисунок 3-9. Панель Behavioral Analysis отчёта malwr.com по нашему образцу, показывающая время вызовов API во время динамического выполнения

`Malwr.com` прекрасно подходит для динамического анализа отдельных образцов, но гораздо хуже приспособлен к анализу большого их количества. Выполнение множества образцов в динамической среде важно для машинного обучения и анализа данных, поскольку позволяет выявлять связи между шаблонами их динамического выполнения. Для создания систем машинного обучения, способных обнаруживать вредоносные программы по шаблонам динамического выполнения, необходимо запускать тысячи образцов.

Кроме того, `malwr.com` не предоставляет результаты анализа в пригодных для машинной обработки форматах, например XML или JSON. Для решения этих проблем необходимо настроить и запустить собственный экземпляр CuckooBox. К счастью, CuckooBox распространяется бесплатно с открытым исходным кодом и снабжён пошаговыми инструкциями по созданию собственной среды динамического анализа. Советую сделать это, посетив cuckoosandbox.org. Теперь вы умеете интерпретировать результаты динамического анализа вредоносных программ на `malwr.com`, за кулисами которого работает CuckooBox, а значит, сможете анализировать и результаты

собственного экземпляра CuckooBox после его запуска.

Ограничения базового динамического анализа

Динамический анализ - мощный инструмент, но не панацея от всех проблем анализа вредоносных программ. У него есть серьёзные ограничения. Одно из них состоит в том, что авторы вредоносных программ знают о CuckooBox и других средах динамического анализа и пытаются обходить их, заставляя программу прекращать выполнение при обнаружении CuckooBox. Разработчикам CuckooBox известно об этих попытках, поэтому они стараются препятствовать обходу своей среды. Эта игра в кошки-мышки идёт непрерывно, и некоторые образцы неизбежно будут обнаруживать выполнение в среде динамического анализа и отказываться работать при попытке их запуска.

Другое ограничение заключается в том, что важные действия вредоносной программы могут не проявиться при динамическом анализе даже без попыток обойти среду. Рассмотрим двоичный файл, который после запуска подключается к удалённому серверу и ждёт команд. Например, эти команды могут предписывать искать на компьютере жертвы файлы определённых типов, записывать нажатия клавиш или включить веб-камеру. Если удалённый сервер не отправит команд либо уже не будет работать, ни одно из этих вредоносных действий не проявится. Из-за подобных ограничений динамический анализ не является универсальным решением. Профессиональные аналитики сочетают динамический и статический анализ, чтобы получить наилучшие результаты.

Итоги

В этой главе вы провели динамический анализ образца программы-вымогателя с помощью `malwr.com` и исследовали результаты. Кроме того, вы узнали о преимуществах и недостатках динамического анализа. Овладев его основами, вы готовы погрузиться в анализ данных о вредоносных программах.

Оставшаяся часть книги посвящена анализу данных о вредоносных программах, полученных посредством статического анализа. Я сосредоточусь на статическом анализе, потому что он проще и позволяет легче получить качественные результаты по сравнению с динамическим, а значит, хорошо подходит для первых практических шагов в анализе данных о вредоносных программах. Однако в каждой последующей главе я также буду объяснять, как применять методы анализа данных к сведениям, полученным при динамическом анализе.

Глава 4. Выявление кампаний атак с помощью сетей вредоносных программ

Сетевой анализ вредоносных программ позволяет превращать их наборы данных в ценные сведения об угрозах, раскрывая враждебные кампании атак, общие тактики вредоносных программ и источники образцов. Такой подход заключается в исследовании связей между группами образцов через их общие атрибуты, будь то встроенные IP-адреса, имена узлов, строки печатных символов, графические изображения и другие признаки.

Например, на рис. 4-1 показаны возможности сетевого анализа вредоносных программ: эту диаграмму удалось построить всего за несколько секунд с помощью методов, которые вы освоите в данной главе.

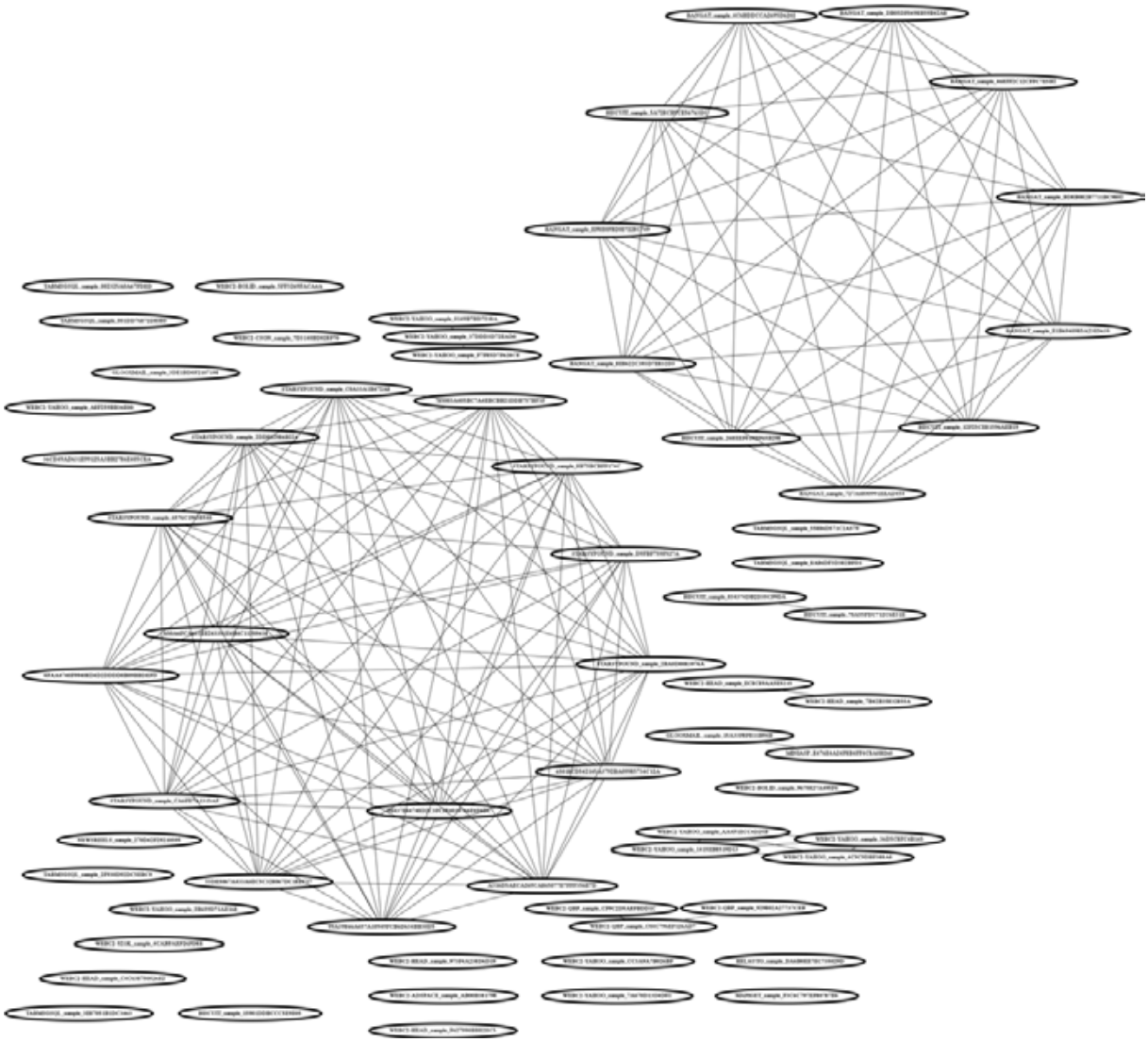

Рисунок 4-1. Социальные сетевые связи вредоносных программ государственного уровня, выявленные посредством анализа общих атрибутов

На рисунке представлена группа образцов вредоносных программ государственного уровня в виде овальных узлов и их «социальные» взаимосвязи в виде линий между узлами. Связи основаны на том, что эти образцы «перезванивают» одним и тем же узлам и IP-адресам, а значит, были развёрнуты одними злоумышленниками. В этой главе вы узнаете, как с помощью подобных связей отличить скоординированную атаку на вашу организацию от несвязанных действий множества

злоумышленников с преступными мотивами.

К концу главы вы узнаете:

• основы теории сетевого анализа применительно к извлечению сведений об угрозах из вредоносных программ;
• способы выявления связей между образцами вредоносных программ посредством визуализаций;
• как создавать и визуализировать сети вредоносных программ, а также извлекать из них сведения с помощью Python и различных наборов инструментов с открытым исходным кодом для анализа и визуализации данных;
• как объединить все эти знания для выявления и анализа кампаний атак в реальных наборах данных о вредоносных программах.

Узлы и рёбра

Прежде чем выполнять анализ общих атрибутов вредоносных программ, необходимо познакомиться с основами сетей. Сеть - это совокупность связанных объектов, называемых узлами. Связи между узлами называются рёбрами. В абстрактной математической сети как узлы, так и рёбра могут представлять практически что угодно. Нас интересует структура взаимосвязей этих узлов и рёбер, поскольку она способна раскрыть важные сведения о вредоносных программах.

При сетевом анализе вредоносных программ каждый отдельный вредоносный файл можно представить узлом, а интересующие отношения, например общий код или сетевое поведение, - рёбрами. Похожие вредоносные файлы имеют общие рёбра и поэтому при применении силовой укладки сетей объединяются в кластеры; далее вы увидите, как именно это происходит.

Есть и другой вариант: считать отдельными узлами как образцы вредоносных программ, так и их атрибуты. Например, собственные узлы будут и у IP-адресов обратного подключения, и у образцов вредоносных программ. Всякий раз, когда образец обращается к определённому IP-адресу, его узел соединяется с узлом этого IP-адреса.

Сети вредоносных программ могут быть сложнее простого набора узлов и рёбер. В частности, с узлами и рёбрами могут быть связаны атрибуты, например доля общего кода у двух соединённых образцов. Распространённым атрибутом ребра является вес: чем он больше, тем сильнее связь между образцами. У узлов тоже могут быть собственные атрибуты, например размер представляемого вредоносного файла, но такие свойства обычно называют просто атрибутами.

Двудольные сети

Двудольной называется сеть, узлы которой можно разделить на две доли, или группы, причём внутри каждой доли связей нет. Такие сети позволяют показывать общие атрибуты образцов вредоносных программ.

На рис. 4-2 приведён пример двудольной сети: узлы образцов вредоносных программ находятся в нижней доле, а доменные имена, к которым образцы «перезванивают», чтобы связаться со злоумышленником, - в другой. Обратите внимание, что адреса обратного подключения никогда не соединяются непосредственно друг с другом, как и образцы вредоносных программ; именно это характерно для двудольной сети.

Даже столь простая визуализация раскрывает важные сведения. По общим серверам обратного подключения можно предположить, что `sample_014`, вероятно, был развёрнут тем же

злоумышленником, что и sample_37D. Можно также предположить, что у sample_37D и sample_F7F один злоумышленник и что sample_014 и sample_F7F, вероятно, тоже связаны с одним злоумышленником, поскольку их соединяет образец sample_37D. В действительности все образцы на рис. 4-2 происходят от одной китайской группы злоумышленников APT1.

Примечание. Мы благодарим компанию Mandiant и Милу Паркур за систематизацию образцов APT1 и предоставление их исследовательскому сообществу.

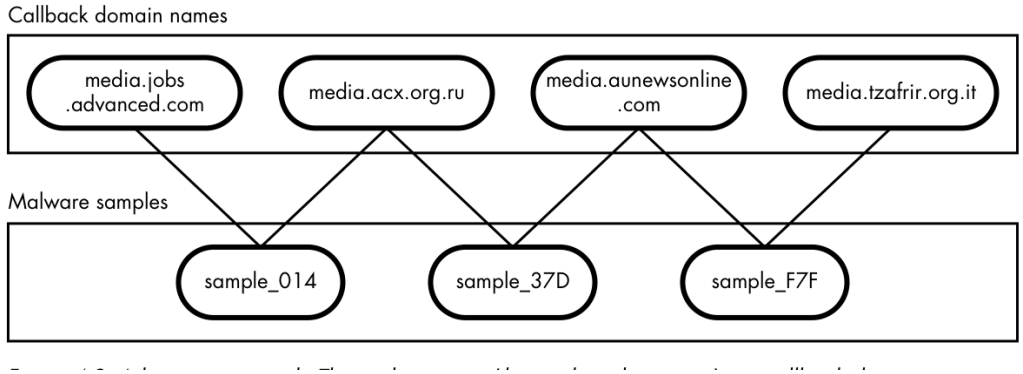

Рисунок 4-2. Двудольная сеть. Верхние узлы, то есть доля атрибутов, представляют доменные имена обратного подключения. Нижние узлы, то есть доля вредоносных программ, представляют их образцы.

Когда количество узлов и связей в сети становится очень большим, может потребоваться увидеть только отношения между образцами вредоносных программ, не исследуя внимательно все связи атрибутов. Сходство образцов можно рассмотреть с помощью проекции двудольной сети - её упрощённой версии, в которой узлы одной доли соединяются, если у них есть общие узлы в другой доле, то есть доле атрибутов. Например, для образцов с рис. 4-1 мы получили бы сеть, где вредоносные программы соединены при наличии общих доменных имён обратного подключения.

На рис. 4-3 показана проекция сети общих серверов обратного подключения для всего упомянутого ранее набора данных китайской группы APT1.

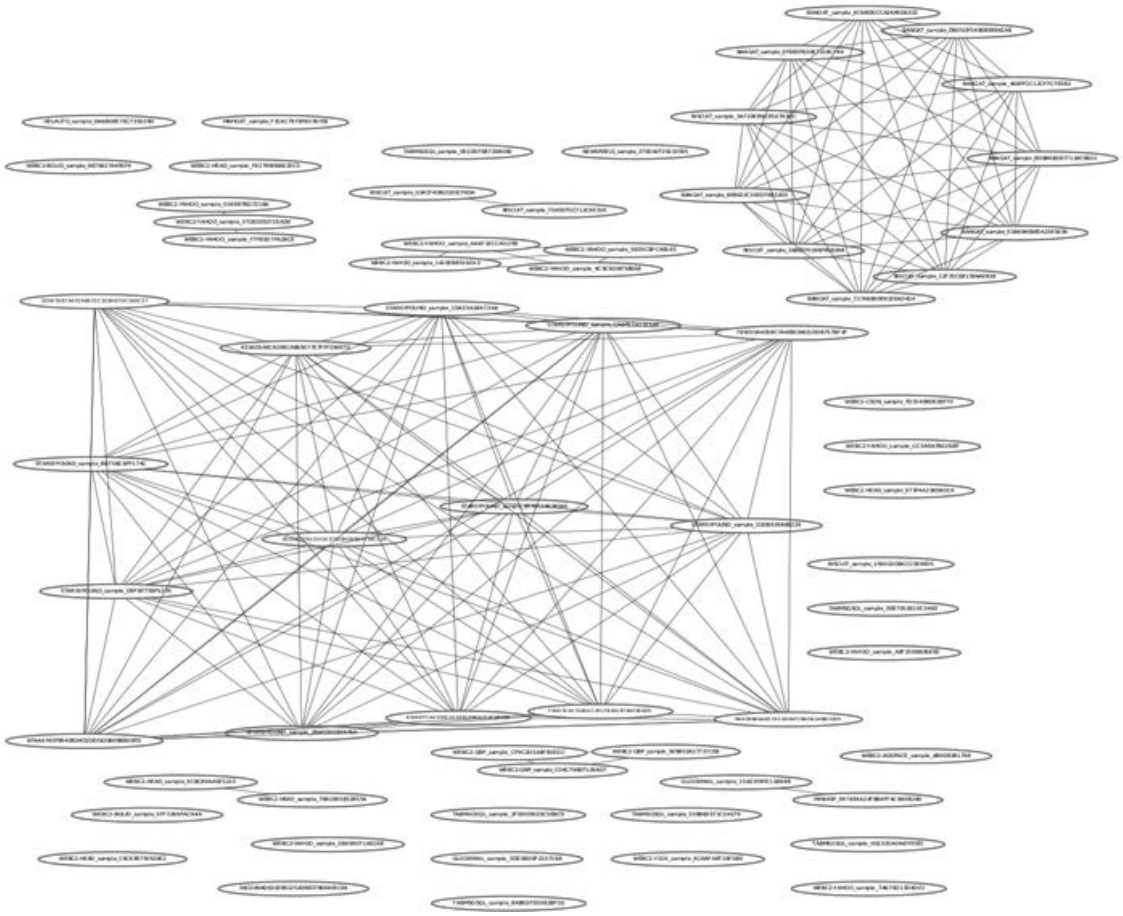

Рисунок 4-3. Проекция образцов из набора данных APT1, где два образца соединяются лишь при наличии хотя бы одного общего сервера. Два крупных кластера использовались в двух разных кампаниях атак.

Узлы здесь представляют образцы вредоносных программ и соединяются, если у них есть хотя бы один общий сервер обратного подключения. Показывая связи только между образцами с общими серверами, мы начинаем видеть общую «социальную сеть» этих вредоносных программ. На рис. 4-3 различимы две большие группы: крупный квадратный кластер слева от центра и круглый кластер в правой верхней части. Дальнейшее исследование показало, что они соответствуют двум разным кампаниям, проведённым за десятилетнюю историю группы APT1.

Визуализация сетей вредоносных программ

При анализе общих атрибутов вредоносных программ с помощью сетей вы будете активно применять программы сетевой визуализации для построения сетей, подобных уже показанным. В этом разделе рассказывается, как создавать такие визуализации с алгоритмической точки зрения.

Основная сложность сетевой визуализации - укладка сети, то есть выбор положения каждого узла в двух- или трёхмерном пространстве координат в зависимости от желаемой размерности визуализации. В идеале узлы следует располагать так, чтобы видимое расстояние между ними было пропорционально длине кратчайшего пути в сети. Иными словами, узлы, разделённые двумя переходами, могли бы находиться примерно в двух дюймах друг от друга, а разделённые тремя переходами - примерно в трёх дюймах. Такой подход позволяет отображать кластеры похожих узлов в соответствии с их настоящими отношениями. Однако, как вы увидите в следующем разделе, этого часто трудно добиться, особенно в сети более чем из трёх узлов.

Проблема искажения

Оказывается, задачу укладки сети зачастую невозможно решить идеально. Эта трудность показана на рис. 4-4.

Во всех этих простых сетях каждый узел соединён со всеми остальными рёбрами одинакового веса 1. В идеальной укладке все узлы находились бы на странице на равном расстоянии друг от друга. Но при переходе к сетям из четырёх, а затем пяти узлов, как на схемах (в) и (г), неравная длина рёбер приводит ко всё большему искажению. К сожалению, это искажение можно лишь минимизировать, но не устранить, и такая минимизация становится одной из главных целей алгоритмов сетевой визуализации.

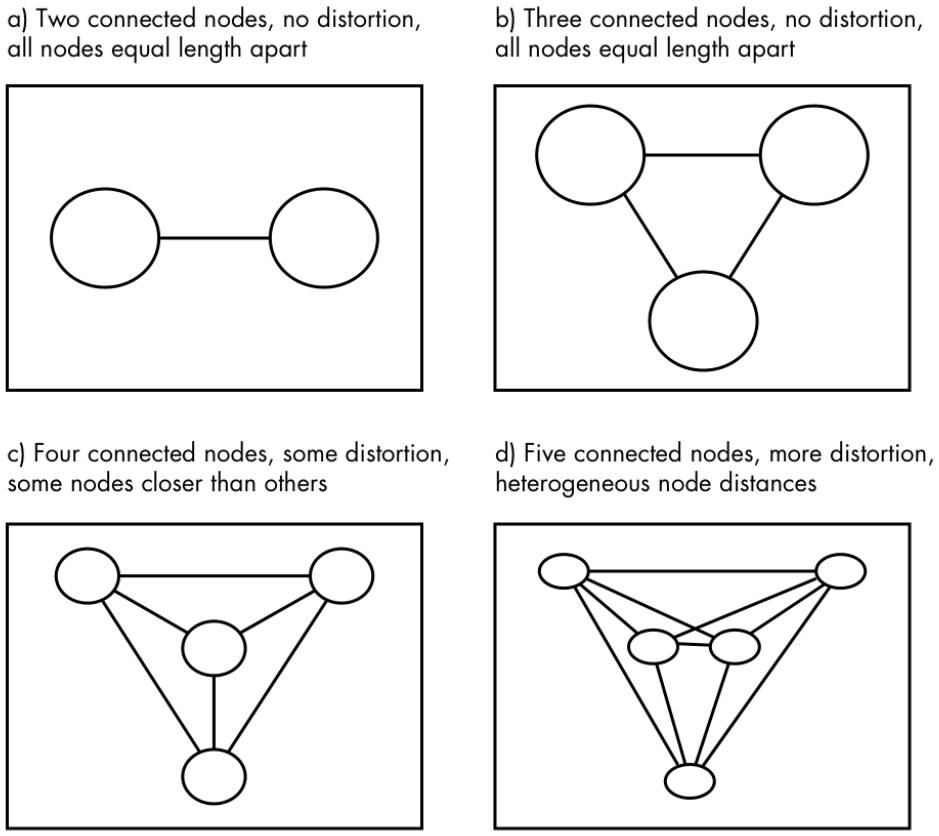

Рисунок 4-4. Идеальная укладка реальной сети вредоносных программ обычно невозможна. В простых случаях (а) и (б) все узлы можно расположить на равном расстоянии. В случае (в) возникает искажение: рёбра имеют уже неодинаковую длину. В случае (г) искажение становится ещё сильнее.

Силовые алгоритмы

Для наиболее эффективного уменьшения искажений укладки специалисты по информатике часто применяют силовые алгоритмы. Они основаны на физическом моделировании сил пружин и магнетизма. Представление рёбер сети в виде физических пружин часто позволяет удачно расположить узлы, поскольку виртуальные пружины толкают и тянут их, стремясь выровнять длину рёбер между узлами. Чтобы лучше понять идею, вспомните поведение пружины: после сжатия или растяжения она «пытается» вернуться к равновесной длине. Эти свойства хорошо согласуются с нашим стремлением сделать все рёбра сети одинаковой длины. Именно силовым алгоритмам посвящена эта глава.

Построение сетей с помощью NetworkX

Теперь, когда вы имеете общее представление о сетях вредоносных программ, можно научиться создавать сети их взаимосвязей с помощью библиотеки сетевого анализа Python с открытым исходным кодом NetworkX и набора средств визуализации сетей GraphViz с открытым исходным кодом. Я покажу, как программно извлекать данные о вредоносных программах, а затем строить, визуализировать и анализировать на их основе сети, представляющие наборы данных.

Начнём с NetworkX - проекта с открытым исходным кодом, который сопровождает команда, сосредоточенная в Лос-Аламосской национальной лаборатории, и фактической стандартной библиотеки Python для обработки сетей. Напомним, что зависимости для этой главы, включая NetworkX, можно установить, перейдя в каталог кода и данных главы и выполнив команду `pip install -r requirements.txt`. Если вы знаете Python, то, вероятно, удивитесь простоте NetworkX. Код из листинга 4-1 импортирует NetworkX и создаёт экземпляр сети.

```
#!/usr/bin/python
import networkx

# создать сеть без узлов и рёбер
network = networkx.Graph()
```

Листинг 4-1. Создание экземпляра сети

В этом коде сеть NetworkX создаётся всего одним вызовом конструктора `Graph`.

> **Примечание.** Библиотека NetworkX иногда употребляет термин *граф* вместо *сети*: в информатике эти термины являются синонимами и обозначают множество узлов, соединённых рёбрами.

Добавление узлов и рёбер

Создав экземпляр сети, добавим в неё несколько узлов. Узлом сети NetworkX может быть любой объект Python. Добавим в нашу сеть узлы разных типов:

```
nodes = ["hello","world",1,2,3]
for node in nodes:
    network.add_node(node)
```

Как видите, мы добавили пять узлов: `"hello"`, `"world"`, `1`, `2` и `3`.

Затем для добавления рёбер вызовем `add_edge()`:

```
(1) network.add_edge("hello","world")
   network.add_edge(1,2)
```

```
    network.add_edge(1,3)
```

Здесь некоторые из пяти узлов соединяются рёбрами. Например, первая строка (1) связывает узлы `"hello"` и `"world"`, создавая между ними ребро.

Добавление атрибутов

NetworkX позволяет без труда связывать атрибуты как с узлами, так и с рёбрами. Чтобы добавить атрибут к узлу и впоследствии обращаться к нему, передайте атрибут как именованный аргумент при добавлении узла в сеть:

```
network.add_node(1,myattribute="foo")
```

Чтобы добавить атрибут позднее, обратитесь к словарю узлов сети следующим образом:

```
network.node[1]["myattribute"] = "foo"
```

Для чтения атрибута также используется словарь узлов:

```
print network.node[1]["myattribute"] # выводит "foo"
```

Как и в случае узлов, атрибуты рёбер можно задавать именованными аргументами при первоначальном добавлении рёбер:

```
network.add_edge("node1","node2",myattribute="attribute of an edge")
```

Атрибут можно добавить и после появления ребра в сети, воспользовавшись словарём рёбер:

```
network.edge["node1"]["node2"]["myattribute"] = "attribute of an edge"
```

Словарь рёбер обладает удобным свойством: к атрибутам можно обращаться и в обратном порядке, не задумываясь о том, какой узел указан первым, как показано в листинге 4-2.

```
(1) network.edge["node1"]["node2"]["myattribute"] = 321
(2) print network.edge["node2"]["node1"]["myattribute"]  # выводит 321
```

Листинг 4-2. Обращение через словарь рёбер к атрибутам узлов независимо от порядка

Первая строка задаёт `myattribute` для ребра между `node1` и `node2` (1), а вторая обращается к `myattribute`, хотя ссылки на `node1` и `node2` переставлены местами (2).

Сохранение сетей на диск

Чтобы визуализировать сеть, её необходимо сохранить из NetworkX на диск в формате `.dot`, широко распространённом в сетевом анализе и поддерживаемом многими наборами средств сетевой визуализации. Для этого достаточно вызвать функцию NetworkX `write_dot()`, как показано в листинге 4-3.

```
#!/usr/bin/python
import networkx
from networkx.drawing.nx_agraph import write_dot

# создать сеть, добавить несколько узлов и соединить их
nodes = ["hello","world",1,2,3]
network = networkx.Graph()
for node in nodes:
    network.add_node(node)
network.add_edge("hello","world")
write_dot((1)network, (2)"network.dot")
```

Листинг 4-3. Сохранение сети на диск с помощью write_dot()

Как видно в конце кода, функции `write_dot()` передаются сохраняемая сеть (1), а также путь или имя файла для её сохранения (2).

Визуализация сетей с помощью GraphViz

После записи сети на диск функцией NetworkX `write_dot()` получившийся файл можно визуализировать с помощью GraphViz. GraphViz - лучший из доступных пакетов командной строки для визуализации сетей. Он поддерживается исследователями AT&T и стал стандартной частью набора средств сетевого анализа, применяемого аналитиками данных. В него входит множество инструментов командной строки для укладки и визуализации сетей. GraphViz заранее установлен на виртуальной машине, прилагаемой к книге; его также можно загрузить по адресу graphviz.gitlab.io. Каждый инструмент GraphViz принимает сеть в формате `.dot` и вызывается со следующим синтаксисом для визуализации сети в файле `.png`:

```
$ <toolname> <dotfile> -T png -o <outputfile.png>
```

Одним из инструментов GraphViz является силовой визуализатор графов `fdp`. Он использует тот же основной интерфейс командной строки, что и все остальные инструменты GraphViz:

```
$ fdp apt1callback.dot -T png -o apt1callback.png
```

Здесь указано, что следует применить инструмент `fdp` к файлу сети `.dot` с именем `apt1callback.dot`, находящемуся в каталоге `~/ch3/` прилагаемых к книге данных. Параметр `-T png` задаёт желаемый формат PNG. Наконец, с помощью `-o apt1callback.png` указывается файл для сохранения результата.

Настройка сетей с помощью параметров

В инструментах GraphViz есть множество параметров для настройки отрисовки сетей. Многие из них задаются флагом командной строки `-G` в следующем формате:

```
G<parametername>=<parametervalue>
```

Два особенно полезных параметра - `overlap` и `splines`. Значение `false` параметра `overlap` запрещает GraphViz накладывать узлы друг на друга. Параметр `splines` предписывает GraphViz рисовать не прямые, а кривые линии, чтобы рёбра сети было легче прослеживать. Ниже приведено несколько способов задать эти параметры в GraphViz.

Следующая команда предотвращает наложение узлов:

```
$ <toolname> <dotfile> -Goverlap=false -T png -o <outputfile.png>
```

Эта команда изображает рёбра кривыми линиями, или сплайнами, для повышения удобочитаемости сети:

```
$ <toolname> <dotfile> -Gsplines=true -T png -o <outputfile.png>
```

Следующая команда одновременно изображает рёбра сплайнами и запрещает визуальное наложение узлов:

```
$ <toolname> <dotfile> -Gsplines=true -Goverlap=false -T png -o <outputfile.png>
```

Обратите внимание, что параметры просто перечисляются друг за другом: `-Gsplines=true -Goverlap=false`, причём порядок не имеет значения, а затем следуют `-T png -o <outputfile.png>`.

В следующем разделе я расскажу о нескольких наиболее полезных инструментах GraphViz помимо `fdp`.

Инструменты командной строки GraphViz

Ниже перечислены инструменты GraphViz, которые я считаю наиболее полезными, и ситуации, в которых уместно применять каждый из них.

fdp

В предыдущем примере мы использовали инструмент укладки `fdp` для создания силовой укладки, описанной в разделе «Силовые алгоритмы» на с. 40. При построении сетей вредоносных программ менее чем из 500 узлов `fdp` за приемлемое время хорошо выявляет структуру сети. Но при работе более чем с 500 узлами, особенно если связи между ними сложны, быстродействие `fdp` довольно резко снижается.

Чтобы испытать `fdp` на сети общих серверов обратного подключения APT1 с рис. 4-3, выполните следующую команду из каталога `ch4` прилагаемых к книге данных; GraphViz должен быть установлен:

```
$ fdp callback_servers_malware_projection.dot -T png -o fdp_servers.png -Goverlap=false
```

Команда создаст файл `.png` с именем `fdp_servers.png`, содержащий сеть, подобную изображённой на рис. 4-5.

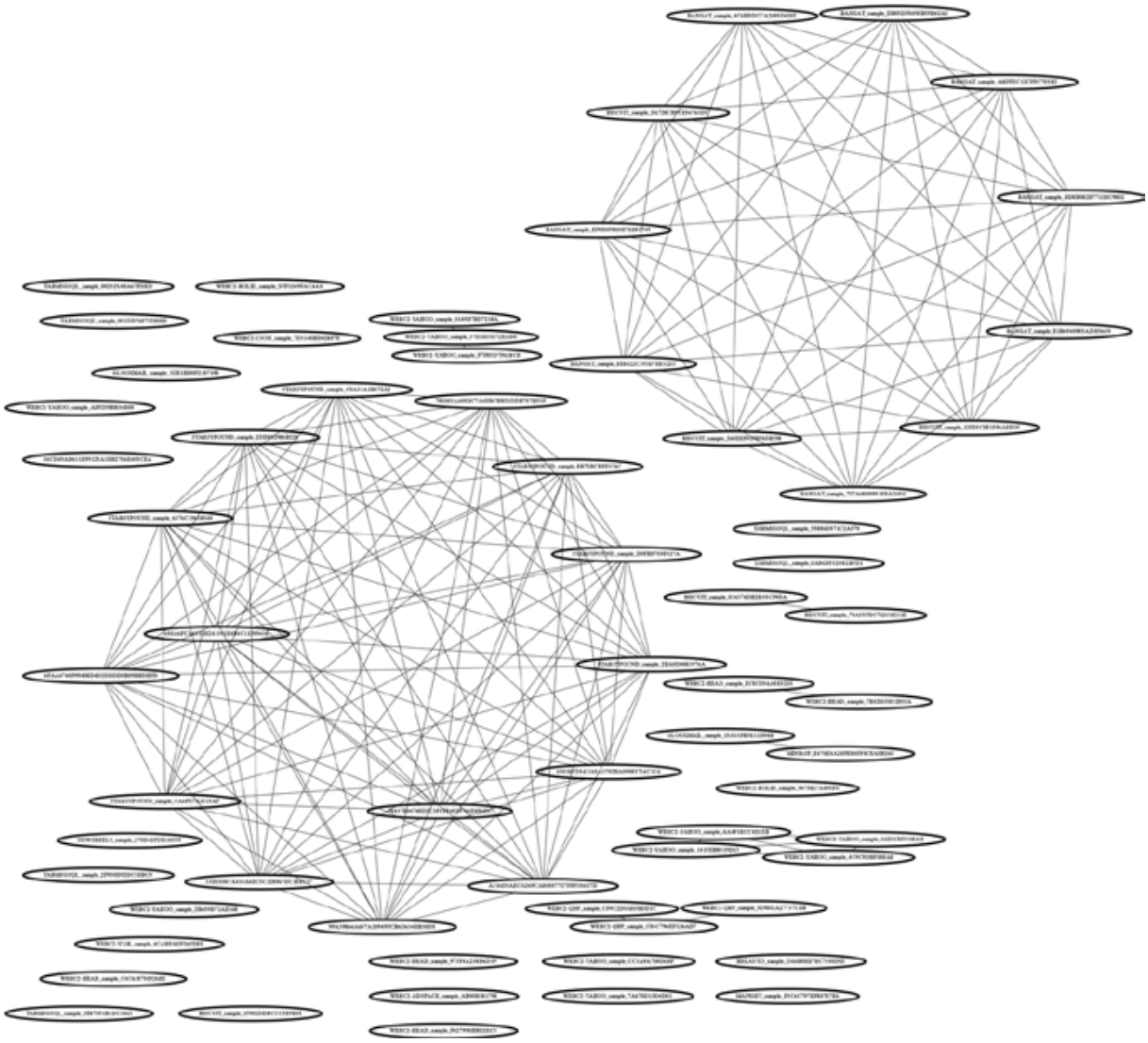

Рисунок 4-5. Укладка образцов APT1 с помощью инструмента fdp

Укладка `fdp` делает наглядными несколько особенностей рисунка. Во-первых, два больших кластера образцов тесно связаны, что хорошо видно в правой верхней и левой нижней частях. Во-вторых, в правой нижней части виден ряд связанных пар образцов. Наконец, многие образцы не демонстрируют явных отношений друг с другом и не соединены ни с какими другими узлами. Важно помнить, что визуализация основана на общих серверах обратного подключения. Не исключено, что несоединённые образцы связаны с другими образцами на рисунке отношениями иного типа, например общим кодом; такие отношения мы рассмотрим в главе 5.

sfdp

Инструмент `sfdp` использует приблизительно тот же подход к укладке, что и `fdp`, но лучше масштабируется, поскольку создаёт иерархию упрощений, называемых огрублениями, в которых близкие узлы объединяются в суперузлы. Завершив огрубление, `sfdp` укладывает объединённые версии графа со значительно меньшим количеством узлов и рёбер, что резко ускоряет процесс. Благодаря этому для поиска наилучших положений узлов `sfdp` выполняет меньше вычислений. В результате на обычном ноутбуке он способен укладывать десятки тысяч узлов и поэтому значительно превосходит остальные алгоритмы при работе с очень большими сетями вредоносных программ.

Однако за масштабируемость приходится платить: иногда `sfdp` создаёт менее ясные укладки, чем `fdp` для сетей того же размера. Например, сравните рис. 4-6, созданный с помощью `sfdp`, с сетью `fdp` на рис. 4-5.

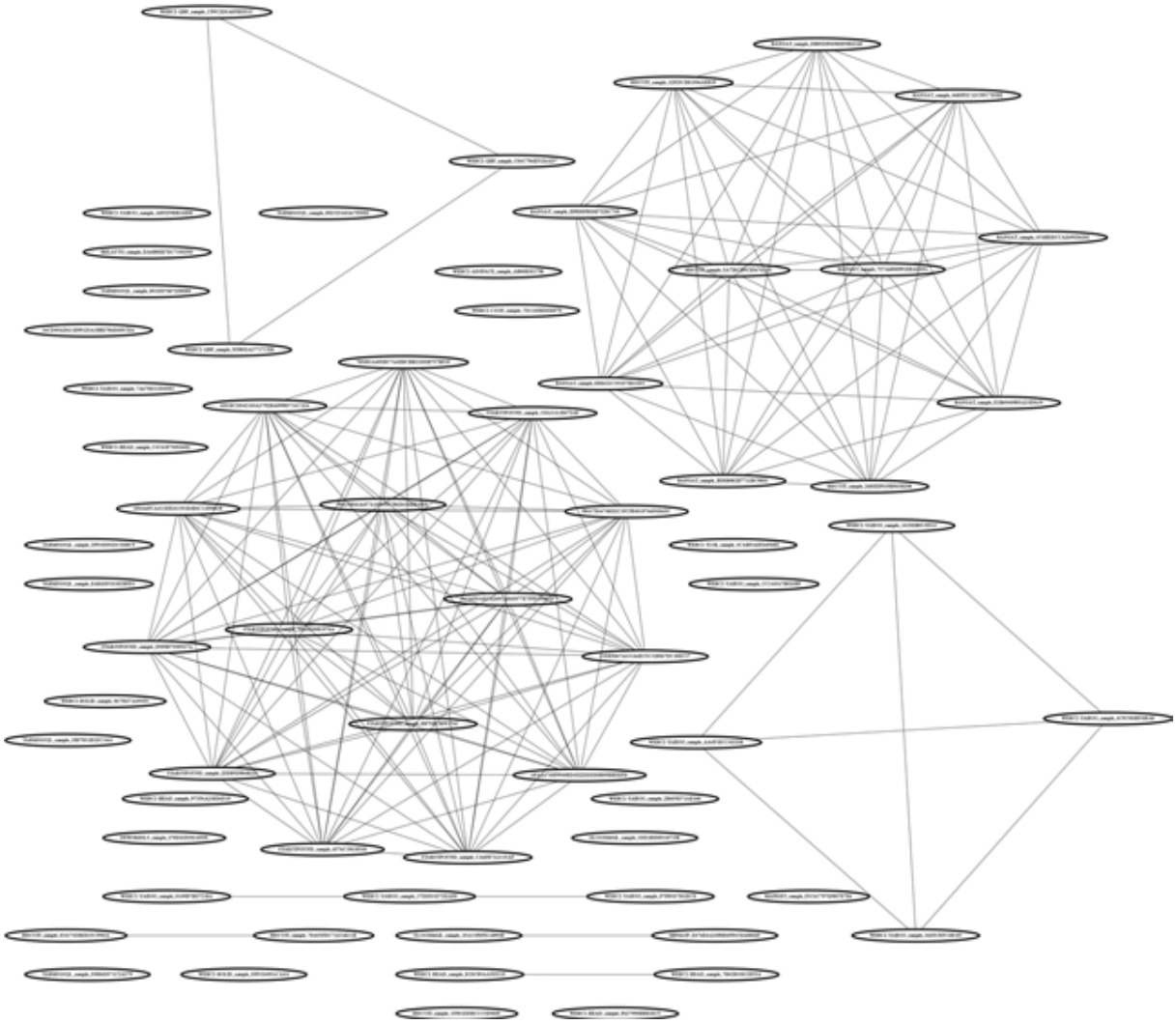

Рисунок 4-6. Укладка сети общих серверов обратного подключения образцов APT1 командой sfdp

Как видите, поверх каждого кластера на рис. 4-6 немного больше визуального шума, поэтому происходящее различить несколько труднее.

Чтобы создать эту сеть, перейдите в каталог `ch4` прилагаемых к книге данных и выполните следующую команду. Она создаст файл изображения `sfdp_servers.png`, показанный на рис. 4-6:

```
$ sfdp callback_servers_malware_projection.dot -T png -o sfdp_servers.png -Goverlap=false
```

Обратите внимание: первый элемент команды указывает, что используется `sfdp`, а не прежний `fdp`. За исключением имени выходного файла, всё остальное осталось без изменений.

neato

Инструмент `neato` реализует в GraphViz другой силовой алгоритм укладки сети. Он создаёт виртуальные пружины между всеми узлами, в том числе несоединёнными, которые помогают вытолкнуть узлы к идеальным положениям, но требуют дополнительных вычислений. Заранее трудно определить, для какой сети `neato` создаст наилучшую укладку. Я советую попробовать его вместе с `fdp` и выбрать более понравившийся результат. На рис. 4-7 показана укладка `neato` для сети общих серверов обратного подключения APT1.

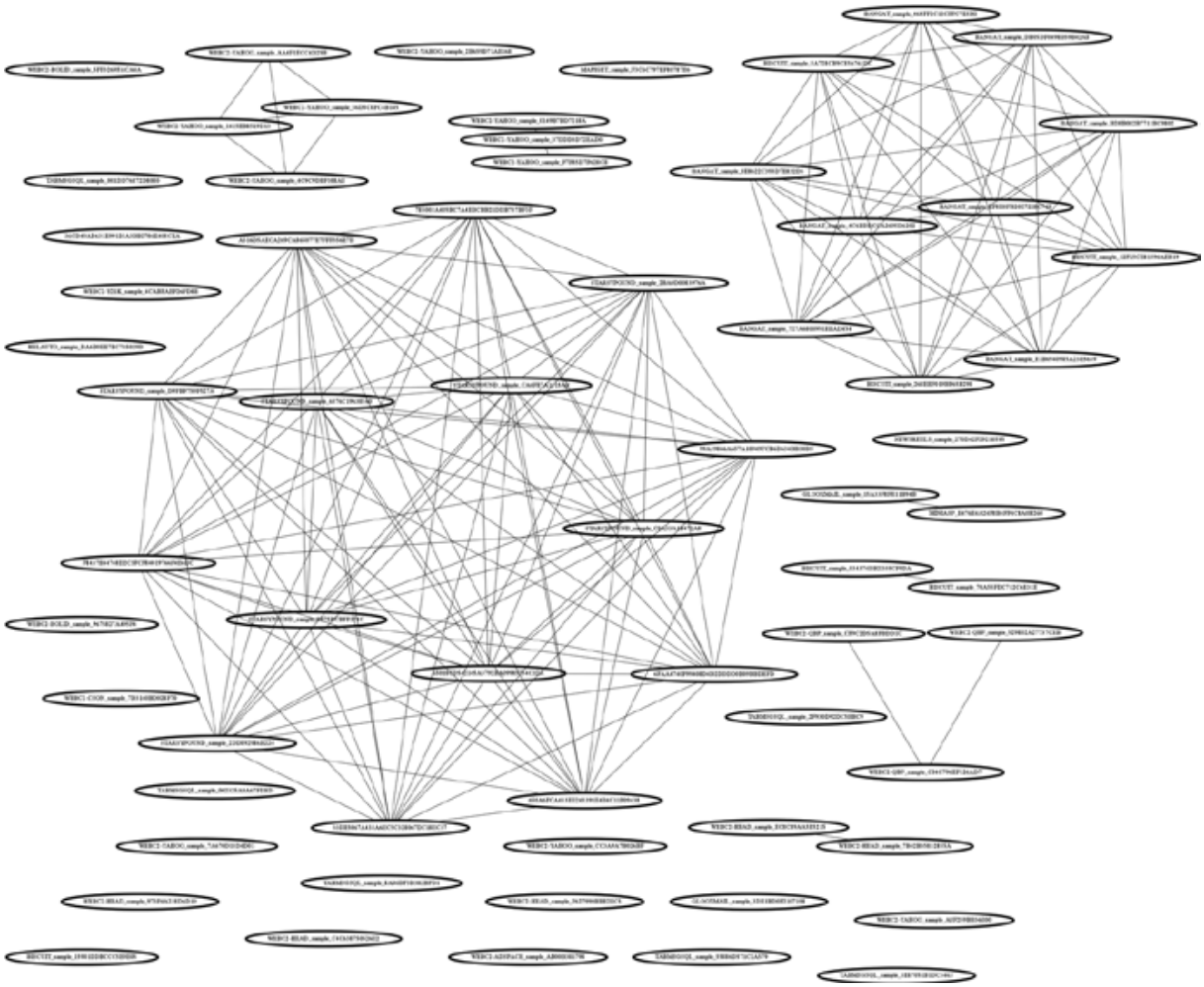

Рисунок 4-7. Укладка сети общих серверов обратного подключения APT1 с помощью neato

В данном случае `neato` создаёт укладку, похожую на результаты `fdp` и `sfdp`. Однако для некоторых наборов данных его результат будет лучше или хуже - инструмент просто нужно испытать на своих данных. Чтобы проверить `neato`, выполните следующую команду из каталога `ch4` прилагаемых к книге данных. Она должна создать файл изображения сети `neato_servers.png`, показанный на рис. 4-7:

```
$ neato callback_servers_malware_projection.dot -T png -o neato_servers.png -
    Goverlap=false
```

Для создания сети мы лишь изменили код, которым строили рис. 4-6: выбрали инструмент `neato` и сохранили файл `.png` под именем `neato_servers.png`. Теперь вы умеете создавать сетевые визуализации, и можно рассмотреть способы их улучшения.

Добавление визуальных атрибутов узлов и рёбер

Помимо выбора общей укладки сети, полезно иметь возможность задавать способ отображения отдельных узлов и рёбер. Например, толщину ребра можно связать с силой связи между двумя узлами, а цвет узла - с инцидентом, к которому относится представляемый им образец. Это позволит лучше видеть кластеры вредоносных программ. NetworkX и GraphViz упрощают эту задачу: визуальные свойства узлов и рёбер задаются значениями набора атрибутов. В следующих разделах я рассматриваю лишь несколько таких атрибутов, хотя сама тема достаточно обширна для отдельной книги.

Толщина ребра

Чтобы задать толщину рамки, которую GraphViz рисует вокруг узла, или линии ребра, присвойте атрибуту узла или ребра `penwidth` выбранное числовое значение, как показано в листинге 4-4.

```
#!/usr/bin/python
import networkx
from networkx.drawing.nx_agraph import write_dot

(1) g = networkx.Graph()
   g.add_node(1)
   g.add_node(2)
   g.add_edge(1,2,(2)penwidth=10) # сделать ребро особенно толстым
   write_dot(g,'network.dot')
```

Листинг 4-4. Задание атрибута penwidth

Здесь я создаю простую сеть (1) из двух узлов, соединённых ребром, и задаю для ребра атрибут `penwidth`, равный 10 (2); значение по умолчанию равно 1. Запустив код, вы должны увидеть изображение, похожее на рис. 4-8.

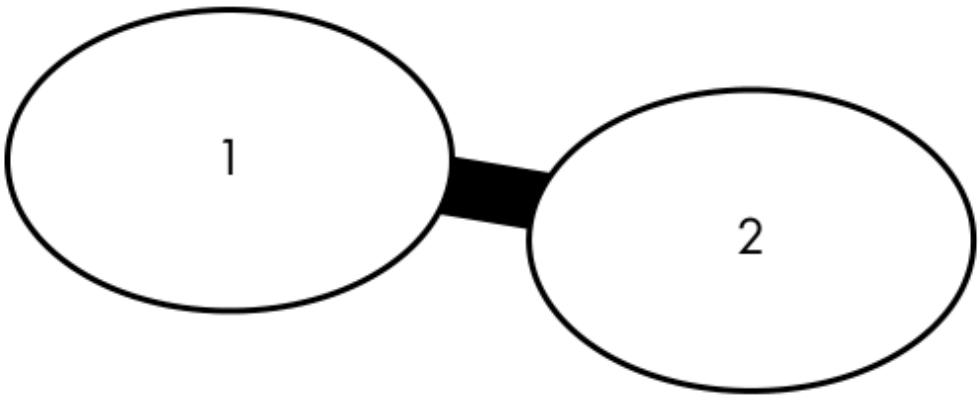

Рисунок 4-8. Простая сеть с ребром, для которого penwidth равен 10

Как видно на рис. 4-8, значение `penwidth`, равное 10, создаёт очень толстое ребро. Толщина ребра или рамки узла, если `penwidth` задан для узла, изменяется пропорционально значению атрибута, поэтому выбирайте его соответственно. Например, если значения силы рёбер изменяются от 1 до 1000, но необходимо видеть все рёбра, атрибуты `penwidth` можно назначать по логарифмической шкале значений силы.

Цвет узла и ребра

Цвет рамки узла или ребра задаётся атрибутом `color`. В листинге 4-5 показано, как это сделать.

```
#!/usr/bin/python

import networkx
from networkx.drawing.nx_agraph import write_dot

g = networkx.Graph()
g.add_node(1,(1)color="blue") # сделать контур узла синим
g.add_node(2,(2)color="pink") # сделать контур узла розовым
g.add_edge(1,2,(3)color="red") # сделать ребро красным
write_dot(g,'network.dot')
```

Листинг 4-5. Задание цветов узлов и рёбер

Здесь я создаю ту же простую сеть, что и в листинге 4-4: два узла и соединяющее их ребро. Для каждого создаваемого узла задаётся значение `color` (1) и (2). Значение цвета ребра (3) также устанавливается при его создании.

На рис. 4-9 показан результат листинга 4-5. Как и ожидалось, первый узел, ребро и второй узел должны иметь разные цвета. Полный перечень доступных цветов приведён по адресу graphviz.org.

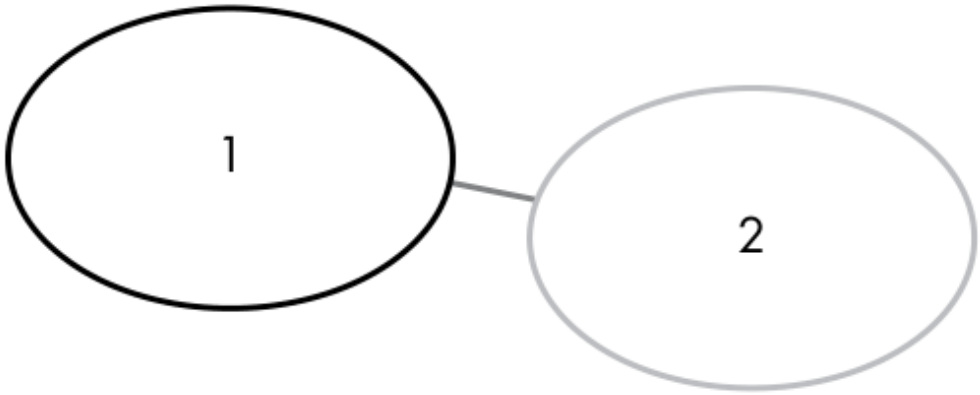

Рисунок 4-9. Простая сеть, демонстрирующая задание цветов узлов и рёбер

Цвета можно применять для обозначения различных классов узлов и рёбер.

Форма узла

Форма узла задаётся атрибутом `shape`, которому присваивается строка с названием фигуры, определённой по адресу graphviz.org. Часто используются значения `box`, `ellipse`, `circle`, `egg`, `diamond`, `triangle`, `pentagon` и `hexagon`. В листинге 4-6 показано, как задать атрибут формы узла.

```
#!/usr/bin/python

import networkx
from networkx.drawing.nx_agraph import write_dot

g = networkx.Graph()
g.add_node(1,(1)shape='diamond')
g.add_node(2,(2)shape='egg')
g.add_edge(1,2)

write_dot(g,'network.dot')
```

Листинг 4-6. Задание формы узлов

Подобно заданию цвета узла, мы просто передаём именованный аргумент `shape` функции `add_node()`, чтобы указать желаемую форму каждого узла. Здесь первый узел получает форму ромба (1), а второй - форму яйца (2). Результат показан на рис. 4-10.

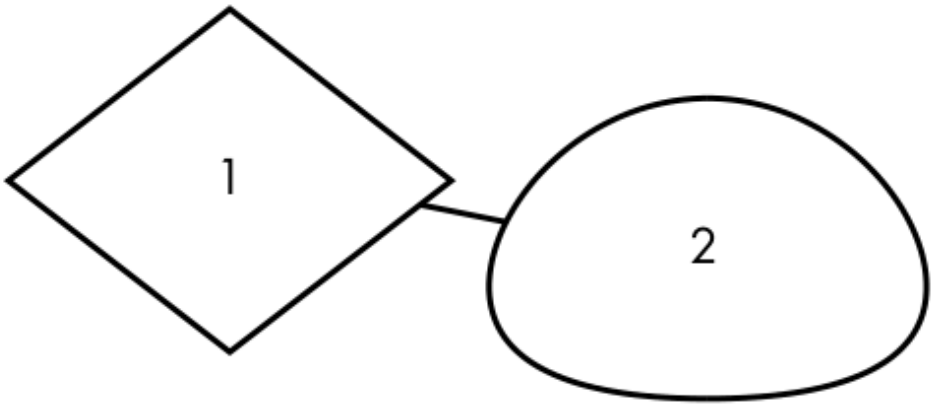

Рисунок 4-10. Простая сеть, демонстрирующая задание формы узлов

Результат в виде ромбовидного и яйцевидного узлов соответствует формам, указанным в листинге 4-6.

Текстовые метки

Наконец, GraphViz позволяет добавлять метки к узлам и рёбрам с помощью атрибута `label`. Узлы автоматически получают метки по назначенным идентификаторам: например, меткой узла, добавленного как `123`, будет `123`. Однако метки можно задавать явно с помощью `label=<my label attribute>`. В отличие от узлов, рёбра по умолчанию не имеют меток, но им также можно назначить атрибут `label`. В листинге 4-7 показано создание уже знакомой сети из двух узлов, где метками снабжены оба узла и соединяющее их ребро.

```
#!/usr/bin/python

import networkx
from networkx.drawing.nx_agraph import write_dot

g = networkx.Graph()
g.add_node(1,(1)label="first node")
g.add_node(2,(2)label="second node")
g.add_edge(1,2,(3)label="link between first and second node")

write_dot(g,'network.dot')
```

Листинг 4-7. Назначение меток узлам и рёбрам

Узлам назначены метки `first node` (1) и `second node` (2) соответственно. Ребро между ними получило метку `link between first and second node` (3). Ожидаемый графический результат показан на рис. 4-11.

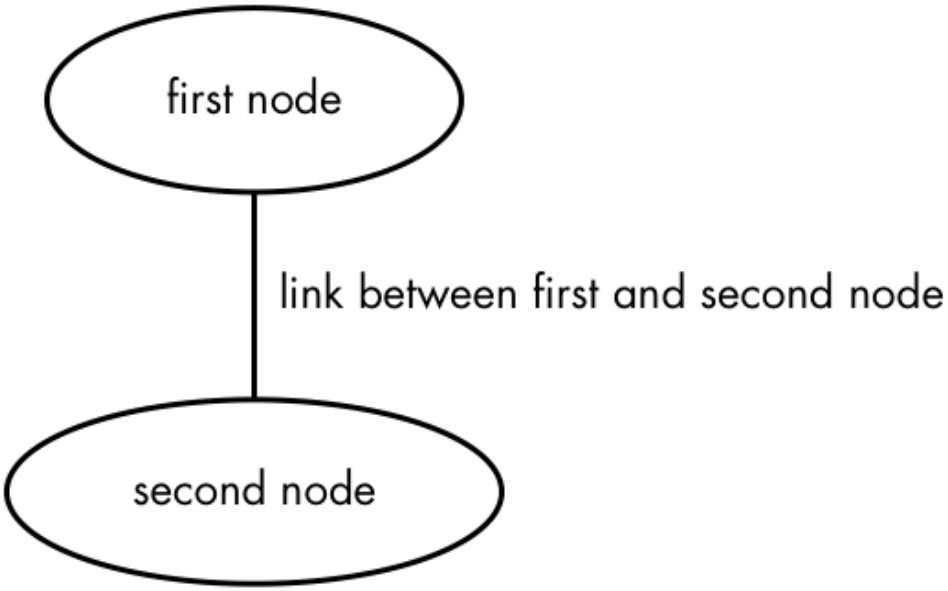

Рисунок 4-11. Простая сеть, демонстрирующая назначение меток узлам и рёбрам

Теперь, когда вы умеете управлять основными атрибутами узлов и рёбер, можно приступать к построению сетей с нуля.

Построение сетей вредоносных программ

Обсуждение построения сетей вредоносных программ мы начнём с воспроизведения и расширения примера общих серверов обратного подключения на рис. 4-1, после чего рассмотрим анализ общих изображений.

Следующая программа извлекает из вредоносных файлов доменные имена обратного подключения, а затем строит двудольную сеть образцов. После этого она создаёт одну проекцию сети, показывающую образцы с общими серверами обратного подключения, и другую проекцию, показывающую серверы, к которым обращаются одни и те же образцы. Наконец, программа сохраняет три сети - исходную двудольную сеть, проекцию образцов и проекцию серверов - в файлы для последующей визуализации с помощью GraphViz.

Я разберу программу по частям. Полный код находится среди прилагаемых к книге данных в файле `ch4/callback_server_network.py`.

В листинге 4-8 показано начало программы с импортом необходимых модулей.

```
#!/usr/bin/python

import pefile  # (1)
import sys
import argparse
import os
import pprint
import networkx  # (2)
import re
from networkx.drawing.nx_agraph import write_dot
import collections
from networkx.algorithms import bipartite
```

Листинг 4-8. Импорт модулей

Из импортированных модулей особенно важны модуль разбора PE-файлов `pefile` (1), применяемый для разбора исследуемых двоичных файлов, и библиотека `networkx` (2), с помощью которой создаётся сеть атрибутов вредоносных программ.

Затем добавим код из листинга 4-9 для разбора аргументов командной строки.

```
args = argparse.ArgumentParser("Visualize shared DLL import relationships
between a directory of malware samples")
args.add_argument((1)"target_path",help="directory with malware samples")
args.add_argument((2)"output_file",help="file to write DOT file to")
args.add_argument((3)"malware_projection",help="file to write DOT file to")
args.add_argument((4)"resource_projection",help="file to write DOT file to")
args = args.parse_args()
```

Листинг 4-9. Разбор аргументов командной строки

К этим аргументам относятся `target_path` (1) - путь к каталогу с анализируемыми вредоносными программами; `output_file` (2) - путь для записи полной сети; `malware_projection` (3) - путь для записи сокращённой версии графа, показывающей образцы с общими атрибутами; `resource_projection` (4) - путь для записи сокращённой версии графа, показывающей атрибуты, которые встречаются совместно в одних образцах.

Теперь можно перейти к основной части программы. В листинге 4-10 приведён код создания сети.

```
#!/usr/bin/python

import pefile
(1) import sys
import argparse
import os
import pprint
import networkx
import re
from networkx.drawing.nx_agraph import write_dot
import collections
from networkx.algorithms import bipartite

args = argparse.ArgumentParser(
"Visualize shared hostnames between a directory of malware samples"
)
args.add_argument("target_path",help="directory with malware samples")
args.add_argument("output_file",help="file to write DOT file to")
args.add_argument("malware_projection",help="file to write DOT file to")
args.add_argument("hostname_projection",help="file to write DOT file to")
args = args.parse_args()
network = networkx.Graph()

valid_hostname_suffixes = map(
lambda string: string.strip(), open("domain_suffixes.txt")
)
valid_hostname_suffixes = set(valid_hostname_suffixes)
(2) def find_hostnames(string):
    possible_hostnames = re.findall(
    r'(?:[a-zA-Z0-9](?:[a-zA-Z0-9\-]{0,61}[a-zA-Z0-9])?\.)+[a-zA-Z]{2,6}',
    string)
    valid_hostnames = filter(
            lambda hostname: hostname.split(".")[-1].lower() \
            in valid_hostname_suffixes,
            possible_hostnames
    )
    return valid_hostnames

# найти в исследуемом каталоге корректные исполняемые PE-файлы Windows
for root,dirs,files in os.walk(args.target_path):
    for path in files:
        # попытаться открыть файл через pefile и убедиться, что это PE-файл
        try:
            pe = pefile.PE(os.path.join(root,path))
        except pefile.PEFormatError:
```

```
        continue
    fullpath = os.path.join(root,path)
    # извлечь из исследуемого образца печатные строки
    (3) strings = os.popen("strings '{0}'".format(fullpath)).read()

    # с помощью функции search_doc из прилагаемого модуля reg
    # найти имена узлов
    (4) hostnames = find_hostnames(strings)
    if len(hostnames):
        # добавить узлы и рёбра двудольной сети
        network.add_node(path,label=path[:32],color='black',penwidth=5,
        bipartite=0)
    for hostname in hostnames:
        (5) network.add_node(hostname,label=hostname,color='blue',
                 penwidth=10,bipartite=1)
        network.add_edge(hostname,path,penwidth=2)
    if hostnames:
        print "Extracted hostnames from:",path
        pprint.pprint(hostnames)
```

Листинг 4-10. Создание сети

Сначала мы создаём новую сеть вызовом конструктора `networkx.Graph()` (1). Затем определяется функция `find_hostnames()`, извлекающая имена узлов из строк (2). Не стоит слишком углубляться в механику этой функции: по существу, это регулярное выражение и код фильтрации строк, который старается как можно точнее распознать домены.

Далее программа перебирает все файлы исследуемого каталога и проверяет, являются ли они PE-файлами, пытаясь загрузить их классом `pefile.PE`; файлы, которые загрузить не удаётся, не анализируются. Наконец, из текущего файла извлекаются атрибуты имён узлов: сначала из файла получают все печатные строки (3), а затем ищут среди них встроенные имена узлов (4). Найденные имена добавляются в сеть как узлы, после чего узел текущего образца соединяется рёбрами с узлами ресурсов имён (5).

Теперь программу можно завершить, как показано в листинге 4-11.

```
# записать dot-файл на диск
(1) write_dot(network, args.output_file)
(2) malware = set(n for n,d in network.nodes(data=True) if d['bipartite']==0)
(3) hostname = set(network)-malware

# функцией проекции двудольной сети из NetworkX создать проекции
# вредоносных программ и имён узлов
(4) malware_network = bipartite.projected_graph(network, malware)
   hostname_network = bipartite.projected_graph(network, hostname)

# записать проекции на диск в места, указанные пользователем
(5) write_dot(malware_network,args.malware_projection)
   write_dot(hostname_network,args.hostname_projection)
```

Листинг 4-11. Запись сетей в файлы

Сначала сеть записывается на диск в место, заданное аргументами командной строки (1). Затем создаются две сокращённые сети - представленные ранее в главе проекции, показывающие отношения образцов вредоносных программ и отношения ресурсов имён узлов. Для этого сначала создаётся множество Python с идентификаторами узлов вредоносных программ (2) и другое множество с идентификаторами узлов ресурсов (3). Затем специальная функция NetworkX `projected_graph()` (4) получает проекции множеств вредоносных программ и ресурсов, после чего сети записываются в указанные места (5).

Вот и всё. Эту программу можно применять к любому набору вредоносных программ из книги, чтобы увидеть отношения между образцами через общие ресурсы имён узлов, встроенные в файлы. Её можно использовать и для собственных наборов данных, выясняя, какие сведения об угрозах удастся получить таким способом.

Построение сети связей по общим изображениям

Помимо общих серверов обратного подключения, вредоносные программы можно анализировать по использованию общих значков и других графических ресурсов. Например, на рис. 4-12 показана часть результатов анализа общих изображений троянских программ из `ch4/data/Trojans`.

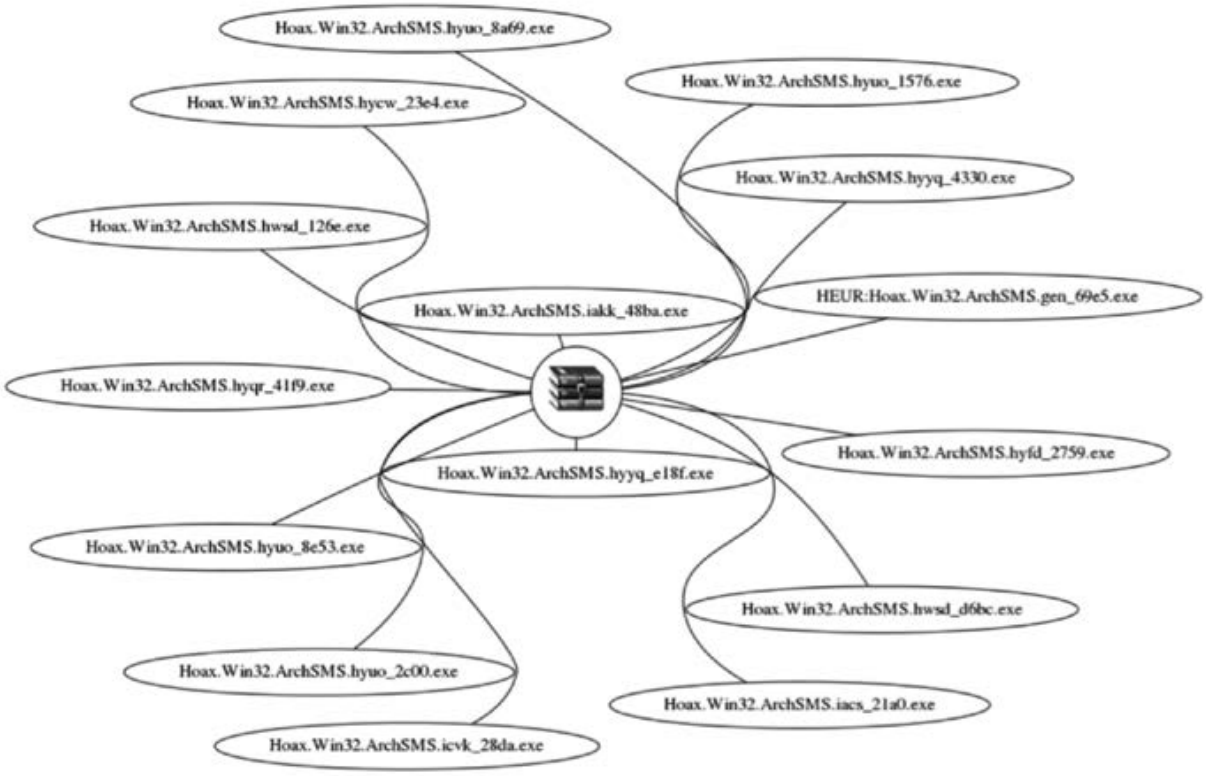

Рисунок 4-12. Визуализация сети общих графических ресурсов для нескольких троянских программ

Видно, что все эти троянские программы выдают себя за архивы и, хотя являются исполняемыми файлами, используют один и тот же значок архива, показанный в центре рисунка. Применение совершенно одинакового изображения для обмана пользователя указывает, что программы, вероятно, происходят от одного злоумышленника. Я подтвердил это, проверив образцы антивирусным механизмом Kaspersky: он отнёс их к одному семейству ArchSMS.

Далее я покажу, как построить визуализацию, подобную рис. 4-12, чтобы увидеть отношения образцов по общим изображениям. Для извлечения изображений мы применим вспомогательную библиотеку `images`, которая, в свою очередь, использует рассмотренный в главе 1 инструмент `wrestool`, и создадим программу `image_network.py`. Напомним, что `wrestool` извлекает изображения из исполняемых файлов Windows.

Разберём создание сети общих изображений, начав с первой части кода в листинге 4-12.

```
#!/usr/bin/python

import pefile
import sys
import argparse
import os
import pprint
import logging
import networkx
```

```
import collections
import tempfile
from networkx.drawing.nx_agraph import write_dot
from networkx.algorithms import bipartite

# разобрать аргументы командной строки с помощью argparse

args = argparse.ArgumentParser(
"Visualize shared image relationships between a directory of malware samples"
)
args.add_argument("target_path",help="directory with malware samples")
args.add_argument("output_file",help="file to write DOT file to")
args.add_argument("malware_projection",help="file to write DOT file to")
args.add_argument("resource_projection",help="file to write DOT file to")
args = args.parse_args()
network = networkx.Graph()

(1) class ExtractImages():
    def __init__(self,target_binary):
        self.target_binary = target_binary
        self.image_basedir = None
        self.images = []

    def work(self):
        self.image_basedir = tempfile.mkdtemp()
        icondir = os.path.join(self.image_basedir,"icons")
        bitmapdir = os.path.join(self.image_basedir,"bitmaps")
        raw_resources = os.path.join(self.image_basedir,"raw")
        for directory in [icondir,bitmapdir,raw_resources]:
            os.mkdir(directory)
        rawcmd = "wrestool -x {0} -o {1} 2> \
                 /dev/null".format(
                 self.target_binary,raw_resources
                 )
        bmpcmd = "mv {0}/*.bmp {1} 2> /dev/null".format(
        raw_resources,bitmapdir
        )
        icocmd = "icotool -x {0}/*.ico -o {1} \
                  2> /dev/null".format(
                  raw_resources,icondir
                  )
        for cmd in [rawcmd,bmpcmd,icocmd]:
            try:
                os.system(cmd)
            except Exception,msg:
                pass
        for dirname in [icondir,bitmapdir]:
            for path in os.listdir(dirname):
                logging.info(path)
                path = os.path.join(dirname,path)
                imagehash = hash(open(path).read())
                if path.endswith(".png"):
                    self.images.append((path,imagehash))
                if path.endswith(".bmp"):
                    self.images.append((path,imagehash))
    def cleanup(self):
        os.system("rm -rf {0}".format(self.image_basedir))

# найти в исследуемом каталоге PE-файлы для извлечения изображений
image_objects = []
for root,dirs,files in os.walk(args.target_path):  # (2)
    for path in files:
        # попытаться разобрать путь и проверить корректность PE-файла
        try:
            pe = pefile.PE(os.path.join(root,path))
        except pefile.PEFormatError:
            continue
```

Листинг 4-12. Разбор начальных аргументов и код загрузки файлов в программе сети общих изображений

Программа начинается почти так же, как только что рассмотренная программа графа имён узлов, начиная с листинга 4-8. Сначала импортируется ряд модулей, в том числе `pefile` и `networkx`. Однако здесь мы также определяем вспомогательный класс `ExtractImages` (1) для извлечения графических ресурсов из исследуемых образцов. Затем программа входит в цикл, перебирающий все исследуемые вредоносные двоичные файлы (2).

Находясь внутри цикла, пора извлечь графические ресурсы из двоичных файлов с помощью класса `ExtractImages`, который служит оболочкой над программами `icoutils`, рассмотренными в главе 1. Соответствующая часть кода приведена в листинге 4-13.

```
fullpath = os.path.join(root,path)
(1) images = ExtractImages(fullpath)
(2) images.work()
image_objects.append(images)

# создать сеть, связывая образцы вредоносных программ с их изображениями
(3) for path, image_hash in images.images:
    # задать атрибут image узлам изображений, чтобы GraphViz
    # отображал изображения внутри этих узлов
    if not image_hash in network:
        (4) network.add_node(image_hash,image=path,label='',type='image')
    node_name = path.split("/")[-1]
    network.add_node(node_name,type="malware")
    (5) network.add_edge(node_name,image_hash)
```

Листинг 4-13. Извлечение графических ресурсов из исследуемых вредоносных программ

Сначала мы передаём классу `ExtractImages` путь к исследуемому вредоносному двоичному файлу (1), а затем вызываем у полученного экземпляра метод `work()` (2). В результате класс `ExtractImages` создаёт временный каталог для хранения изображений вредоносной программы, а затем сохраняет словарь с данными каждого изображения в атрибуте класса `images`.

Получив список извлечённых изображений из `ExtractImages`, мы перебираем его (3). Если хеш изображения ещё не встречался, для него создаётся новый узел сети (4), после чего текущий обрабатываемый образец связывается с изображением в сети (5).

Теперь сеть образцов вредоносных программ, связанных с содержащимися в них изображениями, создана и готова к записи на диск, как показано в листинге 4-14.

```
# записать двудольную сеть, затем создать и записать две проекции
(1) write_dot(network, args.output_file)
malware = set(n for n,d in network.nodes(data=True) if d['type']=='malware')
resource = set(network) - malware
malware_network = bipartite.projected_graph(network, malware)
resource_network = bipartite.projected_graph(network, resource)

(2) write_dot(malware_network,args.malware_projection)
write_dot(resource_network,args.resource_projection)
```

Листинг 4-14. Запись графа на диск

Это выполняется точно так же, как в листинге 4-11. Сначала на диск записывается полная сеть (1), а затем две проекции: проекция вредоносных программ и проекция изображений, которые здесь называются ресурсами (2).

Программу `image_network.py` можно применять для анализа графических ресурсов в любом наборе вредоносных программ из книги либо для извлечения сведений из выбранных вами наборов.

Итоги

В этой главе вы познакомились с инструментами и методами, необходимыми для анализа общих атрибутов собственных наборов вредоносных программ. В частности, вы узнали, как сети, двудольные сети и проекции двудольных сетей помогают выявлять социальные связи между образцами, почему укладка сети играет центральную роль в визуализации и как работают силовые сети. Кроме того, вы научились создавать и визуализировать сети вредоносных программ с помощью Python и таких инструментов с открытым исходным кодом, как NetworkX. В главе 5 вы узнаете, как строить сети вредоносных программ по отношениям общего кода между образцами.

Глава 5. Анализ общего кода

Предположим, вы обнаружили в своей сети новый образец вредоносной программы. С чего начать анализ? Можно отправить его на многомодульный антивирусный сканер, например VirusTotal, и выяснить, к какому семейству он относится. Однако результаты часто бывают неясными и неоднозначными: механизмы нередко присваивают вредоносным программам ничего не значащие общие метки вроде `agent`. Можно также запустить образец в CuckooBox или другой песочнице и получить ограниченный отчёт о серверах обратного подключения и поведении программы.

Когда этих подходов недостаточно, может потребоваться обратная разработка образца. На данном этапе анализ общего кода способен значительно улучшить рабочий процесс. Он показывает, с какими уже исследованными образцами схожа новая вредоносная программа, а следовательно, какой код у них общий. Благодаря этому результаты прежних исследований можно повторно использовать для нового образца, не начиная с нуля. Знание происхождения уже встречавшейся программы может также помочь выяснить, кто развернул новую.

Анализ общего кода, также называемый анализом сходства, - это процесс сравнения двух образцов вредоносных программ посредством оценки доли общего исходного кода, который существовал до компиляции. Он отличается от анализа общих атрибутов, где образцы сравниваются по внешним свойствам, например по значкам рабочего стола или серверам, к которым они обращаются.

При обратной разработке анализ общего кода помогает выявлять образцы, которые можно исследовать совместно, поскольку они созданы одним набором средств разработки вредоносных программ либо представляют разные версии одного семейства. Кроме того, он позволяет определить, могла ли за группой образцов стоять одна команда разработчиков.

Рассмотрим результат в листинге 5-1, полученный программой, которую вы создадите далее в этой главе для демонстрации ценности анализа общего кода. В нём перечислены уже встречавшиеся образцы, которые могут иметь общий код с новым, а также комментарии к этим старым образцам.

```
Showing samples similar to WEBC2-GREENCAT_sample_E54CE5F0112C9FDFE86DB17E85A5E2C5
Sample name                                                    Shared code
[*] WEBC2-GREENCAT_sample_55FB1409170C91740359D1D96364F17B     0.9921875
[*] GREENCAT_sample_55FB1409170C91740359D1D96364F17B           0.9921875
[*] WEBC2-GREENCAT_sample_E83F60FB0E0396EA309FAF0AED64E53F     0.984375
    [comment] This sample was determined to definitely have come from the advanced
    persistent
threat group observed last July on our West Coast network
[*] GREENCAT_sample_E83F60FB0E0396EA309FAF0AED64E53F           0.984375
```

Листинг 5-1. Результаты базового анализа общего кода

Получив новый образец, за несколько секунд можно оценить, с какими программами он, вероятно, имеет общий код и что нам уже о них известно. В данном примере обнаруживается очень похожий образец, относящийся к известной устойчивой развитой угрозе Advanced Persistent Threat (APT), что немедленно предоставляет контекст новой вредоносной программы.

Отношения общего кода между образцами можно также представить сетевой визуализацией, с которой вы познакомились в главе 4. Например, на рис. 5-1 показана сеть отношений общего кода между образцами из набора данных устойчивой развитой угрозы.

Как видно из визуализации, автоматические методы анализа общего кода позволяют быстро обнаруживать семейства вредоносных программ, на выявление которых вручную потребовались бы дни или недели. В этой главе вы научитесь применять такие методы, чтобы:

• выявлять новые семейства вредоносных программ, созданные одними наборами инструментов или одними злоумышленниками;
• определять сходство кода нового образца с уже встречавшимися;

• визуализировать отношения вредоносных программ, чтобы лучше понимать закономерности совместного использования кода и сообщать результаты другим людям;
• применять две созданные мной для книги экспериментальные программы, которые реализуют эти идеи и позволяют видеть отношения общего кода между вредоносными программами.

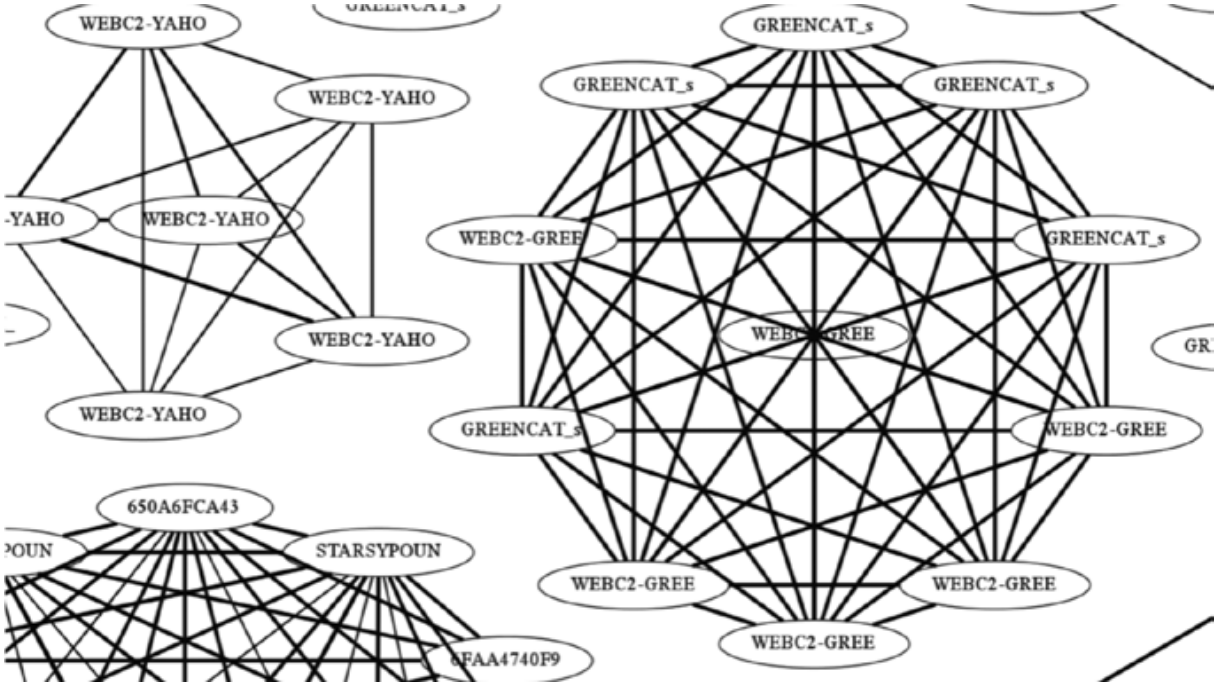

Рисунок 5-1. Пример визуализации, которую вы научитесь создавать в этой главе: отношения общего кода между некоторыми образцами APT1

Сначала я представлю исследуемые в этой главе вредоносные программы: образцы китайской Народно-освободительной армии APT1 из главы 4 и набор образцов криминальных вредоносных программ. Затем вы познакомитесь с математическим сравнением сходства и индексом Жаккара - теоретико-множественным методом сопоставления образцов по общим признакам. Далее я введу понятие признаков и покажу, как совместно с индексом Жаккара приблизительно оценить объём общего кода двух образцов. Вы также научитесь оценивать полезность признаков вредоносных программ. Наконец, опираясь на знания о сетевой визуализации из главы 4, мы создадим визуализации совместного использования кода в разных масштабах, подобные рис. 5-1.

Образцы вредоносных программ, используемые в этой главе

В экспериментах этой главы используются реальные семейства вредоносных программ, имеющие значительные объёмы общего кода. Эти наборы данных доступны благодаря компании Mandiant и Миле Паркур, которые собрали образцы и предоставили их исследовательскому сообществу. В настоящем исследовании вы, возможно, не будете знать, к какому семейству относится образец или насколько новые программы похожи на уже встречавшиеся. Но примеры с известными ответами станут хорошей практикой: они позволяют проверить, согласуются ли автоматические выводы о сходстве с нашими знаниями о том, какие образцы действительно принадлежат одной группе.

Первая группа происходит из набора APT1, который применялся в главе 4 для демонстрации анализа общих ресурсов. Вторая состоит из тысяч криминальных вредоносных программ, разработанных для кражи данных кредитных карт, превращения компьютеров в узлы-зомби ботнетов и других преступных целей. Это реальные образцы из коммерческого потока вредоносных программ, который предоставляется исследователям сведений об угрозах как платная услуга.

Чтобы определить названия семейств, я передал каждый образец антивирусному механизму Kaspersky. Он сумел классифицировать 30 104 образца с помощью устойчивой иерархической классификации, например `trojan.win32.jorik.skor.akr`, указывающей на семейство `jorik.skor`; присвоил класс `unknown` 41 830 образцам; а оставшимся 28 481 назначил общие метки, например просто `win32 Trojan`.

Из-за непоследовательности меток Kaspersky, где некоторые группы вроде семейства jorik охватывают очень широкий спектр вредоносных программ, а другие вроде webprefix обозначают вполне конкретный набор вариантов, а также потому, что Kaspersky иногда пропускает или ошибочно классифицирует вредоносные программы, я выбрал семь классов, которые этот механизм обнаруживает с высокой уверенностью. Это семейства dapato, pasta, skor, vbna, webprefix, xtoober и zango.

Подготовка образцов к сравнению посредством извлечения признаков

Как вообще подойти к оценке объёма кода, который два вредоносных двоичных файла могли иметь общим до компиляции злоумышленниками? Существует множество возможных подходов, однако в сотнях опубликованных по этой теме научных работ сформировалась общая идея: перед сравнением образцы вредоносных программ представляют как «мешки признаков».

Под *признаками* я понимаю любые атрибуты вредоносной программы, которые можно учитывать при оценке сходства кода образцов. Например, признаками могут служить печатные строки, извлечённые из двоичных файлов. Для математического удобства вредоносная программа рассматривается не как взаимосвязанная система функций, импортируемых динамических библиотек и других составляющих, а как мешок независимых признаков, например множество извлечённых строк.

Как работают модели мешка признаков

Чтобы понять принцип мешка признаков, рассмотрим диаграмму Венна двух образцов на рис. 5-2.

Образцы A и B представлены мешками признаков, обозначенных эллипсами внутри диаграммы. Образцы можно сравнить по признакам, общим для обоих. Пересечение двух множеств признаков вычисляется быстро и позволяет оценивать сходство вредоносных программ по любым выбранным нами признакам.

Например, при работе с упакованными вредоносными программами можно использовать признаки из журналов динамического выполнения, поскольку запуск в песочнице заставляет программу распаковать себя. В других случаях сравнение можно выполнять по строкам, извлечённым статически из двоичного файла.

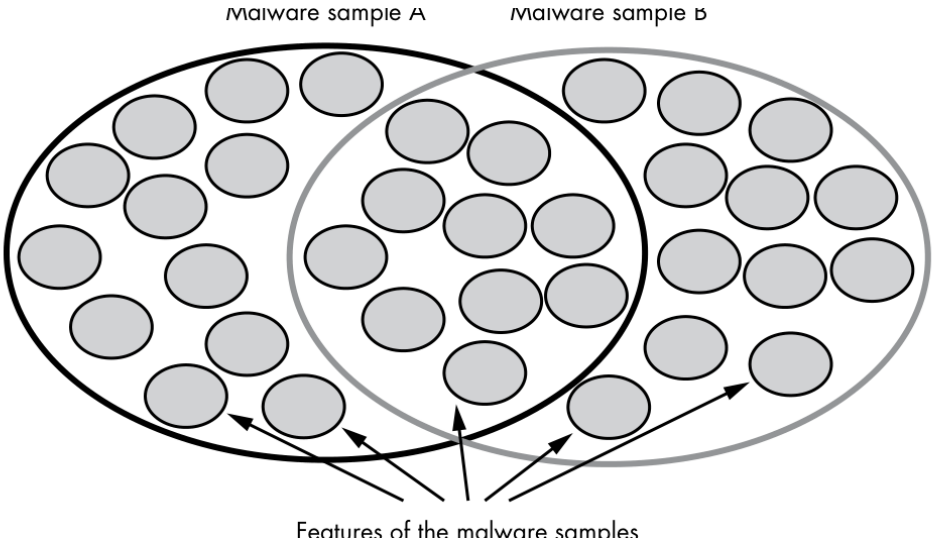

Рисунок 5-2. Модель «мешка признаков» для анализа общего кода вредоносных программ

При динамическом анализе образцы может потребоваться сравнивать не только по общим действиям, но и по порядку этих действий, то есть по *последовательностям поведения*. Распространённый способ учитывать последовательность при сравнении - расширить модель мешка признаков для последовательных данных с помощью n-грамм.

Что такое n-граммы

N-грамма - это подпоследовательность определённой длины N внутри более длинной последовательности событий. Она извлекается путём перемещения окна по последовательным данным. Иными словами, мы перебираем последовательность и на каждом шаге записываем подпоследовательность от события с индексом i до события с индексом i + N - 1, как показано на рис. 5-3.

На рис. 5-3 последовательность целых чисел `(1,2,3,4,5,6,7)` преобразуется в пять подпоследовательностей длиной 3: `(1,2,3)`, `(2,3,4)`, `(3,4,5)`, `(4,5,6)` и `(5,6,7)`.

Разумеется, это можно проделать с любыми последовательными данными. Например, при длине n-граммы 2 предложение `how now brown cow` даёт подпоследовательности `how now`, `now brown` и `brown cow`. При анализе вредоносной программы мы извлекали бы n-граммы последовательных вызовов API, сделанных образцом. Затем представили бы программу мешком признаков и сравнили её n-граммы с n-граммами другого образца, тем самым включив сведения о последовательности в модель сравнения мешков признаков.

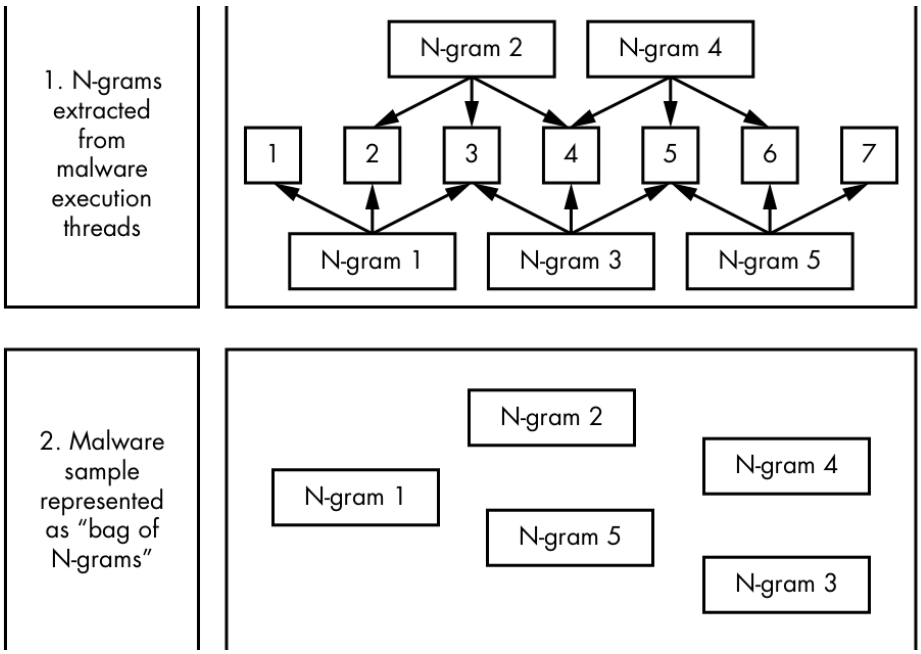

Рисунок 5-3. Наглядное объяснение извлечения n-грамм из инструкций на языке ассемблера и последовательностей динамических вызовов API вредоносной программы при N = 3

У включения сведений о последовательности в сравнение образцов есть достоинства и недостатки. Если порядок важен, например нужно учесть, что сначала наблюдался вызов API A, затем B, а потом C, этот подход позволяет его сохранить. Но когда порядок несущественен, например программа при каждом запуске случайным образом меняет порядок вызовов A, B и C, он способен значительно ухудшить оценку общего кода. Решение об учёте порядка зависит от разновидности исследуемой вредоносной программы и требует экспериментов.

Количественная оценка сходства с помощью индекса Жаккара

Представив вредоносную программу мешком признаков, необходимо измерить степень его сходства с мешком признаков другого образца. Для оценки объёма общего кода двух программ используется функция сходства, которая должна обладать следующими свойствами:

• Она возвращает нормализованное значение, чтобы все попарные сравнения образцов можно было поместить на общую шкалу. По соглашению значение должно изменяться от 0, то есть отсутствия общего кода, до 1, когда образцы имеют 100 процентов общего кода.
• Функция должна помогать точно оценивать общий код двух образцов; это можно определить эмпирически посредством экспериментов.
• Должно быть легко понять, почему функция хорошо моделирует сходство кода: она не должна представлять собой сложный математический «чёрный ящик», требующий больших усилий для понимания или объяснения.

Индекс Жаккара - простая функция с этими свойствами. Хотя исследователи безопасности пробовали и другие математические подходы к оценке сходства кода, например косинусное расстояние, расстояние L1 и евклидово расстояние L2, индекс Жаккара стал самым распространённым, и не без причины. Он просто и наглядно выражает степень пересечения двух множеств признаков: количество уникальных признаков, общих для обоих множеств, делится на количество уникальных признаков, присутствующих хотя бы в одном из них.

На рис. 5-4 показаны примеры значений индекса Жаккара.

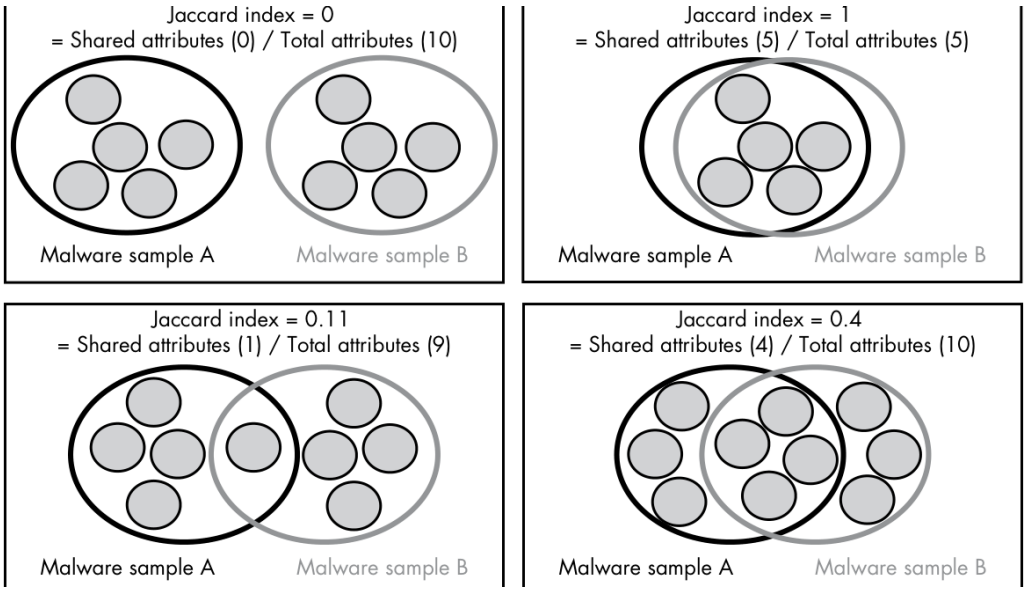

Рисунок 5-4. Наглядное представление идеи индекса Жаккара

На рисунке представлены четыре пары множеств признаков, извлечённых из четырёх пар вредоносных программ. В каждом случае показаны общие и необщие признаки двух множеств, а также полученный индекс Жаккара для соответствующей пары образцов и признаков. Индекс равен количеству общих признаков, делённому на общее количество признаков на диаграмме Венна.

Оценка методов определения общего кода с помощью матриц сходства

Рассмотрим четыре метода определения принадлежности двух образцов одному семейству: сходство последовательностей инструкций, сходство строк, сходство таблиц адресов импорта и сходство динамических вызовов API. Для сравнения этих методов мы применим визуализацию в виде матрицы сходства. Наша цель - сопоставить относительные достоинства и недостатки каждого метода при выявлении отношений общего кода между образцами.

Для начала разберём понятие матрицы сходства. На рис. 5-5 с её помощью сравнивается вымышленный набор из четырёх образцов вредоносных программ.

	Sample 1	Sample 2	Sample 3	Sample 4
Sample 1	Similarity between 1 and 1	Similarity between 1 and 2	Similarity between 1 and 3	Similarity between 1 and 4
Sample 2	Similarity between 2 and 1	Similarity between 2 and 2	Similarity between 2 and 3	Similarity between 2 and 4
Sample 3	Similarity between 3 and 1	Similarity between 3 and 2	Similarity between 3 and 3	Similarity between 3 and 4
Sample 4	Similarity between 4 and 1	Similarity between 4 and 2	Similarity between 4 and 3	Similarity between 4 and 4

Рисунок 5-5. Условная матрица сходства

Матрица позволяет увидеть отношения сходства между всеми образцами. Часть места в ней расходуется впустую. Например, нас не интересуют значения в затенённых ячейках, поскольку там каждый образец сравнивается сам с собой. Кроме того, сведения по обе стороны от затенённых ячеек повторяются, поэтому достаточно рассматривать лишь одну сторону.

На рис. 5-6 приведён реальный пример матрицы сходства вредоносных программ. Из-за большого количества образцов каждое значение сходства представлено оттенком одного пикселя. Вместо названий отдельных образцов по горизонтальной и вертикальной осям указаны названия их семейств. Идеальная матрица выглядела бы как цепочка белых квадратов, проходящая по диагонали от левого верхнего угла к правому нижнему: строки и столбцы каждого семейства сгруппированы, и ожидается, что все представители одного семейства похожи друг на друга, но не на образцы других семейств.

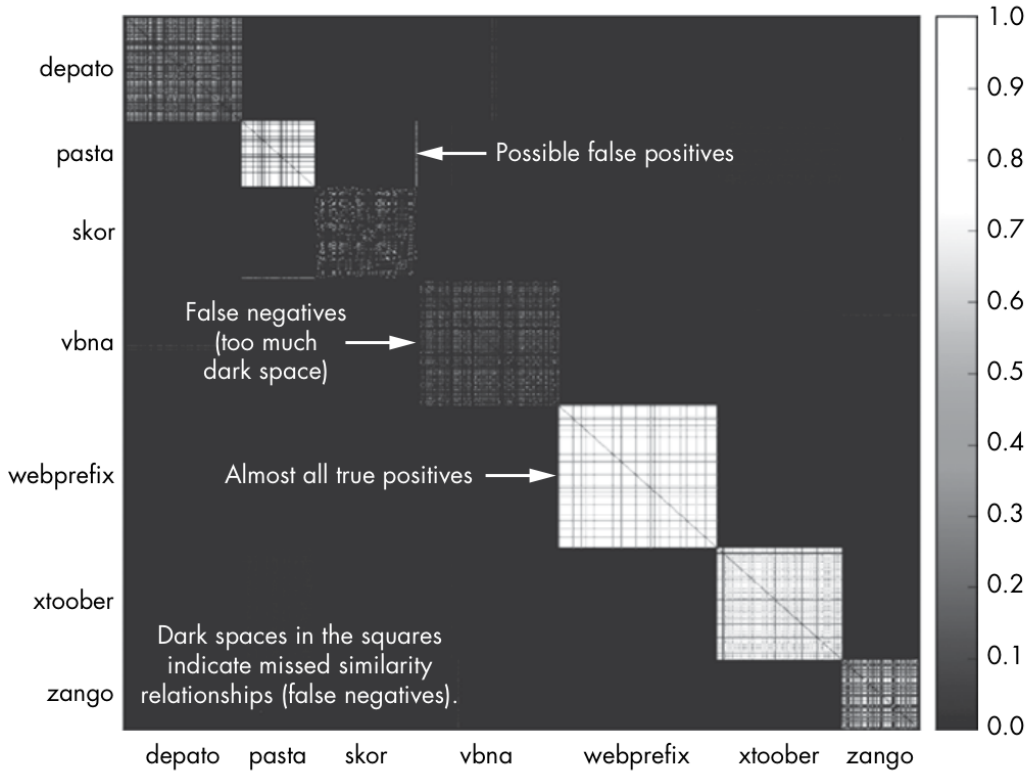

Рисунок 5-6. Реальная матрица сходства, вычисленная для семи семейств вредоносных программ

На рис. 5-6 квадраты некоторых семейств полностью белые. Это хороший результат: белые пиксели внутри квадрата семейства обозначают выявленное сходство его образцов. Другие квадраты гораздо темнее, то есть сильные отношения сходства не обнаружены. Наконец, иногда за пределами квадратов семейств встречаются линии пикселей: они свидетельствуют либо о родстве семейств, либо о ложноположительных результатах, когда общий код обнаружен у изначально разных семейств.

Далее с помощью матриц, подобных рис. 5-6, мы сравним результаты четырёх методов оценки общего кода, начав со сходства последовательностей инструкций.

Сходство на основе последовательностей инструкций

Самый очевидный способ сравнить два вредоносных двоичных файла по объёму общего кода - сопоставить последовательности их инструкций на языке ассемблера x86. Если у образцов есть общие последовательности инструкций, вполне вероятно, что до компиляции у них был общий исходный код. Для этого необходимо дизассемблировать программы, например методом линейного дизассемблирования из главы 2. Затем рассмотренным ранее способом извлечь n-граммы последовательных инструкций в порядке их расположения в секции `.text`. Наконец, по n-граммам инструкций вычислить индексы Жаккара для пар образцов и оценить предполагаемый объём общего кода. Значение N при извлечении n-грамм зависит от целей анализа. Чем больше N, тем длиннее извлекаемые подпоследовательности инструкций и тем труднее последовательностям разных образцов совпасть. Большое N помогает выявлять только программы с очень высокой вероятностью общего кода. С другой стороны, N можно уменьшить для поиска слабого сходства либо при подозрении, что образцы меняют порядок инструкций, чтобы затруднить анализ.

На рис. 5-7 N равно 5. Это жёсткий параметр, затрудняющий совпадение образцов.

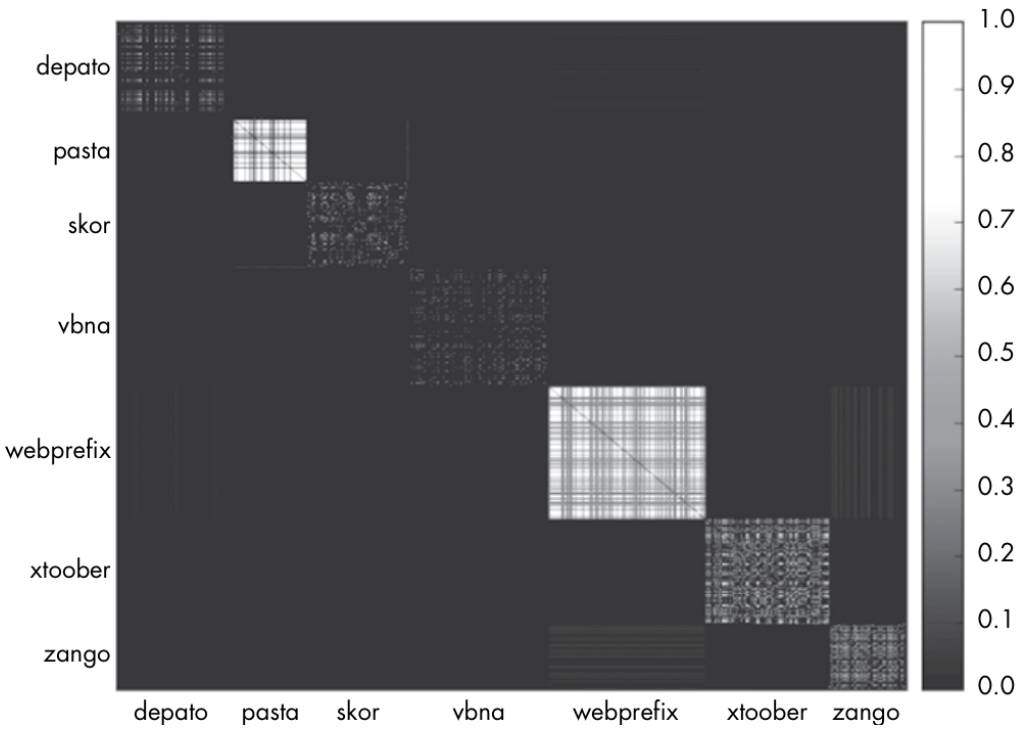

Рисунок 5-7. Матрица сходства по признакам n-грамм инструкций. При N = 5 мы полностью пропускаем отношения сходства многих семейств, но получаем хорошие результаты для webprefix и pasta.

Результаты на рис. 5-7 не слишком убедительны. Анализ на основе инструкций правильно выявляет сходство внутри некоторых семейств, но не обнаруживает его в других, например находит мало отношений в dapato, skor и vbna. Важно, однако, что ложноположительных результатов здесь немного: речь идёт о ложных выводах о сходстве образцов разных семейств в противоположность правильным выводам о сходстве внутри одного семейства.

Как видите, анализ общего кода по подпоследовательностям инструкций может пропустить множество отношений общего кода. Причина в том, что образцы могут быть упакованы, и основная часть инструкций станет видимой лишь после выполнения и самостоятельной распаковки программ. Без распаковки метод оценки общего кода по последовательностям инструкций, вероятно, будет работать плохо.

Даже после распаковки подход может оказаться проблемным из-за шума, вносимого компиляцией исходного кода. Компиляторы действительно способны преобразовать один исходный текст в радикально различающиеся последовательности инструкций. Рассмотрим простую функцию на C:

```
int f(void) {
    int a = 1;
    int b = 2;
(1) return (a*b)+3;
}
```

Можно предположить, что любой компилятор сведёт функцию к одной последовательности ассемблерных инструкций. На самом деле результат сильно зависит не только от выбранного компилятора, но и от его параметров. Например, компиляция этой функции компилятором clang со стандартными настройками создаёт для строки (1) следующие инструкции:

```
movl    $1, -4(%rbp)
movl    $2, -8(%rbp)
movl    -4(%rbp), %eax
imull   -8(%rbp), %eax
addl    $3, %eax
```

Напротив, компиляция той же функции с флагом `-O3`, предписывающим оптимизировать код по скорости, создаёт для той же строки исходного текста всего одну инструкцию:

```
movl    $5, %eax
```

Различие возникло потому, что во втором случае компилятор заранее вычислил результат функции, а не стал явно выполнять вычисления, как в первом. При сравнении по последовательностям инструкций эти функции выглядели бы совершенно непохожими, хотя в действительности скомпилированы из одного и того же исходного кода.

Помимо того что одинаковый код C или C++ может выглядеть совершенно по-разному на уровне ассемблерных инструкций, при таком сравнении двоичных файлов возникает ещё одна проблема: сегодня многие вредоносные программы написаны на высокоуровневых языках вроде C#. В двоичных файлах содержится стандартный шаблонный ассемблерный код, который лишь интерпретирует байт-код этих языков. Поэтому, хотя двоичные файлы на одном высокоуровневом языке могут иметь очень похожие инструкции x86, их фактический байт-код способен свидетельствовать о происхождении из совершенно разного исходного кода.

Сходство на основе строк

Сходство вредоносных программ по строкам можно вычислить, извлекая из образцов все непрерывные последовательности печатных символов и рассчитывая индекс Жаккара для каждой пары по отношениям общих строк.

Такой подход обходит проблему компилятора, потому что извлечённые из двоичного файла строки обычно являются строками форматирования, заданными программистом, а компиляторы, как правило, их не преобразуют независимо от выбранного компилятора и параметров. Например, типичная строка вредоносной программы может выглядеть так: `Started key logger at %s on %s and time %s.` При любых настройках компилятора она, вероятно, будет совершенно одинаковой в нескольких двоичных файлах, что зависит от происхождения этих файлов из общей базы исходного кода.

На рис. 5-8 показано, насколько хорошо метрика на основе строк выявляет правильные отношения общего кода в наборе криминальных вредоносных программ.

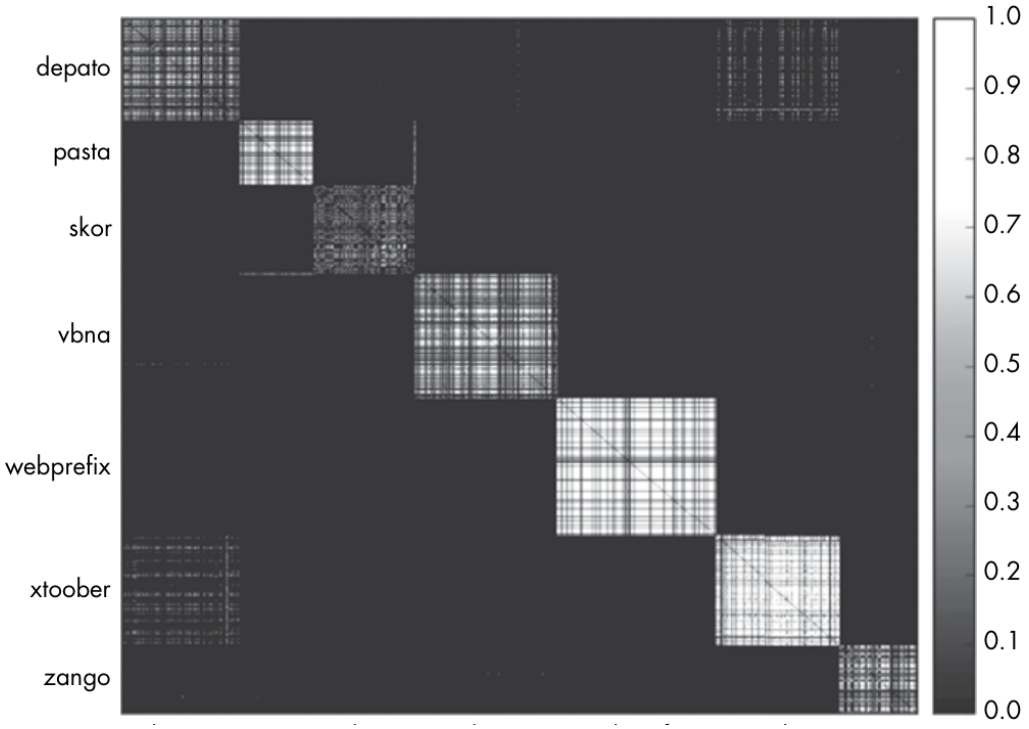

Рисунок 5-8. Матрица сходства, построенная по строковым признакам

На первый взгляд, этот метод гораздо лучше основанного на инструкциях выявляет семейства вредоносных программ и правильно восстанавливает значительную долю отношений во всех семи семействах. Однако, в отличие от метода сходства инструкций, здесь есть несколько ложноположительных результатов: ошибочно предполагается некоторый объём общего кода у xtoober и dapato. Кроме того, метод не обнаружил сходство части образцов внутри некоторых семейств и особенно плохо сработал для zango, skor и dapato.

Сходство на основе таблицы адресов импорта

То, что я называю сходством на основе таблицы адресов импорта, можно вычислить, сравнивая DLL, импортируемые вредоносными двоичными файлами. Идея состоит в том, что даже после перестановки инструкций, обфускации секции инициализированных данных и внедрения методов против анализа, отладчика и виртуальной машины автор мог оставить совершенно одинаковые объявления импорта. Результаты метода таблицы адресов импорта показаны на рис. 5-9.

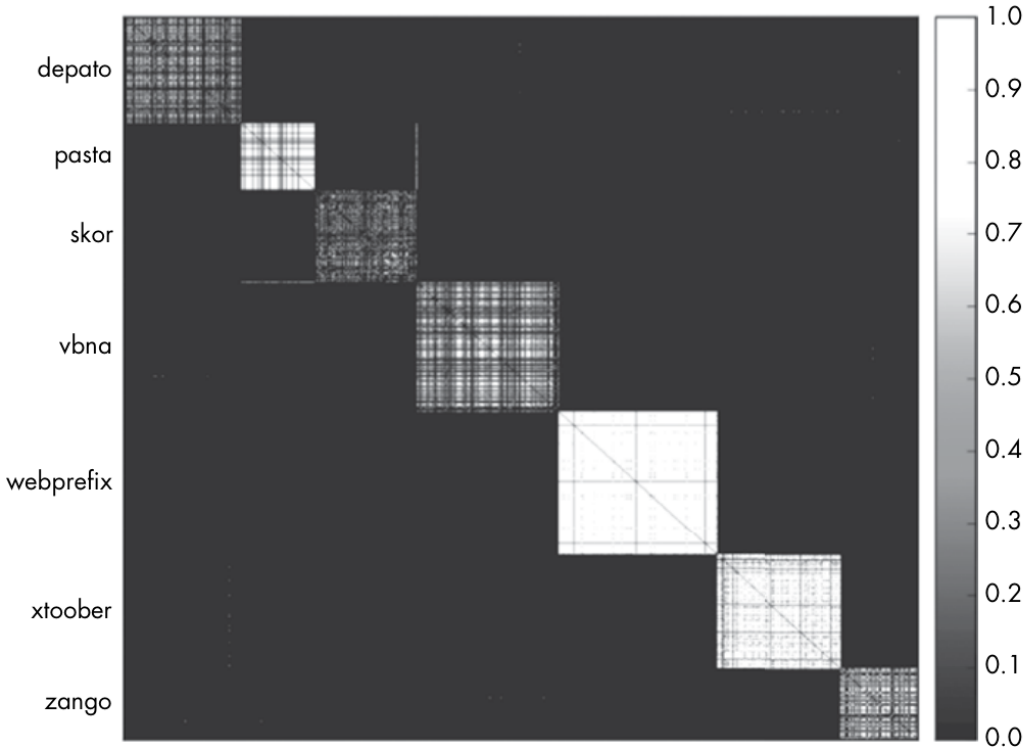

Рисунок 5-9. Матрица сходства, построенная по признакам таблицы адресов импорта

Рисунок показывает, что метод таблицы адресов импорта оценивает отношения сходства образцов webprefix и xtoober лучше всех предыдущих и в целом работает очень хорошо, хотя пропускает множество отношений skor, dapato и vbna. Примечательно также небольшое количество ложноположительных результатов в экспериментальном наборе.

Сходство на основе динамических вызовов API

Последний рассматриваемый в главе метод сравнения - динамическое сходство вредоносных программ. Преимущество сравнения динамических последовательностей состоит в том, что даже сильно обфусцированные или упакованные образцы обычно выполняют сходные последовательности действий в песочнице виртуальной машины, если происходят из одного кода или заимствуют код друг у друга. Для реализации подхода необходимо запустить образцы в песочнице и записать сделанные ими вызовы API, извлечь n-граммы вызовов из динамических журналов и, наконец, сравнить программы посредством индекса Жаккара между их мешками n-грамм.

Рисунок 5-10 показывает, что в большинстве случаев метод динамических n-грамм работает примерно так же хорошо, как методы импорта и строк.

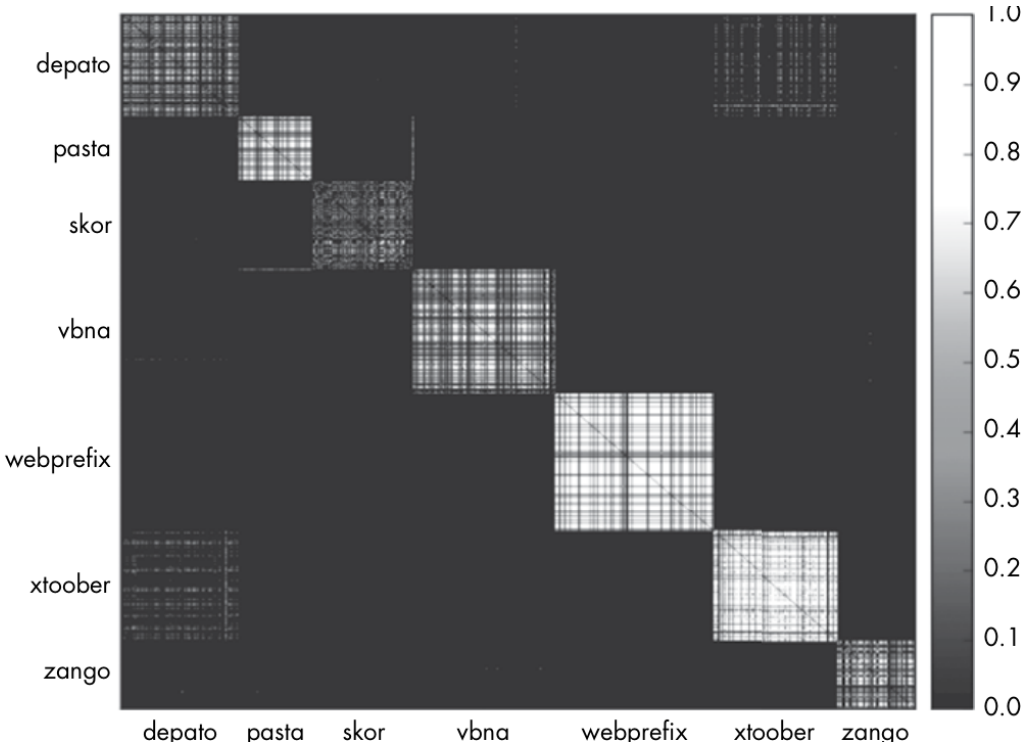

Рисунок 5-10. Матрица сходства, построенная по признакам n-грамм динамических вызовов API

Неидеальные результаты показывают, что и этот метод не является панацеей. Простого запуска в песочнице недостаточно, чтобы вызвать многие действия вредоносной программы. Например, разные варианты инструмента командной строки могут включать или не включать важный модуль кода и поэтому выполнять разные последовательности действий, хотя большая часть их кода будет общей.

Другая проблема в том, что некоторые образцы обнаруживают песочницу и немедленно завершают работу, оставляя слишком мало сведений для сравнения. Итак, сходство последовательностей динамических вызовов API, как и другие описанные подходы, не идеально, но способно предоставить впечатляющие сведения о сходстве образцов.

Построение графа сходства

Теперь, когда вы понимаете идеи методов выявления общего кода вредоносных программ, построим простую систему, которая выполняет такой анализ набора данных.

Сначала необходимо оценить объём общего кода образцов, извлекая выбранные признаки. Это могут быть любые рассмотренные признаки: функции из таблицы адресов импорта, строки, n-граммы инструкций или n-граммы динамического поведения. Здесь мы воспользуемся печатными строками, поскольку они дают хорошие результаты и просты для извлечения и понимания.

После извлечения строковых признаков необходимо перебрать все пары образцов и сравнить их признаки посредством индекса Жаккара. Затем следует построить граф общего кода. Для этого сначала выбирается порог объёма общего кода двух образцов; стандартное значение, которое я использую в своих исследованиях, равно 0,8. Если индекс Жаккара пары превышает порог, между образцами создаётся связь для визуализации. На последнем этапе граф исследуется для выявления образцов, соединённых отношениями общего кода.

В листингах 5-2 - 5-6 приведена наша демонстрационная программа. Поскольку она длинная, я разделю её на части и буду объяснять каждую по ходу изложения. Листинг 5-2 импортирует

используемые библиотеки и объявляет функцию `jaccard()`, вычисляющую индекс Жаккара двух множеств признаков.

```
#!/usr/bin/python

import argparse
import os
import networkx
from networkx.drawing.nx_pydot import write_dot
import itertools

def jaccard(set1, set2):
    """
    Вычислить расстояние Жаккара между двумя множествами: получить
    их пересечение и объединение, а затем разделить количество
    элементов пересечения на количество элементов объединения.
    """
    intersection = set1.intersection(set2)
    intersection_length = float(len(intersection))
    union = set1.union(set2)
    union_length = float(len(union))
    return intersection_length / union_length
```

Листинг 5-2. Импортируемые модули и вспомогательная функция для вычисления индекса Жаккара двух образцов

Далее, в листинге 5-3, объявляются ещё две служебные функции: `getstrings()`, находящая множество последовательностей печатных символов в анализируемых файлах, и `pecheck()`, проверяющая, что исследуемые файлы действительно являются PE-файлами Windows. Эти функции понадобятся позднее при извлечении признаков из исследуемых вредоносных двоичных файлов.

```
def getstrings(fullpath):
    """
    Извлечь строки из двоичного файла, указанного параметром fullpath,
    и вернуть множество уникальных строк этого файла.
    """
    strings = os.popen("strings '{0}'".format(fullpath)).read()
    strings = set(strings.split("\n"))
    return strings

def pecheck(fullpath):
    """
    Выполнить быструю проверку и убедиться, что fullpath указывает
    на исполняемый PE-файл Windows (PE-файлы начинаются двумя
    байтами MZ).
    """
    return open(fullpath).read(2) == "MZ"
```

Листинг 5-3. Объявление функций для извлечения признаков

Затем в листинге 5-4 разбираются аргументы командной строки пользователя. К ним относятся исследуемый каталог с анализируемыми вредоносными программами, выходной файл `.dot` для построенной сети общего кода и порог индекса Жаккара, определяющий, насколько высоким должен быть индекс двух образцов, чтобы программа решила, что они происходят из общей базы кода.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Identify similarities between malware samples and build similarity
    graph"
    )

    parser.add_argument(
        "target_directory",
        help="Directory containing malware"
    )
    parser.add_argument(
```

```
        "output_dot_file",
        help="Where to save the output graph DOT file"
    )

    parser.add_argument(
        "--jaccard_index_threshold", "-j", dest="threshold", type=float,
        default=0.8, help="Threshold above which to create an 'edge' between samples"
    )

    args = parser.parse_args()
```

Листинг 5-4. Разбор аргументов командной строки пользователя

Далее, в листинге 5-5, ранее объявленные вспомогательные функции выполняют основную работу программы: находят PE-файлы в исследуемом каталоге, извлекают из них признаки и создают сеть для выражения отношений сходства между двоичными файлами.

```
malware_paths = []  # здесь будут храниться пути к вредоносным файлам
malware_features = dict()  # здесь будут храниться строки вредоносных программ
graph = networkx.Graph()  # граф сходства

for root, dirs, paths in os.walk(args.target_directory):
    # обойти дерево исследуемого каталога и сохранить все пути к файлам
    for path in paths:
        full_path = os.path.join(root, path)
        malware_paths.append(full_path)

# исключить пути, не указывающие на PE-файлы
malware_paths = filter(pecheck, malware_paths)

# получить и сохранить строки всех вредоносных PE-файлов
for path in malware_paths:
    features = getstrings(path)
    print "Extracted {0} features from {1} ...".format(len(features), path)
    malware_features[path] = features

    # добавить каждый вредоносный файл в граф
    graph.add_node(path, label=os.path.split(path)[-1][:10])
```

Листинг 5-5. Извлечение признаков из PE-файлов исследуемого каталога и создание сети общего кода

После извлечения признаков необходимо перебрать все пары образцов и сравнить их признаки с помощью индекса Жаккара. Это выполняется в листинге 5-6. Кроме того, мы строим граф общего кода, где образцы соединяются, если их индекс превышает заданный пользователем порог. В моих исследованиях наилучшим оказался порог 0,8.

```
# перебрать все пары вредоносных программ
for malware1, malware2 in itertools.combinations(malware_paths, 2):

    # вычислить расстояние Жаккара для текущей пары
    jaccard_index = jaccard(malware_features[malware1], malware_features[malware2])

    # если расстояние Жаккара превышает порог, добавить ребро
    if jaccard_index > args.threshold:
        print malware1, malware2, jaccard_index
        graph.add_edge(malware1, malware2, penwidth=1+(jaccard_index-args.threshold)*10)

# записать граф на диск для визуализации
write_dot(graph, args.output_dot_file)
```

Листинг 5-6. Создание графа общего кода в Python

При применении к образцам APT1 код из листингов 5-2 - 5-6 создаёт диаграмму на рис. 5-11. Чтобы её визуализировать, следует воспользоваться рассмотренным в главе 4 инструментом GraphViz `fdp` и выполнить команду `fdp -Tpng network.dot -o network.png`.

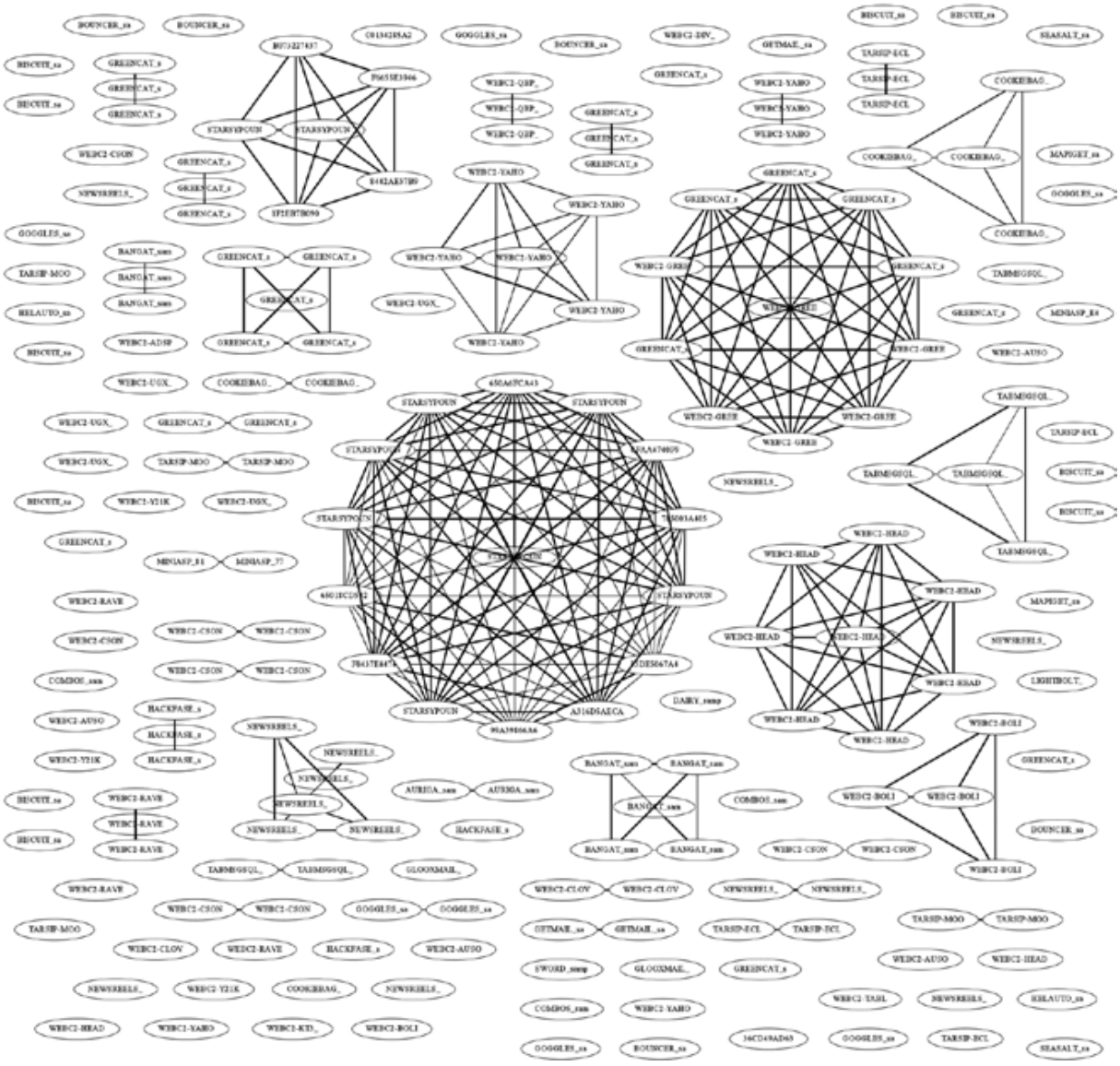

Рисунок 5-11. Полный граф сходства образцов APT1 на основе строк

Поразительно, что всего за несколько минут мы воспроизвели значительную часть кропотливой ручной работы, проделанной исходными аналитиками APT1 при подготовке отчёта, и выявили многие семейства вредоносных программ, применявшиеся злоумышленниками государственного уровня.

Мы знаем, что наш метод дал точные результаты в сравнении с ручной обратной разработкой аналитиков, потому что узлы носят названия, присвоенные специалистами Mandiant. Это видно по группировке образцов с похожими именами на сетевой визуализации рис. 5-11, например образцов `STARSYPOUN` в центральном круге. Поскольку вредоносные программы автоматически объединяются в группы, согласующиеся с названиями семейств, наш метод, по-видимому, «согласен» с аналитиками Mandiant.

Код из листингов 5-2 - 5-6 можно расширить и применить к собственным вредоносным программам для получения аналогичных сведений.

Масштабирование сравнений сходства

Хотя код из листингов 5-2 - 5-6 хорошо работает с небольшими наборами вредоносных программ, для большого количества образцов он не подходит. Причина в том, что число попарных сравнений растёт квадратично с количеством образцов. В частности, число вычислений индекса Жаккара для матрицы сходства набора размера n выражается формулой:

```
(n^2 - n) / 2
```

Вернёмся, например, к матрице сходства на рис. 5-5 и определим, сколько индексов Жаккара нужно вычислить для четырёх образцов. На первый взгляд ответ равен 16, то есть 4 в квадрате, поскольку столько ячеек в матрице. Однако нижний треугольник дублирует верхний, поэтому повторные вычисления не нужны и из общего количества можно вычесть 6. Кроме того, не требуется сравнивать образцы с самими собой, а значит, можно исключить диагональ и вычесть ещё четыре вычисления.

Требуемое количество вычислений равно:

```
(4^2 - 4) / 2 = (16 - 4) / 2 = 6
```

Это кажется приемлемым, пока набор не вырастет, например, до 10 000 образцов, для которых понадобится 49 995 000 вычислений. Набор из 50 000 образцов потребует 1 249 975 000 вычислений индекса Жаккара.

Для масштабирования сравнений сходства необходимы рандомизированные алгоритмы приближённого сравнения. Основная идея - допустить некоторую погрешность вычислений в обмен на сокращение времени. Для наших целей прекрасно подходит приближённый метод MinHash. Он позволяет приближённо вычислять индекс Жаккара и не сравнивать заведомо непохожие образцы ниже заранее заданного порога, благодаря чему можно анализировать отношения общего кода миллионов программ.

Прежде чем читать объяснение MinHash, учтите: это непростой алгоритм, на понимание которого может потребоваться время. Если вы решите пропустить раздел «MinHash в подробностях», прочитайте хотя бы раздел «MinHash в двух словах» и используйте предоставленный код - тогда масштабирование анализа общего кода не вызовет проблем.

MinHash в двух словах

MinHash берёт признаки образца и хеширует их k хеш-функциями. Для каждой функции сохраняется лишь минимальное значение среди хешей всех признаков, так что множество признаков сокращается до массива фиксированного размера из k целых чисел, называемых минимальными хешами. Чтобы получить приближённый индекс Жаккара двух образцов по их массивам, достаточно определить, сколько из k минимальных хешей совпадает, и разделить это число на k.

Удивительным образом полученное значение оказывается близким приближением настоящего индекса Жаккара двух образцов. Преимущество MinHash перед буквальным вычислением индекса состоит в значительно более высокой скорости.

Более того, MinHash позволяет хитроумно индексировать вредоносные программы в базе данных, чтобы сравнивать лишь те образцы, которые, вероятно, похожи, поскольку у них совпал хотя бы один хеш. Это резко ускоряет вычисление сходства внутри наборов.

MinHash в подробностях

Теперь подробно рассмотрим математические основы MinHash. На рис. 5-12 показаны множества признаков двух образцов, обозначенные затенёнными кругами, их хеширование и сортировка по значениям хешей, а затем сравнение по первому элементу каждого списка.

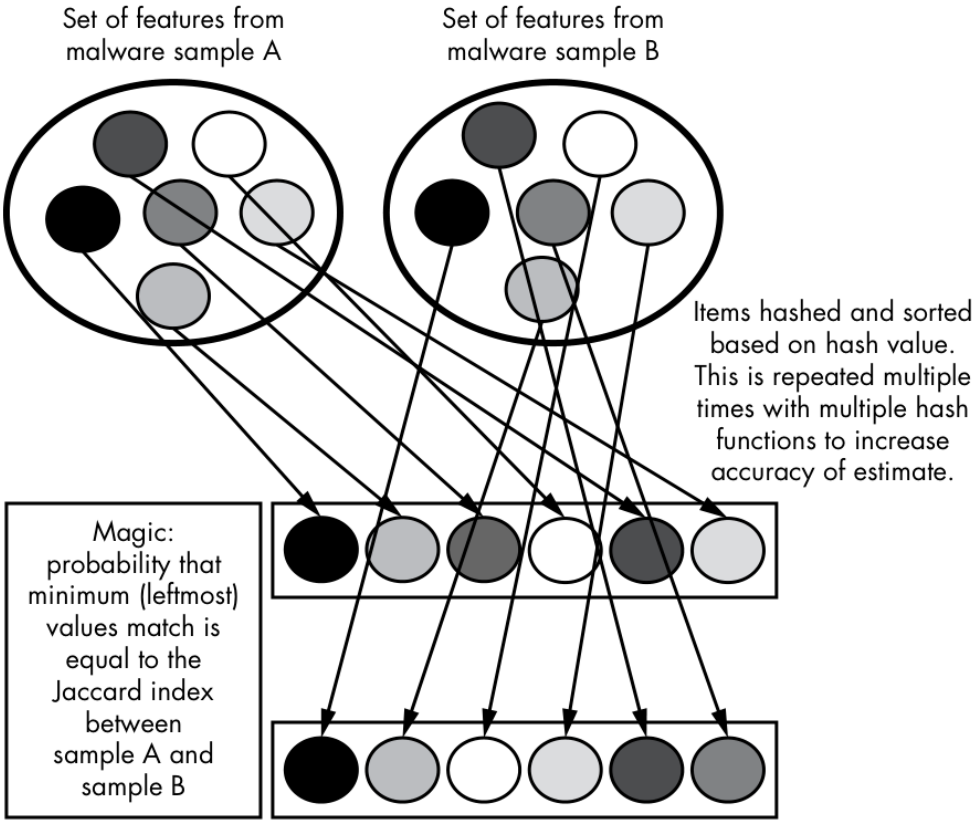

Рисунок 5-12. Наглядное представление идеи MinHash

Вероятность совпадения первых элементов равна индексу Жаккара образцов. Объяснение причины выходит за рамки книги, но именно этот удачный факт позволяет приближённо вычислять индекс с помощью хешей.

Разумеется, однократное хеширование, сортировка и проверка первого элемента мало что сообщают: хеши либо совпали, либо нет, и по одному совпадению невозможно точно угадать исходный индекс Жаккара. Для более точной оценки необходимо использовать k хеш-функций, повторить операцию k раз, а затем разделить количество совпадений первых элементов на k. Ожидаемая погрешность оценки определяется формулой:

```
1.0 / sqrt(k)
```

Таким образом, чем больше повторений, тем выше уверенность. Обычно я задаю k равным 256, и тогда средняя погрешность оценки составляет 6 процентов.

Предположим, мы вычислили массив MinHash для каждого образца в наборе из миллиона программ. Как использовать эти значения для ускорения поиска семейств? Можно было бы перебрать все пары и сравнить массивы, но это привело бы к 499 999 500 000 сравнений. Хотя массивы MinHash сравниваются быстрее, чем вычисляется индекс Жаккара, такое число всё равно чрезмерно для современного оборудования. Необходимо дополнительно использовать минимальные хеши для оптимизации.

Стандартный подход состоит в совместном применении эскизов и индексации базы данных, создающем систему, где сравниваются только образцы с уже известной высокой вероятностью сходства. Эскиз создаётся хешированием нескольких минимальных хешей вместе.

При поступлении нового образца мы проверяем, есть ли в базе эскизы, совпадающие с его эскизами. Если есть, новый образец сравнивается с совпавшими по массивам MinHash, чтобы приблизительно вычислить индекс Жаккара между ним и более старыми похожими программами. Таким образом не требуется сопоставлять новый образец со всей базой: сравниваются лишь программы с высокой вероятностью большого индекса Жаккара.

Построение постоянной системы поиска похожих вредоносных программ

Теперь вы знаете достоинства и недостатки различных типов признаков для оценки отношений общего кода между образцами. Вы также познакомились с индексом Жаккара, матрицами сходства и тем, как MinHash делает вычисление сходства практически осуществимым даже для очень больших наборов. Этих фундаментальных понятий достаточно для создания масштабируемой системы поиска общего кода вредоносных программ.

В листингах 5-7 - 5-12 приведён пример простой системы, индексирующей образцы по строковым признакам. В собственной работе вы сможете изменить её для использования других признаков или расширить дополнительными средствами визуализации. Листинг длинный, поэтому разбит на части, которые мы рассмотрим по очереди.

Сначала листинг 5-7 импортирует необходимые пакеты Python.

```
#!/usr/bin/python

import argparse
import os
import murmur
import shelve
import numpy as np
from listings_5_2_to_5_6 import *

NUM_MINHASHES = 256
SKETCH_RATIO = 8
```

Листинг 5-7. Импорт модулей Python и объявление констант, связанных с MinHash

Здесь импортируются такие пакеты, как `murmur`, `shelve` и `sim_graph`. Например, `murmur` - библиотека хеширования, применяемая для вычисления только что рассмотренного алгоритма MinHash. Простой модуль базы данных `shelve` из стандартной библиотеки Python хранит сведения об образцах и их минимальных хешах, используемых при вычислении сходства. Файл `listings_5_2_to_5_6.py` предоставляет функции вычисления сходства образцов.

Кроме того, в листинге 5-7 объявлены две константы: `NUM_MINHASHES` и `SKETCH_RATIO`. Они задают количество минимальных хешей и отношение числа минимальных хешей к количеству эскизов, вычисляемых для каждого образца. Напомним: чем больше минимальных хешей и эскизов, тем точнее вычисление сходства. Например, 256 минимальных хешей и отношение 8:1, то есть 32 эскиза, обеспечивают приемлемую точность при небольших вычислительных затратах.

В листинге 5-8 реализованы функции базы данных для создания, открытия и удаления базы `shelve`, хранящей сведения об образцах.

```
(1) def wipe_database():
    """
    Эта программа использует базу данных shelve стандартной библиотеки Python
```

```
    для постоянного хранения сведений в файле samples.db в одном каталоге
    со сценарием. wipe_database удаляет этот файл, фактически сбрасывая систему.
    """
    dbpath = "/".join(__file__.split('/')[:-1] + ['samples.db'])
    os.system("rm -f {0}".format(dbpath))

(2) def get_database():
    """
    Вспомогательная функция получения базы shelve, представляющей собой
    простое хранилище пар «ключ - значение».
    """
    dbpath = "/".join(__file__.split('/')[:-1] + ['samples.db'])
    return shelve.open(dbpath,protocol=2,writeback=True)
```

Листинг 5-8. Вспомогательные функции базы данных

Функция `wipe_database()` (1) удаляет базу программы, если требуется стереть сохранённые сведения об образцах и начать заново. Функция `get_database()` (2) открывает базу, при необходимости создавая её, и возвращает объект для сохранения и получения сведений о вредоносных программах.

Листинг 5-9 реализует ключевую часть анализа общего кода - MinHash.

```
def minhash(features):
    """
    Здесь происходит магия MinHash: вычисляются минимальные хеши признаков
    образца и эскизы этих хешей. Их количество задаётся глобальными
    переменными NUM_MINHASHES и NUM_SKETCHES в начале сценария.
    """
    minhashes = []
    sketches = []
    (1) for i in range(NUM_MINHASHES):
        minhashes.append(
        (2) min([murmur.string_hash(`feature`,i) for feature in features])
        )
    (3) for i in xrange(0,NUM_MINHASHES,SKETCH_RATIO):
        (4) sketch = murmur.string_hash(`minhashes[i:i+SKETCH_RATIO]`)
        sketches.append(sketch)
    return np.array(minhashes),sketches
```

Листинг 5-9. Получение минимальных хешей и эскизов образца

Мы выполняем цикл `NUM_MINHASHES` раз (1) и на каждой итерации добавляем одно минимальное значение. Оно вычисляется хешированием всех признаков с последующим выбором минимального хеша. Для хеширования используется функция `string_hash()` пакета `murmur`, а минимум списка выбирается функцией Python `min()` (2).

Второй аргумент `string_hash` - начальное значение, благодаря которому функция отображает данные в разные хеши в зависимости от этого значения. Для каждого минимального хеша нужна уникальная хеш-функция, иначе все 256 значений окажутся одинаковыми. Поэтому на каждой итерации `string_hash` получает значение счётчика `i`, и признаки всякий раз отображаются в другие хеши.

Затем мы перебираем вычисленные минимальные хеши и создаём по ним эскизы (3). Напомним, эскизы - это хеши нескольких минимальных хешей, применяемые для индексирования образцов в базе данных, чтобы запрос быстро находил программы с высокой вероятностью сходства. В следующем листинге мы перебираем все минимальные хеши образца с шагом `SKETCH_RATIO` и хешируем каждый фрагмент, получая эскизы. Наконец, функция `string_hash` пакета `murmur` хеширует минимальные хеши совместно (4).

Листинг 5-10 использует `get_database()` из листинга 5-8, функцию `getstrings()` из импортированного модуля `sim_graph` и `minhash()` из листинга 5-9, чтобы создать функцию индексирования образцов в базе системы.

```
def store_sample(path):
```

```
    """
    Сохранить образец, его минимальные хеши и эскизы в базе shelve.
    """
    (1) db = get_database()
    (2) features = getstrings(path)
    (3) minhashes,sketches = minhash(features)

    (4) for sketch in sketches:
        sketch = str(sketch)
        (5) if not sketch in db:
            db[sketch] = set([path])
        else:
            obj = db[sketch]
            (6) obj.add(path)
            db[sketch] = obj
        db[path] = {'minhashes':minhashes,'comments':[]}
        db.sync()

    print "Extracted {0} features from {1} ...".format(len(features),path)
```

Листинг 5-10. Сохранение минимальных хешей образца в базе shelve с использованием эскизов в качестве ключей

Мы вызываем `get_database()` (1), `getstrings()` (2) и `minhash()` (3), а затем, начиная с (4), перебираем эскизы образца. Для индексирования применяется метод *обратного индекса*, позволяющий сохранять образцы по значениям эскизов, а не по идентификатору. Точнее, для каждого из 32 эскизов программа находит его запись в базе и добавляет идентификатор образца в связанный с эскизом список. Идентификатором здесь служит путь в файловой системе.

Реализация видна в коде: программа перебирает вычисленные эскизы (4); создаёт запись, если она отсутствует, одновременно связывая с ней образец (5); а если запись существует, добавляет путь образца в связанное с эскизом множество путей (6).

В листинге 5-11 объявлены две важные функции: `comment_sample()` и `search_sample()`.

```
(1) def comment_sample(path):
    """
    Позволить пользователю прокомментировать образец. Комментарий показывается,
    когда образец встречается в списке программ, похожих на новый образец,
    и позволяет повторно использовать знания о базе вредоносных программ.
    """
    db = get_database()
    comment = raw_input("Enter your comment:")
    if not path in db:
        store_sample(path)
    comments = db[path]['comments']
    comments.append(comment)
    db[path]['comments'] = comments
    db.sync()
    print "Stored comment:", comment

(2) def search_sample(path):
    """
    Найти образцы, похожие на указанный аргументом path, и вывести их
    комментарии, имена файлов и значения сходства.
    """
    db = get_database()
    features = getstrings(path)
    minhashes, sketches = minhash(features)
    neighbors = []

    (3) for sketch in sketches:
        sketch = str(sketch)

        if not sketch in db:
            continue

        (4) for neighbor_path in db[sketch]:
            neighbor_minhashes = db[neighbor_path]['minhashes']
```

```
            similarity = (neighbor_minhashes == minhashes).sum() / float(NUM_MINHASHES)
            neighbors.append((neighbor_path, similarity))

    neighbors = list(set(neighbors))
    (5) neighbors.sort(key=lambda entry:entry[1], reverse=True)
    print ""
    print "Sample name".ljust(64), "Shared code estimate"
    for neighbor, similarity in neighbors:
        short_neighbor = neighbor.split("/")[-1]
        comments = db[neighbor]['comments']
        print str("[*] "+short_neighbor).ljust(64), similarity
        for comment in comments:
            print "\t[comment]",comment
```

Листинг 5-11. Объявление функций комментирования образцов и поиска программ, похожих на исследуемый образец

Как и ожидалось, `comment_sample()` (1) добавляет заданный пользователем комментарий к записи образца в базе. Это полезно, поскольку полученные при обратной разработке выводы сохраняются в базе. Когда позднее встретится новый образец, похожий на прокомментированные, комментарии помогут быстрее понять его происхождение и назначение.

Функция `search_sample()` (2) использует MinHash для поиска программ, похожих на образец запроса. Сначала извлекаются строковые признаки, минимальные хеши и эскизы нового образца. Затем перебираются его эскизы и в базе ищутся программы с тем же эскизом (3). Для каждого совпавшего образца по минимальным хешам вычисляется приближённый индекс Жаккара (4). Наконец, пользователю сообщаются наиболее похожие программы и все сохранённые для них комментарии (5).

Листинг 5-12 завершает программу, реализуя разбор аргументов.

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="""
Simple code-sharing search system which allows you to build up
a database of malware samples (indexed by file paths) and
then search for similar samples given some new sample
"""
    )

    parser.add_argument(
        "-l", "--load", dest="load", default=None,
        help="Path to malware directory or file to store in database"
    )

    parser.add_argument(
        "-s", "--search", dest="search", default=None,
        help="Individual malware file to perform similarity search on"
    )

    parser.add_argument(
        "-c", "--comment", dest="comment", default=None,
        help="Comment on a malware sample path"
    )

    parser.add_argument(
        "-w", "--wipe", action="store_true", default=False,
        help="Wipe sample database"
    )

    args = parser.parse_args()
    (1) if args.load:
        malware_paths = []  # здесь будут храниться пути к вредоносным файлам
        malware_features = dict()  # здесь будут храниться строки вредоносных программ
        for root, dirs, paths in os.walk(args.load):
            # обойти дерево исследуемого каталога и сохранить все пути к файлам
            for path in paths:
                full_path = os.path.join(root,path)
```

```
                malware_paths.append(full_path)

        # исключить пути, не указывающие на PE-файлы
        malware_paths = filter(pecheck, malware_paths)

        # получить и сохранить строки всех вредоносных PE-файлов
        for path in malware_paths:
            store_sample(path)

    (2) if args.search:
        search_sample(args.search)

    (3) if args.comment:
        comment_sample(args.comment)

    (4) if args.wipe:
        wipe_database()
```

Листинг 5-12. Обновление базы сходства и выполнение запросов в соответствии с аргументами командной строки пользователя

Здесь пользователь может загрузить в базу образцы, которые будут сравниваться с новыми программами при последующем поиске (1). Затем предоставляется возможность найти в базе программы, похожие на переданный образец, и вывести результаты в терминал (2). Пользователь также может комментировать уже сохранённые образцы (3). Наконец, можно очистить существующую базу (4).

Запуск системы поиска сходства

После реализации кода систему можно запускать. Она поддерживает четыре простые операции:

• **Загрузка.** Загруженные образцы сохраняются в базе для будущего поиска общего кода. Можно загрузить отдельный файл или указать каталог, который система рекурсивно просмотрит в поисках PE-файлов. Чтобы загрузить образцы, выполните из каталога кода главы следующую команду:

```
python listings_5_7_to_5_12.py -l <path to directory or individual malware sample>
```

• **Комментирование.** Комментарий позволяет сохранить знания об образце. Когда позднее встретятся похожие программы, поиск покажет комментарии к старому образцу и ускорит работу. Чтобы прокомментировать вредоносную программу, выполните:

```
python listings_5_7_to_5_12.py -c <path to malware sample>
```

• **Поиск.** Для одного образца поиск находит в базе все похожие программы и печатает их в порядке убывания сходства вместе со всеми оставленными комментариями. Команда имеет следующий вид:

```
python listings_5_7_to_5_12.py -s <path to malware sample>
```

• **Очистка.** Эта операция удаляет из системной базы все записи:

```
python listings_5_7_to_5_12.py -w
```

В листинге 5-13 показана загрузка образцов APT1 в систему.

```
mds@mds:~/malware_data_science/ch5/code$ python listings_5_7_to_5_12.py -l ../
data
Extracted 240 attributes from ../data/APT1_MALWARE_FAMILIES/WEBC2-YAHOO/WEBC2-
YAHOO_sample/WEBC2-YAHOO_sample_A8F259BB36E00D124963CFA9B86F502E ...
Extracted 272 attributes from ../data/APT1_MALWARE_FAMILIES/WEBC2-YAHOO/WEBC2-
YAHOO_sample/WEBC2-YAHOO_sample_0149B7BD7218AAB4E257D28469FDDB0D ...
Extracted 236 attributes from ../data/APT1_MALWARE_FAMILIES/WEBC2-YAHOO/WEBC2-
YAHOO_sample/WEBC2-YAHOO_sample_CC3A9A7B026BFE0E55FF219FD6AA7D94 ...
Extracted 272 attributes from ../data/APT1_MALWARE_FAMILIES/WEBC2-YAHOO/WEBC2-
```

```
YAHOO_sample/WEBC2-YAHOO_sample_1415EB8519D13328091CC5C76A624E3D ...
Extracted 236 attributes from ../data/APT1_MALWARE_FAMILIES/WEBC2-YAHOO/WEBC2-
YAHOO_sample/WEBC2-YAHOO_sample_7A670D13D4D014169C4080328B8FEB86 ...
Extracted 243 attributes from ../data/APT1_MALWARE_FAMILIES/WEBC2-YAHOO/WEBC2-
YAHOO_sample/WEBC2-YAHOO_sample_37DDD3D72EAD03C7518F5D47650C8572 ...
--snip--
```

Листинг 5-13. Пример вывода при загрузке данных в реализованную в этой главе систему поиска сходства

Далее в листинге 5-14 показано выполнение поиска сходства.

```
mds@mds:~/malware_data_science/ch5/code$ python listings_5_7_to_5_12.py -s \
../data/APT1_MALWARE_FAMILIES/GREENCAT/GREENCAT_sample/GREENCAT_sample_AB20\
8F0B517BA9850F1551C9555B5313
Sample name                                                    Shared code
estimate
[*] GREENCAT_sample_5AEAA53340A281074FCB539967438E3F           1.0
[*] GREENCAT_sample_1F92FF8711716CA795FBD81C477E45F5           1.0
[*] GREENCAT_sample_3E69945E5865CCC861F69B24BC1166B6           1.0
[*] GREENCAT_sample_AB208F0B517BA9850F1551C9555B5313           1.0
[*] GREENCAT_sample_3E6ED3EE47BCE9946E2541332CB34C69           0.99609375
[*] GREENCAT_sample_C044715C2626AB515F6C85A21C47C7DD           0.6796875
[*] GREENCAT_sample_871CC547FEB9DBEC0285321068E392B8           0.62109375
[*] GREENCAT_sample_57E79F7DF13C0CB01910D0C688FCD296           0.62109375
```

Листинг 5-14. Пример вывода реализованной в этой главе системы поиска сходства

Обратите внимание: система правильно определила, что образец запроса из семейства greencat имеет общий код с другими представителями greencat. Если бы мы не знали заранее, что все они относятся к одному семейству, система сэкономила бы огромный объём работы по обратной разработке.

Эта система - лишь небольшой пример того, как могла бы выглядеть промышленная система поиска сходства. Но полученных знаний достаточно, чтобы без особых затруднений добавить в неё визуализацию и поддержку нескольких методов поиска.

Итоги

В этой главе вы научились выявлять отношения общего кода между вредоносными программами, вычислять сходство для тысяч образцов с целью обнаружения новых семейств, определять сходство кода новой программы с тысячами уже встречавшихся и визуализировать отношения вредоносных программ для понимания закономерностей совместного использования кода.

Теперь вы должны уверенно добавлять анализ общего кода в свой набор средств анализа вредоносных программ. Он позволяет быстро получать сведения о больших объёмах программ и ускорять рабочий процесс.

В главах 6, 7 и 8 вы научитесь создавать системы машинного обучения для обнаружения вредоносных программ. Сочетание этих методов обнаружения с уже полученными знаниями поможет выявлять сложные вредоносные программы, пропущенные другими инструментами, а также анализировать их отношения с известными образцами и получать подсказки о том, кто развернул программу и каковы цели злоумышленника.

Глава 6. Понимание детекторов вредоносных программ на основе машинного обучения

Современные инструменты машинного обучения с открытым исходным кодом позволяют сравнительно легко создавать собственные детекторы вредоносных программ на основе машинного обучения, используя их как основное средство обнаружения либо как дополнение к коммерческим решениям.

Но зачем создавать собственные инструменты машинного обучения, если уже существуют коммерческие антивирусы? Имея примеры конкретных угроз, например вредоносных программ определённой группы злоумышленников, нацеленной на вашу сеть, можно построить собственную технологию на основе машинного обучения и обнаруживать новые экземпляры этих угроз.

Коммерческие антивирусные механизмы, напротив, могут пропустить такие угрозы, если для них ещё нет сигнатур. Кроме того, коммерческие средства представляют собой «закрытые книги»: мы не обязательно знаем, как они работают, и располагаем ограниченными возможностями настройки. Создавая собственные методы обнаружения, мы понимаем их устройство и можем настраивать их для уменьшения количества ложноположительных или ложноотрицательных результатов. Это полезно, поскольку в одних применениях допустимо больше ложноположительных результатов в обмен на меньшее число ложноотрицательных, например при поиске подозрительных файлов в сети для последующей ручной проверки. В других применениях допустимо больше ложноотрицательных результатов ради уменьшения ложноположительных, например если программа блокирует выполнение файлов, признанных вредоносными, и ложные срабатывания мешают пользователям.

В этой главе вы в общих чертах познакомитесь с процессом разработки собственных средств обнаружения. Сначала я объясню основные идеи машинного обучения: пространства признаков, границы решений, обучающие данные, недообучение и переобучение. Затем сосредоточусь на четырёх фундаментальных подходах - логистической регрессии, k ближайших соседях, деревьях решений и случайном лесе - и их применении к обнаружению.

Полученные знания понадобятся в главе 7 для оценки точности систем машинного обучения, а в главе 8 - для реализации таких систем на Python. Приступим.

Этапы создания детектора на основе машинного обучения

Между машинным обучением и другими видами компьютерных алгоритмов есть принципиальное различие. Традиционные алгоритмы сообщают компьютеру, что делать, а системы машинного обучения учатся решать задачу по примерам. Например, вместо простого применения набора заранее заданных правил система обнаружения угроз может обучиться определять вредоносность файла по примерам вредоносных и безвредных файлов.

Машинное обучение привлекательно для компьютерной безопасности тем, что автоматизирует создание сигнатур и способно обнаруживать вредоносные программы точнее сигнатурных подходов, особенно если речь идёт о новых, ранее не встречавшихся программах.

По существу, создание любого детектора на основе машинного обучения, включая дерево решений, сводится к следующим этапам:

1. Собрать примеры вредоносных и безвредных программ. Эти *обучающие примеры* будут использоваться для обучения системы распознавать вредоносные программы.

2. Извлечь признаки каждого обучающего примера и представить его массивом чисел. Сюда же относится исследование и проектирование хороших признаков, которые помогут системе делать точные выводы.
3. Обучить систему распознавать вредоносные программы по извлечённым признакам.
4. Проверить подход на данных, не входивших в обучающие примеры, чтобы оценить работу системы обнаружения.

В следующих разделах каждый этап рассматривается подробнее.

Сбор обучающих примеров

Успех или неудача детектора машинного обучения определяется предоставленными обучающими данными. Способность распознавать подозрительные двоичные файлы во многом зависит от количества и качества примеров. При создании детектора будьте готовы потратить значительную часть времени на их сбор: чем больше примеров получает система, тем точнее она, вероятно, будет.

Качество примеров не менее важно. Собранные вредоносные и безвредные программы должны соответствовать тем файлам, которые детектор будет видеть позднее, когда ему придётся решать, является новый файл вредоносным или нет.

Например, если требуется обнаруживать программы определённой группы субъектов угроз, необходимо собрать для обучения как можно больше вредоносных программ этой группы. Если цель - выявлять широкий класс, например программы-вымогатели, крайне важно собрать максимально представительную выборку этого класса.

Точно так же безвредные обучающие примеры должны соответствовать безвредным файлам, которые детектор будет анализировать после развёртывания. Например, систему обнаружения вредоносных программ в университетской сети следует обучать на широкой выборке безвредных программ студентов и сотрудников, чтобы избежать ложноположительных результатов. В неё войдут компьютерные игры, редакторы документов, специальные программы университетского ИТ-отдела и другие невредоносные приложения.

Приведу пример из своей повседневной работы. Мы создали детектор вредоносных документов Office и потратили около половины времени проекта на сбор обучающих данных, включая безвредные документы более чем тысячи сотрудников компании. Обучение на этих примерах значительно уменьшило долю ложноположительных результатов.

Извлечение признаков

Чтобы классифицировать файлы как хорошие или плохие, системы машинного обучения знакомят с признаками двоичных файлов - атрибутами, помогающими различать эти классы. Например, для определения вредоносности файла можно использовать следующие признаки:

• наличие цифровой подписи;
• наличие неправильно сформированных заголовков;
• наличие зашифрованных данных;
• наблюдался ли файл более чем на 100 рабочих станциях сети.

Эти признаки необходимо извлечь из файлов. Например, можно написать код, определяющий наличие цифровой подписи, ошибочных заголовков, зашифрованных данных и так далее. В анализе данных безопасности детекторы часто используют огромное количество признаков. Например, можно создать отдельный признак для каждого вызова библиотеки Win32 API: двоичный

файл будет обладать признаком, если содержит соответствующий вызов. К извлечению признаков мы вернёмся в главе 8, где рассмотрим более сложные понятия и их применение при реализации систем машинного обучения на Python.

Проектирование хороших признаков

Наша цель - выбрать признаки, обеспечивающие наиболее точные результаты. Ниже приведены несколько общих правил.

Во-первых, выбирайте признаки, которые, по вашему обоснованному предположению, помогут системе отличать плохие файлы от хороших. Например, признак «содержит зашифрованные данные» может быть хорошим указателем вредоносной программы, поскольку нам известно, что такие программы часто содержат зашифрованные данные, а в безвредных они, как мы предполагаем, встречаются реже. Прелесть машинного обучения в том, что, если гипотеза ошибочна и безвредные программы содержат шифрованные данные столь же часто, система практически проигнорирует признак. Если гипотеза верна, она научится применять его для обнаружения.

Во-вторых, не используйте столько признаков, чтобы их множество стало слишком большим по сравнению с количеством обучающих примеров. Специалисты называют это «проклятием размерности». Например, при тысяче признаков и всего тысяче примеров, вероятно, данных недостаточно, чтобы система поняла значение каждого признака для двоичного файла. Статистика говорит, что лучше предоставить системе сравнительно немного признаков относительно доступных примеров и позволить сформировать обоснованные представления о том, какие признаки действительно указывают на вредоносность.

Наконец, признаки должны представлять широкий диапазон гипотез о свойствах вредоносных и безвредных программ. Можно создать признаки, связанные с шифрованием, например использование соответствующих вызовов API или инфраструктуры открытых ключей Public Key Infrastructure (PKI), но для подстраховки нужны и признаки, не связанные с шифрованием. Тогда, если система не обнаружит вредоносную программу по одному типу, она сможет воспользоваться другими.

Обучение систем машинного обучения

После извлечения признаков из обучающих двоичных файлов систему пора обучить. Конкретный алгоритм полностью зависит от выбранного подхода. Например, дерево решений, которое вскоре будет рассмотрено, обучается иначе, чем логистическая регрессия, также рассматриваемая далее.

К счастью, все детекторы машинного обучения предоставляют один основной интерфейс. Им передаются обучающие данные с признаками образцов и соответствующие метки, сообщающие алгоритму, какие двоичные файлы вредоносны, а какие безвредны. После этого алгоритмы учатся определять класс новых, ранее не встречавшихся файлов. Обучение подробнее рассматривается далее в этой главе.

> **Примечание.** В книге основное внимание уделяется классу алгоритмов, называемых *обучением с учителем*. При обучении моделей мы сообщаем, какие примеры вредоносны, а какие безвредны. Другой класс - алгоритмы обучения без учителя - не требует заранее знать класс примеров в обучающем наборе. Они значительно менее эффективны при обнаружении вредоносных программ и поведения, поэтому в этой книге не рассматриваются.

Тестирование систем машинного обучения

Обучив систему, необходимо проверить её точность. Для этого обученную систему запускают на данных, которые при обучении не использовались, и оценивают, насколько хорошо она определяет вредоносные и безвредные двоичные файлы. В сфере безопасности системы обычно обучают на файлах, собранных до определённого момента времени, а затем проверяют на появившихся позднее, чтобы измерить способность обнаруживать новые вредоносные программы и избегать ложноположительных результатов на новых безвредных файлах. Большинство исследований машинного обучения состоит из тысяч итераций: систему создают, проверяют, изменяют, снова обучают и снова проверяют, пока результат не станет удовлетворительным. Подробно тестирование будет рассмотрено в главе 8.

Теперь обсудим работу различных алгоритмов машинного обучения. Это самая трудная часть главы, но она же принесёт наибольшую пользу, если уделить время пониманию. Сначала я расскажу об объединяющих алгоритмы идеях, а затем подробно разберу каждый из них.

Пространства признаков и границы решений

Понять все алгоритмы обнаружения на основе машинного обучения помогают две простые геометрические идеи: пространство признаков и граница решения. *Пространство признаков* - это геометрическое пространство, определённое выбранными признаками. *Граница решения* - геометрическая структура в этом пространстве, по одну сторону которой двоичные файлы считаются вредоносными, а по другую - безвредными. Чтобы классифицировать файл, алгоритм извлекает признаки, помещает образец в пространство и проверяет, с какой стороны границы он находится.

Геометрическое понимание точно для пространств одного, двух и трёх измерений, то есть признаков, но справедливо и для миллионов измерений, хотя представить такое пространство невозможно. В этой главе мы ограничимся двумя измерениями ради наглядности, но реальные системы машинного обучения в безопасности почти всегда используют сотни, тысячи или миллионы измерений, и обсуждаемые двумерные принципы остаются применимыми.

Создадим учебную задачу обнаружения вредоносных программ, чтобы пояснить границу решения в пространстве признаков. Предположим, есть обучающий набор вредоносных и безвредных образцов. Из каждого двоичного файла извлекаются два признака: доля файла, которая выглядит сжатой, и количество подозрительных импортируемых функций. Обучающий набор можно представить, как на рис. 6-1; данные на графике искусственно созданы для примера.

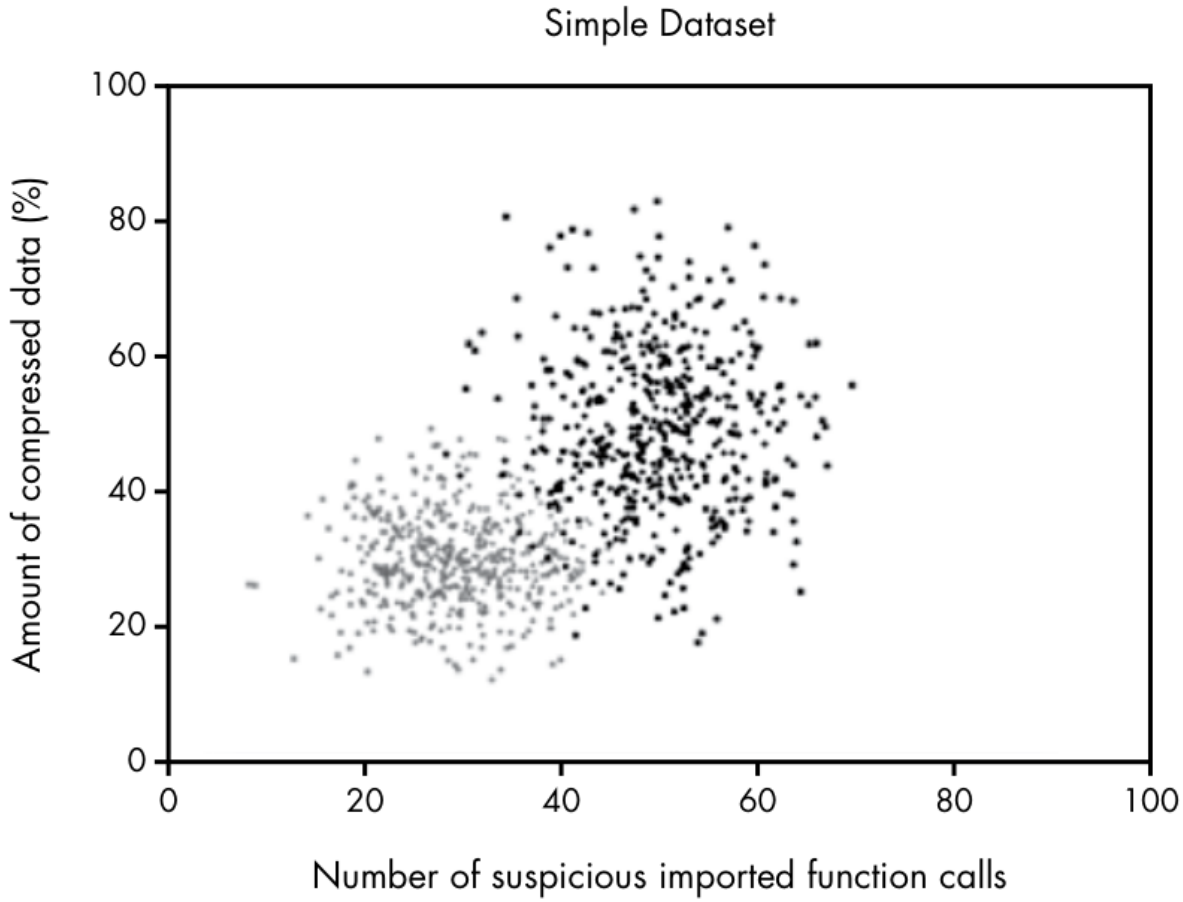

Рисунок 6-1. Учебный набор данных из этой главы: серые точки обозначают безвредные программы, чёрные - вредоносные

Двумерное пространство на рис. 6-1, определённое двумя признаками, является пространством признаков учебного набора. Видна закономерность: чёрные точки, то есть вредоносные программы, в основном расположены в правой верхней части. В целом у них больше подозрительных импортируемых функций и сжатых данных, чем у безвредных программ, преимущественно находящихся в левой нижней части. Если бы вас попросили создать детектор только на этих признаках, из данных следовало бы очевидное правило: двоичный файл с большим количеством как сжатых данных, так и подозрительных импортируемых функций является вредоносным, а файл без большого количества обоих признаков - безвредным.

Геометрически правило можно представить диагональной линией, разделяющей вредоносные и безвредные образцы в пространстве признаков. Двоичные файлы с достаточным количеством сжатых данных и импортируемых функций, определяемые как вредоносные, находятся выше линии, остальные - ниже. Такая линия, называемая границей решения, показана на рис. 6-2.

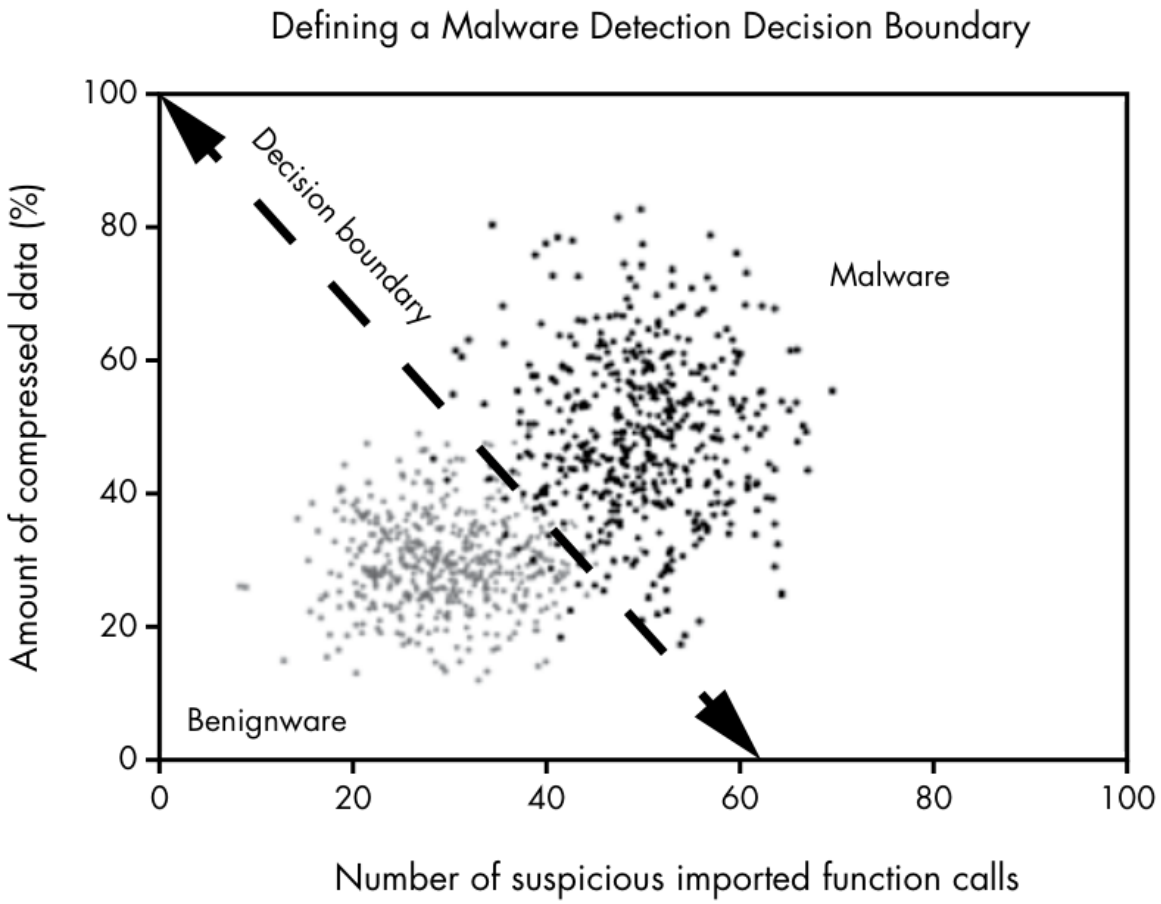

Рисунок 6-2. Граница решения в учебном наборе, задающая правило обнаружения вредоносных программ

Как видно, большинство чёрных точек находится по одну сторону границы, а серых - по другую. Провести линию, разделяющую абсолютно все образцы, невозможно, поскольку чёрное и серое облака перекрываются. Но, судя по примеру, эта линия в большинстве случаев правильно классифицирует новые вредоносные и безвредные программы, если они следуют показанной закономерности.

На рис. 6-2 граница проведена вручную. Но что, если требуется более точная граница, построенная автоматически? Именно это и делает машинное обучение. Все алгоритмы изучают данные и автоматически определяют идеальную границу, которая с наибольшей вероятностью правильно обнаружит новые, ранее не встречавшиеся данные.

Рассмотрим, как распространённый реальный алгоритм определяет границу решения в учебных данных на рис. 6-3. Этот алгоритм называется логистической регрессией.

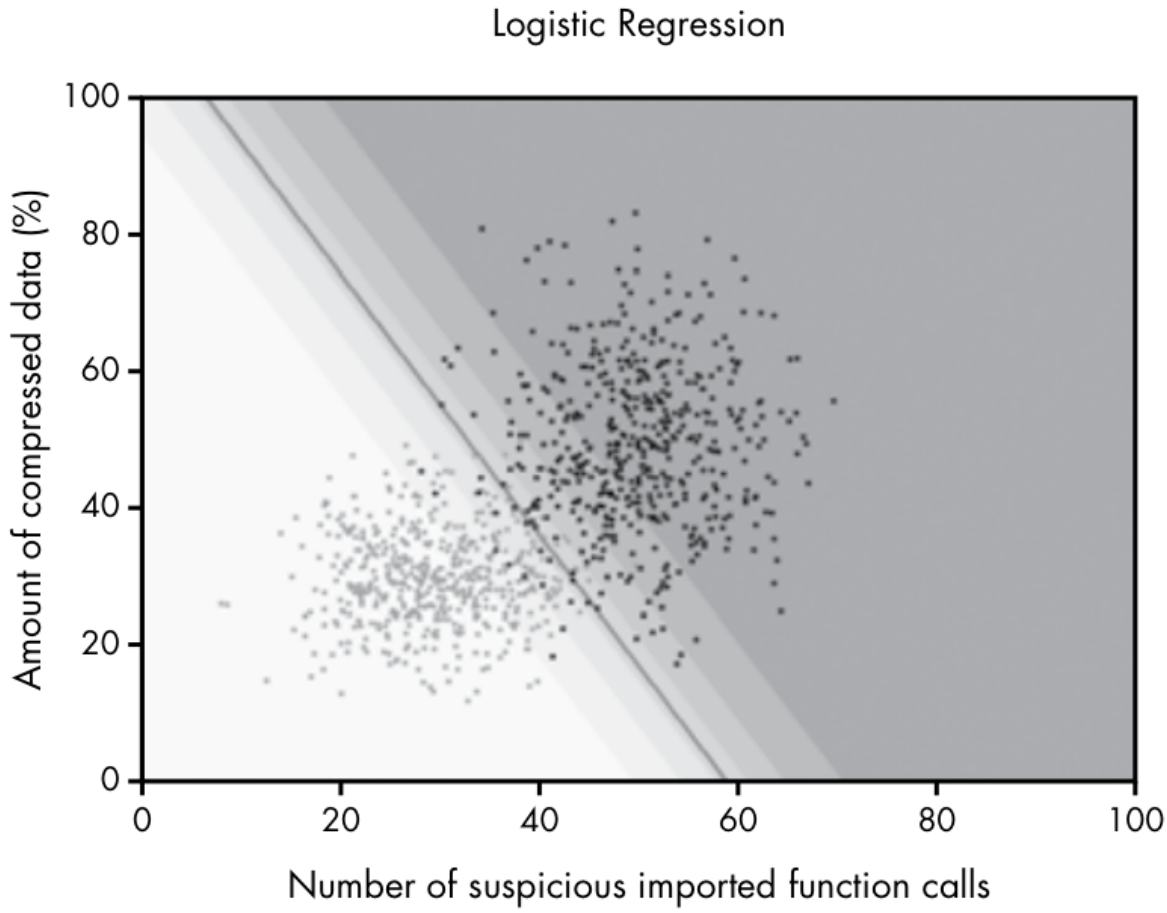

Рисунок 6-3. Граница решения, автоматически построенная при обучении модели логистической регрессии

Здесь используются те же данные, что на предыдущих графиках: серые точки обозначают безвредные программы, чёрные - вредоносные. Линия через центр - граница, которую алгоритм логистической регрессии изучил по данным. Справа от неё алгоритм назначает двоичным файлам вероятность вредоносности более 50 процентов, слева - менее 50 процентов.

Обратите внимание на затенённые области. В тёмно-серой области модель с высокой уверенностью считает файлы вредоносными. Любой новый файл с попавшими туда признаками должен иметь высокую вероятность вредоносности. По мере приближения к границе уверенность модели в классе уменьшается. Логистическая регрессия позволяет легко перемещать линию вверх, в более тёмную область, или вниз, в более светлую, в зависимости от желаемой агрессивности обнаружения. При перемещении вниз будет найдено больше вредоносных программ, но возрастёт количество ложноположительных результатов. При перемещении вверх будет найдено меньше вредоносных программ и получено меньше ложноположительных результатов.

Важно подчеркнуть: логистическая регрессия, как и все остальные алгоритмы, может работать в пространстве признаков произвольно большой размерности. На рис. 6-4 показана работа в пространстве немного большей размерности.

В трёхмерном пространстве граница представляет собой не линию, а плоскость, разделяющую точки объёма. В четырёх и более измерениях логистическая регрессия создаёт гиперплоскость - n-мерную плоскоподобную структуру, разделяющую точки вредоносных и безвредных программ в многомерном пространстве.

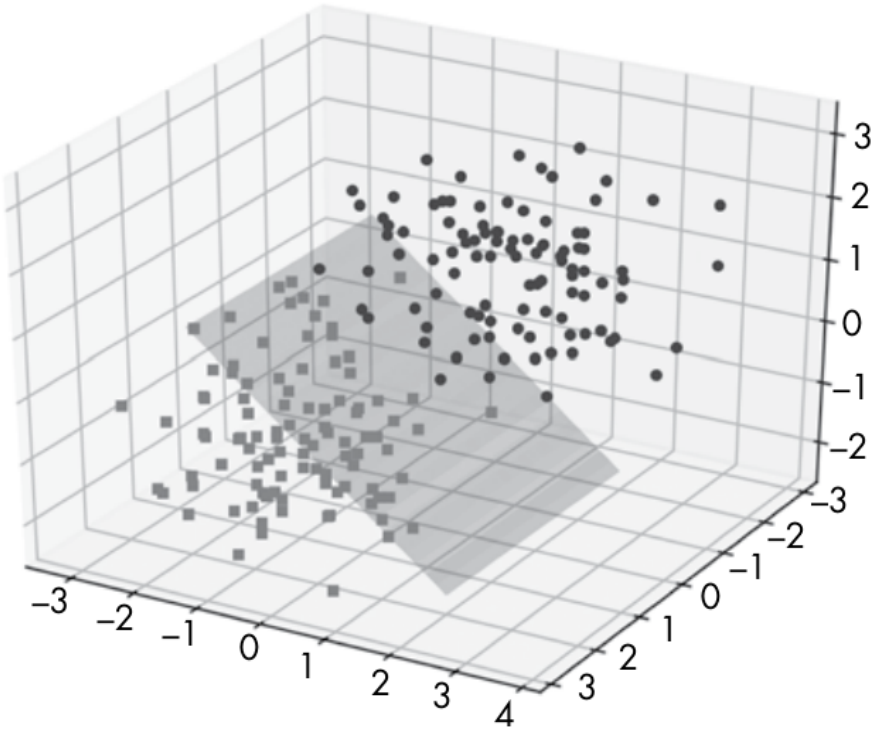

Рисунок 6-4. Плоская граница решения в гипотетическом трёхмерном пространстве признаков, созданная логистической регрессией

Поскольку логистическая регрессия - сравнительно простой алгоритм, она создаёт только простые геометрические границы: линии, плоскости и многомерные плоскости. Другие алгоритмы способны строить более сложные структуры. Рассмотрим, например, границу на рис. 6-5, полученную методом k ближайших соседей, который вскоре будет подробно описан.

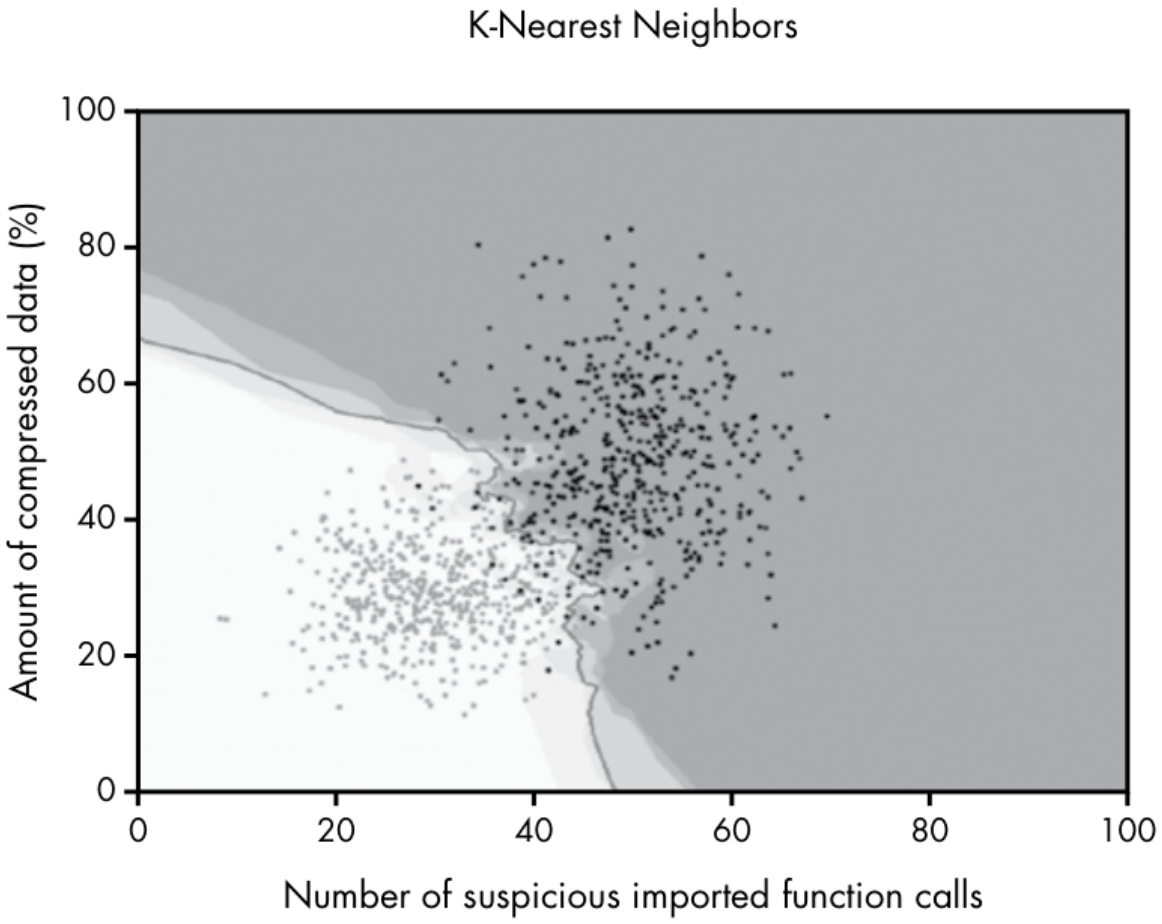

Рисунок 6-5. Граница решения, созданная алгоритмом k ближайших соседей

Как видите, эта граница не является плоскостью, а представляет собой крайне нерегулярную структуру. Некоторые алгоритмы также способны создавать несвязные границы, определяющие отдельные области пространства как вредоносные или безвредные, даже если области не соприкасаются. На рис. 6-6 показана такая нерегулярная граница для другого учебного набора с более сложным расположением вредоносных и безвредных программ.

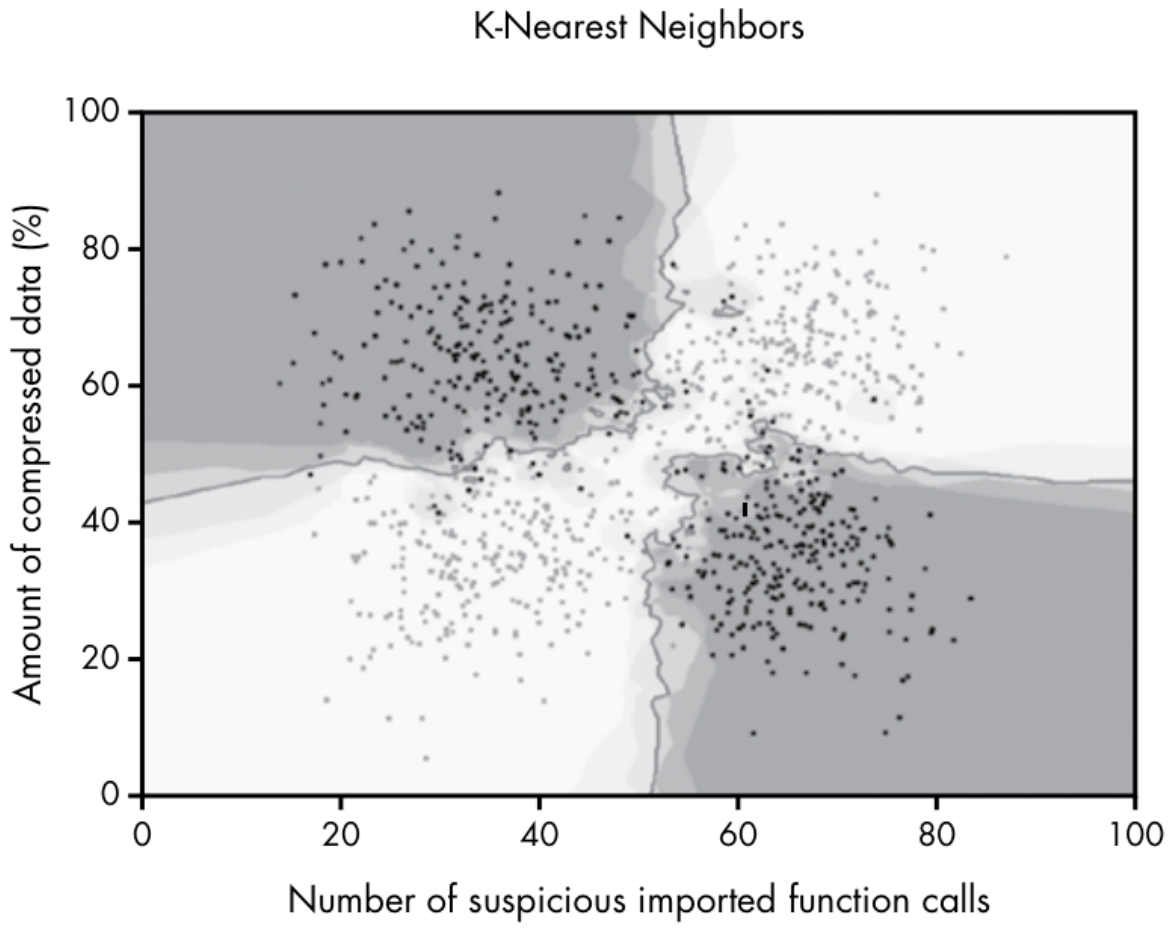

Рисунок 6-6. Несвязная граница решения, созданная алгоритмом k ближайших соседей

Несмотря на несвязность, на языке машинного обучения подобные структуры всё равно называют просто границами решений. Разные алгоритмы выражают разные типы границ, и различие в выразительности заставляет выбирать для конкретного проекта один алгоритм вместо другого.

Теперь, рассмотрев основные понятия пространств признаков и границ решений, обсудим переобучение и недообучение.

Что делает модели хорошими или плохими: переобучение и недообучение

Значение переобучения и недообучения в машинном обучении невозможно переоценить. Хороший алгоритм избегает обоих случаев. Точные модели обнаружения улавливают общую закономерность обучающих данных, отличающую вредоносные программы от безвредных, и не отвлекаются на выбросы или исключения, подтверждающие правило.

Недообученные модели игнорируют выбросы, но не улавливают общую закономерность и поэтому плохо работают на новых двоичных файлах. Переобученные модели отвлекаются на выбросы, которые не отражают закономерность, и также дают низкую точность на ранее не встречавшихся файлах. Создание детектора сводится к улавливанию общего различия между вредоносным и безвредным.

Проиллюстрируем понятия на недообученной, хорошо обученной и переобученной моделях с рис. 6-7, 6-8 и 6-9. На рис. 6-7 показана недообученная модель.

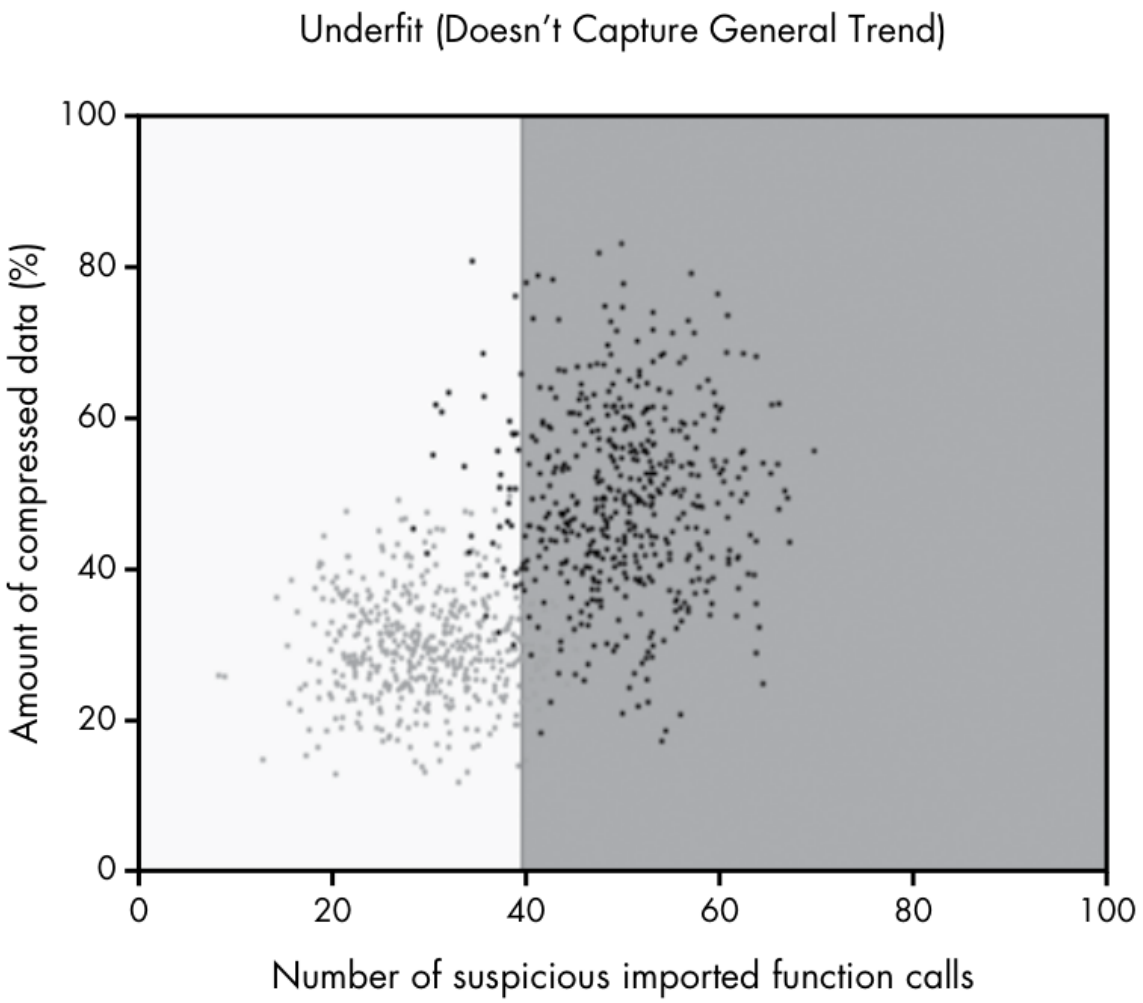

Рисунок 6-7. Недообученная модель машинного обучения

Чёрные точки, то есть вредоносные программы, образуют кластер в правой верхней области, а серые, то есть безвредные, - в левой нижней. Однако модель просто разрезает множество точек посередине, грубо разделяя данные и не учитывая диагональную закономерность. Поэтому модель называется недообученной.

Обратите внимание: во всех областях графика модель выражает лишь два уровня уверенности - тёмно-серый или белый. Иными словами, она либо абсолютно уверена во вредоносности точек, либо абсолютно уверена в их безвредности. Неспособность правильно выражать уверенность - ещё одна причина считать модель недообученной.

Сравним её с хорошо обученной моделью на рис. 6-8.

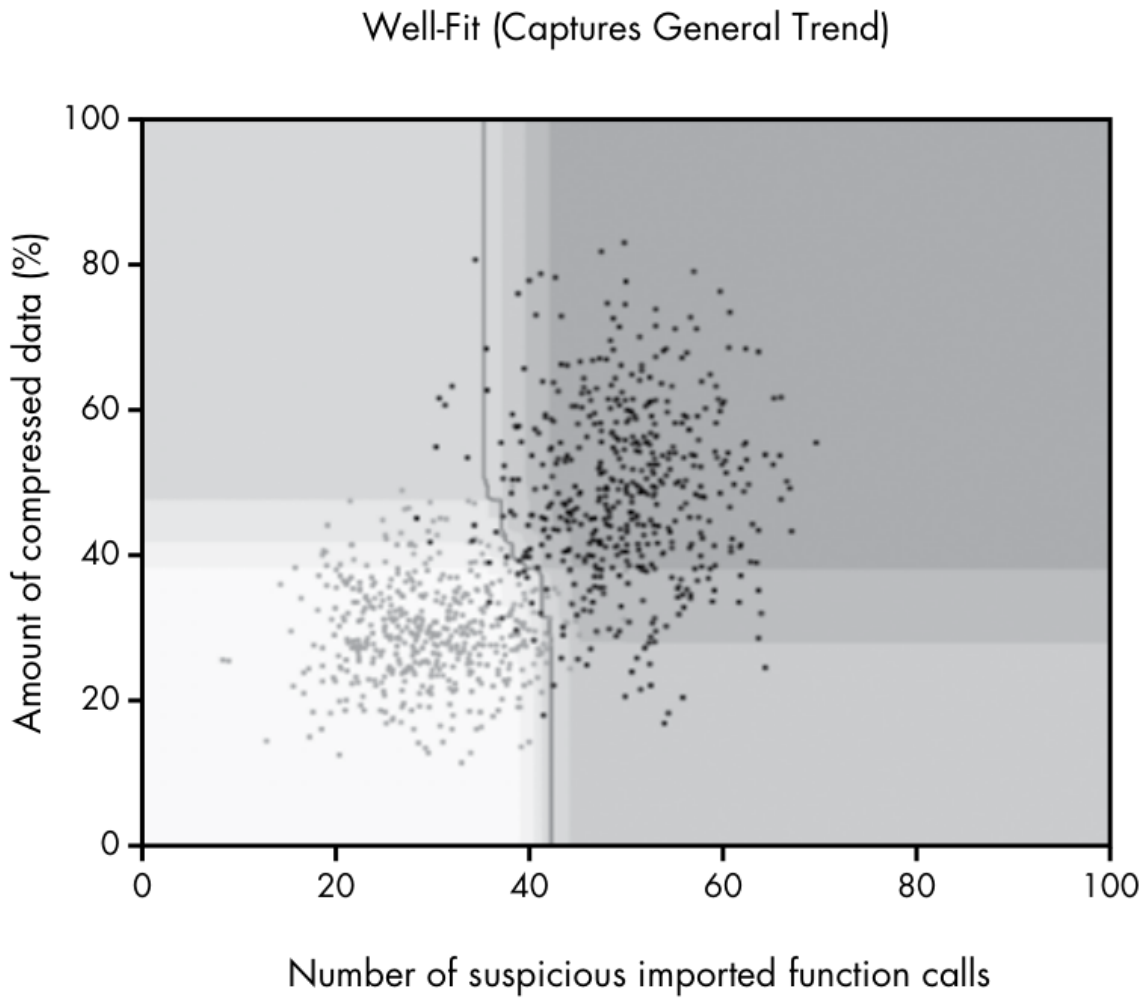

Рисунок 6-8. Хорошо обученная модель машинного обучения

Здесь модель не только улавливает общую закономерность данных, но и разумно выражает уверенность относительно областей пространства, которые определённо вредоносны, определённо безвредны или находятся в серой зоне.

Обратите внимание на линию решения сверху вниз. Модель придерживается простой теории разделения классов: вертикальная линия с диагональной выемкой посередине графика. Затенённые области показывают, что модель уверенно считает вредоносными лишь данные в правой верхней части, а безвредными - только двоичные файлы в левом нижнем углу.

Наконец, сравним переобученную модель на рис. 6-9 с недообученной на рис. 6-7 и хорошо обученной на рис. 6-8.

Переобученная модель не улавливает общую закономерность. Вместо этого она одержима исключениями: несколькими чёрными точками, то есть вредоносными обучающими примерами, внутри кластера серых безвредных точек и строит вокруг них границы. Точно так же она окружает немногочисленные безвредные примеры внутри вредоносного кластера.

В результате новый файл, признаки которого случайно окажутся рядом с такими выбросами, будет признан вредоносным, хотя почти наверняка безвреден, и наоборот. На практике эта модель окажется менее точной, чем могла бы быть.

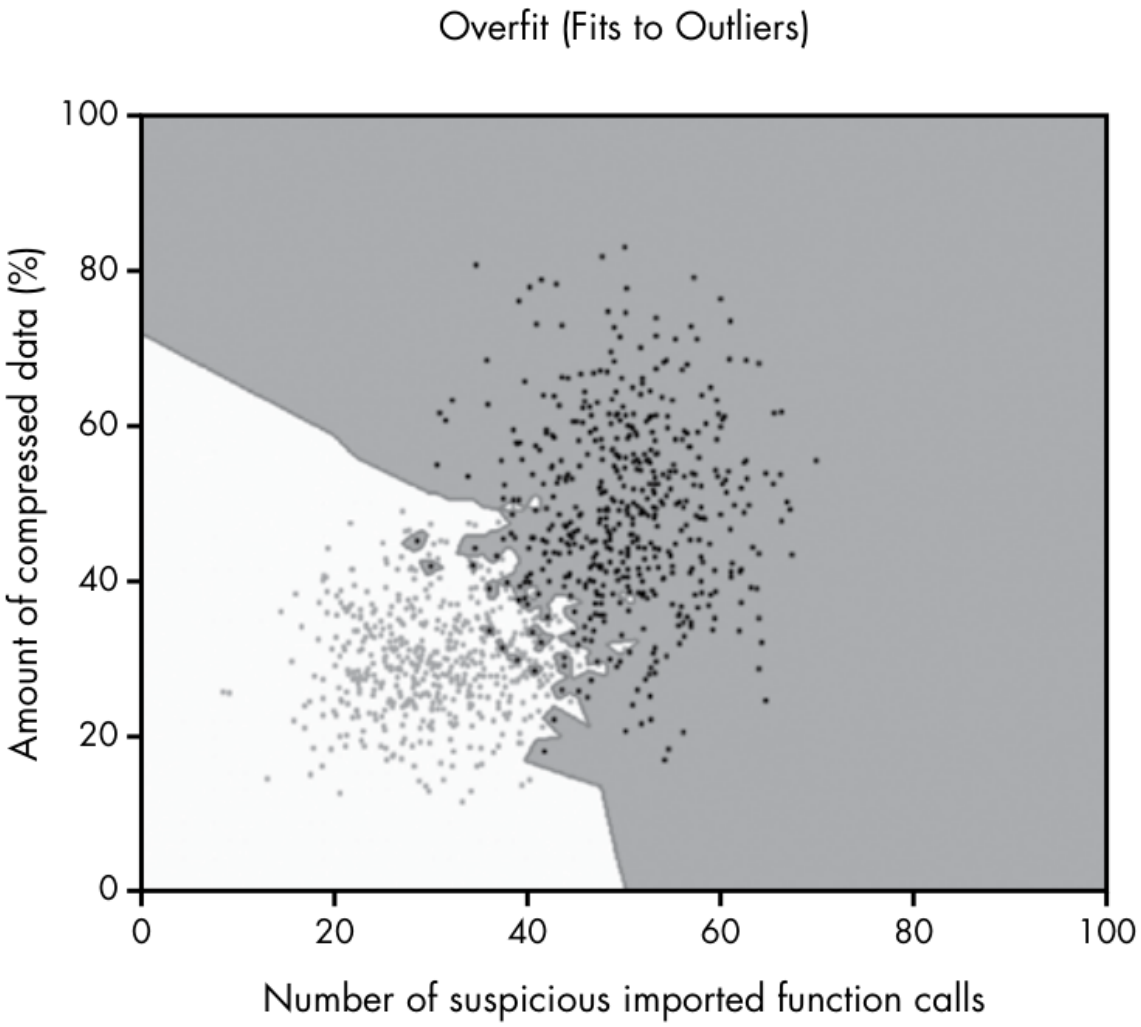

Рисунок 6-9. Переобученная модель машинного обучения

Основные типы алгоритмов машинного обучения

До сих пор машинное обучение обсуждалось очень общо, с упоминанием двух методов: логистической регрессии и k ближайших соседей. В оставшейся части главы мы подробнее рассмотрим логистическую регрессию, k ближайших соседей, деревья решений и случайный лес. Эти алгоритмы часто применяются в анализе данных безопасности. Они сложны, но лежащие в основе идеи понятны и наглядны.

Сначала рассмотрим учебные наборы на рис. 6-10, по которым будут изучаться достоинства и недостатки каждого алгоритма.

Я создал эти наборы специально для примера. Слева находится простой набор, уже использованный на рис. 6-7, 6-8 и 6-9. Чёрные обучающие примеры вредоносных программ здесь можно отделить от серых безвредных простой геометрической структурой вроде линии.

Набор справа, уже показанный на рис. 6-6, сложен: вредоносные и безвредные программы нельзя разделить простой линией. Тем не менее в данных есть ясная закономерность, просто для границы нужны более сложные методы. Посмотрим, как разные алгоритмы работают с обоими наборами.

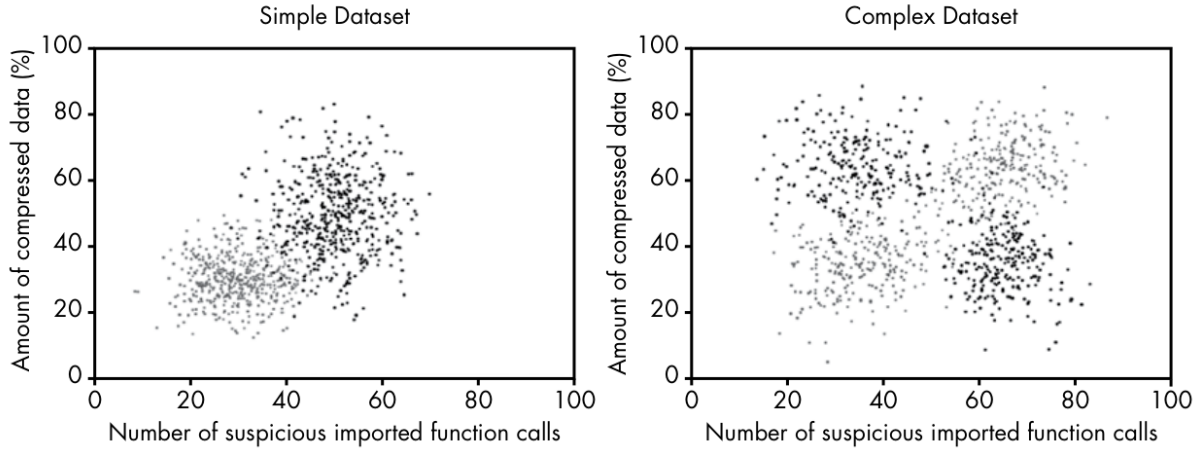

Рисунок 6-10. Два учебных набора из этой главы: чёрные точки обозначают вредоносные программы, серые - безвредные

Логистическая регрессия

Как вы уже узнали, логистическая регрессия создаёт линию, плоскость или гиперплоскость в зависимости от количества признаков, которая геометрически разделяет вредоносные и безвредные обучающие программы. При проверке нового файла обученная модель определяет, с какой стороны границы он находится, и классифицирует его.

Ограничение логистической регрессии состоит в том, что она не подходит, если данные невозможно просто разделить линией или гиперплоскостью. Возможность её применения зависит от данных и признаков. Например, если в задаче много отдельных признаков, каждый из которых сам по себе служит сильным указателем вредоносности или безвредности, логистическая регрессия может оказаться превосходным подходом. Если же для решения нужны сложные отношения между признаками, разумнее использовать k ближайших соседей, деревья решений или случайный лес.

Чтобы показать достоинства и недостатки логистической регрессии, рассмотрим её работу на двух наборах на рис. 6-11. В простом наборе слева алгоритм очень эффективно разделяет вредоносные и безвредные программы. В сложном наборе справа результат неэффективен: алгоритм сбит с толку, поскольку способен выразить только линейную границу. По обе стороны линии видны двоичные файлы обоих типов, а серые полосы уверенности не соответствуют данным. Для такого набора нужен алгоритм, способный выражать более сложные геометрические структуры.

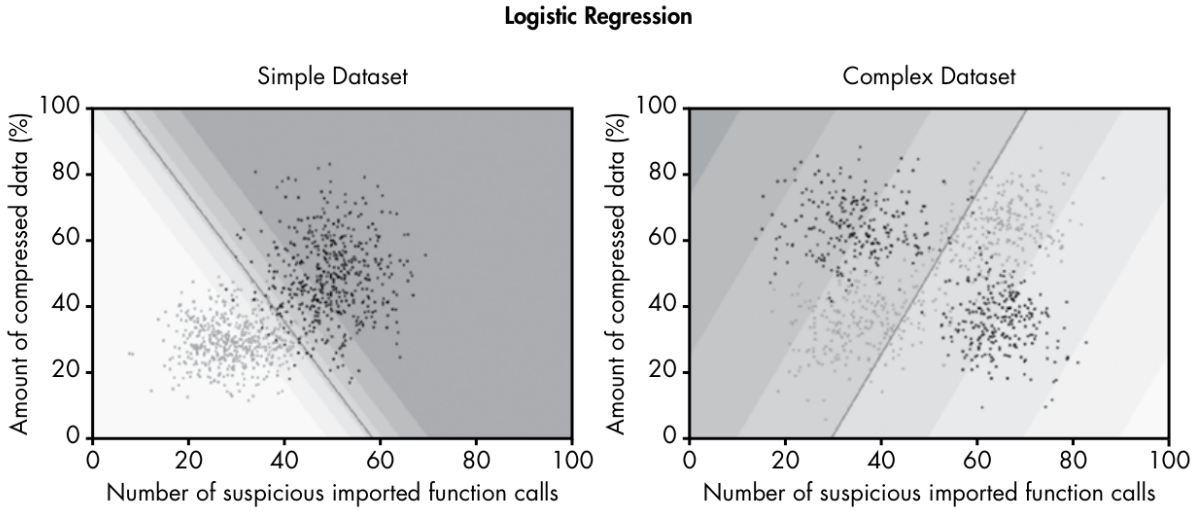

Рисунок 6-11. Граница решения, проведённая через учебные наборы с помощью логистической регрессии

Математические основы логистической регрессии

Теперь рассмотрим математику, с помощью которой логистическая регрессия обнаруживает вредоносные программы. В листинге 6-1 приведён псевдокод Python, вычисляющий вероятность вредоносности двоичного файла.

```
def logistic_regression(compressed_data, suspicious_calls, learned_parameters):  # (1)
    compressed_data = compressed_data * learned_parameters["compressed_data_weight"]  # (
    2)
    suspicious_calls = suspicious_calls * learned_parameters["suspicious_calls_weight"]
    score = compressed_data + suspicious_calls + bias  # (3)
    return logistic_function(score)

def logistic_function(score):  # (4)
    return 1/(1.0+math.e**(-score))
```

Листинг 6-1. Псевдокод вычисления вероятности с помощью логистической регрессии

Разберём код. Сначала определяется функция `logistic_regression` (1) и её параметры. Параметры `compressed_data` и `suspicious_calls` - признаки двоичного файла, представляющие соответственно объём сжатых данных и количество подозрительных вызовов. `learned_parameters` представляет элементы логистической функции, изученные при обучении модели. Способ их изучения будет рассмотрен далее; пока достаточно знать, что они получены из обучающих данных.

Затем признак `compressed_data` (2) умножается на параметр `compressed_data_weight`. Вес увеличивает или уменьшает масштаб признака в зависимости от того, насколько сильно логистическая функция считает его указателем вредоносности. Вес может быть отрицательным, что означает: модель считает признак указателем безвредности.

В следующей строке тот же шаг выполняется для параметра `suspicious_calls`. Затем два взвешенных признака складываются (3), и к ним прибавляется параметр, называемый смещением, который также изучается по обучающим данным. Итак, мы берём признак `compressed_data`, масштабированный в соответствии с тем, насколько сильным указателем вредоносности мы его считаем, прибавляем признак `suspicious_calls`, масштабированный таким же образом, а затем прибавляем смещение, отражающее общую степень подозрительности файлов с точки зрения модели логистической регрессии. Результат этих сложений и умножений представляет собой оценку вероятности вредоносности данного файла.

Наконец, функция `logistic_function` (4) преобразует оценку подозрительности в вероятность. На рис. 6-12 показано, как работает эта функция.

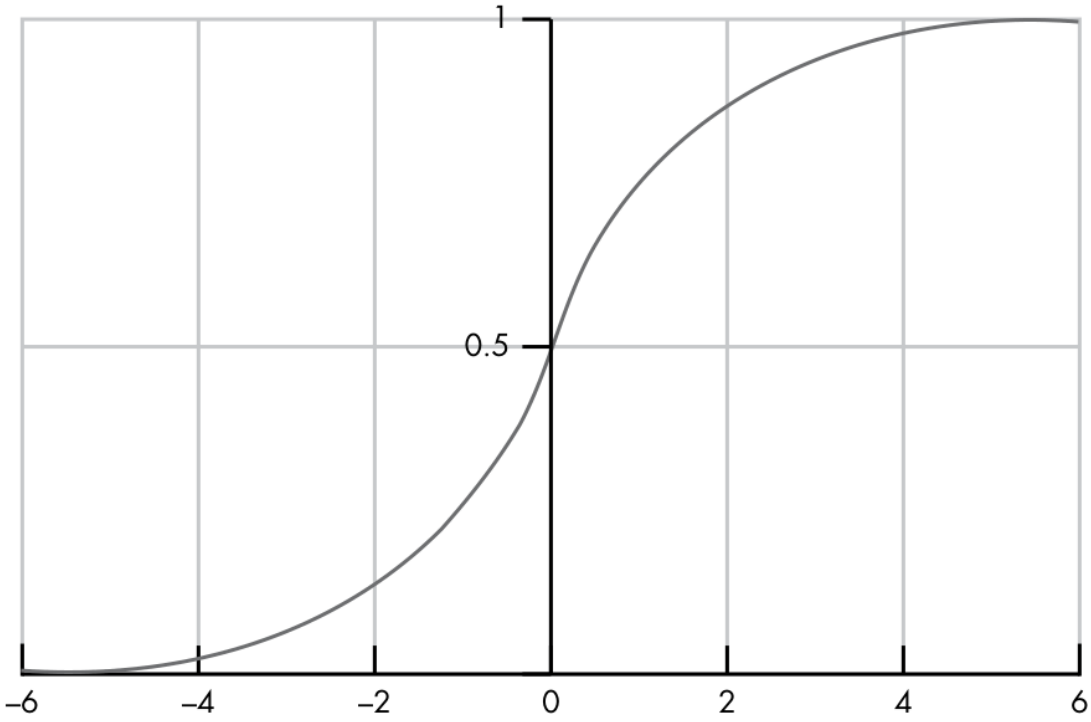

Рисунок 6-12. График логистической функции, используемой в логистической регрессии

Здесь логистическая функция принимает оценку, показанную по оси x, и преобразует её в значение, ограниченное диапазоном от 0 до 1, то есть в вероятность.

Как работает математика

Вернёмся к границам решения на рис. 6-11 и посмотрим, как эта математика действует на практике. Напомним, что вероятность вычисляется так:

```
logistic_function(feature1_weight * feature1 + feature2_weight*feature2 + bias)
```

Если, например, построить значения вероятности в каждой точке пространств признаков на рис. 6-11 с теми же весами признаков и параметром смещения, получатся показанные на рисунке затенённые области. Они отображают, где, по мнению модели, находятся вредоносные и безвредные образцы и насколько модель в этом уверена.

Если затем установить порог 0,5 - напомним, при вероятности выше 50 процентов файлы считаются вредоносными, - линия на рис. 6-11 станет границей решения. Советую поэкспериментировать с моим примером кода: подставить разные веса признаков и смещение и проверить результат самостоятельно.

> **Примечание.** Логистическая регрессия не ограничивает нас двумя признаками. На практике с ней обычно используют десятки, сотни и даже тысячи признаков. Математика при этом не меняется: для любого количества признаков вероятность вычисляется так:

```
logistic_function(feature1 * feature1_weight + feature2 * feature2_weight +
feature3 * feature3_weight ... + bias)
```

Как именно логистическая регрессия учится размещать границу решения в правильном месте на основании обучающих данных? Она использует итеративный подход на основе математического анализа, называемый градиентным спуском. Подробности этого подхода выходят за рамки книги, но основная идея такова: линия, плоскость или гиперплоскость, в зависимости от количества признаков, итеративно корректируется так, чтобы максимизировать вероятность правильного ответа модели на вопрос, является ли точка обучающего набора вредоносной или безвредной программой.

При обучении можно склонить алгоритм логистической регрессии к более простым или более сложным теориям о том, что представляет собой вредоносная и безвредная программа. Эти методы обучения выходят за рамки книги, но, если они вас заинтересовали, советую поискать в интернете материалы по запросу "logistic regression and regularization".

Когда применять логистическую регрессию

У логистической регрессии есть явные достоинства и недостатки по сравнению с другими алгоритмами машинного обучения. Одно из достоинств состоит в том, что представления модели о вредоносных и безвредных программах легко интерпретировать. Например, модель можно понять, изучив веса её признаков. Признаки с большим положительным весом модель воспринимает как вредоносные, а признаки с отрицательным весом - как безвредные. Логистическая регрессия сравнительно проста и хорошо работает, если данные содержат ясные указатели вредоносности. Однако на более сложных данных она часто терпит неудачу.

Теперь рассмотрим другой простой метод машинного обучения, способный выражать гораздо более сложные границы решения: k ближайших соседей.

K ближайших соседей

Алгоритм k ближайших соседей основан на идее, что двоичный файл, расположенный в пространстве признаков близко к другим вредоносным файлам, является вредоносным, а файл, признаки которого помещают его рядом с безвредными файлами, является безвредным. Точнее, если большинство из k ближайших к неизвестному двоичному файлу соседей вредоносны, то и сам файл считается вредоносным. Здесь k означает количество ближайших соседей, которое мы выбираем самостоятельно исходя из того, сколько соседей, по нашему мнению, должно участвовать в определении безвредности или вредоносности образца.

В реальном мире такая идея интуитивно понятна. Например, в наборе данных о массе и росте баскетболистов и игроков в настольный теннис значения баскетболистов, скорее всего, будут ближе друг к другу, чем к значениям игроков в настольный теннис. Точно так же в сфере безопасности признаки вредоносных программ часто похожи на признаки других вредоносных программ, а признаки безвредных - на признаки других безвредных программ.

Эту идею можно превратить в алгоритм k ближайших соседей, который определяет вредоносность двоичного файла следующим образом:

1. Извлечь признаки двоичного файла и найти k образцов, ближайших к нему в пространстве признаков.
2. Разделить количество ближайших вредоносных образцов на k и получить долю вредоносных программ среди ближайших соседей.
3. Если вредоносных образцов достаточно много, признать образец вредоносным.

На рис. 6-13 показан общий принцип работы алгоритма k ближайших соседей.

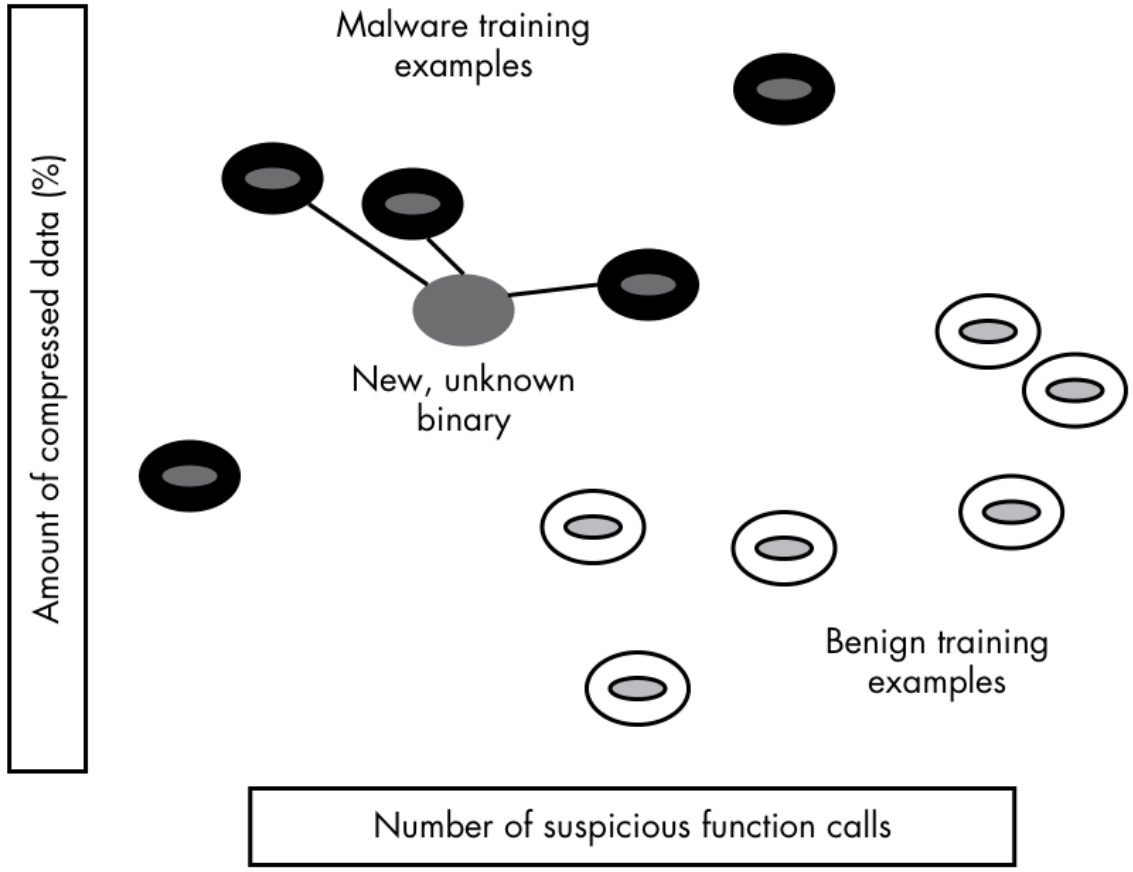

Рисунок 6-13. Применение k ближайших соседей для обнаружения ранее неизвестной вредоносной программы

В левой верхней части показана группа обучающих примеров вредоносных программ, а в правой нижней - группа примеров безвредных программ. Новый неизвестный двоичный файл соединён с тремя ближайшими соседями. В данном случае k равно 3, поэтому рассматриваются три ближайших соседа неизвестного файла. Поскольку все трое вредоносны, новый файл также классифицируется как вредоносный.

Математические основы k ближайших соседей

Теперь обсудим математику, позволяющую вычислить расстояние между признаками нового неизвестного файла и образцами обучающего набора. Для этого используется функция расстояния, которая сообщает расстояние между новым примером и примерами обучающего набора. Наиболее распространено евклидово расстояние - длина кратчайшего пути между двумя точками пространства признаков. В листинге 6-2 приведён псевдокод вычисления евклидова расстояния в нашем двумерном пространстве признаков.

```
import math
def euclidean_distance(compression1,suspicious_calls1, compression2, suspicious_calls2):
    # (1)
    comp_distance = (compression1-compression2)**2  # (2)
    call_distance = (suspicious_calls1-suspicious_calls2)**2  # (3)
    return math.sqrt(comp_distance + call_distance)  # (4)
```

Листинг 6-2. Псевдокод функции `euclidean_distance`

Разберём математику этого кода. Листинг 6-2 принимает пару образцов и вычисляет расстояние между ними по различиям признаков. Сначала вызывающая сторона передаёт признаки двоичных файлов (1): `compression1` - признак сжатия первого примера, `suspicious_calls1` - признак подозрительных вызовов первого примера, `compression2` - признак сжатия второго примера, а `suspicious_calls2` - признак подозрительных вызовов второго примера.

Затем вычисляется квадрат разности признаков сжатия двух образцов (2) и квадрат разности признаков подозрительных вызовов (3). Причина использования квадрата расстояния здесь не рассматривается, но обратите внимание: полученная разность всегда положительна. Наконец, вычисляется квадратный корень из суммы двух разностей - евклидово расстояние между двумя векторами признаков - и возвращается вызывающей стороне (4). Существуют и другие способы вычислять расстояния между примерами, но с алгоритмом k ближайших соседей чаще всего применяется евклидово расстояние, которое хорошо подходит для задач анализа данных безопасности.

Выбор количества голосующих соседей

Теперь посмотрим, какие границы решения и вероятности создаёт алгоритм k ближайших соседей для используемых в главе наборов данных. На рис. 6-14 значение k установлено равным 5, поэтому "голосуют" пять ближайших соседей.

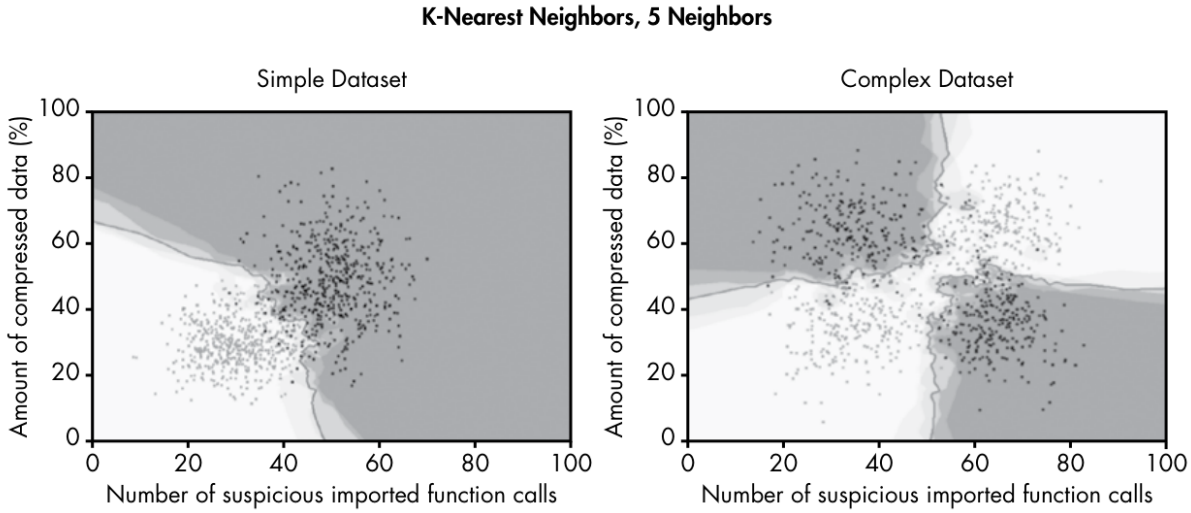

Рисунок 6-14. Границы решения, созданные методом k ближайших соседей при k, равном 5

На рис. 6-15 значение k установлено равным 50, поэтому "голосуют" 50 ближайших соседей.

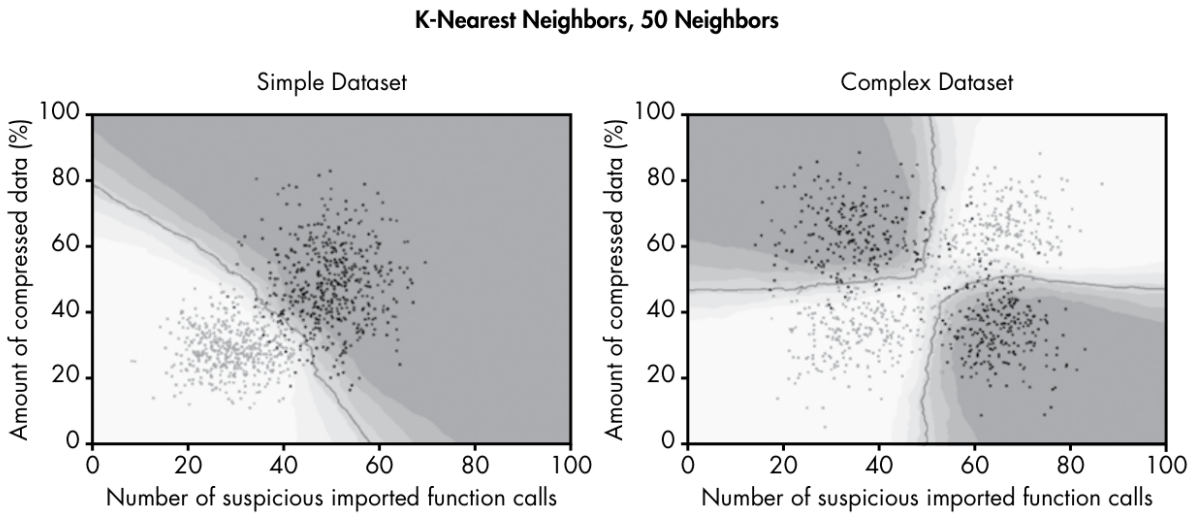

Рисунок 6-15. Границы решения, созданные методом k ближайших соседей при k, равном 50

Обратите внимание, насколько резко модели различаются в зависимости от количества голосующих соседей. Модель на рис. 6-14 проводит для обоих наборов неровную, сложную границу решения. Она переобучена, поскольку очерчивает локальные границы вокруг выбросов, и одновременно недообучена, поскольку не улавливает простые общие закономерности. В отличие от неё, модель на рис. 6-15 хорошо соответствует обоим наборам: она не отвлекается на выбросы и ясно выявляет общие закономерности.

Как видно, k ближайших соседей способен создавать гораздо более сложную границу решения, чем логистическая регрессия. Изменяя k, то есть количество соседей, голосующих за вредоносность или безвредность образца, можно управлять сложностью границы и защищаться как от переобучения, так и от недообучения. Если модель логистической регрессии на рис. 6-11 полностью ошиблась, то k ближайших соседей хорошо разделяет вредоносные и безвредные программы, особенно при 50 голосующих соседях. Алгоритм не ограничен линейной структурой и принимает решение, просто рассматривая ближайших соседей каждой точки. Поэтому он может создавать границы решения произвольной формы и гораздо эффективнее моделировать сложные наборы данных.

Когда применять k ближайших соседей

Алгоритм k ближайших соседей полезен, если отдельные признаки данных не отображаются прямо на понятие подозрительности, но близость к вредоносным образцам служит сильным указателем вредоносности. Например, при классификации вредоносных программ по семействам с общим кодом этот алгоритм стоит попробовать, поскольку образец следует отнести к семейству, если его признаки похожи на признаки известных представителей этого семейства.

Другая причина использовать k ближайших соседей состоит в том, что алгоритм ясно объясняет свои решения. Чтобы понять, почему неизвестный образец был признан вредоносным или безвредным, легко найти и сравнить его сходства с ближайшими образцами.

Деревья решений

Деревья решений - ещё один часто применяемый метод машинного обучения для задач обнаружения. В процессе обучения дерево решений автоматически создаёт последовательность вопросов и по ответам определяет, является ли двоичный файл вредоносным, подобно игре "Двадцать вопросов". На рис. 6-16 показано дерево решений, автоматически обученное на простом наборе данных из этой главы. Проследим ход его логики.

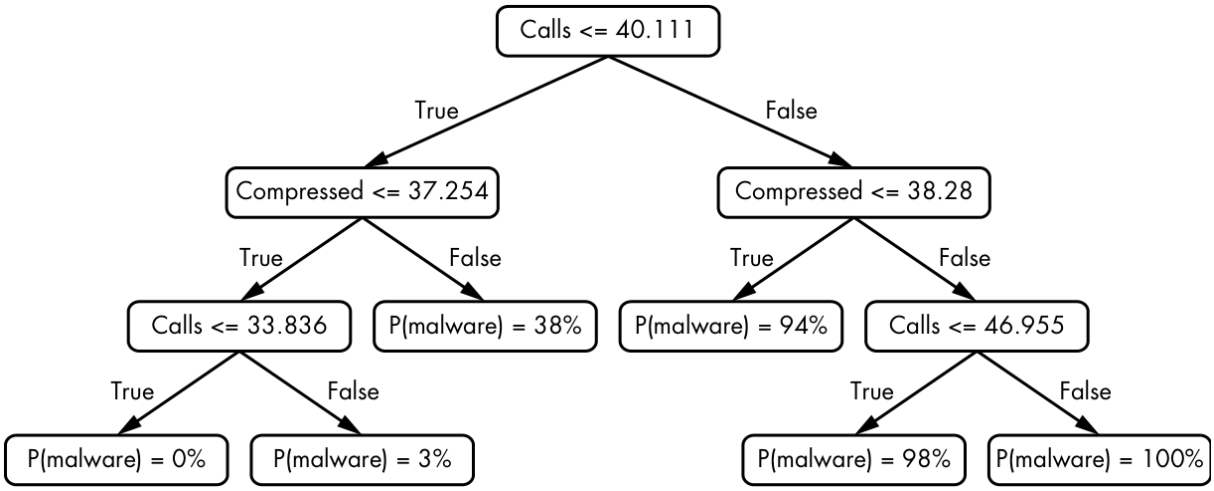

Рисунок 6-16. Дерево решений, изученное на примере простого набора данных

Работа начинается с передачи в дерево признаков, извлечённых из нового, ранее неизвестного двоичного файла. Затем дерево задаёт последовательность вопросов об этих признаках. Верхний прямоугольник дерева, называемый узлом, содержит первый вопрос: не превышает ли количество подозрительных вызовов 40,111? Здесь используется число с плавающей точкой, потому что количество подозрительных вызовов каждого файла нормализовано в диапазоне от 0 до 100.

Если ответ "да", задаётся следующий вопрос: не превышает ли доля сжатых данных в файле 37,254? При ответе "да" следует ещё один вопрос: не превышает ли количество подозрительных вызовов двоичного файла 33,836? Если и здесь ответ "да", достигается конец дерева решений. В этой точке вероятность вредоносности файла равна 0 процентов.

На рис. 6-17 показана геометрическая интерпретация этого дерева решений.

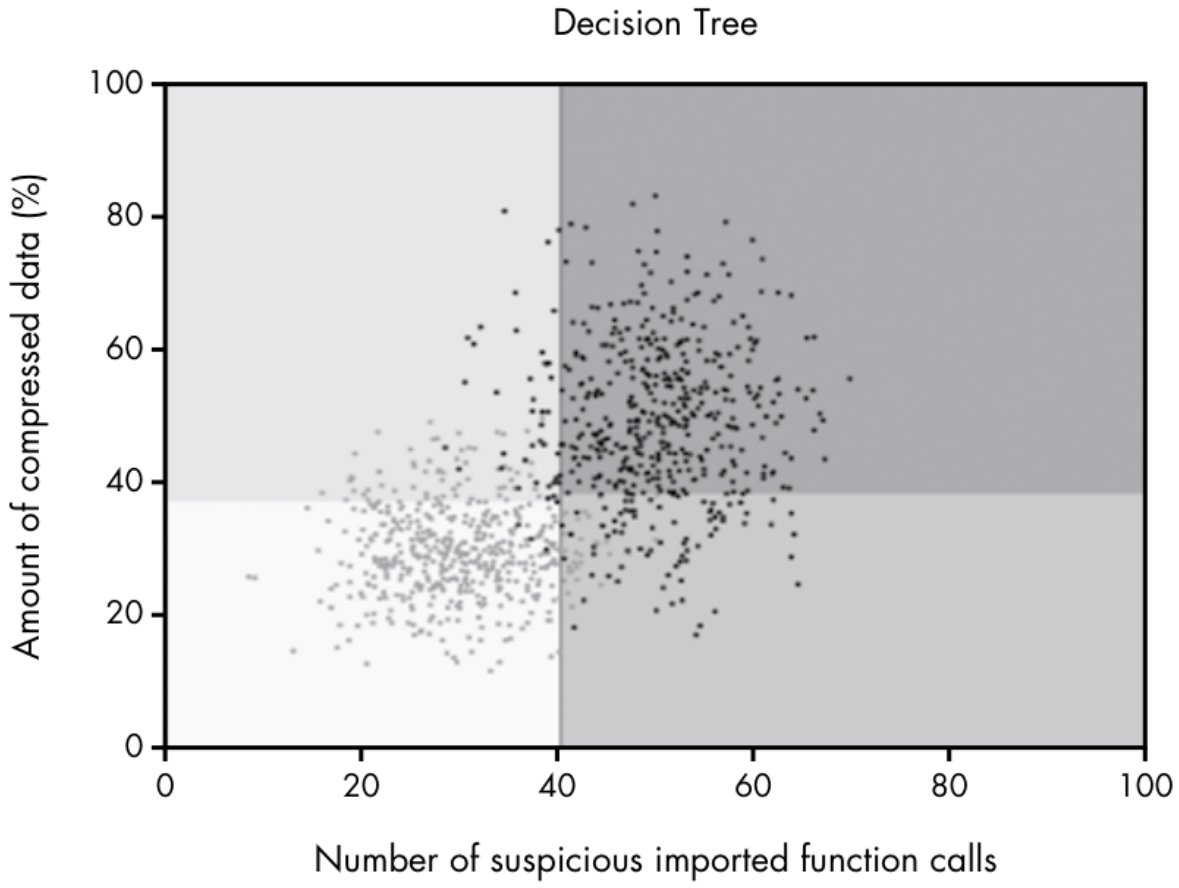

Рисунок 6-17. Граница решения, созданная деревом решений для примера простого набора данных

Затенённые области показывают, где дерево считает образцы вредоносными. Более светлые области соответствуют безвредным образцам. Вероятности, назначенные последовательностью вопросов и ответов на рис. 6-16, должны соответствовать вероятностям в затенённых областях рис. 6-17.

Выбор хорошего корневого узла

Как с помощью алгоритма машинного обучения создать такое дерево из обучающих данных? Основная идея состоит в том, что дерево начинается с исходного вопроса, называемого корневым узлом. Лучший корневой узел - тот, для которого большинство или все образцы одного типа дают ответ "да", а большинство или все образцы другого типа - ответ "нет". Например, корневой узел на рис. 6-16 спрашивает, имеет ли ранее неизвестный двоичный файл не более 40,111 вызова. Напомним, количество вызовов нормализовано в диапазоне от 0 до 100, поэтому допустимы значения с плавающей точкой. Как показывает вертикальная линия на рис. 6-17, у большинства безвредных данных подозрительных вызовов меньше этого количества, а у большинства вредоносных - больше, поэтому такой первый вопрос выбран удачно.

Выбор последующих вопросов

После корневого узла следующие вопросы выбираются методом, похожим на выбор корневого. Корневой вопрос разделил образцы на две группы: с количеством подозрительных вызовов не более 40,111, то есть отрицательная область пространства признаков, и с количеством больше 40,111, то есть положительная область. Теперь нужны вопросы, которые дальше разделят образцы в каждой области на вредоносные и безвредные обучающие примеры.

Это видно по структуре дерева на рис. 6-16 и 6-17. После исходного корневого вопроса о количестве подозрительных вызовов двоичных файлов задаются вопросы об объёме сжатых данных. Рисунок 6-17 объясняет причину: первый вопрос о подозрительных вызовах создаёт грубую границу, отделяющую большинство вредоносных программ от большинства безвредных. Чтобы уточнить границу последующими вопросами, нагляднее всего спросить об объёме сжатых данных в файлах.

Когда перестать задавать вопросы

На некотором этапе создания дерева необходимо решить, когда прекратить задавать вопросы и просто определить безвредность или вредоносность файла исходя из уверенности в ответе. Можно ограничить количество вопросов или глубину дерева, то есть максимальное количество вопросов для любого файла. Другой вариант - позволить дереву расти, пока его структура не даст абсолютную уверенность в типе каждого примера обучающего набора.

Преимущество ограничения размера состоит в том, что простое дерево с большей вероятностью даст верный ответ: вспомните бритву Оккама, согласно которой более простая теория лучше. Иными словами, небольшое дерево с меньшей вероятностью переобучится на обучающих данных.

С другой стороны, максимальное дерево полезно при недообучении. Дальнейший рост повышает сложность границы решения, что и требуется, если модель недообучена. Обычно специалисты по машинному обучению пробуют несколько значений глубины либо допускают максимальную глубину и проверяют результат на ранее неизвестных двоичных файлах, повторяя процесс, пока не добьются наибольшей точности.

Исследование алгоритма создания дерева решений с помощью псевдокода

Теперь рассмотрим автоматизированный алгоритм создания дерева решений. Как вы узнали, его основная идея - создать корневой узел, найдя вопрос, который сильнее всего повышает нашу уверенность в том, вредоносны или безвредны обучающие примеры, а затем найти последующие вопросы, ещё больше повышающие уверенность. Алгоритм должен прекратить задавать вопросы и принять решение, когда уверенность превысит заранее установленный порог.

Программно это можно реализовать рекурсивно. Python-подобный псевдокод в листинге 6-3 в упрощённом виде показывает весь процесс построения дерева.

```
tree = Tree()
def add_question(training_examples):
    question = pick_best_question(training_examples)  # (1)
    uncertainty_yes,yes_samples=ask_question(question,training_examples,"yes")  # (2)
    uncertainty_no,no_samples=ask_question(question,training_examples,"no")  # (3)
    if not uncertainty_yes < MIN_UNCERTAINTY:  # (4)
        add_question(yes_samples)
    if not uncertainty_no < MIN_UNCERTAINTY:  # (5)
        add_question(no_samples)
add_question(training_examples)  # (6)
```

Листинг 6-3. Псевдокод алгоритма построения дерева решений

Псевдокод рекурсивно добавляет вопросы к дереву решений, начиная с корневого узла и двигаясь вниз, пока алгоритм не решит, что дерево способно с высокой уверенностью определить, безвреден или вредоносен новый файл.

В начале построения дерева функция `pick_best_question()` выбирает корневой узел (1); пока не будем разбирать, как она работает. Затем оценивается неопределённость относительно обучающих образцов, для которых ответ на первый вопрос равен "да" (2). Это помогает решить, нужно ли задавать о них дальнейшие вопросы или уже можно предсказать их вредоносность либо безвредность. То же выполняется для образцов, давших на первый вопрос ответ "нет" (3).

Далее проверяется, достаточно ли мала неопределённость `uncertainty_yes` относительно образцов с ответом "да", чтобы определить их вредоносность или безвредность (4). Если да, дополнительные вопросы не задаются. Если нет, функция `add_question()` вызывается снова с `yes_samples`, то есть с образцами, ответившими "да". Это классический пример рекурсии, при которой функция вызывает саму себя. С её помощью тот же процесс, который выполнялся для корневого узла, повторяется с подмножеством обучающих примеров. Следующая инструкция `if` делает то же для примеров с ответом "нет" (5). Наконец, функция построения дерева вызывается с нашими обучающими примерами (6).

Точная работа `pick_best_question()` связана с математикой, выходящей за рамки книги, но идея проста. В любой точке построения дерева рассматриваются обучающие примеры, относительно которых ещё остаётся неопределённость, перечисляются все возможные вопросы и выбирается тот, который лучше всего уменьшает неопределённость в отношении вредоносности или безвредности примеров. Это уменьшение измеряется статистической величиной, называемой информационным выигрышем. Такой простой метод выбора лучшего вопроса работает удивительно хорошо.

> **Примечание.** Это упрощённый пример работы реальных алгоритмов машинного обучения, создающих деревья решений. Здесь опущена математика, необходимая для вычисления того, насколько конкретный вопрос повышает уверенность в том, что файл вредоносен.

Теперь посмотрим, как деревья решений ведут себя на двух учебных наборах этой главы. На рис. 6-18 показаны границы, изученные детектором на основе дерева решений.

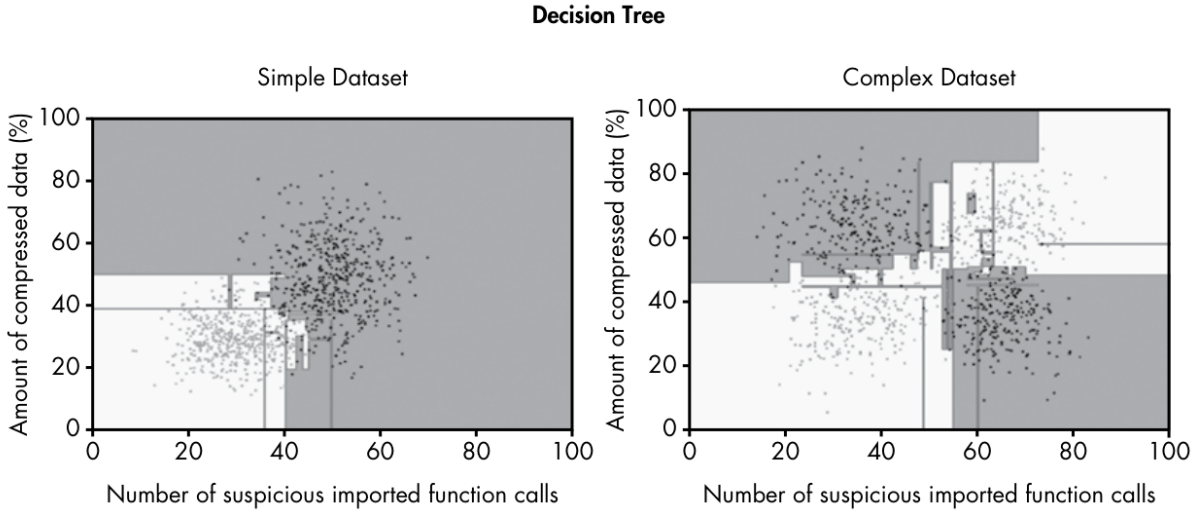

Рисунок 6-18. Границы решения для учебных наборов, созданные методом дерева решений

Здесь вместо установки максимальной глубины деревьям позволено расти до тех пор, пока относительно обучающих данных не останется ни ложноположительных, ни ложноотрицательных

результатов, то есть пока каждый обучающий образец не будет классифицирован правильно.

Обратите внимание: деревья решений способны проводить в пространстве признаков только горизонтальные и вертикальные линии, даже если из данных явно следует, что уместнее была бы кривая или диагональ. Причина в том, что деревья решений позволяют выражать лишь простые условия для отдельных признаков, например "больше или равно" и "меньше или равно", а такие условия всегда дают горизонтальные или вертикальные линии.

Также видно, что, хотя в этих примерах деревья успешно отделяют безвредные программы от вредоносных, границы получились крайне неровными и содержат странные артефакты. Например, область вредоносных программ необычным образом вторгается в область безвредных и наоборот. Положительная сторона состоит в том, что на сложном наборе дерево решений создаёт намного лучшую границу, чем логистическая регрессия.

Сравним деревья на рис. 6-18 с моделями на рис. 6-19.

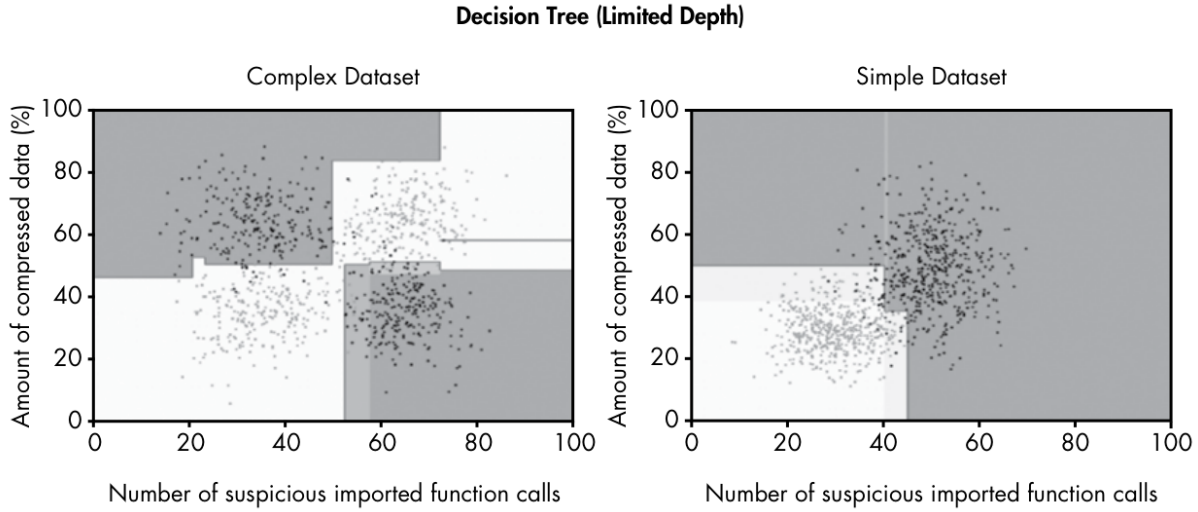

Рисунок 6-19. Границы решения для учебных наборов, созданные деревом решений ограниченной глубины

Деревья на рис. 6-19 созданы тем же алгоритмом, что и на рис. 6-18, но их глубина ограничена пятью узлами. Это означает, что о признаках любого двоичного файла можно задать не более пяти вопросов.

Результат разителен. Деревья на рис. 6-18 явно переобучены: они сосредоточены на выбросах и проводят чрезмерно сложные границы, не отражающие общую закономерность. Деревья на рис. 6-19 гораздо изящнее соответствуют данным и выявляют общую закономерность обоих наборов, не сосредотачиваясь на выбросах, за одним исключением: более узкой области решения в правом верхнем углу простого набора. Как видно, удачный выбор максимальной глубины способен сильно повлиять на детектор машинного обучения на основе дерева решений.

Когда применять деревья решений

Деревья решений выразительны и просты: с помощью вопросов, допускающих ответы "да" или "нет", они способны изучать как простые, так и крайне неровные границы. Максимальная глубина позволяет управлять простотой или сложностью их теорий о различиях вредоносных и безвредных программ.

К сожалению, недостаток деревьев решений состоит в том, что они нередко создают не слишком точные модели. Причина сложна, но связана с тем, что неровные границы решений плохо

соответствуют обучающим данным таким образом, который обобщался бы на ранее неизвестные примеры.

Кроме того, деревья обычно плохо изучают вероятности вблизи своих границ. Это видно по затенённым областям вокруг границы на рис. 6-19. Переход не выглядит естественным или постепенным и происходит не там, где должен, то есть не в областях перекрытия вредоносных и безвредных примеров.

Далее рассматривается случайный лес, объединяющий несколько деревьев решений и дающий гораздо лучшие результаты.

Случайный лес

Хотя сообщество специалистов по безопасности широко применяет деревья решений для обнаружения вредоносных программ, отдельные деревья почти никогда не используются. Вместо этого сотни или тысячи деревьев совместно выполняют обнаружение методом, называемым случайным лесом. Обучается не одно, а множество деревьев, обычно сто или больше, причём каждое дерево обучается по-своему и получает собственный взгляд на данные. Для определения вредоносности нового файла деревья голосуют. Вероятность того, что файл вредоносен, равна количеству положительных голосов, делённому на общее количество деревьев.

Разумеется, если все деревья одинаковы, они будут голосовать одинаково, и случайный лес всего лишь воспроизведёт результат отдельного дерева. Поэтому деревьям придают разные взгляды на природу вредоносных и безвредных программ двумя описанными ниже способами. Такое разнообразие создаёт в модели эффект "мудрости толпы", обычно повышающий её точность.

Алгоритм случайного леса создаётся следующим образом:

1. Обучение: для каждого дерева из запланированного количества, обычно 100 или больше:
• случайно выбрать часть обучающих примеров из обучающего набора;
• построить дерево решений по случайной выборке;
• при рассмотрении каждого возможного вопроса для строящегося дерева учитывать лишь небольшое случайно выбранное подмножество признаков, игнорируя остальные.
2. Обнаружение ранее неизвестного двоичного файла:
• выполнить обнаружение этого файла в каждом отдельном дереве;
• определить, является ли файл вредоносным, по количеству деревьев, проголосовавших "да".

Чтобы разобраться подробнее, рассмотрим результаты случайного леса для двух учебных наборов на рис. 6-20. Они получены с использованием 100 деревьев решений.

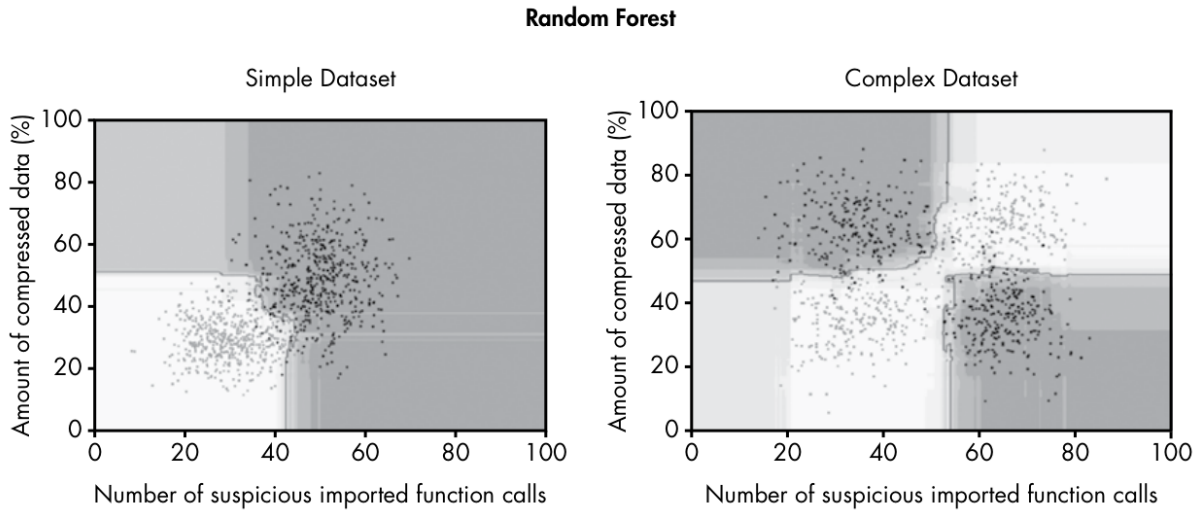

Рисунок 6-20. Границы решения, созданные методом случайного леса

В отличие от отдельных деревьев на рис. 6-18 и 6-19, случайный лес способен выражать для простого и сложного наборов гораздо более плавные и интуитивно понятные границы. В самом деле, модель очень аккуратно соответствует обучающим данным, без неровных краёв, и, по-видимому, изучила хорошие теории различия вредоносных и безвредных программ для обоих наборов.

Затенённые области также интуитивно понятны. Например, чем дальше точка находится от безвредных или вредоносных примеров, тем меньше случайный лес уверен в том, к какому классу она относится. Это хороший признак для работы случайного леса с ранее неизвестными файлами. Как станет видно в следующей главе, среди всех рассмотренных здесь методов именно случайный лес лучше всего работает на ранее неизвестных двоичных файлах.

Чтобы понять, почему случайный лес создаёт столь чистые границы по сравнению с отдельными деревьями, подумаем о работе 100 деревьев. Каждое видит лишь около двух третей обучающих данных и при выборе очередного вопроса рассматривает только случайно выбранный признак. Поэтому за кулисами существуют 100 разных границ решений, которые усредняются и образуют итоговые границы и затенённые области. Эффект "мудрости толпы" создаёт совокупное мнение, способное выявлять закономерности в данных гораздо сложнее и точнее, чем отдельные деревья.

Итоги

В этой главе вы получили общее представление об обнаружении вредоносных программ на основе машинного обучения, а также о четырёх основных подходах: логистической регрессии, k ближайших соседях, деревьях решений и случайных лесах. Системы обнаружения на основе машинного обучения способны автоматизировать написание сигнатур и на практике часто превосходят сигнатуры, написанные вручную.

В следующих главах показано, как эти подходы работают в реальных задачах обнаружения вредоносных программ. Вы научитесь с помощью открытого программного обеспечения для машинного обучения создавать детекторы, которые точно классифицируют файлы как вредоносные или безвредные, а также применять базовую статистику для оценки работы детекторов на ранее неизвестных двоичных файлах.

Глава 7. Оценка систем обнаружения вредоносных программ

В предыдущей главе вы узнали, как машинное обучение помогает создавать детекторы вредоносных программ. В этой главе рассматриваются основные понятия, необходимые для прогнозирования работы систем обнаружения вредоносных программ. Они играют решающую роль в совершенствовании любой созданной вами системы, потому что без способа измерять её работу вы не узнаете, как её улучшить. Обратите внимание: хотя эта глава посвящена введению в базовые понятия оценки, глава 8 продолжает эту тему и знакомит с такими важными понятиями, как перекрёстная проверка.

Сначала я изложу основные идеи оценки точности обнаружения, а затем более сложные идеи, связанные с учётом среды развёртывания системы при оценке её работы. Для этого мы вместе оценим гипотетическую систему обнаружения вредоносных программ.

Четыре возможных результата обнаружения

Предположим, вы запускаете систему обнаружения для программного двоичного файла и получаете её "мнение" о том, вредоносен файл или безвреден. Как показано на рис. 7-1, возможны четыре результата.

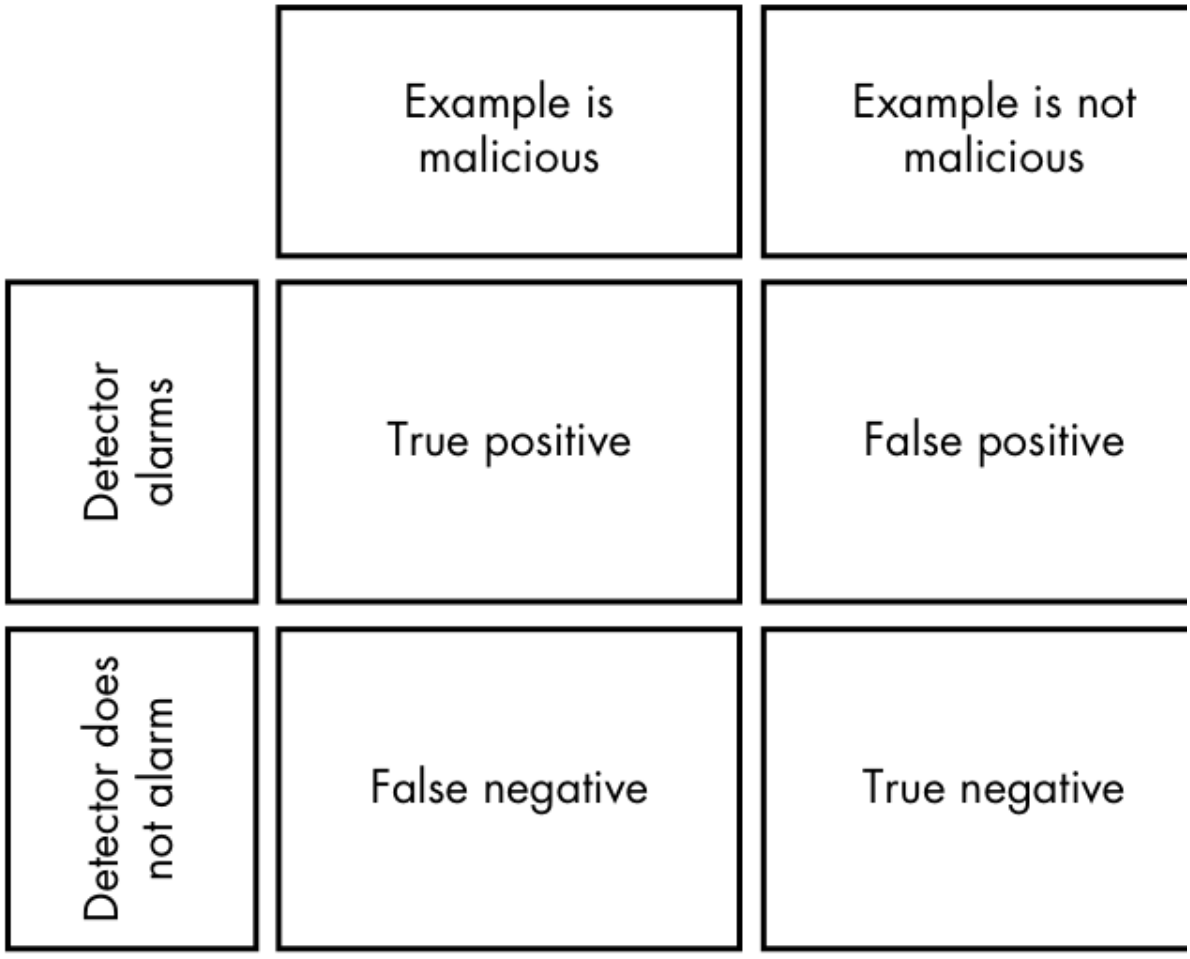

Рисунок 7-1. Четыре возможных результата обнаружения

Эти результаты определяются следующим образом:

• **Истинно положительный.** Двоичный файл является вредоносной программой, и система сообщает, что это вредоносная программа.
• **Ложноотрицательный.** Двоичный файл является вредоносной программой, а система сообщает, что это не вредоносная программа.
• **Ложноположительный.** Двоичный файл не является вредоносной программой, а система сообщает, что это вредоносная программа.
• **Истинно отрицательный.** Двоичный файл не является вредоносной программой, и система сообщает, что это не вредоносная программа.

Как видите, система обнаружения может выдать неточный результат в двух случаях: ложноотрицательном и ложноположительном. На практике желательны истинно положительные и истинно отрицательные результаты, однако добиться их часто трудно.

Эти термины будут использоваться на протяжении всей главы. Фактически большая часть теории оценки обнаружения построена на этом простом словаре.

Доли истинно положительных и ложноположительных результатов

Теперь предположим, что вы хотите проверить точность системы обнаружения на наборе безвредных и вредоносных программ. Можно запустить детектор для каждого двоичного файла и подсчитать, сколько раз на всём проверочном наборе получен каждый из четырёх возможных результатов. Затем потребуются сводные статистические показатели, дающие общее представление о точности системы, то есть о вероятности ложноположительных или ложноотрицательных результатов.

Один из таких показателей - доля истинно положительных результатов системы обнаружения. Она вычисляется делением количества истинно положительных результатов в проверочном наборе на общее количество образцов вредоносных программ в этом наборе. Этот показатель вычисляет процент образцов, которые система смогла обнаружить, поэтому измеряет её способность распознавать вредоносную программу, когда она её "видит".

Однако одного знания о том, что система поднимает тревогу при встрече с вредоносной программой, недостаточно для оценки её точности. Например, если единственным критерием была бы доля истинно положительных результатов, простая функция, отвечающая "да, это вредоносная программа" для каждого файла, получила бы идеальный показатель. Настоящая проверка системы состоит в том, говорит ли она "да, это вредоносная программа" при встрече с вредоносной программой и "нет, это не вредоносная программа" при встрече с безвредной.

Чтобы измерить способность системы распознавать то, что *не является* вредоносной программой, нужно также определить долю ложноположительных результатов, то есть частоту, с которой система поднимает тревогу при встрече с безвредной программой. Она вычисляется делением количества безвредных образцов, которые система отметила как вредоносные, на общее количество проверенных безвредных образцов.

Связь долей истинно положительных и ложноположительных результатов

При проектировании системы обнаружения желательно свести долю ложноположительных результатов к минимуму, а долю истинно положительных - довести до максимума. Если только вам не удастся создать действительно безошибочную систему обнаружения, что фактически невозможно из-за непрерывной эволюции вредоносных программ, между стремлением к высокой доле истинно положительных и низкой доле ложноположительных результатов всегда будет существовать противоречие.

Чтобы понять причину, представьте систему обнаружения, которая перед решением о вредоносности файла суммирует все свидетельства его вредоносности и создаёт оценку подозрительности. Назовём эту гипотетическую систему MalDetect. На рис. 7-2 приведён пример значений, которые MalDetect могла бы вывести для 12 двоичных файлов, обозначенных кружками. Чем правее находится файл, тем выше назначенная ему MalDetect оценка подозрительности.

Рисунок 7-2. Оценки подозрительности отдельных двоичных файлов, выданные гипотетической системой MalDetect

Оценки подозрительности полезны, однако для вычисления долей истинно положительных и ложноположительных результатов MalDetect на наших файлах их необходимо преобразовать в ответы "да" или "нет" на вопрос о вредоносности данного двоичного файла. Для этого используется пороговое правило. Например, можно решить, что файл поднимает тревогу, если оценка подозрительности больше или равна некоторому числу. Если оценка ниже порога, тревога не поднимается.

Пороговое правило - стандартный способ преобразовать оценку подозрительности в двоичное решение, но где установить порог? Правильного ответа нет. На рис. 7-3 показано затруднение: чем выше порог, тем меньше вероятность ложноположительного результата, но тем больше вероятность ложноотрицательного.

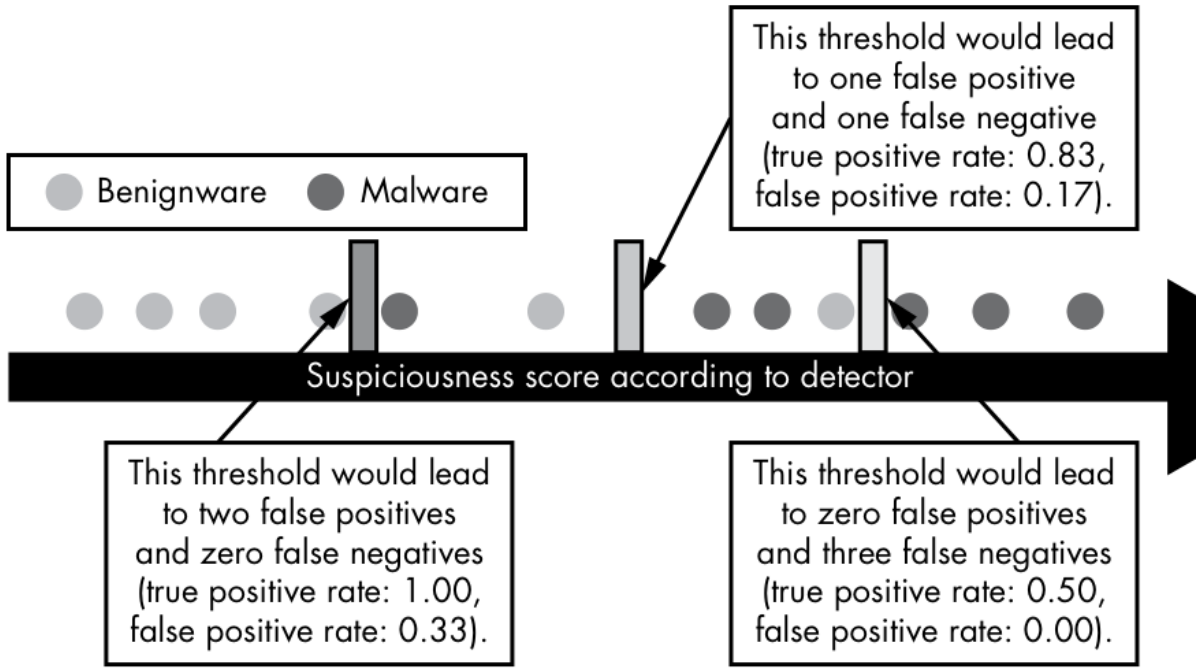

Рисунок 7-3. Связь долей ложноположительных и истинно положительных результатов при выборе порогового значения

Рассмотрим крайний левый порог на рис. 7-3, где файлы слева от порога считаются безвредными, а справа - вредоносными. Порог низок, поэтому доля истинно положительных результатов превосходна: правильно классифицированы 100 процентов вредоносных образцов. Но доля ложноположительных результатов ужасна: 33 процента безвредных образцов ошибочно признаны вредоносными.

Интуиция может подсказать повысить порог, чтобы вредоносными считались лишь образцы с более высокой оценкой подозрительности. Такое решение показано средним порогом на рис. 7-3. Доля ложноположительных результатов снижается до 0,17, но, к сожалению, доля истинно положительных также падает до 0,83. Если продолжить сдвигать порог вправо, как показано крайним правым порогом, ложноположительные результаты исчезнут, но будет обнаружено лишь 50 процентов вредоносных программ.

Как видите, идеального порога не существует. Порог, обеспечивающий низкую долю ложноположительных результатов, что хорошо, обычно пропускает больше вредоносных программ и даёт низкую долю истинно положительных результатов, что плохо. И наоборот, порог с высокой долей истинно положительных результатов, что хорошо, повышает и долю ложноположительных, что плохо.

ROC-кривые

Компромисс между долями истинно положительных и ложноположительных результатов - универсальная проблема всех детекторов, не только детекторов вредоносных программ. Инженеры и статистики долго размышляли над этим явлением и создали для его описания и анализа ROC-кривую, полное английское название которой - Receiver Operating Characteristic.

> **Примечание.** Если выражение Receiver Operating Characteristic кажется вам непонятным, не беспокойтесь. Оно действительно сбивает с толку и связано с первоначальной областью разработки ROC-кривых - радиолокационным обнаружением физических

объектов.

ROC-кривые описывают систему обнаружения, сопоставляя долям ложноположительных результатов соответствующие доли истинно положительных при разных значениях порога. Это помогает оценить компромисс между снижением доли ложноположительных и повышением доли истинно положительных результатов и тем самым выбрать "лучший" порог для конкретной ситуации.

Например, для гипотетической системы MalDetect с рис. 7-3 доля истинно положительных результатов равна 0,5 при доле ложноположительных результатов 0, то есть при низком пороге, а доля истинно положительных результатов равна 1,00 при доле ложноположительных 0,33, то есть при высоком пороге.

На рис. 7-4 этот принцип показан подробнее.

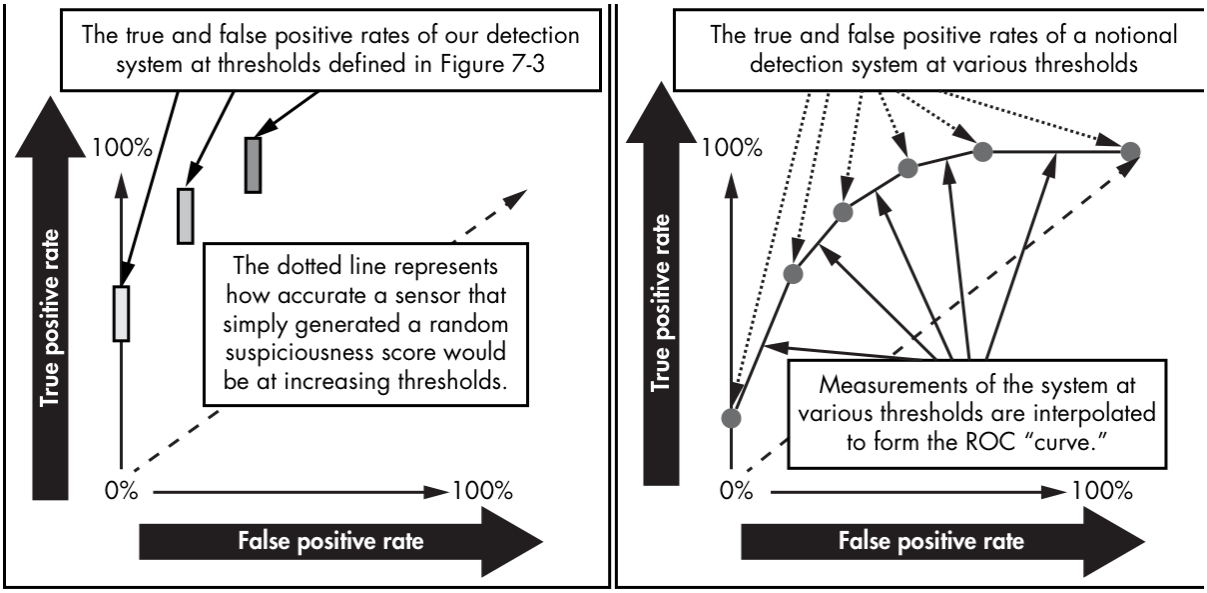

Рисунок 7-4. Смысл ROC-кривых и способ их построения

Для построения ROC-кривой берутся три порога с рис. 7-3 и наносятся полученные для них доли ложноположительных и истинно положительных результатов, как показано в левой части рис. 7-4. График справа показывает то же самое для всех возможных порогов. Как видно, чем выше доля ложноположительных результатов, тем выше доля истинно положительных. Аналогично, чем ниже доля ложноположительных результатов, тем ниже доля истинно положительных.

"Кривая" ROC-кривой - это линия на двумерном ROC-графике, представляющая предполагаемую долю истинно положительных результатов системы для всех возможных значений доли ложноположительных, а также предполагаемую долю ложноположительных результатов для всех возможных значений доли истинно положительных. Существует несколько способов построения такой кривой, но они выходят за рамки книги.

Один простой способ состоит в том, чтобы попробовать множество пороговых значений, наблюдать соответствующие доли ложноположительных и истинно положительных результатов, нанести их на график и соединить точки линией. Эта соединённая линия, показанная на правом графике рис. 7-4, и становится ROC-кривой.

Учёт базовых частот при оценке

Как вы узнали, ROC-кривые сообщают, с какой частотой система правильно называет вредоносные файлы вредоносными, то есть долю истинно положительных результатов, и с какой частотой называет вредоносными безвредные файлы, то есть долю ложноположительных. Однако ROC-кривая не сообщает, какой процент тревог системы будет истинно положительным. Этот показатель называется точностью системы. Точность связана с долей встречаемых системой двоичных файлов, которые в действительности являются вредоносными. Эта доля называется базовой частотой. Определим оба термина:

• **Точность.** Процент тревог системы обнаружения, являющихся истинно положительными, то есть действительно указывающих на вредоносные программы. Иными словами, при проверке на некотором наборе двоичных файлов точность системы равна: количество истинно положительных результатов / (истинно положительные + ложноположительные результаты).
• **Базовая частота.** Процент переданных системе данных, обладающих искомым свойством. В нашем случае базовая частота означает процент двоичных файлов, которые в действительности являются вредоносными программами.

В следующем разделе рассматривается связь этих двух показателей.

Как базовая частота влияет на точность

Хотя доли истинно положительных и ложноположительных результатов системы не изменяются при изменении базовой частоты, её точность меняется, нередко очень сильно. Чтобы понять причину, рассмотрим два случая.

Предположим, доля ложноположительных результатов MalDetect равна 1 проценту, а доля истинно положительных - 100 процентам. Теперь запустим MalDetect в сети, о которой заранее известно, что в ней нет вредоносных программ, например в только что созданной лабораторной сети. Поскольку вредоносных программ заведомо нет, каждая тревога MalDetect по определению будет ложноположительной: система встретит только безвредные двоичные файлы. Иными словами, точность составит 0 процентов.

Напротив, если запустить MalDetect на наборе, целиком состоящем из вредоносных программ, ни одна тревога не будет ложноположительной: такой результат просто не сможет возникнуть, поскольку в наборе программ нет безвредных файлов. Следовательно, точность составит 100 процентов.

В обоих крайних случаях базовая частота оказывает огромное влияние на точность MalDetect, то есть на вероятность того, что тревога системы окажется ложноположительной.

Оценка точности в среде развёртывания

Теперь вы знаете, что в зависимости от доли вредоносных программ в проверочном наборе, то есть базовой частоты, система показывает очень разную точность. Как оценить точность системы по предполагаемой базовой частоте среды её развёртывания? Достаточно подставить оценку базовой частоты среды в переменные формулы точности: истинно положительные / (истинно положительные + ложноположительные результаты). Потребуются три числа:

• доля истинно положительных результатов системы, TPR, то есть процент образцов вредоносных программ, правильно обнаруживаемых системой;

• доля ложноположительных результатов системы, FPR, то есть процент безвредных образцов, для которых система ошибочно поднимает тревогу;
• базовая частота BR двоичных файлов, с которыми будет использоваться система. Например, если система будет проверять файлы с пиратских сайтов, это ожидаемый процент вредоносных программ среди загруженных оттуда файлов.

Числитель формулы точности, то есть количество истинно положительных результатов, можно оценить произведением доли истинно положительных результатов и базовой частоты. Оно даёт процент вредоносных программ, которые система обнаружит правильно. Знаменатель, то есть сумма истинно положительных и ложноположительных результатов, можно оценить как доля истинно положительных результатов x базовая частота + доля ложноположительных результатов x (1 - базовая частота). Полученное значение представляет процент всех двоичных файлов, для которых система поднимет тревогу: количество правильно обнаруженных вредоносных файлов плюс доля безвредных файлов, вызвавших ложную тревогу.

Итак, ожидаемая точность системы вычисляется следующим образом:

```
                доля истинно положительных результатов x базовая частота
точность = ------------------------------------------------------------------------
           доля истинно положительных результатов x базовая частота +
           доля ложноположительных результатов x (1 - базовая частота)
```

Рассмотрим ещё один пример глубокого влияния базовой частоты на работу системы обнаружения. Предположим, доля истинно положительных результатов системы равна 80 процентам, доля ложноположительных - 10 процентам, а ожидаемая доля вредоносных программ среди проверяемых файлов - 50 процентов. Тогда ожидаемая точность составит 89 процентов. Но при базовой частоте 10 процентов точность упадёт до 47 процентов.

Что произойдёт при очень низкой базовой частоте? Например, в современной корпоративной сети лишь очень немногие двоичные файлы в действительности вредоносны. Если подставить в формулу базовую частоту 1 процент, то есть один вредоносный файл на 100, точность составит около 7,5 процента, а это означает, что 92,5 процента тревог системы будут ложноположительными! При базовой частоте 0,1 процента, то есть одном вероятно вредоносном файле на 1000, точность составит 1 процент: 99 процентов тревог окажутся ложными! Наконец, при базовой частоте 0,01 процента, то есть одном вероятно вредоносном файле на 10 000, что, пожалуй, наиболее реалистично для корпоративной сети, ожидаемая точность упадёт до 0,1 процента. Подавляющее большинство предупреждений системы будет ложноположительным.

Один из выводов состоит в том, что системы с высокой долей ложноположительных результатов почти никогда не полезны в корпоративных средах, поскольку их точность слишком мала. Поэтому ключевая цель при создании систем обнаружения вредоносных программ - настолько снизить долю ложноположительных результатов, чтобы точность системы стала приемлемой.

Другой связанный вывод: при анализе ROC-кривой, представленном выше, следует фактически игнорировать доли ложноположительных результатов выше, скажем, 1 процента, если система предназначена для корпоративной среды. Любая более высокая доля, скорее всего, приведёт к столь низкой точности, что система окажется бесполезной.

Итоги

В этой главе вы изучили основные понятия оценки обнаружения, включая доли истинно положительных и ложноположительных результатов, ROC-кривые, базовые частоты и точность. Вы увидели, что при создании системы обнаружения вредоносных программ важно как максимизировать долю истинно положительных результатов, так и минимизировать долю ложноположительных. Из-за влияния базовой частоты на точность снижение доли ложноположительных результатов особенно важно, если систему планируется развернуть в организации.

Если вы ещё не чувствуете себя совершенно уверенно в этих понятиях, не беспокойтесь. В следующей главе вы получите дополнительную практику, создавая и оценивая систему обнаружения вредоносных программ с нуля. В процессе вы изучите дополнительные понятия оценки, относящиеся к машинному обучению и помогающие совершенствовать детекторы на его основе.

Глава 8. Создание детекторов на основе машинного обучения

Сегодня благодаря высококачественному открытому программному обеспечению, которое берёт на себя сложные математические вычисления при реализации систем машинного обучения, применять машинное обучение может любой человек, знающий основы Python и понимающий ключевые понятия.

В этой главе я покажу, как создавать системы обнаружения вредоносных программ с помощью scikit-learn - самого популярного и, на мой взгляд, лучшего открытого пакета машинного обучения. Глава содержит много примеров кода. Основные блоки кода находятся в каталоге `malware_data_science/ch8/code`, а соответствующие данные - в каталоге `malware_data_science/ch8/data` среди кода и данных, а также на виртуальной машине, прилагаемых к книге.

Если вы будете следовать изложению, изучать примеры кода и самостоятельно их запускать, то к концу главы сможете уверенно создавать и оценивать собственные системы машинного обучения. Вы также научитесь создавать общий детектор вредоносных программ и пользоваться инструментами, необходимыми для построения детекторов конкретных семейств. Полученные навыки имеют широкое применение и позволят использовать машинное обучение в других задачах безопасности, например для обнаружения вредоносных писем или подозрительных сетевых потоков.

Сначала вы изучите термины и понятия, необходимые перед началом работы со scikit-learn. Затем реализуете в scikit-learn базовый детектор на основе дерева решений, используя понятия из главы 6. После этого научитесь объединять код извлечения признаков со scikit-learn и создадите практический детектор, который извлекает реальные признаки и обнаруживает вредоносные программы методом случайного леса. Наконец, на примере этого детектора вы научитесь оценивать системы машинного обучения средствами scikit-learn.

Термины и понятия

Сначала повторим некоторые термины. Открытая библиотека scikit-learn, сокращённо sklearn, стала популярной в сообществе машинного обучения благодаря сочетанию мощности и простоты. Многие специалисты по анализу данных применяют её как в компьютерной безопасности, так и в других областях, а для многих она служит основным набором инструментов машинного обучения. Хотя sklearn постоянно пополняется новыми подходами, единообразный программный интерфейс упрощает их применение.

Как и многим другим платформам машинного обучения, sklearn требуются обучающие данные в форме векторов. Векторы - это массивы чисел, где каждому индексу соответствует один признак обучающего двоичного файла. Например, если детектор использует два признака программного файла - *сжат* и *содержит зашифрованные данные*, - вектор признаков обучающего примера может выглядеть как `[0,1]`. Первый индекс сообщает, сжат ли файл, причём ноль означает "нет", а второй - содержит ли он зашифрованные данные, причём единица означает "да".

Работать с векторами неудобно, поскольку приходится помнить соответствие индексов признакам. К счастью, sklearn предоставляет вспомогательный код, преобразующий другие представления данных в векторную форму. Например, класс sklearn `DictVectorizer` преобразует словарное представление обучающих данных, скажем `{"is compressed":1,"contains encrypted data":0}`,

в используемое sklearn векторное представление вроде `[0,1]`. Позже с помощью `DictVectorizer` можно восстановить соответствие между индексами вектора и исходными именами признаков.

Для обучения детектора на основе sklearn нужно передать библиотеке два отдельных объекта: описанные выше векторы признаков и вектор меток. Вектор меток содержит по одному числу для каждого обучающего примера и в нашем случае сообщает, является пример вредоносной или безвредной программой. Например, передав три обучающих примера, а затем вектор меток `[0,1,0]`, мы сообщим sklearn, что первый образец безвреден, второй вредоносен, а третий безвреден. По соглашению специалисты по машинному обучению обозначают обучающие данные прописной переменной `X`, а метки - строчной переменной `y`. Разный регистр отражает математическое соглашение обозначать прописными буквами матрицы, которые можно представить как массивы векторов, а строчными - отдельные векторы. Вы встретите это соглашение в примерах машинного обучения в интернете; оно будет использоваться и далее в книге, чтобы вы к нему привыкли.

Платформа sklearn применяет и другие непривычные термины. Детекторы на основе машинного обучения в ней называются не "детекторами", а "классификаторами". В этом контексте классификатор - всего лишь система машинного обучения, распределяющая объекты по двум или более категориям. Следовательно, детектор, как я называю его в книге, представляет собой частный вид классификатора, который распределяет объекты по двум категориям, например вредоносным и безвредным программам. Кроме того, вместо термина *обучение* в документации и API sklearn часто используется термин *подгонка*. Например, выражение "подогнать классификатор машинного обучения по обучающим примерам" равнозначно выражению "обучить детектор машинного обучения на обучающих примерах".

Наконец, применительно к классификаторам вместо термина *обнаружить* sklearn использует термин *предсказать*. В платформе sklearn и вообще в сообществе машинного обучения этот термин употребляется всякий раз, когда система выполняет задачу, будь то предсказание стоимости акции через неделю или обнаружение вредоносности неизвестного двоичного файла.

Создание учебного детектора на основе дерева решений

Теперь, познакомившись с технической терминологией sklearn, создадим с помощью этой платформы простое дерево решений, подобное рассмотренному в главе 6. Напомним, дерево решений играет в подобие игры "Двадцать вопросов": задаёт последовательность вопросов о входных векторах и приходит к решению об их вредоносности или безвредности. Мы пошагово построим классификатор на основе дерева решений, а затем рассмотрим пример полной программы. В листинге 8-1 показан импорт необходимых модулей sklearn.

```
from sklearn import tree
from sklearn.feature_extraction import DictVectorizer
```

Листинг 8-1. Импорт модулей sklearn

Первый импортируемый модуль `tree` - модуль деревьев решений sklearn. Второй модуль `feature_extraction` - вспомогательный модуль sklearn, из которого импортируется класс `DictVectorizer`. Этот класс удобно преобразует переданные в читаемой словарной форме обучающие данные в векторное представление, необходимое sklearn для фактического обучения детекторов.

После импорта нужных модулей sklearn создадим экземпляры необходимых классов, как показано в листинге 8-2.

```
classifier = tree.DecisionTreeClassifier()  # (1)
vectorizer = DictVectorizer(sparse=False)  # (2) (3)
```

Листинг 8-2. Инициализация классификатора на основе дерева решений и векторизатора

Первый созданный класс `DecisionTreeClassifier` (1) представляет детектор. Sklearn предоставляет ряд параметров, управляющих точной работой дерева, но здесь параметры не выбираются и используются настройки по умолчанию.

Следующим создаётся экземпляр `DictVectorizer` (2). В конструкторе параметру `sparse` присваивается `False` (3), что запрещает sklearn использовать разреженные векторы. Они экономят память, но сложнее в работе, а модуль деревьев решений sklearn не умеет их использовать, поэтому данная возможность отключается.

После создания экземпляров классов можно инициализировать учебные обучающие данные, как в листинге 8-3.

```
# Объявить учебные обучающие данные
training_examples = [  # (1)
    {'packed':1,'contains_encrypted':0},
    {'packed':0,'contains_encrypted':0},
    {'packed':1,'contains_encrypted':1},
    {'packed':1,'contains_encrypted':0},
    {'packed':0,'contains_encrypted':1},
    {'packed':1,'contains_encrypted':0},
    {'packed':0,'contains_encrypted':0},
    {'packed':0,'contains_encrypted':0},
]
ground_truth = [1,1,1,1,0,0,0,0]  # (2)
```

Листинг 8-3. Объявление обучающих векторов и вектора меток

В этом примере инициализируются две структуры, вместе составляющие обучающие данные: векторы признаков и вектор меток. Векторы признаков, присвоенные переменной `training_examples` (1), заданы в словарной форме. Используются два простых признака. Первый, `packed`, указывает, упакован ли файл, а второй, `contains_encrypted`, - содержит ли файл зашифрованные данные. Вектор меток в переменной `ground_truth` (2) сообщает, является каждый обучающий пример вредоносным или безвредным. В этой книге и вообще среди специалистов по анализу данных безопасности 0 всегда означает безвредный файл, а 1 - вредоносный. Здесь вектор объявляет первые четыре вектора признаков вредоносными, а вторые четыре - безвредными.

Обучение классификатора на основе дерева решений

После объявления обучающих векторов и вектора меток обучим модель, вызвав метод `fit` экземпляра класса дерева решений, как показано в листинге 8-4.

```
# Инициализировать векторизатор обучающими данными
vectorizer.fit(training_examples)  # (1)
# Преобразовать обучающие примеры в векторную форму
X = vectorizer.transform(training_examples)  # (2)
y = ground_truth # По соглашению назовём эталонные метки 'y'
```

Листинг 8-4. Инициализация класса векторизатора обучающими данными

Сначала код листинга 8-4 инициализирует класс векторизатора, экземпляр которого был создан в листинге 8-2, вызовом метода `fit` (1). Здесь этот метод приказывает sklearn создать соответствие между признаками `packed` и `contains_encrypted` и индексами векторного массива. Затем словарные векторы признаков преобразуются в числовую векторную форму вызовом метода `transform` класса векторизатора (2). Напомним, по принятому в сообществе соглашению векторы признаков присваиваются переменной `X`, а вектор меток - переменной `y`.

Теперь, подготовив обучающие данные, можно обучить детектор дерева решений, вызвав метод `fit` экземпляра классификатора:

```
# Обучить классификатор, то есть "подогнать" его
classifier.fit(X,y)
```

Как видите, обучить детектор sklearn очень просто. Но за кулисами sklearn выполняет алгоритмический процесс поиска хорошего дерева решений, способного точно определять вредоносность или безвредность новых программ, подобно алгоритму из предыдущей главы.

После обучения детектора воспользуемся кодом листинга 8-5, чтобы определить вредоносность двоичного файла.

```
test_example = {'packed':1,'contains_encrypted':0}  # (1)
test_vector = vectorizer.transform(test_example)  # (2)
print classifier.predict(test_vector) # Выводит [1] (3)
```

Листинг 8-5. Определение вредоносности двоичного файла

Здесь создаётся словарный вектор признаков гипотетического двоичного файла (1), переводится в числовую форму ранее объявленным векторизатором (2), а затем передаётся созданному детектору дерева решений (3) для определения вредоносности. При запуске кода классификатор "считает" новый файл вредоносным, поскольку выводит 1. Причина станет ясна после визуализации дерева решений.

Визуализация дерева решений

Дерево, автоматически созданное sklearn по обучающим данным, можно визуализировать, как показано в листинге 8-6.

```
# Визуализировать дерево решений
with open("classifier.dot","w") as output_file:  # (1)
    tree.export_graphviz(  # (2)
        classifier,
        feature_names=vectorizer.get_feature_names(),
        out_file=output_file
    )

import os
os.system("dot classifier.dot -Tpng -o classifier.png")
```

Листинг 8-6. Создание изображения дерева решений с помощью GraphViz

Здесь открывается файл `classifier.dot` (1), в который функция `export_graphviz()` модуля деревьев sklearn записывает сетевое представление дерева решений. Затем вызывается `tree.export_graphviz` (2), записывающая в `classifier.dot` файл GraphViz `.dot` с сетевым представлением дерева на диске. Наконец, программа командной строки GraphViz `dot` создаёт изображение дерева в форме, соответствующей материалу главы 6. После запуска должен появиться файл `classifier.png`, похожий на рис. 8-1.

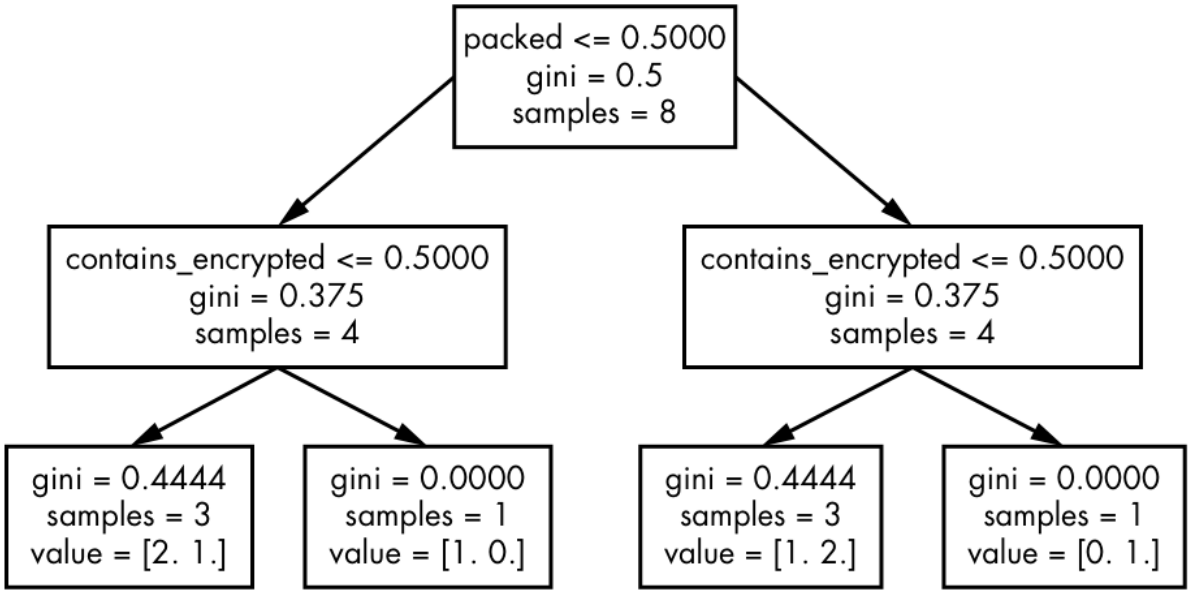

Рисунок 8-1. Визуализация дерева решений

Визуализация должна быть знакома по главе 6, но здесь присутствуют новые термины. Первая строка каждого прямоугольника содержит имя признака, о котором узел задаёт вопрос; на языке машинного обучения говорят, что узел "разделяет по" этому признаку. Например, первый узел разделяет по признаку `packed`: если файл не упакован, движение продолжается по левой стрелке, иначе - по правой.

Вторая строка каждого прямоугольника относится к индексу Джини узла, который измеряет неравенство между вредоносными и безвредными обучающими примерами, соответствующими узлу. Чем выше индекс Джини, тем сильнее соответствующие узлу образцы смещены в сторону либо безвредных, либо вредоносных. Следовательно, высокий индекс Джини в каждом узле полезен: чем сильнее обучающие примеры смещены к одному из классов, тем увереннее можно определить, являются новые проверочные примеры вредоносными или безвредными. Третья строка каждого прямоугольника просто сообщает количество обучающих примеров, соответствующих узлу.

Обратите внимание: текст в листовых узлах дерева отличается. Эти узлы не "задают вопрос", а отвечают на вопрос "является ли двоичный файл вредоносным или безвредным?" Например, крайний левый листовой узел содержит `value = [2. 1.]`. Это означает, что узлу соответствуют два безвредных обучающих примера, не упакованных и не зашифрованных, и один вредоносный. Если достигнут этот узел, файлу назначается вероятность вредоносности 33 процента: один вредоносный образец / три образца всего = 33 процента. Значение Джини в таких прямоугольниках показывает, сколько информации о вредоносности или безвредности файла даёт разделение по вопросу, непосредственно ведущему к этим узлам. Как видите, изучение созданных sklearn визуализаций помогает понять, как деревья решений выполняют обнаружение.

Полный пример кода

В листинге 8-7 приведён полный код описанного процесса работы с деревом решений. После пошагового разбора он должен быть вам вполне понятен.

```
#!/usr/bin/python

# Импортировать модули sklearn
from sklearn import tree
from sklearn.feature_extraction import DictVectorizer
```

```
# Инициализировать классификатор дерева решений и векторизатор
classifier = tree.DecisionTreeClassifier()
vectorizer = DictVectorizer(sparse=False)

# Объявить учебные обучающие данные
training_examples = [
{'packed':1,'contains_encrypted':0},
{'packed':0,'contains_encrypted':0},
{'packed':1,'contains_encrypted':1},
{'packed':1,'contains_encrypted':0},
{'packed':0,'contains_encrypted':1},
{'packed':1,'contains_encrypted':0},
{'packed':0,'contains_encrypted':0},
{'packed':0,'contains_encrypted':0},
]
ground_truth = [1,1,1,1,0,0,0,0]

# Инициализировать векторизатор обучающими данными
vectorizer.fit(training_examples)

# Преобразовать обучающие примеры в векторную форму
X = vectorizer.transform(training_examples)
y = ground_truth # По соглашению назовём эталонные метки 'y'
# Обучить классификатор, то есть "подогнать" его
classifier.fit(X,y)

test_example = {'packed':1,'contains_encrypted':0}
test_vector = vectorizer.transform(test_example)
print classifier.predict(test_vector) # Выводит [1]

# Визуализировать дерево решений
with open("classifier.dot","w") as output_file:
    tree.export_graphviz(
        classifier,
        feature_names=vectorizer.get_feature_names(),
        out_file=output_file
    )

import os
os.system("dot classifier.dot -Tpng -o classifier.png")
```

Листинг 8-7. Полный пример процесса работы с деревом решений

Рассмотренный учебный детектор показывает, как начать пользоваться возможностями sklearn, но ему недостаёт некоторых функций, необходимых практическому детектору вредоносных программ. Теперь разберём, что требуется такому детектору.

Создание практических детекторов машинного обучения с помощью sklearn

Для практического детектора нужны полноценные признаки программных двоичных файлов и код, извлекающий их из файлов. Полноценные признаки отражают содержимое двоичных файлов во всей его сложности, поэтому необходимо использовать сотни или тысячи признаков. Под "извлечением" признаков я понимаю написание кода, который определяет их присутствие в двоичных файлах. Кроме того, требуются тысячи обучающих примеров и масштабное обучение модели. Наконец, нужно применять более совершенные методы обнаружения sklearn, поскольку рассмотренное простое дерево решений не обеспечивает достаточной точности.

Извлечение практических признаков

Прежние признаки вроде *упакован* и *содержит зашифрованные данные* - простые учебные примеры, и двух таких признаков недостаточно для рабочего детектора. Как уже говорилось, практические системы используют сотни, тысячи и даже миллионы признаков. Например, детектор может использовать в качестве признаков миллионы строк символов из двоичных файлов. Он также может использовать значения заголовков формата Portable Executable, импортируемые файлом функции либо некоторое сочетание всех этих данных. В этой главе мы будем работать только со строковыми признаками, но сначала кратко рассмотрим распространённые категории признаков для обнаружения вредоносных программ, начиная со строк.

Строковые признаки

Строковые признаки программного двоичного файла - это все непрерывные последовательности печатных символов не короче некоторой минимальной длины; в этой книге минимум равен пяти символам. Предположим, файл содержит следующие последовательности печатных символов:

```
["A", "The", "PE executable", "Malicious payload"]
```

В таком случае признаками могут служить строки `"PE executable"` и `"Malicious payload"`, поскольку каждая содержит больше пяти символов.

Чтобы преобразовать строковые признаки в понятный sklearn формат, их нужно поместить в словарь Python. Сами строки используются как ключи словаря, а их значения устанавливаются в 1, указывая, что данный файл содержит строку. Например, предыдущий учебный файл получит вектор признаков `{"PE executable": 1, "Malicious payload": 1}`. Разумеется, большинство двоичных файлов содержит не две, а сотни печатных строк, способных нести богатую информацию о работе программы.

В самом деле, строковые признаки хорошо подходят для обнаружения на основе машинного обучения, поскольку заключают в себе много информации о двоичных файлах. Если вредоносный образец упакован, в нём, скорее всего, будет мало информативных строк, что само по себе может выдать его вредоносность. Если же части раздела ресурсов файла не упакованы и не обфусцированы, их строки многое расскажут о поведении файла. Например, если программа выполняет HTTP-запросы, среди её строк часто встречается `"GET %s"`.

Однако у строковых признаков есть ограничения. Например, они ничего не сообщают о фактической логике двоичной программы, поскольку не содержат её исполняемый код. Поэтому, хотя строки полезны даже в упакованных файлах, они не раскрывают фактические действия таких файлов. В результате основанные на строках детекторы не идеальны для обнаружения упакованных вредоносных программ.

Признаки заголовка Portable Executable (PE)

Признаки заголовка PE извлекаются из его метаданных, присутствующих в каждом файле Windows `.exe` и `.dll`. Подробнее формат заголовков описан в главе 1. Чтобы извлечь признаки PE из статических двоичных файлов, можно использовать приведённый там код, а затем закодировать признаки файла словарём Python, где имя поля заголовка служит ключом, а значение поля - соответствующим значением словаря.

Признаки заголовка PE хорошо дополняют строковые. Например, строки часто хорошо отражают вызовы функций и сетевые передачи программы, как в примере `"GET %s"`, а признаки заголовка PE

содержат такие сведения, как временная метка компиляции, структура разделов PE, отметки исполняемости этих разделов и их размер на диске. Они также отражают объём памяти, выделяемый программой при запуске, и многие другие характеристики двоичного файла во время выполнения, которых нет в строковых признаках.

Даже для упакованных файлов признаки заголовка PE позволяют достаточно хорошо отличать вредоносные программы от безвредных. Хотя из-за обфускации код упакованных файлов не виден, по-прежнему видны занимаемый им объём на диске, расположение двоичного файла и способ сжатия по нескольким файловым разделам. Эти показательные детали могут помочь системе машинного обучения различить вредоносные и безвредные программы. Недостаток признаков заголовка PE состоит в том, что они не отражают фактические инструкции, выполняемые программой, и вызываемые ею функции.

Признаки таблицы адресов импорта (IAT)

Таблица адресов импорта, IAT, с которой вы познакомились в главе 1, также служит важным источником признаков. Она содержит список функций и библиотек, импортируемых двоичным файлом из внешних DLL. Поэтому IAT заключает важную информацию о поведении программы, способную дополнить описанные выше признаки заголовка PE.

Чтобы использовать IAT как источник признаков, каждый файл нужно представить словарём, где имя импортируемой библиотеки и функции служит ключом, а значение 1 указывает, что файл содержит конкретный импорт. Пример ключа: `"KERNEL32.DLL:LoadLibraryA"`, где `KERNEL32.DLL` - DLL, а `LoadLibraryA` - вызов функции. Полученный для образца словарь признаков IAT будет выглядеть как `{KERNEL32.DLL:LoadLibraryA: 1, ...}`, причём каждому обнаруженному в файле ключу назначается 1.

По моему опыту создания детекторов, признаки IAT редко хорошо работают сами по себе. Хотя они отражают полезную высокоуровневую информацию о поведении, вредоносные программы часто обфусцируют IAT, чтобы выглядеть безвредными. Даже без обфускации вредоносные программы нередко импортируют те же вызовы DLL, что и безвредные, поэтому различить их только по IAT трудно. Наконец, если вредоносная программа упакована, то есть сжата или зашифрована и настоящий код становится виден только после запуска и распаковки либо расшифрования, IAT содержит лишь импорты упаковщика, а не вредоносной программы. Тем не менее в сочетании с признаками заголовка PE и строками признаки IAT могут повысить точность системы.

N-граммы

До сих пор рассматривались признаки машинного обучения, не учитывающие порядок. Например, строковые признаки проверяют присутствие конкретной строки в файле, но не то, расположена ли она на диске до или после другой строки. Иногда порядок имеет значение. Например, важное семейство вредоносных программ может импортировать только обычные функции, но в строго определённом порядке. Увидев их в таком порядке, мы поймём, что перед нами это семейство, а не безвредная программа. Для отражения порядка используется понятие машинного обучения, называемое N-граммой.

N-граммы звучат экзотичнее, чем устроены. Признаки располагаются в порядке появления, а затем по последовательности скользит окно длины n; последовательность признаков внутри окна на каждом шаге считается единым составным признаком. Например, для последовательности `["how", "now", "brown", "cow"]` и N-грамм длины 2, то есть n = 2, признаками будут `[('how','now'), ('now','brown'), ('brown','cow')]`.

В обнаружении вредоносных программ некоторые виды данных естественнее всего представлять N-граммами. Например, после дизассемблирования файла в последовательность инструкций вроде `["inc", "dec", "sub", "mov"]` имеет смысл использовать N-граммы для отражения их последовательностей, поскольку последовательность инструкций помогает обнаружить конкретную реализацию вредоносной программы. Аналогично при выполнении файлов для изучения динамического поведения N-граммы могут представлять последовательности вызовов API или высокоуровневых действий.

Я рекомендую экспериментировать с N-граммами в системах обнаружения всякий раз, когда данные имеют некоторую последовательность. Чтобы определить n, то есть длину N-граммы, часто требуются пробы и ошибки. Нужно изменять n и смотреть, какое значение даёт наибольшую точность на проверочных данных. При правильном числе N-граммы способны эффективно отражать фактическое последовательное поведение двоичных программ и повышать точность системы.

Почему нельзя использовать все возможные признаки

Узнав достоинства и недостатки разных категорий, вы можете спросить, почему бы не использовать все признаки одновременно и не создать лучший возможный детектор. На то есть несколько причин.

Во-первых, извлечение всех рассмотренных признаков занимает много времени и снижает скорость сканирования файлов. Что важнее, избыточное количество признаков может вызвать нехватку памяти и чрезмерно увеличить время обучения алгоритма. Поэтому при создании систем я советую пробовать разные признаки и сосредоточиваться на тех, которые хорошо работают для искомых вредоносных программ и безвредных программ, на которых нужно избегать ложных срабатываний.

К сожалению, даже если сосредоточиться на одной категории, например строках, признаков часто будет больше, чем способно обработать большинство алгоритмов. При использовании строк нужен отдельный признак для каждой уникальной строки в обучающих данных. Например, если обучающий образец A содержит `"hello world"`, а образец B - `"hello world!"`, эти строки придётся считать двумя разными признаками. Поэтому при работе с тысячами обучающих образцов быстро появятся тысячи уникальных строк, и система будет использовать столько же признаков.

Сжатие признаков с помощью трюка хеширования

Проблему избыточного количества признаков можно решить популярным и простым методом, называемым трюком хеширования или хешированием признаков. Предположим, обучающий набор содержит миллион уникальных строковых признаков, но алгоритм и оборудование способны обработать лишь 4000 уникальных признаков во всём наборе. Нужен способ сжать миллион признаков в вектор длиной 4000 элементов.

Трюк хеширования помещает миллион признаков в пространство размерностью 4000, хешируя каждый признак в один из 4000 индексов. Затем значение исходного признака прибавляется к числу по этому индексу в 4000-мерном векторе. Разумеется, признаки часто сталкиваются, поскольку их значения складываются в одном измерении. Это может повлиять на точность, так как алгоритм больше не "видит" значения отдельных признаков. Но на практике снижение точности часто очень мало, а польза сжатого представления значительно превосходит эту небольшую потерю.

Реализация трюка хеширования

Чтобы прояснить идею, разберём пример кода, реализующего трюк хеширования. Здесь код показан для объяснения алгоритма; позже мы воспользуемся реализацией sklearn. Пример начинается с объявления функции:

```
def apply_hashing_trick(feature_dict, vector_size=2000):
```

Функция `apply_hashing_trick()` принимает два параметра: исходный словарь признаков и размер вектора, в котором после хеширования будет храниться уменьшенный вектор признаков.

Затем создаётся новый массив признаков:

```
new_features = [0 for x in range(vector_size)]
```

Массив `new_features` хранит информацию о признаках после применения трюка хеширования. Его основные операции выполняются внутри цикла `for`, как показано в листинге 8-8.

```
for key in feature_dict:  # (1)
    array_index = hash(key) % vector_size  # (2)
    new_features[array_index] += feature_dict[key]  # (3)
```

Листинг 8-8. Выполнение хеширования в цикле `for`

Здесь цикл `for` перебирает каждый признак словаря (1). Для этого ключи словаря, а в случае строковых признаков - отдельные строки двоичного файла, хешируются по модулю `vector_size`, чтобы хеш-значения были ограничены диапазоном от нуля до `vector_size - 1` (2). Результат сохраняется в переменной `array_index`.

Внутри цикла значение элемента массива `new_features` с индексом `array_index` увеличивается на значение из исходного массива признаков (3). Для строковых признаков, где значение 1 указывает на присутствие конкретной строки в файле, элемент увеличивается на единицу. Для признаков заголовка PE, имеющих разные значения, например объём памяти, занимаемый разделом PE, к элементу прибавляется значение признака.

Наконец, вне цикла просто возвращается словарь `new_features`:

```
return new_features
```

Теперь sklearn может работать с `new_features`, используя лишь тысячи, а не миллионы уникальных признаков.

Полный код трюка хеширования

В листинге 8-9 приведён полный, теперь уже знакомый вам код трюка хеширования.

```
def apply_hashing_trick(feature_dict,vector_size=2000):
    # Создать массив нулей длиной 'vector_size'
    new_features = [0 for x in range(vector_size)]

    # Перебрать все признаки в словаре
    for key in feature_dict:

        # Получить индекс в новом массиве признаков
        array_index = hash(key) % vector_size

        # Прибавить значение признака к элементу нового массива
        # с индексом, полученным с помощью трюка хеширования
        new_features[array_index] += feature_dict[key]

    return new_features
```

Листинг 8-9. Полный код реализации трюка хеширования

Как вы увидели, хеширование признаков легко реализовать самостоятельно, что помогает понять его работу. Но можно просто воспользоваться реализацией sklearn, которая удобна и лучше оптимизирована.

Применение FeatureHasher из sklearn

Чтобы воспользоваться встроенной реализацией sklearn вместо собственной, сначала нужно импортировать класс `FeatureHasher`:

```
from sklearn.feature_extraction import FeatureHasher
```

Затем создаётся экземпляр класса:

```
hasher = FeatureHasher(n_features=2000)
```

Параметр `n_features` задаёт размер нового массива, получаемого после хеширования.

Чтобы применить трюк хеширования к векторам признаков, достаточно передать их методу `transform` класса `FeatureHasher`:

```
features = [{'how': 1, 'now': 2, 'brown': 4},{'cow': 2, '.': 5}]
hashed_features = hasher.transform(features)
```

Результат фактически совпадает с результатом собственной реализации в листинге 8-9. Различие лишь в использовании реализации sklearn: хорошо сопровождаемая библиотека машинного обучения удобнее собственного кода. Полный пример приведён в листинге 8-10.

```
from sklearn.feature_extraction import FeatureHasher
hasher = FeatureHasher(n_features=10)
features = [{'how': 1, 'now': 2, 'brown': 4},{'cow': 2, '.': 5}]
hashed_features = hasher.transform(features)
```

Листинг 8-10. Реализация с помощью `FeatureHasher`

Перед продолжением отметим несколько особенностей хеширования признаков. Во-первых, как вы, вероятно, догадались, оно скрывает информацию о передаваемых алгоритму признаках, поскольку складывает значения лишь на основании совпадения хеш-корзины. Следовательно, в общем случае чем меньше корзин используется или чем больше признаков хешируется в фиксированное количество корзин, тем хуже работает алгоритм. Удивительно, но даже с трюком хеширования алгоритмы могут работать хорошо. Поскольку современное оборудование просто не позволяет обрабатывать миллионы или миллиарды признаков, в анализе данных безопасности обычно приходится применять этот трюк.

Другой недостаток состоит в том, что при анализе внутреннего устройства модели восстановить исходные хешированные признаки трудно или невозможно. Возьмём дерево решений: поскольку в каждый элемент вектора хешируются произвольные признаки, неизвестно, какой из сложенных в этом элементе признаков заставил дерево выполнить по нему разделение. Любое количество признаков могло привести алгоритм к выводу, что такое разделение полезно. Это существенное ограничение, но специалисты по анализу данных безопасности мирятся с ним из-за огромного преимущества: возможности сжать миллионы признаков до управляемого количества.

Теперь, рассмотрев строительные блоки практического детектора, создадим сквозную систему обнаружения вредоносных программ на основе машинного обучения.

Создание детектора промышленного уровня

С точки зрения требований к программному обеспечению практический детектор должен выполнять три задачи: извлекать из двоичных файлов признаки для обучения и обнаружения, обучаться обнаруживать вредоносные программы по обучающим данным и фактически выполнять обнаружение для новых файлов. Пошагово разберём код каждой задачи и увидим, как всё соединяется.

Код этого раздела находится в `malware_data_science/ch8/code/complete_detector.py` среди материалов книги и по тому же пути на прилагаемой виртуальной машине. Однострочный сценарий оболочки `malware_data_science/ch8/code/run_complete_detector.sh` показывает, как запустить детектор из оболочки.

Извлечение признаков

Первым делом реализуется код извлечения признаков из обучающих двоичных файлов; здесь я пропускаю шаблонный код и сосредоточиваюсь на основных функциях программы. Для извлечения нужно получить необходимые данные из файлов, сохранить признаки в словаре Python и, если количество уникальных признаков может стать непомерным, преобразовать их реализацией трюка хеширования из sklearn.

Для простоты используются только строковые признаки и трюк хеширования. В листинге 8-11 показано и то и другое.

```
def get_string_features(path,hasher):  # (1) (2)
    # Извлечь строки из двоичного файла регулярным выражением
    chars = r" -~"
    min_length = 5
    string_regexp = '[%s]{%d,}' % (chars, min_length)
    file_object = open(path)
    data = file_object.read()
    pattern = re.compile(string_regexp)
    strings = pattern.findall(data)

    # Сохранить строковые признаки в словарной форме
    string_features = {}  # (3)
    for string in strings:
        string_features[string] = 1

    # Хешировать признаки с помощью трюка хеширования
    hashed_features = hasher.transform([string_features])  # (4)

    # Преобразовать данные, чтобы получить массив признаков
    hashed_features = hashed_features.todense()
    hashed_features = numpy.asarray(hashed_features)
    hashed_features = hashed_features[0]

    # Вернуть хешированные строковые признаки
    print "Extracted {0} strings from {1}".format(len(string_features),path)
    return hashed_features  # (5)
```

Листинг 8-11. Определение функции `get_string_features`

Здесь объявляется функция `get_string_features`, принимающая путь к целевому двоичному файлу (1) и экземпляр класса хеширования признаков sklearn (2). Затем регулярное выражение извлекает из файла все печатные строки длиной не менее пяти символов. Признаки сохраняются в словаре Python (3): каждой строке присваивается значение 1, просто указывающее на присутствие признака в файле.

Далее признаки хешируются реализацией sklearn посредством вызова `hasher`. Обратите внимание: словарь `string_features` перед передачей экземпляру хешера оборачивается списком Python (4), потому что sklearn требует список словарей, а не один словарь.

Поскольку словарь передан как список словарей, признаки возвращаются списком массивов. Кроме того, они возвращаются в разреженном формате - сжатом представлении, полезном для обработки больших матриц, но не рассматриваемом в книге. Данные нужно вернуть к обычному вектору numpy.

Для этого вызываются `.todense()` и `.asarray()`, а затем выбирается первый массив списка результатов хешера, что даёт итоговый вектор признаков. Последний шаг функции - вернуть вызывающей стороне вектор `hashed_features` (5).

Обучение детектора

Поскольку основную сложную работу по обучению выполняет sklearn, после извлечения признаков из целевых файлов для обучения детектора требуется немного кода.

Сначала нужно извлечь признаки из обучающих примеров, а затем создать экземпляры хешера и выбранного детектора sklearn, в данном случае классификатора случайного леса. После этого метод `fit` детектора обучает его на двоичных файлах примеров. Наконец, детектор и хешер сохраняются на диск, чтобы впоследствии использовать их для сканирования файлов.

В листинге 8-12 приведён код обучения детектора.

```
def get_training_data(benign_path,malicious_path,hasher):  # (1)
    def get_training_paths(directory):  # (2)
        targets = []
        for path in os.listdir(directory):
            targets.append(os.path.join(directory,path))
        return targets
    malicious_paths = get_training_paths(malicious_path)  # (3)
    benign_paths = get_training_paths(benign_path)  # (4)
    X = [get_string_features(path,hasher)  # (5)
        for path in malicious_paths + benign_paths]
    y = ([1 for i in range(len(malicious_paths))]
        + [0 for i in range(len(benign_paths))])
    return X, y
def train_detector(X,y,hasher):  # (6)
    classifier = tree.RandomForestClassifier()
    classifier.fit(X,y)  # (7)
    pickle.dump((classifier,hasher),open("saved_detector.pkl","w+"))  # (8)
```

Листинг 8-12. Программирование sklearn для обучения детектора

Сначала объявляется функция `get_training_data()` (1), извлекающая признаки из предоставленных обучающих примеров. Она принимает три аргумента: путь к каталогу с безвредными двоичными программами `benign_path`, путь к каталогу с вредоносными программами `malicious_path` и экземпляр класса sklearn `FeatureHasher`, используемый для хеширования, `hasher`.

Затем объявляется локальная вспомогательная функция `get_training_paths()` (2), возвращающая список абсолютных путей к файлам данного каталога. В следующих двух строках с её помощью получаются списки путей из каталогов вредоносных (3) и безвредных (4) обучающих примеров.

Наконец, извлекаются признаки и создаётся вектор меток. Для каждого пути обучающего примера вызывается функция `get_string_features` из листинга 8-11 (5). Обратите внимание: вектор меток содержит 1 для каждого вредоносного пути и 0 для каждого безвредного, поэтому число по каждому индексу вектора меток соответствует метке вектора признаков по тому же индексу массива `X`.

Именно в такой форме sklearn ожидает признаки и метки, позволяя сообщить библиотеке метку каждого вектора.

После завершения извлечения и создания вектора признаков `X` и вектора меток `y` можно приказать sklearn обучить детектор на этих векторах.

Для этого служит функция `train_detector()` (6) с тремя аргументами: векторами признаков обучающих примеров `X`, вектором меток `y` и экземпляром хешера `hasher`. В теле создаётся детектор sklearn `tree.RandomForestClassifier`. Затем `X` и `y` передаются методу `fit` для обучения (7), после чего модуль Python `pickle` (8) сохраняет детектор и хешер для будущего использования.

Запуск детектора для новых двоичных файлов

Теперь рассмотрим применение только что обученного и сохранённого детектора к новым программным файлам. В листинге 8-13 показана соответствующая функция `scan_file()`.

```
def scan_file(path):
    if not os.path.exists("saved_detector.pkl"):
        print "Train a detector before scanning files."
        sys.exit(1)
    with open("saved_detector.pkl") as saved_detector:  # (1)
        classifier, hasher = pickle.load(saved_detector)
    features = get_string_features(path,hasher)  # (2)
    result_proba = classifier.predict_proba(features)[1]  # (3)
    # Если пользователь указал malware_paths и
    # benignware_paths, обучить детектор
    if result_proba > 0.5:  # (4)
        print "It appears this file is malicious!", result_proba
    else:
        print "It appears this file is benign.", result_proba
```

Листинг 8-13. Запуск детектора для новых двоичных файлов

Здесь объявляется функция `scan_file()`, которая сканирует файл и определяет его вредоносность или безвредность. Единственный аргумент - путь к сканируемому файлу. Сначала функция загружает сохранённые детектор и хешер из файла `pickle` (1).

Затем из целевого файла извлекаются признаки функцией `get_string_features` (2), определённой в листинге 8-11.

Наконец, для решения о вредоносности файла с учётом извлечённых признаков вызывается метод детектора `predict`. Точнее, применяется метод `predict_proba` экземпляра классификатора (3) и выбирается второй элемент возвращённого массива, соответствующий вероятности вредоносности. Если она выше 0,5, то есть 50 процентов (4), файл объявляется вредоносным; иначе пользователю сообщается, что он безвреден. Для минимизации ложных срабатываний порог можно сделать гораздо выше.

Что реализовано к этому моменту

В листинге 8-14 приведён полный код этого небольшого, но реалистичного детектора. Надеюсь, после разбора отдельных частей он читается легко.

```
#!/usr/bin/python

import os
import sys
import pickle
import argparse
import re
```

```
import numpy
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_extraction import FeatureHasher

def get_string_features(path,hasher):
    # Извлечь строки из двоичного файла регулярными выражениями
    chars = r" -~"
    min_length = 5
    string_regexp = '[%s]{%d,}' % (chars, min_length)
    file_object = open(path)
    data = file_object.read()
    pattern = re.compile(string_regexp)
    strings = pattern.findall(data)

    # Сохранить строковые признаки в словарной форме
    string_features = {}
    for string in strings:
        string_features[string] = 1

    # Хешировать признаки с помощью трюка хеширования
    hashed_features = hasher.transform([string_features])

    # Преобразовать данные, чтобы получить массив признаков
    hashed_features = hashed_features.todense()
    hashed_features = numpy.asarray(hashed_features)
    hashed_features = hashed_features[0]

    # Вернуть хешированные строковые признаки
    print "Extracted {0} strings from {1}".format(len(string_features),path)
    return hashed_features

def scan_file(path):
    # Просканировать файл и определить его вредоносность или безвредность
    if not os.path.exists("saved_detector.pkl"):
        print "Train a detector before scanning files."
        sys.exit(1)
    with open("saved_detector.pkl") as saved_detector:
        classifier, hasher = pickle.load(saved_detector)
    features = get_string_features(path,hasher)
    result_proba = classifier.predict_proba([features])[:,1]
    # Если пользователь указал malware_paths и
    # benignware_paths, обучить детектор
    if result_proba > 0.5:
        print "It appears this file is malicious!", result_proba
    else:
        print "It appears this file is benign.", result_proba

def train_detector(benign_path,malicious_path,hasher):
    # Обучить детектор на указанных обучающих данных
    def get_training_paths(directory):
        targets = []
        for path in os.listdir(directory):
            targets.append(os.path.join(directory,path))
        return targets
    malicious_paths = get_training_paths(malicious_path)
    benign_paths = get_training_paths(benign_path)
    X = [get_string_features(path,hasher) for path in malicious_paths + benign_paths]
    y = [1 for i in range(len(malicious_paths))] + [0 for i in range(len(benign_paths))]
    classifier = tree.RandomForestClassifier(64)
    classifier.fit(X,y)
    pickle.dump((classifier,hasher),open("saved_detector.pkl","w+"))

def get_training_data(benign_path,malicious_path,hasher):
    def get_training_paths(directory):
        targets = []
        for path in os.listdir(directory):
            targets.append(os.path.join(directory,path))
        return targets
    malicious_paths = get_training_paths(malicious_path)
    benign_paths = get_training_paths(benign_path)
    X = [get_string_features(path,hasher) for path in malicious_paths + benign_paths]
```

```
    y = [1 for i in range(len(malicious_paths))] + [0 for i in range(len(benign_paths))]
    return X, y

parser = argparse.ArgumentParser("get windows object vectors for files")
parser.add_argument("--malware_paths",default=None,help="Path to malware training files")
parser.add_argument("--benignware_paths",default=None,help="Path to benignware training
    files")
parser.add_argument("--scan_file_path",default=None,help="File to scan")
args = parser.parse_args()

hasher = FeatureHasher(20000)
if args.malware_paths and args.benignware_paths:
    train_detector(args.benignware_paths,args.malware_paths,hasher)
elif args.scan_file_path:
    scan_file(args.scan_file_path)
else:
    print "[*] You did not specify a path to scan," \
        " nor did you specify paths to malicious and benign training files" \
        " please specify one of these to use the detector.\n"
    parser.print_help()
```

Листинг 8-14. Код базового детектора вредоносных программ на основе машинного обучения

Написать детектор на основе машинного обучения замечательно, но перед уверенным развёртыванием необходимо оценить и улучшить его работу. Далее рассматриваются разные способы оценки работы детектора.

Оценка работы детектора

К счастью, sklearn содержит код, упрощающий оценку систем с помощью таких показателей, как изученные в главе 7 ROC-кривые. Библиотека также предоставляет дополнительные средства специально для оценки систем машинного обучения. Например, её функции выполняют перекрёстную проверку - мощный метод прогнозирования качества детектора после развёртывания.

В этом разделе вы научитесь с помощью sklearn строить ROC-кривые, показывающие точность детектора, а также познакомитесь с перекрёстной проверкой и её реализацией в sklearn.

Применение ROC-кривых для оценки эффективности детектора

Напомним, ROC-кривые измеряют изменение доли истинно положительных результатов детектора, то есть процента успешно обнаруженных вредоносных программ, и доли ложноположительных результатов, то есть процента безвредных программ, ошибочно отмеченных как вредоносные, при регулировке чувствительности.

Чем выше чувствительность, тем больше ложноположительных результатов, но тем выше доля обнаружения. Чем ниже чувствительность, тем меньше ложных срабатываний, но и меньше обнаружений. Для вычисления ROC-кривой нужен детектор, способный выводить оценку угрозы, причём более высокое значение должно означать большую вероятность вредоносности файла. Реализации деревьев решений, логистической регрессии, k ближайших соседей, случайных лесов и других рассмотренных в книге методов sklearn позволяют выводить такую оценку, отражающую вероятность вредоносности или безвредности файла. Посмотрим, как с помощью ROC-кривой определить точность детектора.

Вычисление ROC-кривых

Чтобы вычислить ROC-кривую для детектора из листинга 8-14, необходимо сделать две вещи: определить постановку эксперимента и реализовать его модулем `metrics` библиотеки sklearn. В базовом эксперименте обучающие примеры делятся пополам: первая половина используется для обучения, а вторая - для вычисления ROC-кривой. Такое разделение имитирует задачу обнаружения вредоносных программ нулевого дня. По сути, мы говорим программе: "Покажи мне один набор безвредных и вредоносных программ, по которому я научусь их распознавать, а затем покажи другой набор и проверь, насколько хорошо я усвоил понятия вредоносной и безвредной программы". Поскольку детектор никогда не видел вредоносных и безвредных программ проверочного набора, этот способ оценки просто прогнозирует его работу с действительно новыми вредоносными программами.

Реализовать разделение в sklearn несложно. Сначала в анализатор аргументов программы детектора добавляется параметр, сообщающий о необходимости оценить точность:

```
parser.add_argument("--evaluate",default=False,
action="store_true",help="Perform cross-validation")
```

Затем в часть программы, обрабатывающую аргументы командной строки, добавляется ещё одна ветвь `elif`, как показано в листинге 8-15. Она обрабатывает случай, когда пользователь добавил к аргументам `--evaluate`.

```
elif args.malware_paths and args.benignware_paths and args.evaluate:
    hasher = FeatureHasher()  # (1)
    X, y = get_training_data(  # (2)
        args.benignware_paths,args.malware_paths,hasher)
    evaluate(X,y,hasher)
def evaluate(X,y,hasher):  # (3)
    import random
    from sklearn import metrics
    from matplotlib import pyplot
```

Листинг 8-15. Запуск детектора для новых двоичных файлов

Разберём код подробнее. Сначала создаётся экземпляр хешера sklearn (1), получаются необходимые эксперименту обучающие данные (2), а затем вызывается функция `evaluate` (3). Она принимает обучающие данные `X`, `y` и экземпляр хешера `hasher`, после чего импортирует три модуля, необходимые для оценки. Модуль `random` случайно выбирает обучающие примеры для обучения и проверки детектора. Модуль `metrics` из sklearn вычисляет ROC-кривую, а модуль `pyplot` из matplotlib, фактического стандарта библиотек визуализации данных Python, отображает её.

Разделение данных на обучающий и проверочный наборы

После случайной сортировки массивов `X` и `y`, соответствующих обучающим данным, их можно разделить на одинаковые по размеру обучающий и проверочный наборы, как показано в продолжающем функцию `evaluate()` листинге 8-16.

```
X, y = numpy.array(X), numpy.array(y)  # (1)
indices = range(len(y))  # (2)
random.shuffle(indices)  # (3)
X, y = X[indices], y[indices]  # (4)
splitpoint = len(X) * 0.5
splitpoint = int(splitpoint)  # (5)
training_X, test_X = X[:splitpoint], X[splitpoint:]  # (6)
training_y, test_y = y[:splitpoint], y[splitpoint:]
```

Листинг 8-16. Разделение данных на обучающий и проверочный наборы

Сначала X и y преобразуются в массивы numpy (1), после чего создаётся список индексов, соответствующий количеству их элементов (2). Индексы случайно перемешиваются (3), а X и y переупорядочиваются согласно новому порядку (4). Это позволяет случайно распределить образцы между обучающим и проверочным наборами, а не разделить их лишь по порядку в каталоге экспериментальных данных. Для завершения случайного разделения определяется индекс, делящий набор пополам, округляется до ближайшего целого функцией `int()` (5), после чего массивы X и y фактически разделяются на обучающие и проверочные наборы (6).

Теперь можно создать экземпляр детектора дерева решений и обучить его на обучающем наборе:

```
classifier = RandomForestClassifier()
classifier.fit(training_X,training_y)
```

Затем обученный классификатор выдаёт для проверочных примеров оценки вероятности их вредоносности:

```
scores = classifier.predict_proba(test_X)[:,-1]
```

Здесь метод `predict_proba()` классификатора предсказывает вероятность безвредности или вредоносности проверочных примеров. Затем с помощью индексирования numpy выбираются только вероятности вредоносности, а не безвредности. Помните, эти вероятности избыточны: например, если вероятность вредоносности примера равна 0,99, то вероятность безвредности равна 0,01, поскольку в сумме они дают 1,00. Поэтому здесь нужна лишь вероятность вредоносности.

Вычисление ROC-кривой

Получив от детектора вероятности вредоносности, которые также можно называть оценками, пора вычислить ROC-кривую. Сначала вызывается функция `roc_curve` модуля `metrics` библиотеки sklearn:

```
fpr, tpr, thresholds = metrics.roc_curve(test_y, scores)
```

Функция `roc_curve` проверяет разные пороги решения, то есть пороги оценок, выше которых двоичный файл считается вредоносным, и измеряет доли ложноположительных и истинно положительных результатов детектора при использовании каждого порога.

Функция принимает два аргумента: вектор меток проверочных примеров `test_y` и массив `scores`, содержащий мнение детектора о вредоносности каждого примера. Она возвращает три связанных массива: `fpr`, `tpr` и `thresholds`. Все они имеют одинаковую длину, поэтому доля ложноположительных результатов, доля истинно положительных и порог решения по каждому индексу соответствуют друг другу.

Теперь вычисленную ROC-кривую можно визуализировать с помощью matplotlib. Для этого вызывается метод `plot` модуля `pyplot`:

```
pyplot.plot(fpr,tpr,'r-')
pyplot.xlabel("Detector false positive rate")
pyplot.ylabel("Detector true positive rate")
pyplot.title("Detector ROC Curve")
pyplot.show()
```

Методы `xlabel`, `ylabel` и `title` задают подписи осей и заголовок графика, а метод `show` открывает окно. Полученная ROC-кривая показана на рис. 8-2.

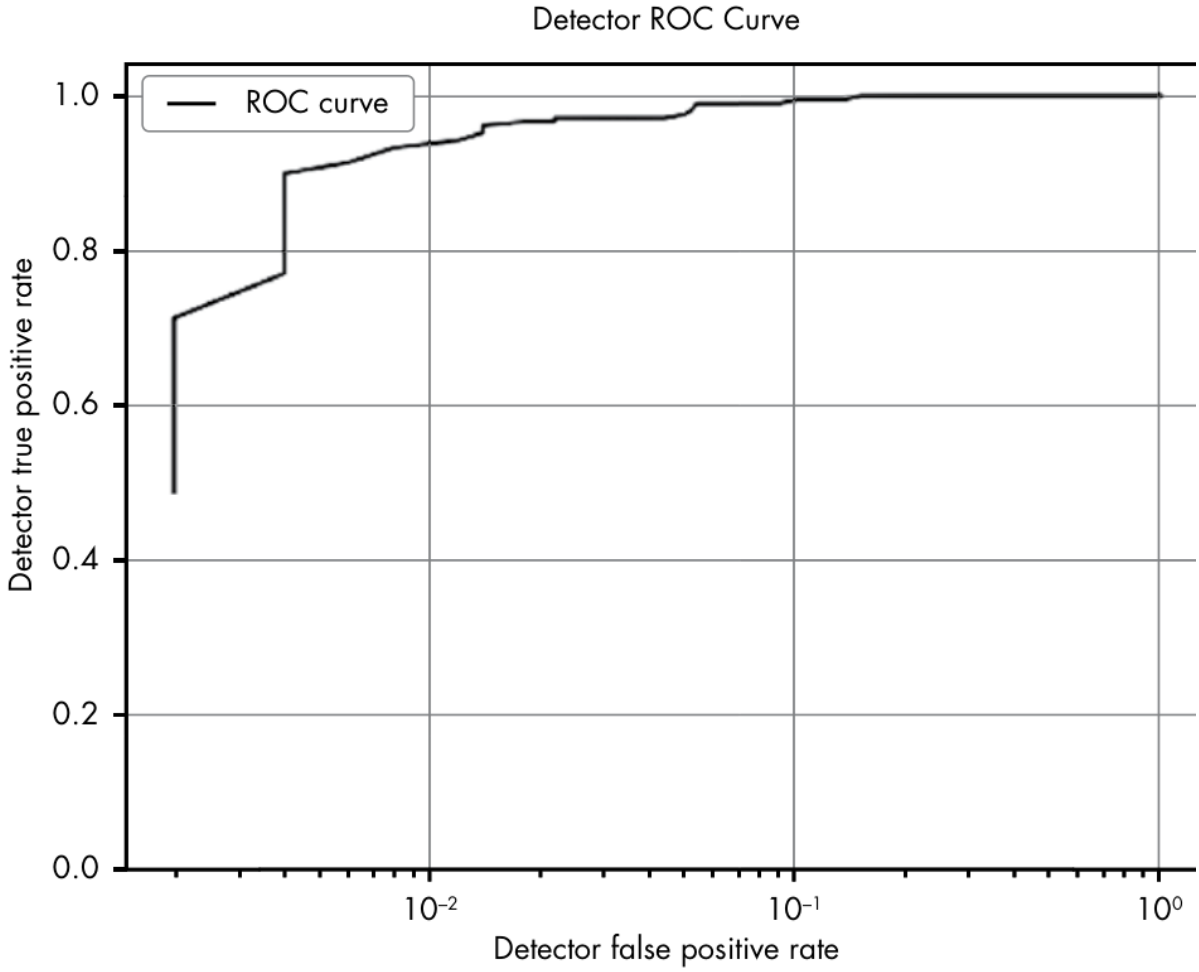

Рисунок 8-2. Визуализация ROC-кривой детектора

Как видно на графике, для столь простого примера детектор работает хорошо. При доле ложноположительных результатов около 1 процента, то есть 10^-2, он обнаруживает примерно 94 процента вредоносных образцов проверочного набора. Здесь он обучается всего на нескольких сотнях примеров; для повышения точности потребовались бы десятки тысяч, сотни тысяч или даже миллионы примеров. Увы, масштабирование машинного обучения до такой степени выходит за рамки книги.

Перекрёстная проверка

Визуализация ROC-кривой полезна, но реальную точность детектора можно предсказать лучше, проведя на обучающих данных множество экспериментов, а не один. Напомним, для проверки мы разделили обучающие примеры пополам, обучили детектор на первой половине и проверили на второй. На самом деле это недостаточная проверка. В реальном мире будет измеряться не точность на конкретном наборе проверочных примеров, а точность на новых, ранее неизвестных вредоносных программах. Чтобы лучше представить будущую работу, нужно провести не один эксперимент на одном проверочном наборе, а множество экспериментов на множестве наборов и оценить общую тенденцию точности.

Для этого служит перекрёстная проверка. Основная идея состоит в разделении обучающих примеров на несколько блоков; здесь используются три, но можно взять больше. Например, если есть 300 примеров и решено создать три блока, первые 100 образцов попадут в первый, вторые

100 - во второй, а третьи 100 - в третий.

Затем проводятся три проверки. В первой система обучается на блоках 2 и 3, а проверяется на блоке 1. Во второй процесс повторяется, но система обучается на блоках 1 и 3, а проверяется на блоке 2. Как вы, вероятно, уже догадались, в третьей система обучается на блоках 1 и 2, а проверяется на блоке 3. Этот процесс показан на рис. 8-3.

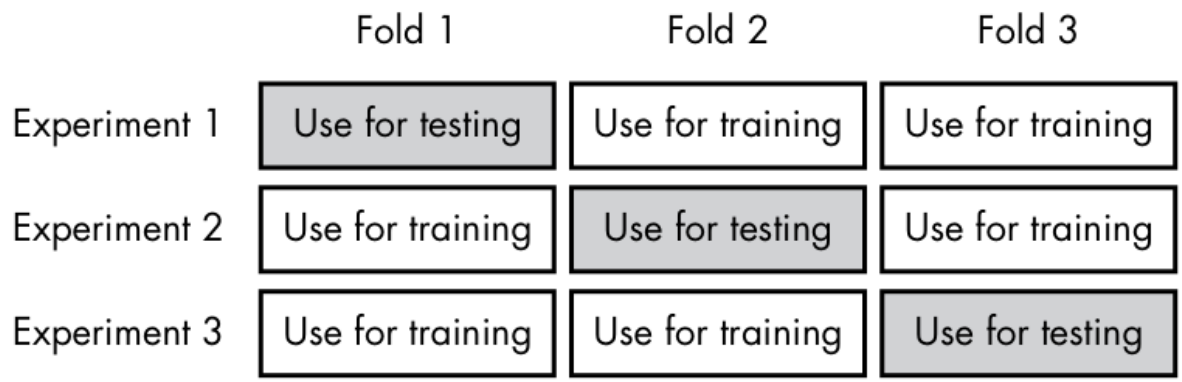

Рисунок 8-3. Пример процесса перекрёстной проверки

Sklearn упрощает реализацию перекрёстной проверки. Для этого перепишем функцию `evaluate` из листинга 8-15 под именем `cv_evaluate`.

```
def cv_evaluate(X,y,hasher):
    import random
    from sklearn import metrics
    from matplotlib import pyplot
    from sklearn.cross_validation import KFold
```

Функция `cv_evaluate()` начинается так же, как исходная функция оценки, но дополнительно импортируется класс `KFold` из модуля `cross_validation` библиотеки sklearn. K-блочная перекрёстная проверка, сокращённо `KFold`, равнозначна только что описанному виду и представляет собой самый распространённый способ перекрёстной проверки.

Затем обучающие данные преобразуются в массивы numpy, чтобы воспользоваться расширенным индексированием массивов:

```
X, y = numpy.array(X), numpy.array(y)
```

Следующий код фактически начинает перекрёстную проверку:

```
fold_counter = 0
for train, test in KFold(len(X),3,shuffle=True):  # (1)
    training_X, training_y = X[train], y[train]  # (2)
    test_X, test_y = X[test], y[test]
```

Сначала создаётся экземпляр класса `KFold`: первым параметром передаётся количество обучающих примеров, а вторым - требуемое количество блоков. Третий аргумент `shuffle=True` (1) приказывает sklearn случайно отсортировать обучающие данные перед разделением на три блока. Экземпляр `KFold` фактически представляет собой итератор, на каждой итерации выдающий новое разделение обучающих и проверочных примеров. Внутри цикла `for` соответствующие элементы назначаются массивам обучающих и проверочных экземпляров `training_X`, `training_y`, `test_X` и `test_y` (2).

После подготовки данных можно создать и обучить `RandomForestClassifier`, как вы уже делали в этой главе:

```
classifier = RandomForestClassifier()
classifier.fit(training_X,training_y)
```

Наконец, для данного блока вычисляется ROC-кривая и строится представляющая её линия:

```
scores = classifier.predict_proba(test_X)[:,-1]
fpr, tpr, thresholds = metrics.roc_curve(test_y, scores)
pyplot.semilogx(fpr,tpr,label="Fold number {0}".format(fold_counter))
fold_counter += 1
```

Обратите внимание: метод matplotlib show пока не вызывается. Он будет вызван после оценки всех блоков, когда можно будет показать три линии одновременно. Как и в предыдущем разделе, подпишем оси и дадим графику заголовок:

```
pyplot.xlabel("Detector false positive rate")
pyplot.ylabel("Detector true positive rate")
pyplot.title("Detector Cross-Validation ROC Curves")
pyplot.legend()
pyplot.grid()
pyplot.show()
```

Полученная ROC-кривая показана на рис. 8-4.

Как видите, результаты на всех блоках похожи, хотя между ними определённо есть различия. При доле ложноположительных результатов 1 процент средняя доля обнаружения, то есть доля истинно положительных результатов, в трёх запусках составляет около 90 процентов. Эта оценка учитывает все три эксперимента и точнее оценки, полученной одним экспериментом. В последнем случае результат в некоторой степени зависел бы от случайного выбора образцов для обучения и проверки. Большее количество экспериментов позволяет надёжнее оценить эффективность решения.

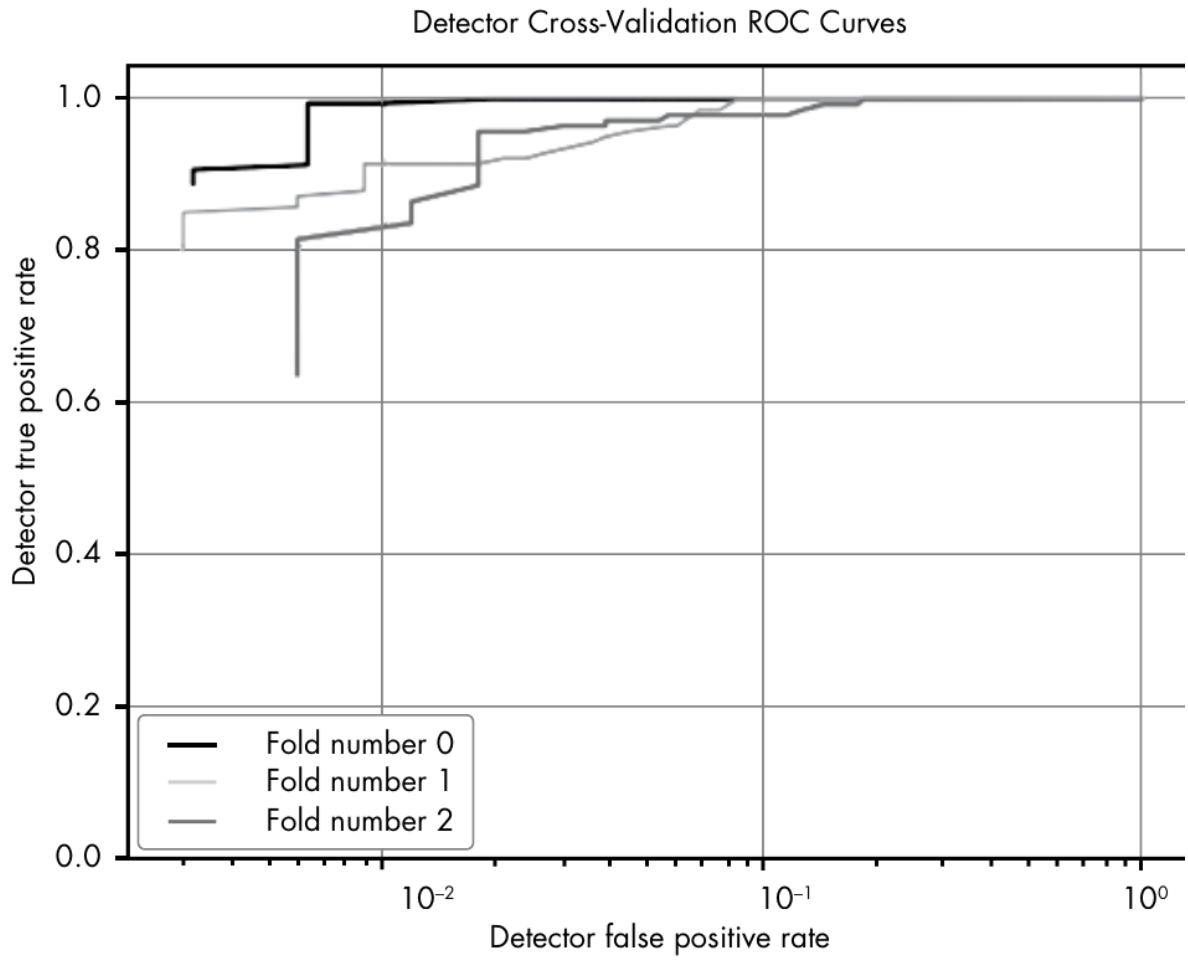

Рисунок 8-4. Построение ROC-кривой детектора с помощью перекрёстной проверки

Обратите внимание: эти результаты не выдающиеся, потому что обучение проводится на очень небольшом объёме данных - нескольких сотнях вредоносных и безвредных образцов. На моей основной работе, где мы обучаем крупномасштабные системы обнаружения, обычно используются сотни миллионов образцов. Для собственного детектора сотни миллионов не нужны, но для действительно хорошей работы стоит собрать не меньше десятков тысяч образцов, например чтобы достичь 90-процентной доли обнаружения при доле ложноположительных результатов 0,1 процента.

Дальнейшие шаги

До сих пор рассматривалось применение Python и sklearn для извлечения признаков из обучающего набора программных двоичных файлов, а затем обучения и оценки метода машинного обучения на основе дерева решений. Систему можно улучшить, используя вместо печатных строк или вместе с ними другие признаки, например признаки заголовка PE, N-граммы инструкций или таблицу адресов импорта, рассмотренные выше, либо применив другой алгоритм машинного обучения.

Для повышения точности я советую выйти за рамки `RandomForestClassifier` библиотеки sklearn, то есть `sklearn.ensemble.RandomForestClassifier`, и попробовать другие классификаторы. Напомним из предыдущей главы, что детекторы случайного леса тоже основаны на деревьях решений, но строят не одно дерево, а множество, случайным образом изменяя способ построения. Чтобы определить вредоносность или безвредность нового файла, каждое дерево принимает отдельное решение. Эти решения складываются и делятся на общее количество деревьев, что даёт средний результат.

Можно использовать и другие алгоритмы sklearn, например логистическую регрессию. Переход к любому из них может сводиться к поиску и замене в примере кода этой главы. Например, здесь дерево решений создаётся и обучается так:

```
classifier = RandomForestClassifier()
classifier.fit(training_X,training_y)
```

Но этот код можно заменить следующим:

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
classifier.fit(training_X,training_y)
```

В результате вместо детектора на основе дерева решений получится детектор логистической регрессии. Выполнив для него новую оценку посредством перекрёстной проверки и сравнив её с результатами на рис. 8-4, можно определить, какой метод работает лучше.

Итоги

В этой главе вы подробно изучили создание детекторов вредоносных программ на основе машинного обучения. В частности, вы узнали, как извлекать из программных двоичных файлов признаки для машинного обучения, сжимать их с помощью трюка хеширования и обучать детекторы на извлечённых признаках. Вы также научились строить ROC-кривые, чтобы исследовать связь порога обнаружения с долями истинно положительных и ложноположительных результатов. Наконец, вы познакомились с более сложным понятием оценки - перекрёстной проверкой - и возможными расширениями детектора из этой главы.

На этом обсуждение обнаружения вредоносных программ средствами sklearn завершается. В главах 10 и 11 будет рассмотрен другой набор методов машинного обучения, известный как глубокое обучение или искусственные нейронные сети. Теперь вы располагаете базовыми знаниями, необходимыми для эффективного применения машинного обучения к идентификации вредоносных программ.

Я призываю вас продолжать читать о машинном обучении. Поскольку компьютерная безопасность во многом представляет собой задачу анализа данных, машинное обучение останется в отрасли и будет и далее полезно не только для обнаружения вредоносных двоичных файлов, но и для выявления вредоносного поведения в сетевом трафике, системных журналах и других контекстах.

В следующей главе мы глубоко погрузимся в визуализацию связей между вредоносными программами, которая помогает быстро понять сходства и различия между большим количеством образцов.

Глава 9. Визуализация тенденций вредоносных программ

Иногда лучший способ проанализировать коллекцию вредоносных программ - визуализировать её. Визуализация данных безопасности позволяет быстро распознавать тенденции как внутри вредоносных программ, так и в общей картине угроз. Такие представления часто гораздо понятнее невизуальной статистики и помогают доносить выводы до самой разной аудитории. Например, в этой главе вы увидите, как визуализация помогает выявлять типы вредоносных программ, преобладающие в наборе данных, тенденции внутри наборов, например распространение программ-вымогателей в 2016 году, и сравнительную эффективность коммерческих антивирусов в обнаружении вредоносных программ.

Работая с этими примерами, вы научитесь создавать собственные визуализации, способные приводить к ценным выводам, с помощью пакета анализа данных Python pandas и пакетов визуализации seaborn и matplotlib. Pandas применяется главным образом для загрузки и обработки данных и сам по себе не слишком связан с визуализацией, но очень полезен для подготовки данных к ней.

Почему визуализация данных о вредоносных программах важна

Чтобы увидеть пользу визуализации, рассмотрим два примера. Первая визуализация отвечает на вопрос: улучшается ли способность антивирусной отрасли обнаруживать программы-вымогатели? Вторая показывает, какие типы вредоносных программ становились распространёнными в течение года. Начнём с первого примера на рис. 9-1.

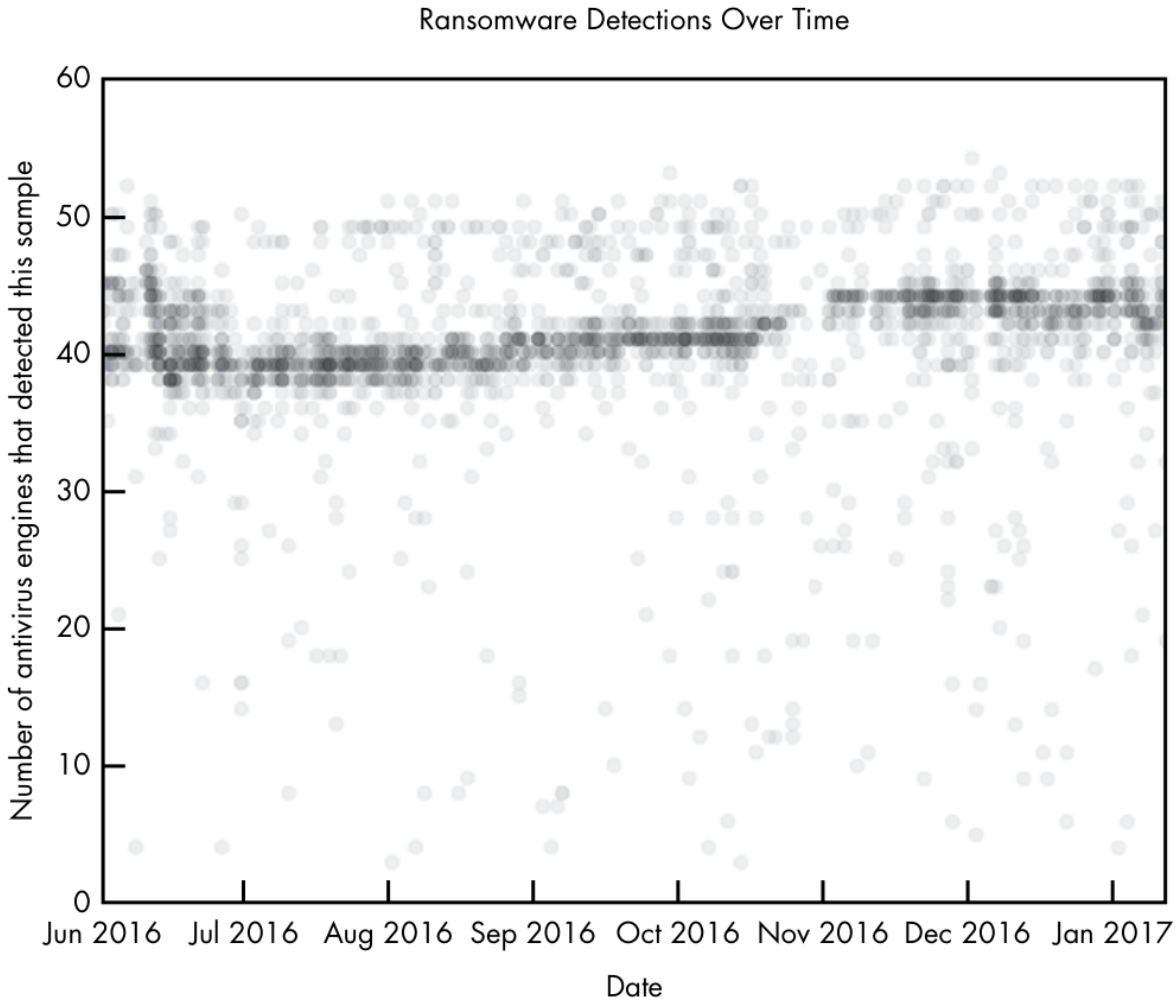

Рисунок 9-1. Визуализация обнаружения программ-вымогателей во времени

Я создал эту визуализацию по данным, собранным для тысяч образцов программ-вымогателей. Данные содержат результаты проверки каждого файла 57 отдельными антивирусными механизмами. Каждый кружок обозначает образец вредоносной программы. Ось y показывает количество обнаружений, или положительных результатов, полученных образцом от антивирусных механизмов при сканировании. Хотя ось y заканчивается значением 60, максимальное количество для одного сканирования равно 57 - общему числу сканеров. Ось x показывает дату, когда каждый образец впервые появился на сайте анализа вредоносных программ VirusTotal.com и был просканирован.

График показывает, что в июне 2016 года способность антивирусного сообщества обнаруживать эти вредоносные файлы была сравнительно высокой, примерно в июле снизилась, а затем до конца года устойчиво росла. К концу 2016 года файлы программ-вымогателей по-прежнему в среднем пропускали около 25 процентов антивирусных механизмов, поэтому можно заключить, что в тот период сообщество безопасности всё ещё обнаруживало их довольно слабо.

Исследование можно продолжить визуализацией того, какие антивирусные механизмы обнаруживают программы-вымогатели, с какой частотой и как их показатели улучшаются со временем. Можно изучить и другую категорию вредоносных программ, например трояны. Такие графики полезны при выборе антивирусного механизма для покупки или типа вредоносных программ, для которого стоит создать собственное средство обнаружения, возможно в дополнение к коммерческому антивирусу. Подробнее о создании собственных средств рассказано в главе 8.

Теперь рассмотрим рис. 9-2 - ещё одну визуализацию, созданную по тому же набору данных, что и рис. 9-1.

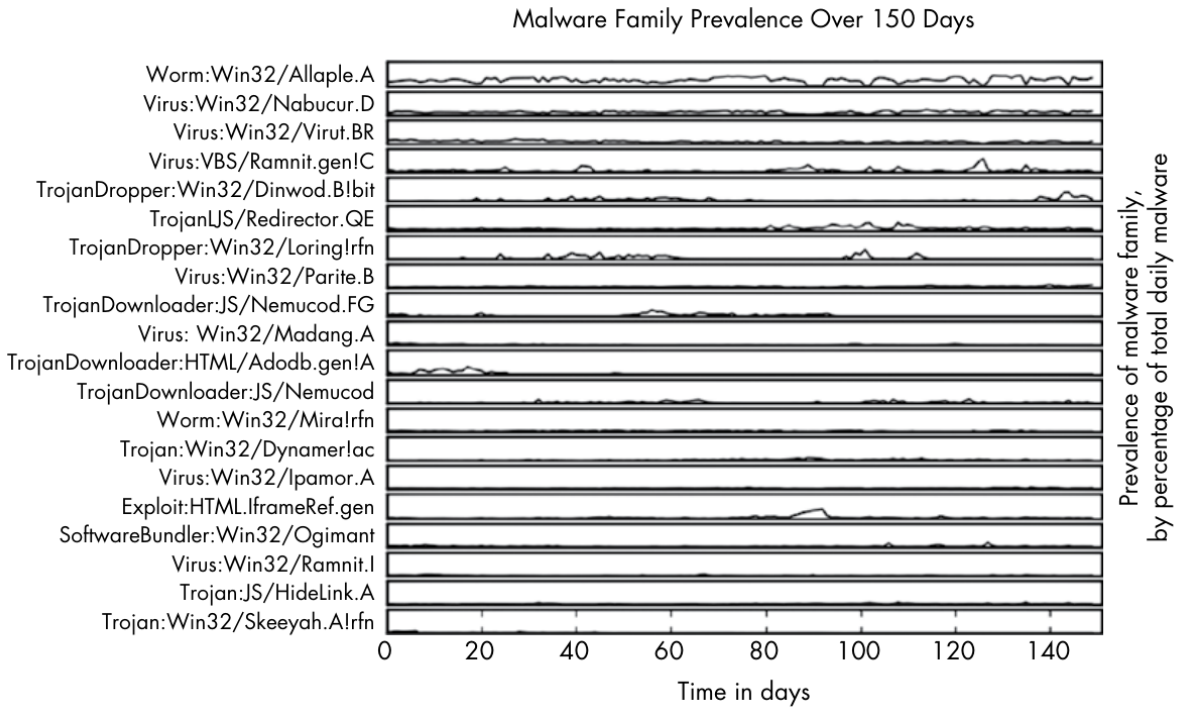

Рисунок 9-2. Визуализация обнаружений отдельных семейств вредоносных программ во времени

На рис. 9-2 показаны 20 самых распространённых семейств и их частота относительно друг друга за 150 дней. График раскрывает несколько важных фактов. Самое распространённое семейство Allaple.A встречалось постоянно в течение всех 150 дней, а другие семейства, например Nemucod.FG, были распространены в течение более коротких промежутков, а затем исчезали. Такой график, построенный по вредоносным программам, обнаруженным в сети вашей организации, может выявить полезные тенденции и показать, какие типы участвовали в атаках с течением времени. Без подобного сравнительного рисунка понять и сопоставить относительные пики и объёмы этих типов во времени было бы трудно и долго.

Эти два примера показывают пользу визуализации вредоносных программ. В оставшейся части главы вы научитесь создавать собственные визуализации. Сначала обсудим используемый учебный набор, затем проанализируем данные с помощью pandas и, наконец, визуализируем их пакетами matplotlib и seaborn.

Знакомство с набором данных о вредоносных программах

Используемый набор содержит сведения о 37 000 уникальных двоичных файлах, собранных сервисом агрегации обнаружений VirusTotal. Каждый файл описан четырьмя полями: количеством антивирусных механизмов из 57, отметивших его как вредоносный, которое я называю количеством положительных результатов образца; размером файла; типом, то есть майнером биткоинов, кейлоггером, программой-вымогателем, трояном или червём; и датой первого обнаружения. Мы увидим, что даже такой сравнительно небольшой объём метаданных позволяет анализировать и визуализировать набор, получая важные выводы.

Загрузка данных в pandas

Популярная библиотека анализа данных Python pandas упрощает загрузку данных в аналитические объекты `DataFrame`, а затем предоставляет методы для выборки, преобразования и анализа переупакованных данных. Мы воспользуемся pandas для загрузки и анализа и подготовим данные к удобной визуализации. В листинге 9-1 определяется и загружается небольшой пример данных в интерпретатор Python.

```
In [135]: import pandas

In [136]: example_data = [{'column1': 1, 'column2': 2},  # (1)
     ...:  {'column1': 10, 'column2': 32},
     ...:  {'column1': 3, 'column2': 58}]

In [137]: pandas.DataFrame(example_data)  # (2)
Out[137]:
   column1  column2
0        1        2
1       10       32
2        3       58
```

Листинг 9-1. Непосредственная загрузка данных в pandas

Здесь данные `example_data` определяются как список словарей Python (1). После создания списка он передаётся конструктору `DataFrame` (2), и получается соответствующий объект pandas. Каждый словарь становится строкой итогового `DataFrame`, а его ключи `column1` и `column2` - столбцами. Это один из способов загрузить данные непосредственно в pandas.

Можно также загрузить данные из внешних CSV-файлов. Код листинга 9-2 загружает набор этой главы, доступный на виртуальной машине и в архиве кода и данных к книге.

```
import pandas
malware = pandas.read_csv("malware_data.csv")
```

Листинг 9-2. Загрузка данных в pandas из внешнего CSV-файла

После импорта `malware_data.csv` объект `malware` должен выглядеть примерно так:

```
      positives      size        type            fs_bucket
0            45    251592      trojan  2017-01-05 00:00:00
1            32    227048      trojan  2016-06-30 00:00:00
2            53    682593        worm  2016-07-30 00:00:00
3            39    774568      trojan  2016-06-29 00:00:00
4            29    571904      trojan  2016-12-24 00:00:00
5            31    582352      trojan  2016-09-23 00:00:00
6            50   2031661        worm  2017-01-04 00:00:00
```

Теперь у нас есть `DataFrame` pandas с набором вредоносных программ. Он содержит четыре столбца: `positives`, количество обнаружений образца антивирусными механизмами из 57; `size`, количество байтов, занимаемых файлом на диске; `type`, тип вредоносной программы, например троян или червь; и `fs_bucket`, дата первого обнаружения.

Работа с DataFrame pandas

Данные находятся в `DataFrame`; теперь посмотрим, как обращаться к ним и обрабатывать их вызовом метода `describe()`, показанным в листинге 9-3.

```
In [51]: malware.describe()
Out[51]:
         positives          size
count  37511.000000  3.751100e+04
mean      39.446536  1.300639e+06
std       15.039759  3.006031e+06
```

```
min        3.000000  3.370000e+02
25%       32.000000  1.653960e+05
50%       45.000000  4.828160e+05
75%       51.000000  1.290056e+06
max       57.000000  1.294244e+08
```

Листинг 9-3. Вызов метода `describe()`

Как показано в листинге, `describe()` выводит полезную статистику `DataFrame`. Первая строка `count` содержит общее количество ненулевых строк `positives` и общее количество ненулевых строк. Вторая строка показывает среднее количество положительных результатов на образец и средний размер образцов. Далее следуют стандартные отклонения `positives` и `size`, а также минимальное значение каждого столбца среди всех образцов. Наконец, выводятся процентили и максимальные значения столбцов.

Предположим, требуется получить данные одного столбца `DataFrame`, например `positives`, чтобы увидеть среднее количество обнаружений каждого файла или построить гистограмму распределения положительных результатов. Достаточно написать `malware['positives']`, и столбец вернётся как список чисел, показанный в листинге 9-4.

```
In [3]: malware['positives']
Out[3]:
0        45
1        32
2        53
3        39
4        29
5        31
6        50
7        40
8        20
9        40
--пропущено--
```

Листинг 9-4. Возврат столбца `positives`

После получения столбца статистику можно вычислять непосредственно. Например, `malware['positives'].mean()` вычисляет среднее, `malware['positives'].max()` - максимум, `malware['positives'].min()` - минимум, а `malware['positives'].std()` - стандартное отклонение. Примеры приведены в листинге 9-5.

```
In [7]: malware['positives'].mean()
Out[7]: 39.446535682866362

In [8]: malware['positives'].max()
Out[8]: 57

In [9]: malware['positives'].min()
Out[9]: 3

In [10]: malware['positives'].std()
Out[10]: 15.039759380778822
```

Листинг 9-5. Вычисление среднего, максимума, минимума и стандартного отклонения

Для более подробного анализа данные можно также разрезать на части. Например, листинг 9-6 вычисляет среднее количество положительных результатов для типов вредоносных программ `trojan`, `bitcoin` и `worm`.

```
In [67]: malware[malware['type'] == 'trojan']['positives'].mean()
Out[67]: 33.43822473365119

In [68]: malware[malware['type'] == 'bitcoin']['positives'].mean()
Out[68]: 35.857142857142854
In [69]: malware[malware['type'] == 'worm']['positives'].mean()
Out[69]: 49.90857904874796
```

Листинг 9-6. Вычисление средних долей обнаружения разных вредоносных программ

Сначала выбираются строки `DataFrame`, где `type` равен `trojan`, с помощью записи `malware[malware['type'] == 'trojan']`. Чтобы выбрать столбец `positives` полученных данных и вычислить среднее, выражение расширяется до `malware[malware['type'] == 'trojan']['positives'].mean()`. Листинг 9-6 даёт интересный результат: черви обнаруживаются чаще майнеров биткоинов и троянов. Поскольку 49,9 > 35,8 и 33,4, вредоносные образцы червей в среднем обнаруживает больше поставщиков, 49,9, чем образцы майнеров и троянов, 35,8 и 33,4.

Фильтрация данных по условиям

Подмножества данных можно выбирать и с помощью других условий. Например, для числовых данных, таких как размер вредоносного файла, можно использовать условия "больше" и "меньше", а затем вычислять статистику полученных подмножеств. Это полезно, если требуется выяснить, связана ли эффективность антивирусных механизмов с размером файла. Проверим это кодом листинга 9-7.

```
In [84]: malware[malware['size'] > 1000000]['positives'].mean()
Out[84]: 33.507073192162373

In [85]: malware[malware['size'] > 2000000]['positives'].mean()
Out[85]: 32.761442050415432

In [86]: malware[malware['size'] > 3000000]['positives'].mean()
Out[86]: 27.20672682526661

In [87]: malware[malware['size'] > 4000000]['positives'].mean()
Out[87]: 25.652548725637182

In [88]: malware[malware['size'] > 5000000]['positives'].mean()
Out[88]: 24.411069317571197
```

Листинг 9-7. Фильтрация результатов по размеру вредоносного файла

Возьмём первую строку кода. Сначала `DataFrame` ограничивается образцами размером больше миллиона: `malware[malware['size'] > 1000000]`. Затем берётся столбец `positives` и вычисляется среднее, `['positives'].mean()`, равное примерно 33,5. По мере повторения операции для всё больших файлов среднее количество обнаружений в каждой группе снижается. Мы выяснили, что между размером вредоносного файла и средним количеством обнаруживающих его антивирусных механизмов действительно существует связь. Это интересный результат, заслуживающий дальнейшего исследования. Далее мы изучим его визуально с помощью matplotlib и seaborn.

Визуализация данных с помощью matplotlib

Основная библиотека визуализации данных Python - matplotlib. Фактически большинство других библиотек визуализации Python представляют собой удобные оболочки над ней. Использовать matplotlib с pandas просто: pandas извлекает, разрезает и обрабатывает данные, а matplotlib строит график. Самая полезная для наших целей функция matplotlib - `plot`. На рис. 9-3 показано, что она умеет.

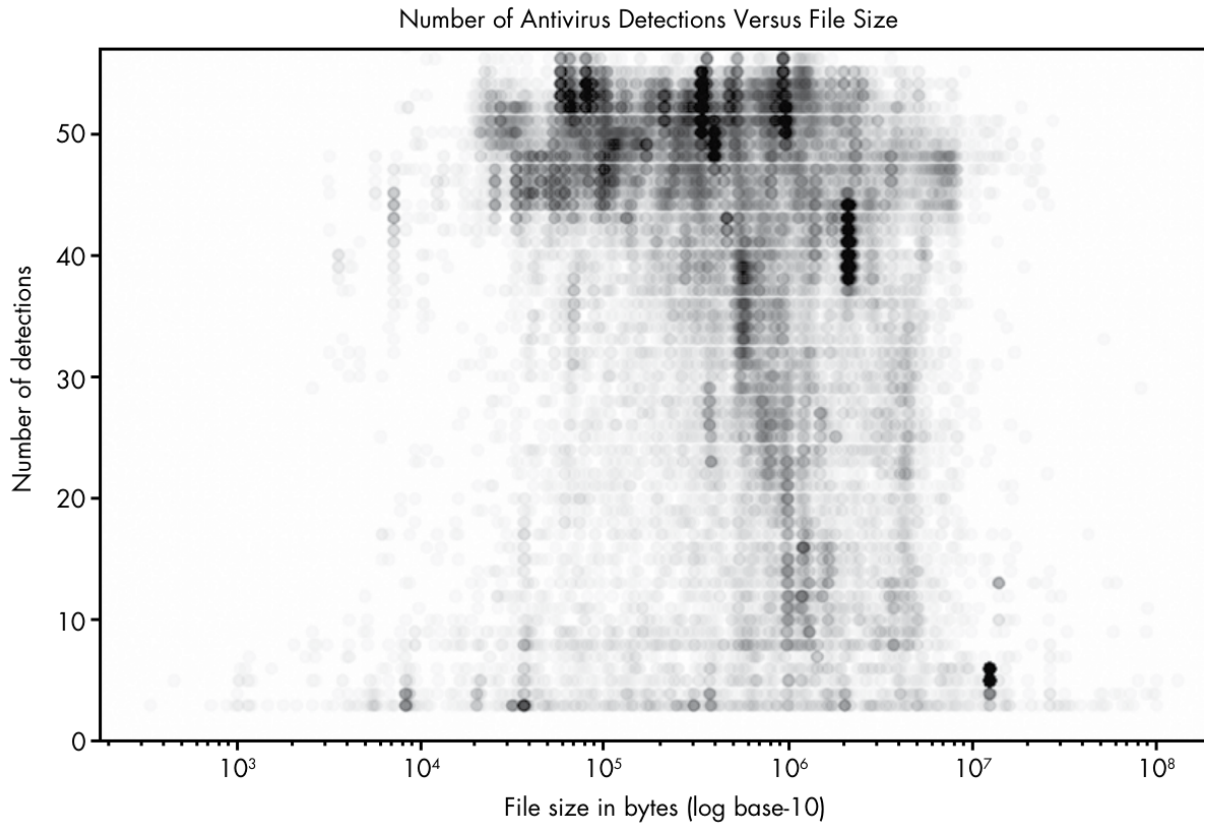

Рисунок 9-3. График размеров образцов вредоносных программ и количества антивирусных обнаружений

Здесь показаны атрибуты `positives` и `size` набора вредоносных программ. Как и предвещало обсуждение pandas в предыдущем разделе, обнаруживается интересный результат. Маленькие и очень большие файлы редко выявляются большинством из 57 проверивших их антивирусных механизмов. Файлы среднего размера, примерно от 10^4,5 до 10^7, обнаруживаются большинством механизмов. Возможно, небольшие файлы содержат недостаточно информации, чтобы механизмы определили их вредоносность, а проверка больших слишком медленна, поэтому многие антивирусы вообще отказываются их сканировать.

Построение связи между размером вредоносной программы и обнаружениями поставщиков

Разберём построение графика с рис. 9-3 по коду листинга 9-8.

```
import pandas  # (1)
from matplotlib import pyplot
malware = pandas.read_csv("malware_data.csv")  # (2)
pyplot.plot(malware['size'], malware['positives'],  # (3) (4)
            'bo', alpha=0.01)  # (5) (6)
pyplot.xscale("log")  # (7)
pyplot.ylim([0,57])  # (8)
pyplot.xlabel("File size in bytes (log base-10)")
pyplot.ylabel("Number of detections")
pyplot.title("Number of Antivirus Detections Versus File Size")
pyplot.show()  # (9)
```

Листинг 9-8. Визуализация данных функцией `plot()`

Для построения требуется совсем немного кода. Разберём каждую строку. Сначала импортируются необходимые библиотеки (1): pandas и модуль `pyplot` библиотеки matplotlib. Затем функция

`read_csv` (2), с которой вы уже знакомы, загружает набор вредоносных программ в `DataFrame` pandas.

Далее вызывается `plot()`. Первый аргумент - данные о размере вредоносных файлов (3), второй - данные `positives` (4), то есть количество положительных обнаружений каждого образца. Эти аргументы определяют отображаемые matplotlib данные: первый соответствует оси x, второй - оси y. Следующий аргумент `'bo'` (5) сообщает цвет и форму данных. Наконец, параметр `alpha`, то есть прозрачность кружков, устанавливается равным 0,1 (6), чтобы плотность данных была видна в разных областях графика даже при полном наложении кружков.

> **Примечание.** Буква `b` в `bo` означает синий цвет, а `o` - кружок: matplotlib получает указание представлять данные синими кружками. Можно также попробовать зелёный (`g`), красный (`r`), голубой (`c`), пурпурный (`m`), жёлтый (`y`), чёрный (`k`) и белый (`w`) цвета. Другие формы включают точку (`.`), один пиксель на точку данных (`,`), квадрат (`s`) и пятиугольник (`p`). Полные сведения приведены в документации matplotlib по адресу matplotlib.org.

После вызова `plot()` масштаб оси x устанавливается логарифмическим (7). Поэтому размер файлов отображается как степени числа 10, что упрощает наблюдение связей между очень маленькими и очень большими файлами.

После построения данных подписываются оси и задаётся заголовок. Ось x представляет размер вредоносного файла, `"File size in bytes (log base-10)"`, а ось y - количество обнаружений, `"Number of detections"`. Поскольку анализируются 57 антивирусных механизмов, шкала оси y устанавливается в диапазон от 0 до 57 (8). Наконец, функция `show()` (9) отображает график. Если бы требовалось сохранить изображение, этот вызов можно было бы заменить на `pyplot.savefig("myplot.png")`.

Разобрав первый пример, перейдём к следующему.

Построение долей обнаружения программ-вымогателей

Теперь попробуем воспроизвести рис. 9-1 с обнаружениями программ-вымогателей, показанный в начале главы. Листинг 9-9 содержит весь код построения обнаружений во времени.

```
import dateutil
import pandas
from matplotlib import pyplot

malware = pandas.read_csv("malware_data.csv")
malware['fs_date'] = [dateutil.parser.parse(d) for d in malware['fs_bucket']]
ransomware = malware[malware['type'] == 'ransomware']
pyplot.plot(ransomware['fs_date'], ransomware['positives'], 'ro', alpha=0.05)
pyplot.title("Ransomware Detections Over Time")
pyplot.xlabel("Date")
pyplot.ylabel("Number of antivirus engine detections")
pyplot.show()
```

Листинг 9-9. Построение долей обнаружения программ-вымогателей во времени

Некоторые части кода уже знакомы, другие - нет. Разберём его построчно:

```
import dateutil
```

Полезный пакет Python dateutil позволяет легко разбирать даты во множестве форматов. Он импортируется потому, что для визуализации потребуется разбирать даты.

```
import pandas
from matplotlib import pyplot
```

Также импортируются модуль `pyplot` библиотеки matplotlib и pandas.

```
malware = pandas.read_csv("malware_data.csv")
malware['fs_date'] = [dateutil.parser.parse(d) for d in malware['fs_bucket']]
ransomware = malware[malware['type'] == 'ransomware']
```

Эти строки читают набор и создают отфильтрованный набор `ransomware`, содержащий только образцы программ-вымогателей, поскольку именно эти данные требуется построить.

```
pyplot.plot(ransomware['fs_date'], ransomware['positives'], 'ro', alpha=0.05)
pyplot.title("Ransomware Detections Over Time")
pyplot.xlabel("Date")
pyplot.ylabel("Number of antivirus engine detections")
pyplot.show()
```

Эти пять строк повторяют код листинга 9-8: строят данные, задают заголовок, подписывают оси x и y и выводят всё на экран, как на рис. 9-4. Чтобы сохранить график на диск, вызов `pyplot.show()` снова можно заменить на `pyplot.savefig("myplot.png")`.

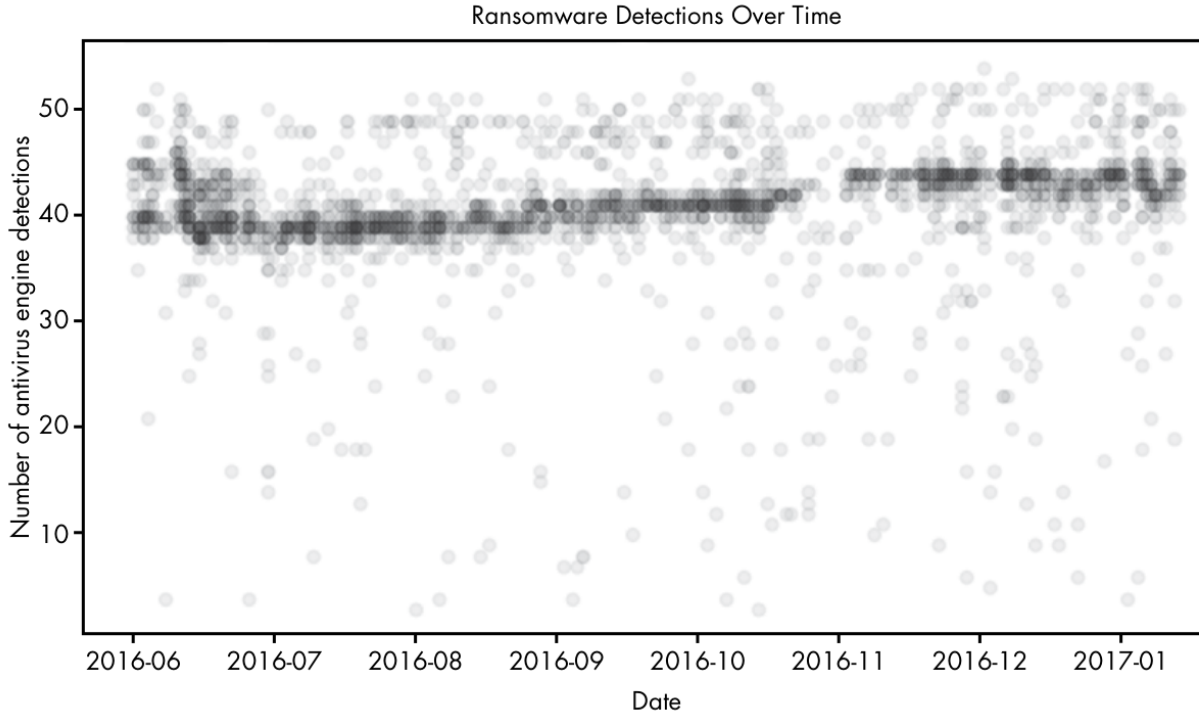

Рисунок 9-4. Визуализация обнаружения программ-вымогателей во времени

Построим функцией `plot()` ещё один график.

Построение долей обнаружения программ-вымогателей и червей

На этот раз вместе с обнаружениями программ-вымогателей построим обнаружения червей. Рисунок 9-5 ясно показывает, что антивирусная отрасль лучше обнаруживает червей, старую тенденцию вредоносных программ, чем программы-вымогатели, более новую тенденцию.

График показывает количество антивирусных механизмов, обнаруживших образцы, то есть ось y, во времени, то есть по оси x. Каждая красная точка обозначает образец с `type="ransomware"`, а синяя - образец с `type="worm"`. В среднем червей обнаруживает больше механизмов, чем программы-вымогатели. Однако количество механизмов, обнаруживающих образцы обоих типов, со временем медленно растёт.

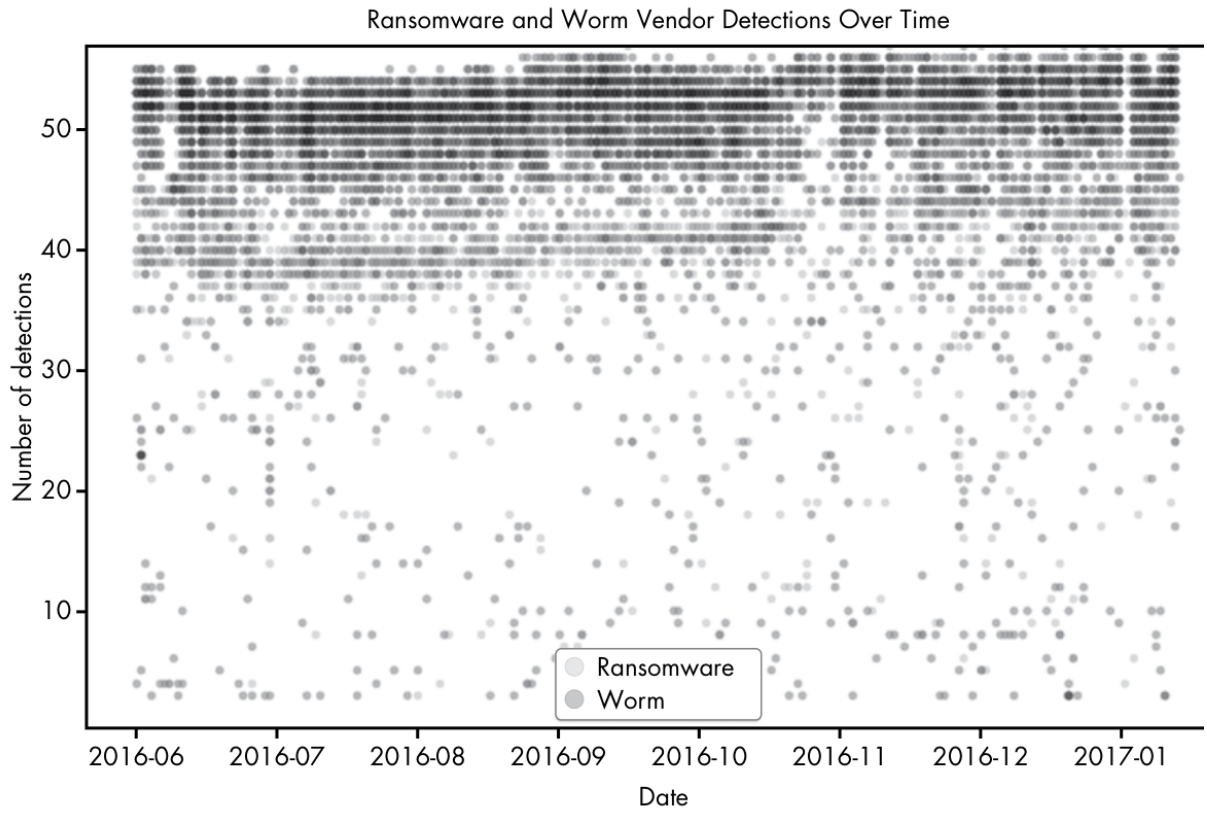

Рисунок 9-5. Визуализация обнаружения программ-вымогателей и червей во времени

Код построения приведён в листинге 9-10.

```
import dateutil
import pandas
from matplotlib import pyplot

malware = pandas.read_csv("malware_data.csv")
malware['fs_date'] = [dateutil.parser.parse(d) for d in malware['fs_bucket']]

ransomware = malware[malware['type'] == 'ransomware']
worms = malware[malware['type'] == 'worm']

pyplot.plot(ransomware['fs_date'], ransomware['positives'],
            'ro', label="Ransomware", markersize=3, alpha=0.05)
pyplot.plot(worms['fs_date'], worms['positives'],
            'bo', label="Worm", markersize=3, alpha=0.05)
pyplot.legend(framealpha=1, markerscale=3.0)
pyplot.xlabel("Date")
pyplot.ylabel("Number of detections")
pyplot.ylim([0, 57])
pyplot.title("Ransomware and Worm Vendor Detections Over Time")
pyplot.show()
```

Листинг 9-10. Построение долей обнаружения программ-вымогателей и червей во времени

Разберём код, начав с первой части листинга 9-10:

```
import dateutil
import pandas
from matplotlib import pyplot

malware = pandas.read_csv("malware_data.csv")
malware['fs_date'] = [dateutil.parser.parse(d) for d in malware['fs_bucket']]

ransomware = malware[malware['type'] == 'ransomware']
worms = malware[malware['type'] == "worm"]  # (1)
--пропущено--
```

Код похож на предыдущий пример. Пока единственное отличие состоит в создании отфильтрованной версии данных о червях (1) тем же методом, которым создаются данные о программах-вымогателях. Рассмотрим оставшуюся часть:

```
--пропущено--
pyplot.plot(ransomware['fs_date'], ransomware['positives'],  # (1)
            'ro', label="Ransomware", markersize=3, alpha=0.05)
pyplot.plot(worms['fs_bucket'], worms['positives'],  # (2)
            'bo', label="Worm", markersize=3, alpha=0.05)
pyplot.legend(framealpha=1, markerscale=3.0)  # (3)
pyplot.xlabel("Date")
pyplot.ylabel("Number of detections")
pyplot.ylim([0,57])
pyplot.title("Ransomware and Worm Vendor Detections Over Time")
pyplot.show()
pyplot.gcf().clf()
```

Главное отличие от листинга 9-9 заключается в двойном вызове `plot()`: сначала для программ-вымогателей с селектором `ro` (1), создающим красные кружки, а затем для червей с селектором `bo` (2), создающим синие. При желании таким же образом можно построить третий набор данных. Кроме того, в отличие от листинга 9-9 здесь создаётся легенда (3), показывающая, что синие отметки соответствуют червям, а красные - программам-вымогателям. Параметр `framealpha` определяет прозрачность фона легенды; значение 1 делает его полностью непрозрачным. Параметр `markerscale` масштабирует размер отметок легенды, в данном случае в три раза.

В этом разделе вы научились создавать простые графики в matplotlib. Но признаем честно: они не великолепны. В следующем разделе используется другая библиотека, позволяющая придать им более профессиональный вид и быстро реализовать более сложные визуализации.

Визуализация данных с помощью seaborn

После pandas и matplotlib перейдём к seaborn - библиотеке визуализации, построенной поверх matplotlib, но заключённой в более изящную оболочку. Она включает встроенные темы оформления и готовые высокоуровневые функции, экономящие время при более сложном анализе. Благодаря этому создавать развитые и красивые графики легко.

Начнём знакомство с seaborn со столбчатой диаграммы количества примеров каждого типа вредоносных программ в наборе, показанной на рис. 9-6.

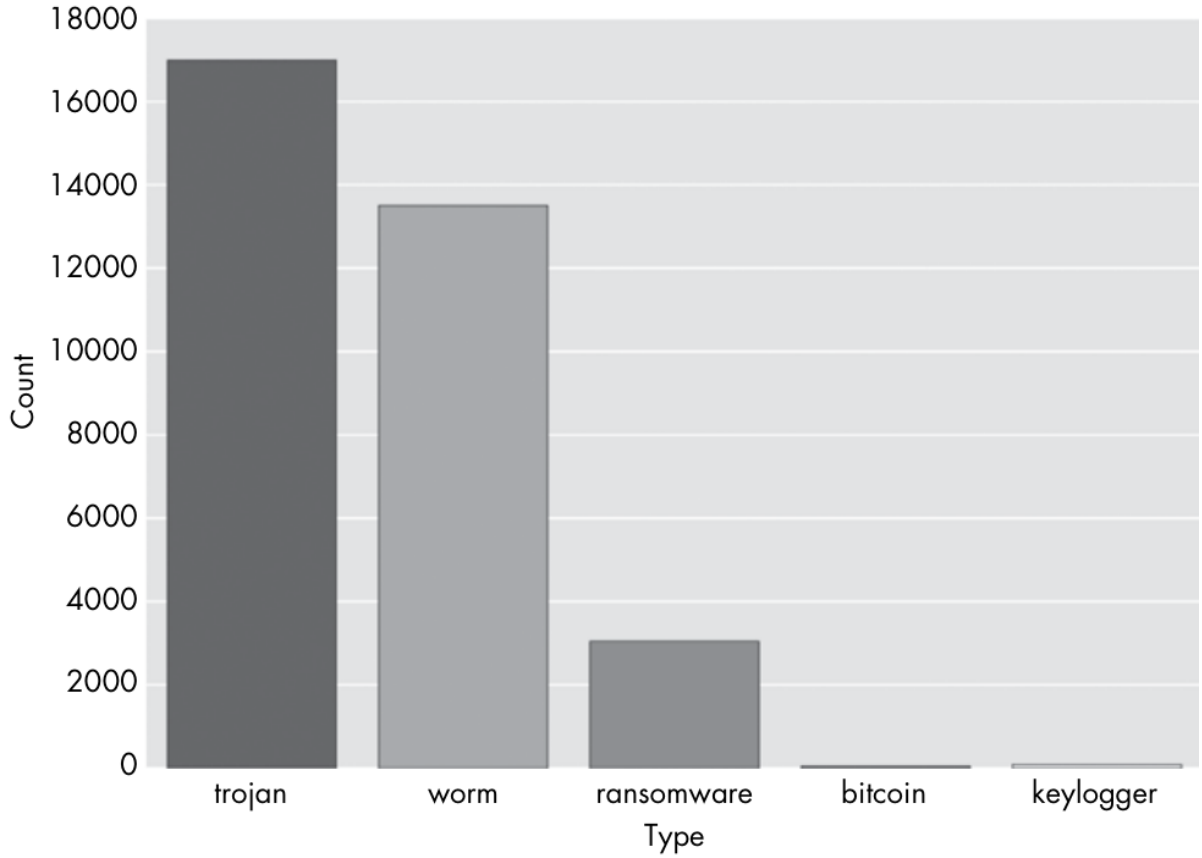

Рисунок 9-6. Столбчатая диаграмма разных видов вредоносных программ в наборе этой главы

Код построения приведён в листинге 9-11.

```
import pandas
from matplotlib import pyplot
import seaborn

malware = pandas.read_csv("malware_data.csv")  # (1)
seaborn.countplot(x='type', data=malware)  # (2)
pyplot.show()  # (3)
```

Листинг 9-11. Создание столбчатой диаграммы количества вредоносных программ по типам

Сначала данные читаются вызовом `pandas.read_csv` (1), затем функция seaborn `countplot` создаёт столбчатую диаграмму столбца `type` объекта `DataFrame` (2). Наконец, график выводится методом `show()` модуля `pyplot` (3). Напомним, seaborn представляет собой оболочку над matplotlib, поэтому отображать фигуры seaborn нужно через matplotlib. Теперь перейдём к более сложному примеру.

Построение распределения антивирусных обнаружений

Предпосылка следующего графика такова: предположим, требуется понять распределение, то есть частоту, антивирусных обнаружений среди образцов набора и выяснить, какой процент вредоносных программ пропускает большинство механизмов, а какой процент обнаруживает большинство. Эти сведения показывают эффективность коммерческой антивирусной отрасли. Для каждого количества обнаружений можно построить столбчатую диаграмму, то есть гистограмму, показывающую долю образцов с таким количеством, как на рис. 9-7.

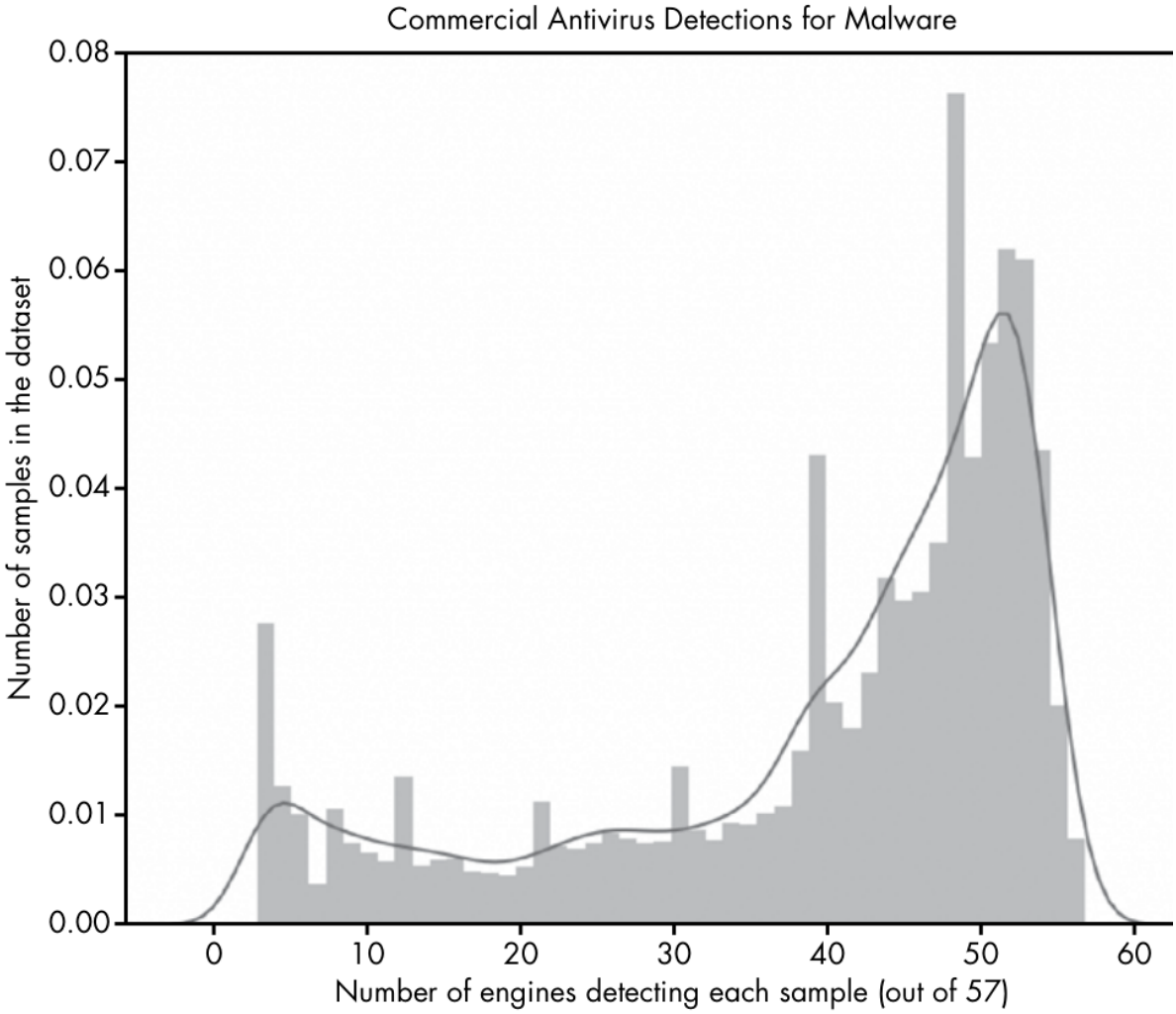

Рисунок 9-7. Визуализация распределения антивирусных обнаружений, то есть положительных результатов

Ось x представляет категории образцов, отсортированные по количеству обнаруживших их механизмов из 57. Если образец признан вредоносным 50 механизмами из 57, он попадает в категорию 50; если лишь 10 механизмами - в категорию 10. Высота каждого столбца пропорциональна общему количеству образцов в этой категории.

График ясно показывает, что многие образцы обнаруживает большинство из 57 механизмов, о чём говорит большой пик частоты в крайней правой верхней области. Но существенное меньшинство образцов обнаруживается лишь небольшим количеством механизмов, что видно в левой области. Образцы, обнаруженные менее чем пятью механизмами, не показаны из-за методики построения набора: вредоносными я считаю образцы, выявленные пятью или более антивирусными механизмами. Значительное число образцов всего с 5-30 обнаружениями говорит о сохраняющемся серьёзном несогласии механизмов. Если образец признан вредоносным 10 механизмами из 57, то либо 47 не смогли его обнаружить, либо 10 ошиблись и выдали ложноположительный результат на безвредный файл. Второй вариант очень маловероятен, поскольку продукты поставщиков имеют чрезвычайно низкие доли ложных срабатываний. Гораздо вероятнее, что большинство механизмов пропустило образец.

Для построения требуется всего несколько строк, как в листинге 9-12.

```
import pandas
import seaborn
from matplotlib import pyplot
```

```
malware = pandas.read_csv("malware_data.csv")
axis = seaborn.distplot(malware['positives'])  # (1)
axis.set(xlabel="Number of engines detecting each sample (out of 57)",  # (2)
         ylabel="Amount of samples in the dataset",
         title="Commercial Antivirus Detections for Malware")
pyplot.show()
```

Листинг 9-12. Построение распределения положительных результатов

В seaborn есть встроенная функция создания графиков распределения, то есть гистограмм, поэтому функции `distplot` просто передаются требуемые данные `malware['positives']` (1). Затем возвращённый seaborn объект оси настраивает заголовок графика и подписи осей x и y (2).

Теперь построим средствами seaborn график двух переменных: количества положительных обнаружений вредоносных файлов, то есть файлов с пятью и более обнаружениями, и их размеров. Ранее такой график был создан в matplotlib на рис. 9-3, но функция seaborn `jointplot` даёт более привлекательный и информативный результат. Рисунок 9-8 богат информацией и сначала требует некоторых усилий для понимания, поэтому разберём его.

Этот график похож на гистограмму рис. 9-7, но вместо распределения одной переменной высотой столбцов он показывает распределения двух переменных, размера вредоносного файла по оси x и количества обнаружений по оси y, насыщенностью цвета. Чем темнее область, тем больше в ней данных. Например, наиболее распространён размер около 10^5,5 и значение `positives` около 53. Вспомогательные графики сверху и справа от основного показывают сглаженные частоты размеров и обнаружений, раскрывая их распределения.

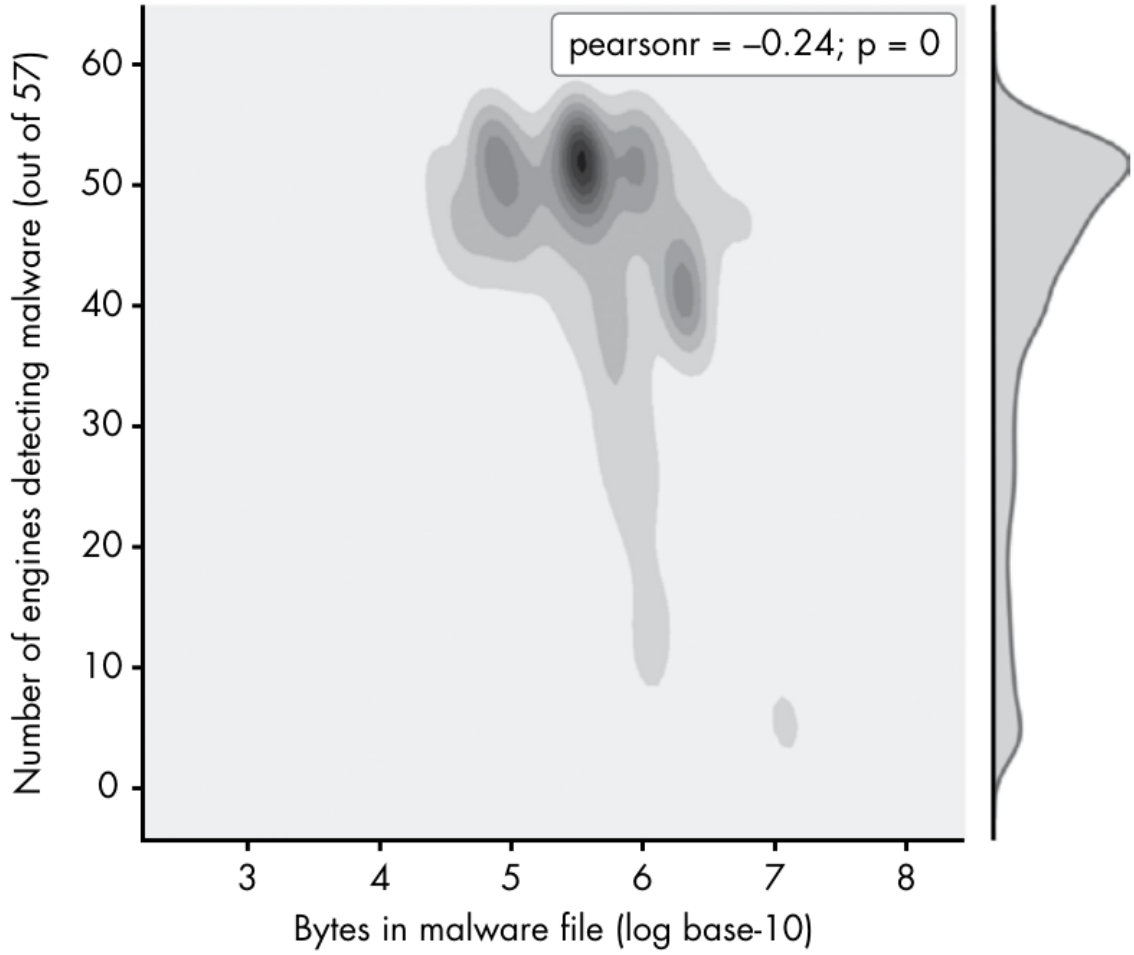

Рисунок 9-8. Визуализация распределения размеров вредоносных файлов и положительных обнаружений

Самый интересный центральный график показывает связь между `size` и `positives`. Вместо отдельных точек данных, как на рис. 9-3 в matplotlib, он гораздо яснее отображает общую тенденцию. Очень большие вредоносные файлы, размером 10^6 и больше, реже обнаруживаются антивирусными механизмами. Возможно, стоит создать собственное решение, специализирующееся на таких вредоносных программах.

Для построения достаточно одного вызова seaborn, как показано в листинге 9-13.

```
import pandas
import seaborn
import numpy
from matplotlib import pyplot

malware = pandas.read_csv("malware_data.csv")
axis=seaborn.jointplot(x=numpy.log10(malware['size']),  # (1)
                       y=malware['positives'],
                       kind="kde")
axis.set_axis_labels("Bytes in malware file (log base-10)",  # (2)
                     "Number of engines detecting malware (out of 57)")
pyplot.show()
```

Листинг 9-13. Построение распределения размеров вредоносных файлов и положительных обнаружений

Здесь функция seaborn `jointplot` создаёт совместный график распределения столбцов `positives` и `size` нашего `DataFrame` (1). Несколько сбивает с толку то, что для подписей осей `jointplot` требует другую функцию, чем в листинге 9-11: `set_axis_labels()` (2), первый аргумент которой задаёт подпись оси x, а второй - оси y.

Создание скрипичного графика

Последний тип графика в этой главе - скрипичный график seaborn. Он позволяет изящно исследовать распределение переменной для нескольких типов вредоносных программ. Например, чтобы увидеть распределение размеров файлов по типам в наборе, можно построить рис. 9-9.

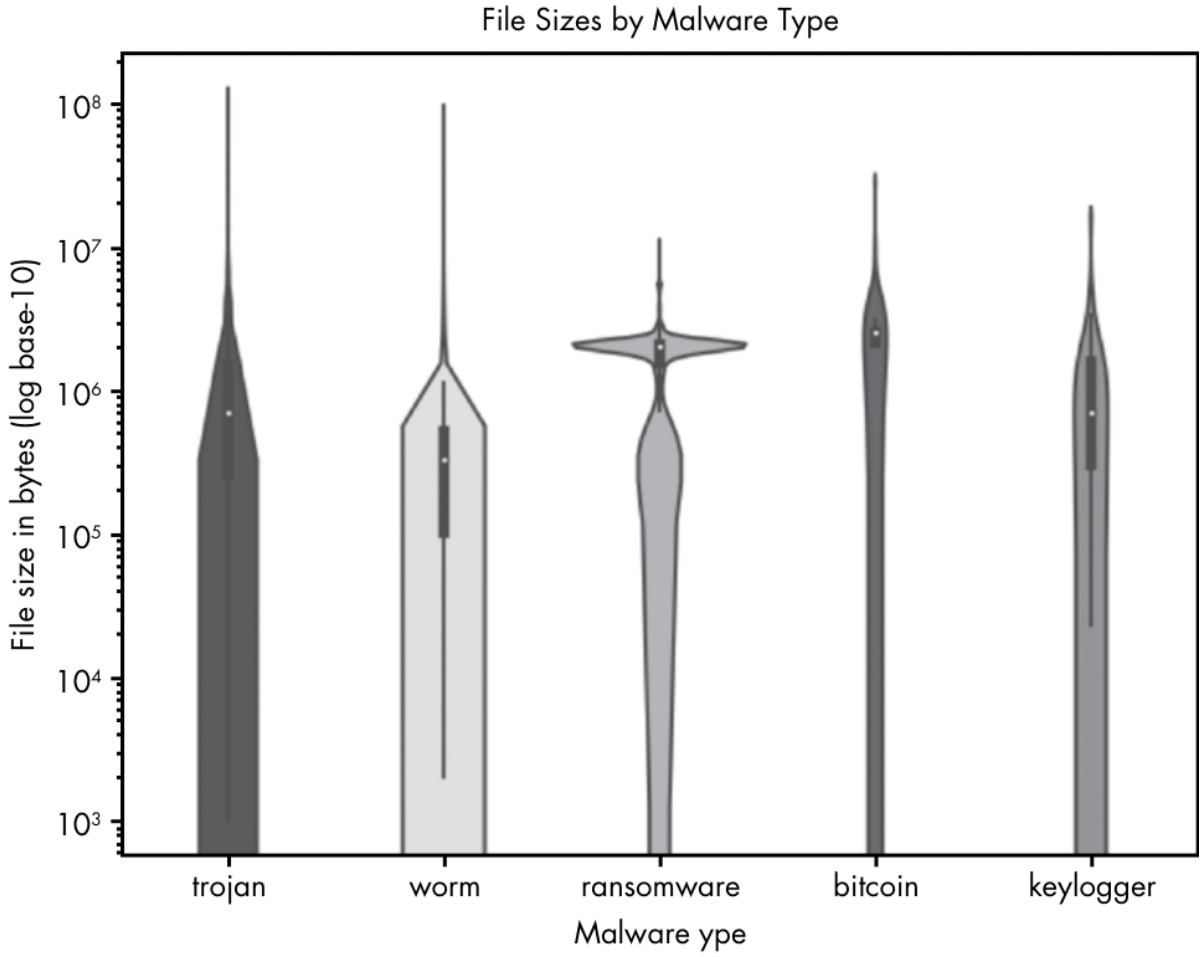

Рисунок 9-9. Визуализация размеров файлов по типам вредоносных программ

По оси y показаны размеры файлов в степенях числа 10, а по оси x перечислены типы вредоносных программ. Толщина столбцов каждого типа меняется на разных уровнях размера и показывает объём данных этого типа с соответствующим размером. Например, видно значительное количество очень больших файлов программ-вымогателей, а черви обычно меньше, вероятно потому, что стремятся быстро распространяться по сети и их авторы уменьшают размер. Знание этих закономерностей может помочь лучше классифицировать неизвестные файлы: большой файл с большей вероятностью окажется программой-вымогателем и с меньшей - червём. Оно также подсказывает, на какие размеры следует нацелить защитное средство для конкретного типа вредоносных программ.

Для создания скрипичного графика достаточно одного вызова, как в листинге 9-14.

```
import pandas
import seaborn
from matplotlib import pyplot

malware = pandas.read_csv("malware_data.csv")

axis = seaborn.violinplot(x=malware['type'], y=malware['size'])  # (1)
axis.set(xlabel="Malware type", ylabel="File size in bytes (log base-10)",  # (2)
         title="File Sizes by Malware Type", yscale="log")
pyplot.show()  # (3)
```

Листинг 9-14. Создание скрипичного графика

В листинге 9-14 сначала создаётся скрипичный график (1). Затем seaborn получает указание задать подписи осей и заголовок, а для оси y установить логарифмический масштаб (2). Наконец, график

выводится (3). Можно построить аналогичный график количества положительных результатов для каждого типа, как на рис. 9-10.

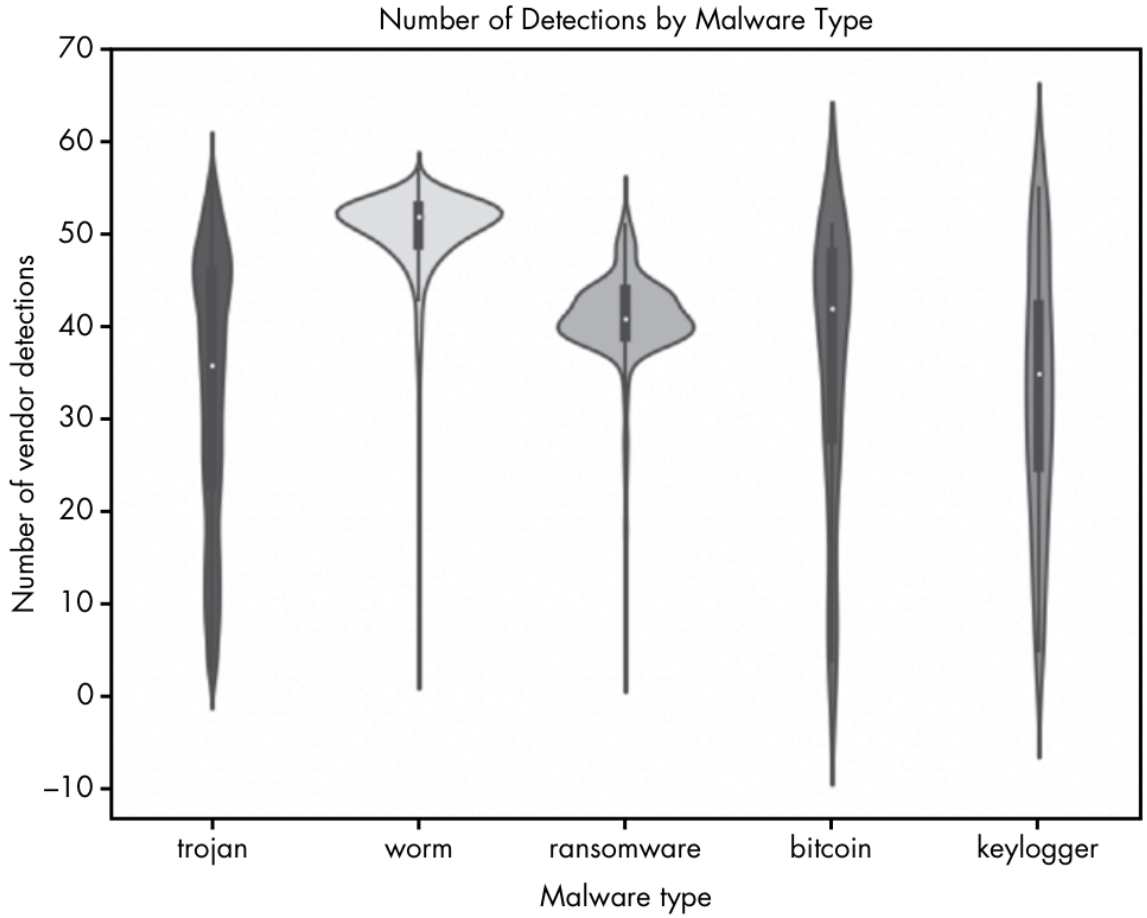

Рисунок 9-10. Визуализация количества положительных результатов антивирусов, то есть обнаружений, для каждого типа вредоносных программ

Единственное различие рис. 9-9 и 9-10 состоит в том, что по оси y вместо размера файла показано количество полученных положительных результатов. График раскрывает несколько интересных тенденций. Например, программы-вымогатели почти всегда обнаруживаются более чем 30 сканерами. Майнеры биткоинов, трояны и кейлоггеры, напротив, значительную часть времени обнаруживаются менее чем 30 сканерами. Следовательно, больше вредоносных программ этих типов проходит через защиту отрасли, а пользователи, у которых не установлены обнаруживающие их сканеры, скорее всего, заражаются такими образцами. В листинге 9-15 показано построение рис. 9-10.

```
import pandas
import seaborn
from matplotlib import pyplot

malware = pandas.read_csv("malware_data.csv")

axis = seaborn.violinplot(x=malware['type'], y=malware['positives'])
axis.set(xlabel="Malware type", ylabel="Number of vendor detections",
         title="Number of Detections by Malware Type")
pyplot.show()
```

Листинг 9-15. Визуализация антивирусных обнаружений по типам вредоносных программ

Код отличается от предыдущего только тем, что функции `violinplot` передаются другие данные, `malware['positives']` вместо `malware['size']`, иначе подписываются оси и называется график,

а логарифмический масштаб оси y не устанавливается.

Итоги

В этой главе вы узнали, как визуализация данных о вредоносных программах позволяет получать общие представления о развивающихся угрозах и эффективности средств безопасности. С помощью pandas, matplotlib и seaborn вы создали собственные визуализации и получили сведения из учебных наборов.

Вы также научились применять такие методы pandas, как `describe()`, для вывода полезной статистики и извлекать подмножества набора. Затем эти подмножества использовались для собственных визуализаций, оценивающих улучшения антивирусного обнаружения, распространение типов вредоносных программ и другие широкие вопросы.

Эти мощные инструменты превращают имеющиеся данные безопасности в практически применимые аналитические сведения, способные направлять разработку новых средств и методов. Надеюсь, вы продолжите изучать визуализацию данных и включите её в процесс анализа вредоносных программ и безопасности.

Глава 10. Основы глубокого обучения

Глубокое обучение - вид машинного обучения, который в последние несколько лет быстро развивался благодаря росту вычислительной мощности и совершенствованию методов. Обычно под глубоким обучением понимают глубокие, то есть многослойные, нейронные сети. Они превосходно выполняют очень сложные задачи, которые исторически были ориентированы на человека, например распознавание изображений и перевод языков.

Например, компьютеру легко определить, содержит ли файл точную копию уже встречавшегося вредоносного кода, и для этого не нужно развитое машинное обучение. Но определить, содержит ли файл вредоносный код, в некоторой степени похожий на виденный ранее, гораздо сложнее. Традиционные схемы на основе сигнатур жёстки и плохо работают с ранее неизвестными или обфусцированными вредоносными программами, тогда как модели глубокого обучения способны видеть сквозь поверхностные изменения и выявлять основные признаки, делающие образец вредоносным. То же относится к сетевой активности, анализу поведения и другим смежным областям. Способность выделять полезные характеристики из массы шума делает глубокое обучение чрезвычайно мощным инструментом кибербезопасности.

Глубокое обучение - всего лишь разновидность машинного обучения, рассмотренного в целом в главах 6 и 7. Однако оно часто создаёт более точные модели, чем обсуждавшиеся там методы, поэтому последние примерно пять лет вся область машинного обучения уделяет ему особое внимание. Если вы хотите работать на переднем крае анализа данных безопасности, необходимо научиться применять глубокое обучение. Но предупреждаем: понять его труднее, чем методы из первой части книги, а для полного понимания потребуются усилия и знание математического анализа на уровне старших классов. Время, вложенное в изучение, окупится способностью создавать более точные системы машинного обучения для анализа данных безопасности. Поэтому настоятельно советуем внимательно читать главу и работать над материалом, пока он не станет понятен. Приступим.

Что такое глубокое обучение?

Модели глубокого обучения учатся видеть обучающие данные как вложенную иерархию понятий, что позволяет им представлять невероятно сложные закономерности. Иными словами, они учитывают не только переданные исходные признаки, но и автоматически объединяют их в новые оптимизированные метапризнаки, затем объединяют метапризнаки в ещё более сложные признаки и так далее.

Слово "глубокое" относится и к используемой архитектуре, обычно состоящей из нескольких слоёв обрабатывающих элементов, каждый из которых принимает выходы предыдущего слоя как входы. Каждый элемент называется нейроном, а архитектура в целом - нейронной сетью или глубокой нейронной сетью, если слоёв много.

Чтобы увидеть пользу такой архитектуры, представим программу, пытающуюся классифицировать изображения как велосипед или одноколёсный велосипед. Для человека задача проста, но запрограммировать компьютер смотреть на сетку пикселей и определять изображённый объект довольно трудно. Пиксели, указывающие на одноколёсный велосипед в одном изображении, могут означать совсем другое в следующем, если объект немного смещён, повёрнут под другим углом или имеет другой цвет.

Модели глубокого обучения обходят трудность, разбивая задачу на более управляемые части. Например, первый слой нейронов глубокой сети может разделить изображение на части и

определить лишь низкоуровневые визуальные признаки: края и границы фигур. Созданные признаки передаются следующему слою, который ищет закономерности среди них. Эти закономерности поступают последующим слоям, пока сеть не начнёт определять общие формы, а в конце - целые объекты. В примере с одноколёсным велосипедом первый слой может найти линии, второй - увидеть образованные линиями окружности, а третий - определить, что некоторые окружности являются колёсами. Таким образом, вместо массы пикселей модель видит в каждом изображении определённое количество метапризнаков "колесо". Она может, например, изучить, что два колеса, скорее всего, означают велосипед, а одно - одноколёсный велосипед.

В этой главе основное внимание уделено фактической работе нейронных сетей, как математической, так и структурной. Сначала на примере очень простой сети я объясню, что представляет собой нейрон и как соединяется с другими нейронами. Затем опишу математические процессы обучения сетей. Наконец, расскажу о нескольких популярных видах нейронных сетей, их особенностях и подходящих задачах. Это хорошо подготовит вас к главе 11, где вы будете создавать модели глубокого обучения на Python.

Как работают нейронные сети

Модели машинного обучения - просто большие математические функции. Например, мы берём входные данные, такие как HTML-файл, представленный последовательностью чисел, применяем функцию машинного обучения, например нейронную сеть, и получаем выход, сообщающий, насколько вредоносным выглядит HTML-файл. Каждая модель - функция с настраиваемыми параметрами, оптимизируемыми в процессе обучения.

Но как на самом деле работает и выглядит функция глубокого обучения? Как следует из названия, нейронные сети - это сети из множества нейронов. Поэтому прежде чем понять работу сети, нужно узнать, что такое нейрон.

Анатомия нейрона

Сам нейрон - всего лишь небольшая простая функция. На рис. 10-1 показан один нейрон.

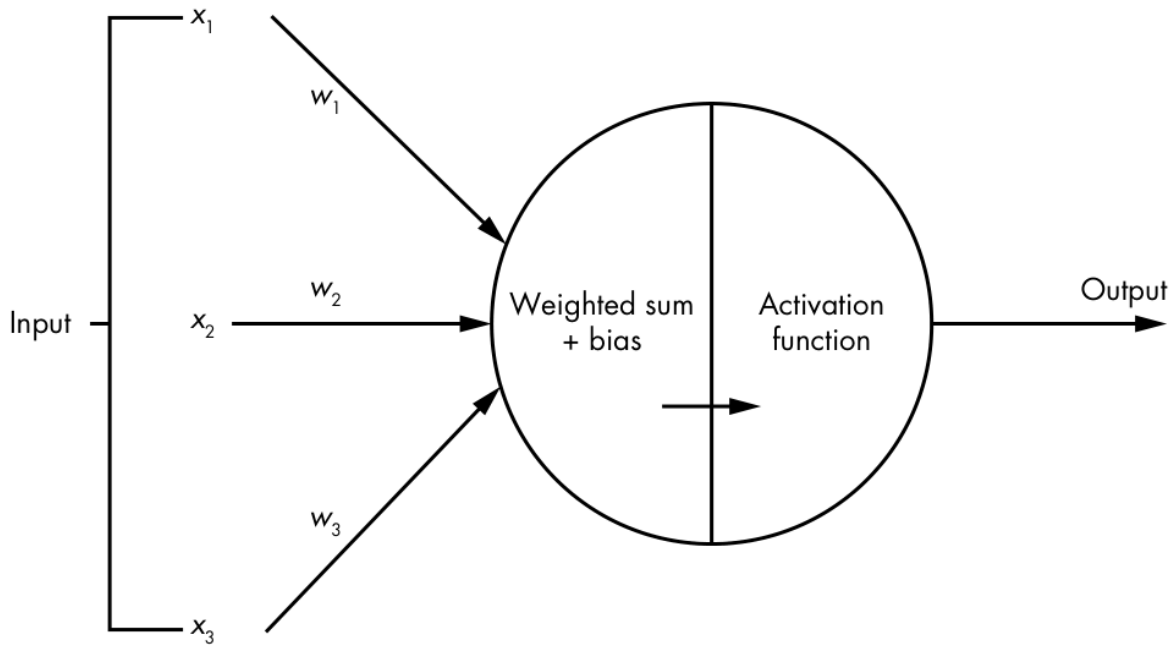

Рисунок 10-1. Визуализация одного нейрона

Входные данные поступают слева, а справа выходит одно число, хотя некоторые виды нейронов создают несколько выходов. Значение выхода представляет собой функцию входных данных и некоторых параметров, оптимизируемых при обучении. Внутри каждого нейрона входные данные преобразуются в выход за два шага.

Сначала вычисляется взвешенная сумма входов. На рис. 10-1 каждое входное число x_i умножается на связанный вес w_i. Полученные значения складываются во взвешенную сумму, к которой прибавляется смещение. Смещение и веса - параметры нейрона, изменяемые во время обучения для оптимизации модели.

Затем к взвешенной сумме со смещением применяется функция активации. Её назначение - выполнить нелинейное преобразование взвешенной суммы, которая сама представляет линейное преобразование входных данных нейрона. Существует множество распространённых и обычно довольно простых функций активации. Единственное требование - дифференцируемость, позволяющая оптимизировать параметры методом обратного распространения ошибки. Вскоре этот процесс будет рассмотрен в разделе "Обучение нейронных сетей" на странице 189 оригинала.

В табл. 10-1 приведены распространённые функции активации и указаны подходящие для них задачи.

Таблица 10-1. Распространённые функции активации

Название	**Уравнение**	**Описание**
Тождественная	`f(x) = x`	Фактически функция активации отсутствует.

Название	Уравнение	Описание
ReLU	`f(x) = 0 при x < 0; x при x >= 0`	Просто `max(0, x)`. ReLU обеспечивают быстрое обучение и устойчивее к проблеме исчезающего градиента, объясняемой далее, чем другие функции вроде сигмоиды.
ReLU с утечкой	`f(x) = alpha*x при x < 0; x при x >= 0`	Подобна обычной ReLU, но вместо 0 возвращает небольшую постоянную долю x. Обычно alpha выбирается очень малой, например 0,01, и остаётся постоянной при обучении.
PReLU	`f(x) = alpha*x при x < 0; x при x >= 0`	Подобна ReLU с утечкой, но alpha является параметром, значение которого оптимизируется при обучении вместе со стандартными весами и смещениями.
ELU	`f(x) = alpha*(e^x - 1) при x < 0; x при x >= 0`	Как и в PReLU, alpha является параметром, но при x < 0 кривая не уходит бесконечно вниз с наклоном alpha, а ограничивается значением alpha, поскольку e^x при x < 0 всегда находится между 0 и 1.
Ступенчатая	`f(x) = 0 при x < 0; 1 при x >= 0`	Простая ступенчатая функция: возвращает 0, если x < 0, иначе 1.
Гауссова	`f(x) = e^(-x^2)`	Колоколообразная кривая с максимальным значением 1 при x = 0.

Таблица 10-1. Распространённые функции активации, продолжение

Название	Уравнение	Описание
Сигмоида	`f(x) = e^x / (e^x + 1)`	Из-за проблемы исчезающего градиента сигмоидальные функции часто применяются лишь в последнем слое нейронной сети. Непрерывный выход ограничен диапазоном от 0 до 1, поэтому сигмоидальные нейроны хорошо приближают выходные вероятности.
Softmax, несколько выходов	`f(x)_j = e^(x_j) / sum_(k=1)^K e^(x_k), j = 1, 2, ..., K`	Выводит несколько значений, сумма которых равна 1. Функции Softmax часто применяются в последнем слое для представления вероятностей классификации, поскольку заставляют сумму всех выходов нейрона равняться 1.

Выпрямленная линейная функция, ReLU, сегодня используется гораздо чаще других и представляет собой просто `max(0, s)`. Предположим, взвешенная сумма со смещением называется s. Если s выше нуля, выход нейрона равен s; если s равна нулю или ниже, выход равен 0. Всю функцию ReLU-нейрона можно записать как `max(0, взвешенная сумма входов + смещение)`, а для n входов конкретнее:

```
max(0, sum_(i=1)^n(w_i * x_i) + b)
```

Нелинейные функции активации - одна из ключевых причин, по которым сети таких нейронов способны приближать любую непрерывную функцию, что во многом объясняет их мощь. Далее вы узнаете, как нейроны соединяются в сеть, а позже поймёте, почему нелинейные функции активации столь важны.

Сеть нейронов

Чтобы создать нейронную сеть, нейроны располагают в ориентированном графе, то есть сети, из нескольких слоёв и соединяют в гораздо более крупную функцию. На рис. 10-2 показан пример небольшой нейронной сети.

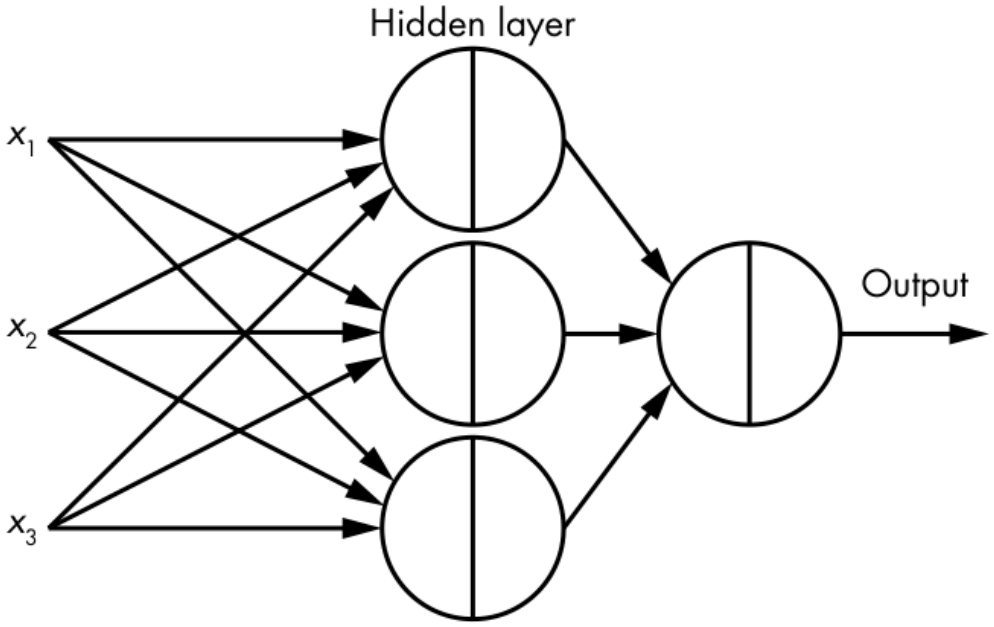

Рисунок 10-2. Очень маленькая нейронная сеть из четырёх нейронов, в которой данные передаются от нейрона к нейрону по соединениям

На рис. 10-2 слева расположены исходные входы x1, x2 и x3. Копии этих значений x_i передаются по соединениям каждому нейрону скрытого слоя, то есть слоя, выход которого не является окончательным выходом модели. В результате получаются три выходных значения, по одному от каждого нейрона. Наконец, все три выхода передаются последнему нейрону, который выдаёт окончательный результат сети.

С каждым соединением сети связан весовой параметр w, а каждый нейрон содержит смещение b, прибавляемое к взвешенной сумме. Поэтому общее количество оптимизируемых параметров базовой сети равно количеству рёбер, соединяющих вход с нейроном, плюс количество нейронов. Например, сеть на рис. 10-2 содержит четыре нейрона и 9 + 3 ребра, всего 16 оптимизируемых параметров. Это лишь пример с очень маленькой сетью; реальные сети часто имеют тысячи нейронов и миллионы соединений.

Теорема об универсальной аппроксимации

Поразительное свойство нейронных сетей состоит в том, что они являются универсальными аппроксиматорами: при достаточном количестве нейронов и правильных весах и смещениях сеть способна имитировать практически любое поведение. Сеть на рис. 10-2 имеет прямое распространение, то есть данные всегда движутся вперёд, слева направо на изображении.

Теорема об универсальной аппроксимации описывает универсальность формально. Она утверждает, что сеть прямого распространения с одним скрытым слоем нейронов и нелинейными функциями активации способна с произвольно малой ошибкой приблизить любую непрерывную функцию на компактном подмножестве R^n.[1] Формулировка непроста, но означает всего лишь следующее: при достаточном количестве нейронов сеть очень точно приближает любую непрерывную ограниченную функцию с конечным количеством входов и выходов.

Иными словами, независимо от требуемой функции теоретически существует нейронная сеть с подходящими параметрами, способная выполнить задачу. Например, если нарисовать извилистую непрерывную функцию f(x), как на рис. 10-3, существует сеть, для которой при любом x выполняется f(x) приблизительно равно network(x), какой бы сложной ни была f(x). Это одна из

причин могущества нейронных сетей.

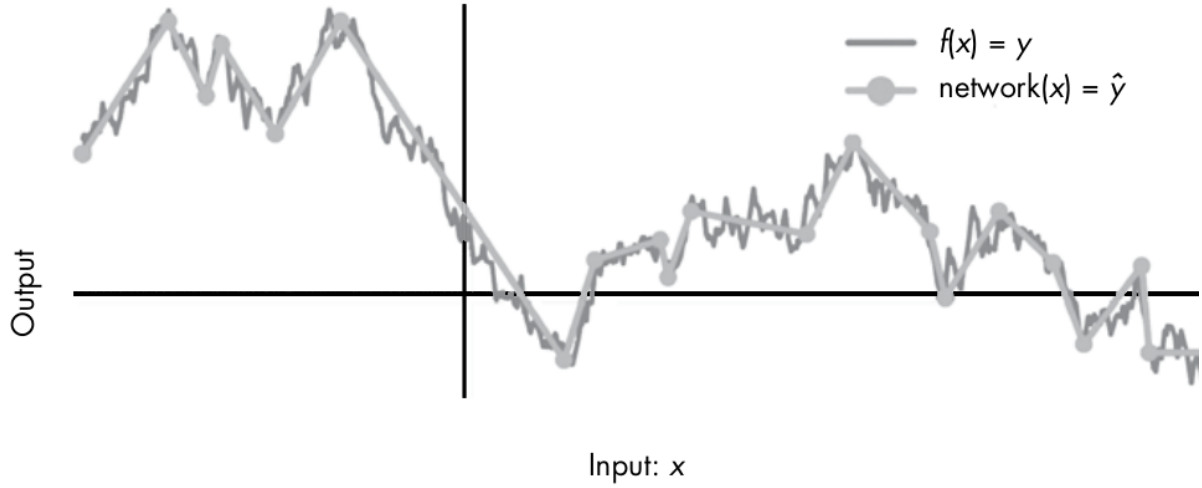

Рисунок 10-3. Пример аппроксимации необычной функции маленькой нейронной сетью. По мере роста количества нейронов разность y и y с крышкой стремится к 0

В следующих разделах мы вручную построим простую сеть, чтобы понять, как и почему при правильных параметрах можно моделировать столь разные виды поведения. Хотя пример имеет очень маленький масштаб, всего один вход и выход, тот же принцип действует с несколькими входами и выходами и невероятно сложным поведением.

Создание собственной нейронной сети

Чтобы увидеть универсальность в действии, попробуем построить собственную сеть. Начнём с двух ReLU-нейронов и единственного входа x, как на рис. 10-4. Затем посмотрим, как разные веса и смещения позволяют моделировать разные функции и результаты.

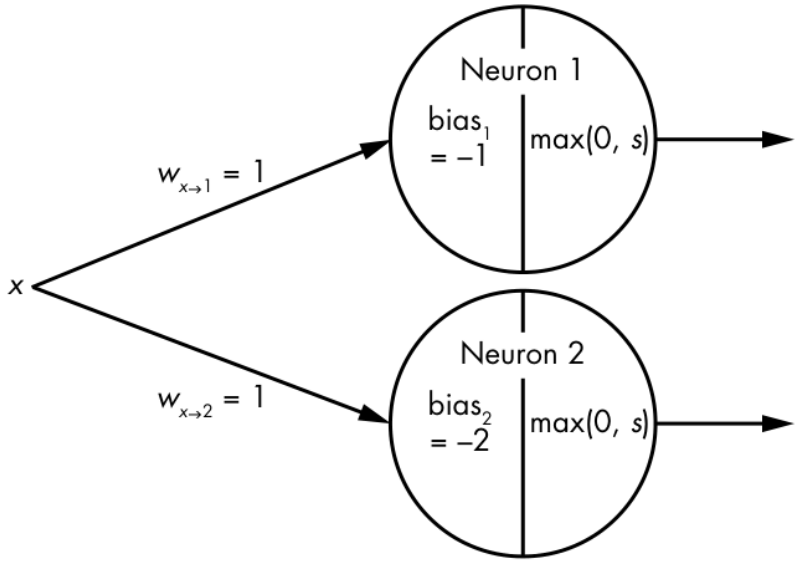

Рисунок 10-4. Два нейрона, получающих входные данные x

Оба нейрона имеют вес 1 и используют функцию ReLU. Единственное различие состоит в том, что `neuron1` применяет смещение -1, а `neuron2` - смещение -2. Посмотрим, что произойдёт при передаче `neuron1` нескольких значений x. Результаты приведены в табл. 10-2.

Таблица 10-2. `Neuron1`

Вход x	**Взвешенная сумма** `x * w(x→1)`	**Взвешенная сумма + смещение** `x * w(x→1) + bias1`	**Выход** `max(0, x * w(x→1) + bias1)`
0	`0 * 1 = 0`	`0 + -1 = -1`	`max(0, -1) = 0`
1	`1 * 1 = 1`	`1 + -1 = 0`	`max(0, 0) = 0`
2	`2 * 1 = 2`	`2 + -1 = 1`	`max(0, 1) = 1`
3	`3 * 1 = 3`	`3 + -1 = 2`	`max(0, 2) = 2`
4	`4 * 1 = 4`	`4 + -1 = 3`	`max(0, 3) = 3`
5	`5 * 1 = 5`	`5 + -1 = 4`	`max(0, 4) = 4`

Первый столбец показывает примеры входа x, второй - полученную взвешенную сумму. В третьем прибавляется смещение, а в четвёртом применяется ReLU и получается выход нейрона для данного x. График функции `neuron1` показан на рис. 10-5.

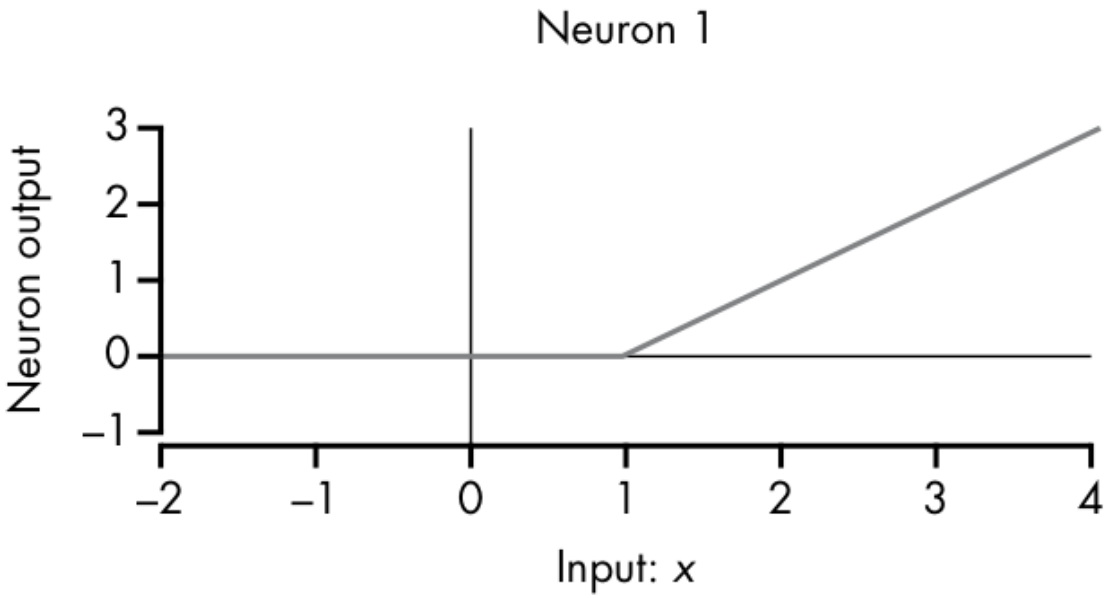

Рисунок 10-5. `Neuron1` *как функция. Ось x представляет единственное входное значение нейрона, а ось y - его выход*

Поскольку смещение `neuron1` равно -1, его выход остаётся нулевым, пока взвешенная сумма не превысит 1, после чего возрастает с определённым наклоном, как на рис. 10-5. Наклон 1 связан со значением веса w(x→1), равным 1. Подумайте, что произойдёт при весе 2: взвешенная сумма удвоится, угол на рис. 10-5 появится при x = 0,5 вместо x = 1, а линия пойдёт вверх с наклоном 2 вместо 1.

Теперь рассмотрим `neuron2` со смещением -2, как в табл. 10-3.

Таблица 10-3. `Neuron2`

Вход x	**Взвешенная сумма** `x * w(x→2)`	**Взвешенная сумма + смещение** `(x * w(x→2)) + bias2`	**Выход** `max(0, (x * w(x→2)) + bias2)`
0	`0 * 1 = 0`	`0 + -2 = -2`	`max(0, -2) = 0`
1	`1 * 1 = 1`	`1 + -2 = -1`	`max(0, -1) = 0`
2	`2 * 1 = 2`	`2 + -2 = 0`	`max(0, 0) = 0`
3	`3 * 1 = 3`	`3 + -2 = 1`	`max(0, 1) = 1`
4	`4 * 1 = 4`	`4 + -2 = 2`	`max(0, 2) = 2`
5	`5 * 1 = 5`	`5 + -2 = 3`	`max(0, 3) = 3`

Поскольку смещение `neuron2` равно -2, угол на рис. 10-6 появляется при x = 2, а не x = 1.

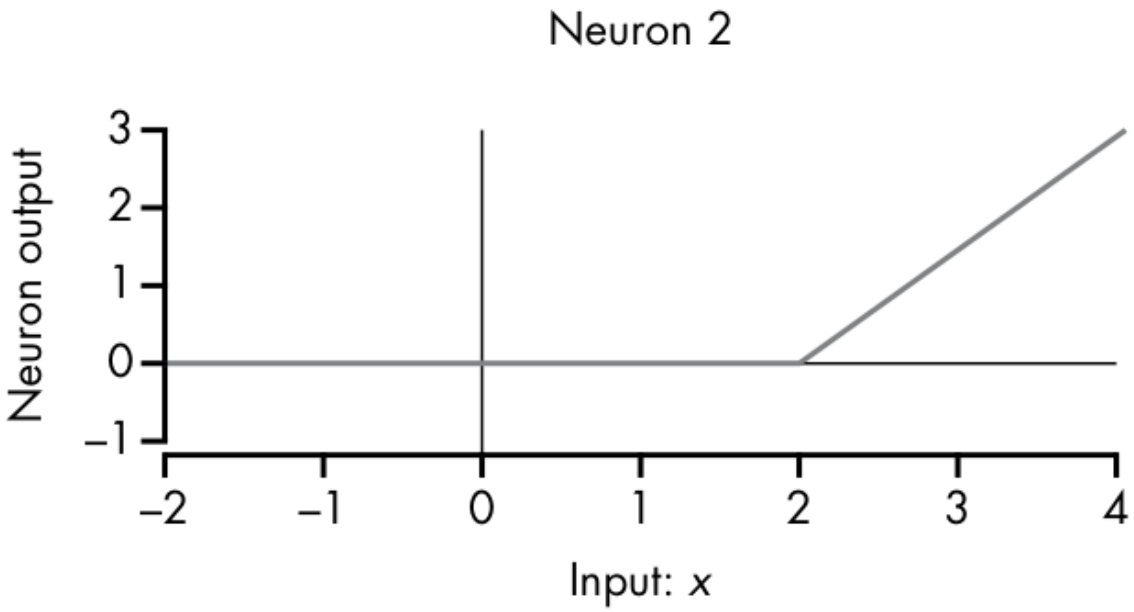

Рисунок 10-6. `Neuron2` *как функция*

Итак, мы построили две очень простые функции, то есть нейроны: обе ничего не делают на некотором интервале, а затем бесконечно возрастают с наклоном 1. Поскольку используются ReLU-нейроны, на наклон функции каждого влияют веса, а на точку начала наклона - и смещение, и вес. Для других функций активации действуют похожие правила. Изменяя параметры, можно как угодно менять угол и наклон функции каждого нейрона.

Однако для достижения универсальности нейроны нужно объединять, что позволит приближать более сложные функции. Соединим наши два нейрона с третьим, как на рис. 10-7. Получится маленькая трёхнейронная сеть с одним скрытым слоем из `neuron1` и `neuron2`.

На рис. 10-7 входные данные x передаются обоим нейронам, а затем их выходы поступают как входы `neuron3`, который выдаёт окончательный выход сети.

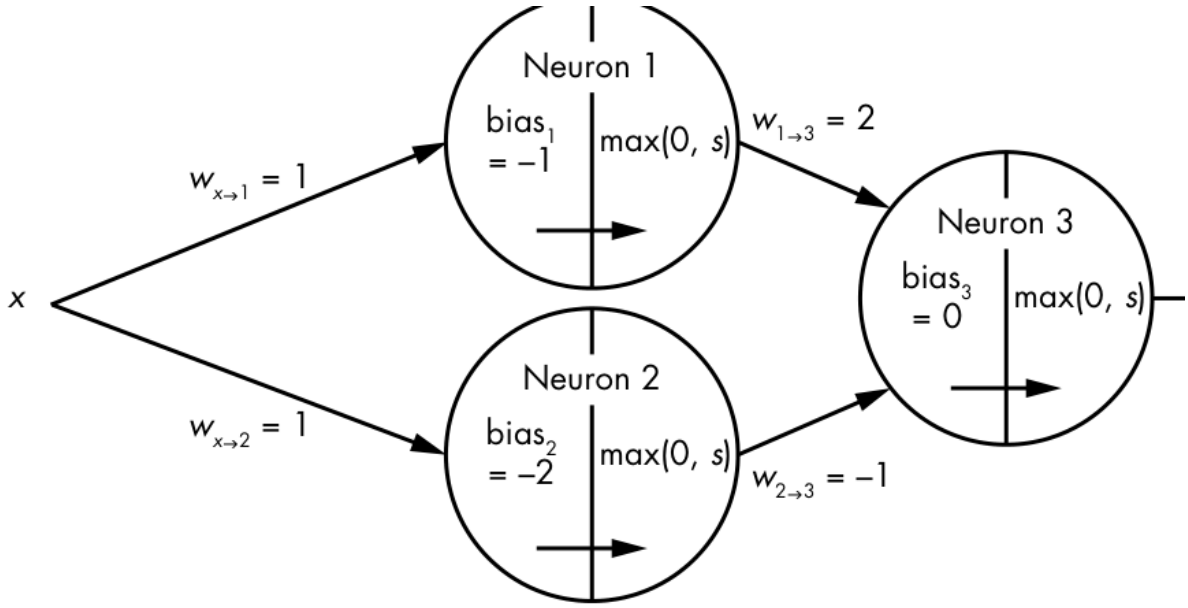

Рисунок 10-7. Маленькая трёхнейронная сеть

Если изучить веса на рис. 10-7, видно, что вес w(1→3) равен 2 и удваивает вклад `neuron1` в `neuron3`. Вес w(2→3) равен -1 и инвертирует вклад `neuron2`. По существу, `neuron3` применяет функцию активации к выражению `neuron1 * 2 - neuron2`. В табл. 10-4 приведены входы и соответствующие выходы сети.

Таблица 10-4. Трёхнейронная сеть

Исходный вход x	`neuron1`	`neuron2`	**Взвешенная сумма** `(neuron1 * w(1→3)) + (neuron2 * w(2→3))`	**Сумма +** `bias3`	**Окончательный выход сети**
0	0	0	`(0 * 2) + (0 * -1) = 0`	`0 + 0 + 0 = 0`	`max(0, 0) = 0`
1	0	0	`(0 * 2) + (0 * -1) = 0`	`0 + 0 + 0 = 0`	`max(0, 0) = 0`
2	1	0	`(1 * 2) + (0 * -1) = 2`	`2 + 0 + 0 = 2`	`max(0, 2) = 2`
3	2	1	`(2 * 2) + (1 * -1) = 3`	`4 + -1 + 0 = 3`	`max(0, 3) = 3`
4	3	2	`(3 * 2) + (2 * -1) = 4`	`6 + -2 + 0 = 4`	`max(0, 4) = 4`

Исходный вход x	`neuron1`	`neuron2`	**Взвешенная сумма** `(neuron1 * w(1→3)) + (neuron2 * w(2→3))`	**Сумма +** `bias3`	**Окончательный выход сети**
5	4	3	`(4 * 2) + (3 * -1) = 5`	`8 + -3 + 0 = 5`	`max(0, 5) = 5`

Первый столбец показывает исходный вход сети x, затем следуют выходы `neuron1` и `neuron2`. Остальные столбцы показывают обработку выходов в `neuron3`: вычисляется взвешенная сумма, прибавляется смещение, а в последнем столбце применяется ReLU, что даёт выход нейрона и сети для каждого исходного x. График функции сети показан на рис. 10-8.

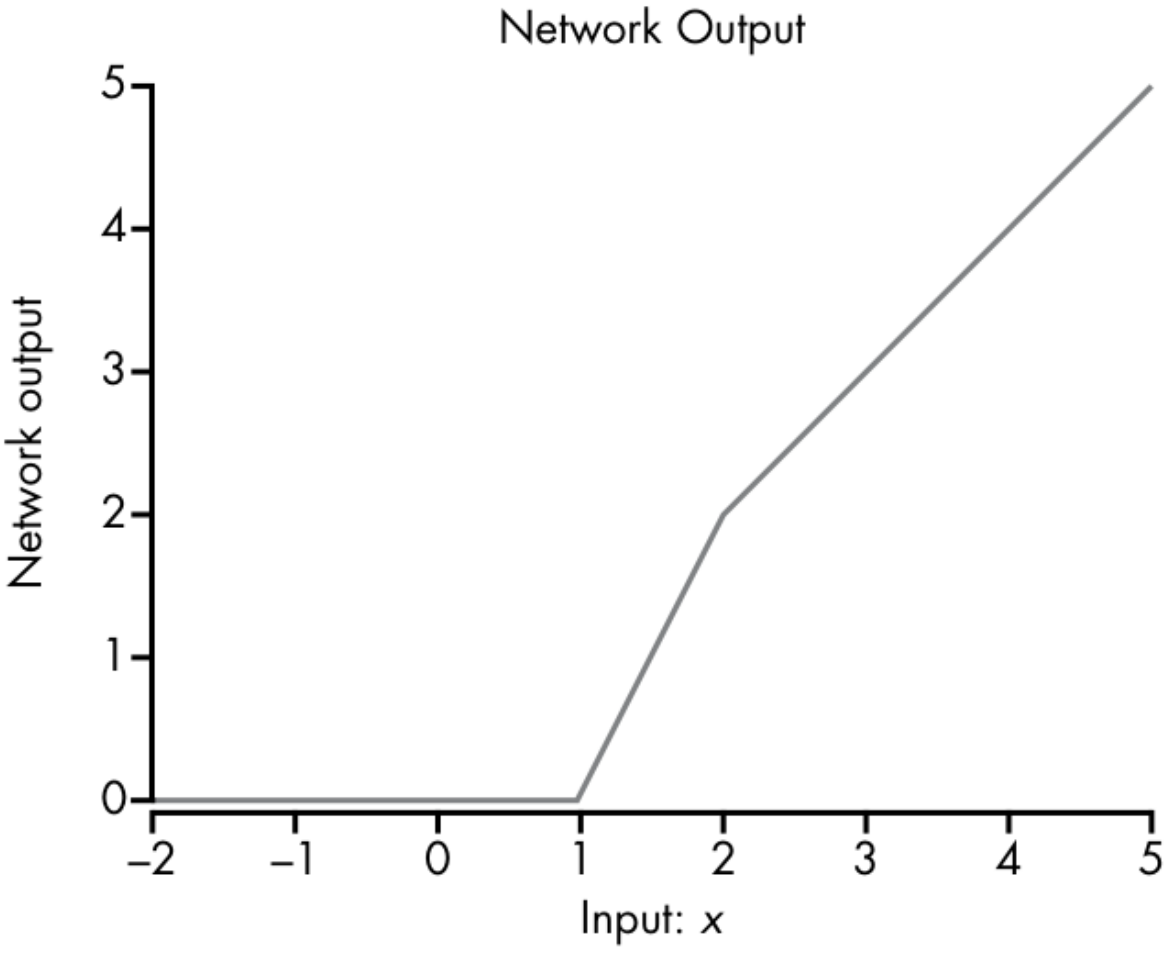

Рисунок 10-8. Входы сети и соответствующие выходы

Как видно, сочетание простых функций позволяет создать график, который поднимается на любом требуемом интервале с любым наклоном, как на рис. 10-8. Иными словами, мы значительно приблизились к способности представить любую конечную функцию нашего входа x.

Добавление ещё одного нейрона в сеть

Мы увидели, как добавлением нейронов заставить график функции сети идти вверх с любым наклоном. Но как направить его вниз? Добавим ещё один нейрон, `neuron4`, как на рис. 10-9.

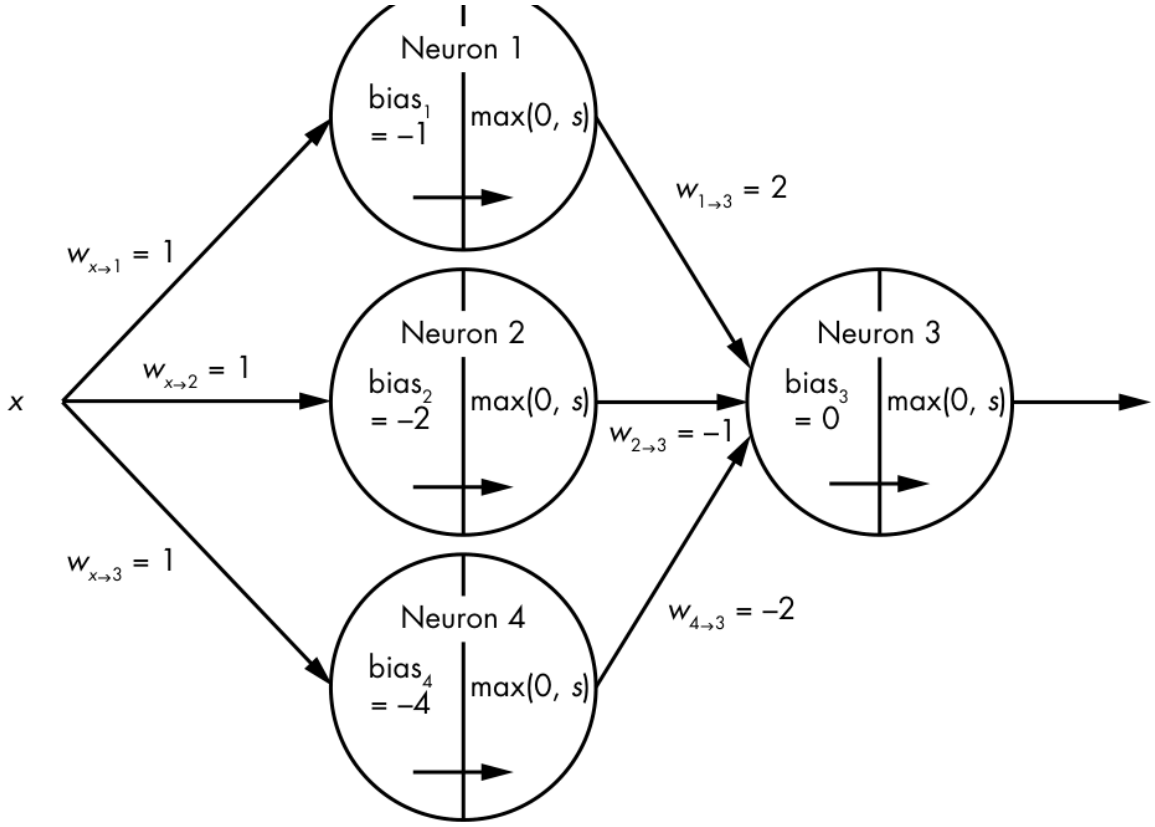

Рисунок 10-9. Маленькая четырёхнейронная сеть с одним скрытым слоем

На рис. 10-9 вход x передаётся `neuron1`, `neuron2` и `neuron4`. Их выходы затем поступают как входы `neuron3`, который выдаёт окончательный выход сети. `Neuron4` такой же, как `neuron1` и `neuron2`, но его смещение равно -4. Выход `neuron4` приведён в табл. 10-5.

Таблица 10-5. `Neuron4`

Вход x	**Взвешенная сумма** `x * w(x→4)`	**Взвешенная сумма + смещение** `(x * w(x→4)) + bias4`	**Выход** `max(0, (x * w(x→4)) + bias4)`
0	`0 * 1 = 0`	`0 + -4 = -4`	`max(0, -4) = 0`
1	`1 * 1 = 1`	`1 + -4 = -3`	`max(0, -3) = 0`
2	`2 * 1 = 2`	`2 + -4 = -2`	`max(0, -2) = 0`
3	`3 * 1 = 3`	`3 + -4 = -1`	`max(0, -1) = 0`
4	`4 * 1 = 4`	`4 + -4 = 0`	`max(0, 0) = 0`
5	`5 * 1 = 5`	`5 + -4 = 1`	`max(0, 1) = 1`

Чтобы график сети пошёл вниз, вычтем функцию `neuron4` из функций `neuron1` и `neuron2` во взвешенной сумме `neuron3`, установив вес соединения `neuron4` с `neuron3` равным -2. Новый выход

всей сети показан в табл. 10-6.

Таблица 10-6. Четырёхнейронная сеть

Исходный вход x	`neuron1`	`neuron2`	`neuron4`	**Взвешенная сумма**	**Сумма +** `bias3`	**Окончательный выход**
0	0	0	0	`(0 * 2) + (0 * -1) + (0 * -2) = 0`	`0 + 0 + 0 + 0 = 0`	`max(0, 0) = 0`
1	0	0	0	`(0 * 2) + (0 * -1) + (0 * -2) = 0`	`0 + 0 + 0 + 0 = 0`	`max(0, 0) = 1`
2	1	0	0	`(1 * 2) + (0 * -1) + (0 * -2) = 2`	`2 + 0 + 0 + 0 = 2`	`max(0, 2) = 2`
3	2	1	0	`(2 * 2) + (1 * -1) + (0 * -2) = 3`	`4 + -1 + 0 + 0 = 3`	`max(0, 3) = 3`
4	3	2	0	`(3 * 2) + (2 * -1) + (0 * -2) = 4`	`6 + -2 + 0 + 0 = 4`	`max(0, 4) = 4`
5	4	3	1	`(4 * 2) + (3 * -1) + (1 * -2) = 5`	`8 + -3 + -2 + 0 = 3`	`max(0, 3) = 3`

На рис. 10-10 показан получившийся график.

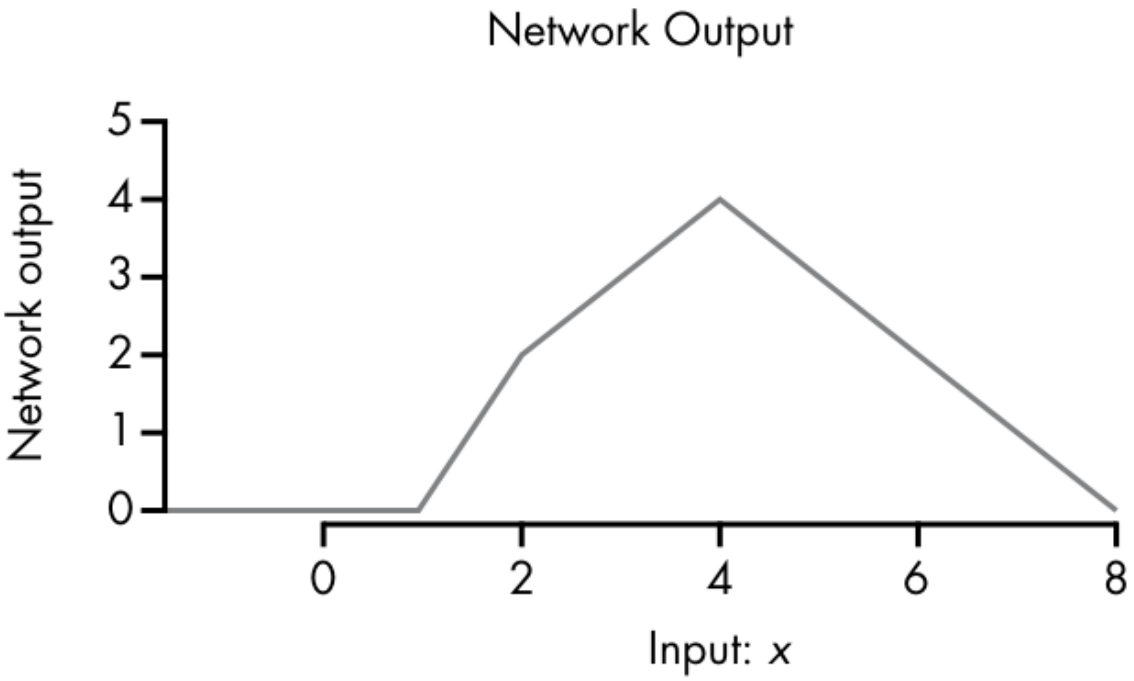

Рисунок 10-10. Визуализация четырёхнейронной сети

Надеюсь, теперь понятно, как архитектура нейронной сети позволяет двигаться вверх и вниз с любой скоростью в любых точках графика, просто объединяя несколько простых нейронов. Это и есть универсальность. Можно продолжать добавлять нейроны и создавать гораздо более сложные функции.

Автоматическое создание признаков

Вы узнали, что нейронная сеть с одним скрытым слоем и достаточным количеством нейронов способна приближать любую конечную функцию. Это весьма мощная идея. Но что происходит при нескольких скрытых слоях? Вкратце, начинается автоматическое создание признаков, возможно ещё более мощное свойство нейронных сетей.

Исторически значительная часть создания моделей машинного обучения приходилась на извлечение признаков. Для HTML-файла много времени уходило на выбор числовых характеристик, способных помочь модели, например количества заголовков разделов или уникальных слов.

Многослойные нейронные сети и автоматическое создание признаков позволяют переложить значительную часть этой работы на модель. В общем случае, если передать сети довольно сырые признаки, такие как символы или слова HTML-файла, каждый слой может научиться представлять их в форме, хорошо подходящей для входов последующих слоёв. Иными словами, если количество букв а в документе особенно важно для обнаружения вредоносности, сеть научится их считать без прямого указания человека.

В примере распознавания велосипеда никто специально не сообщал сети, что края или метапризнаки колёс полезны. Во время обучения модель сама выяснила, что эти признаки полезны для входа следующего слоя. Особенно важно, что последующие слои способны по-разному использовать изученные низкоуровневые признаки. Поэтому глубокие сети могут оценивать множество невероятно сложных закономерностей с гораздо меньшим количеством нейронов и параметров, чем однослойная сеть.

Нейронные сети не только выполняют значительную часть извлечения признаков, прежде требовавшую много времени и усилий, но и делают это оптимизированно и с эффективным использованием пространства под руководством процесса обучения.

Обучение нейронных сетей

До сих пор мы изучали, как нейронная сеть с большим количеством нейронов и правильными весами и смещениями приближает сложные функции. Во всех примерах параметры задавались вручную. Но реальные сети обычно содержат тысячи нейронов и миллионы параметров, поэтому нужен эффективный способ оптимизировать значения.

Обычно обучение начинается с набора данных и сети с множеством неоптимизированных, случайно инициализированных параметров. Обучение состоит в оптимизации параметров для минимизации целевой функции. В обучении с учителем, где модель должна предсказывать метку, например 0 для "безвредно" и 1 для "вредоносно", целевая функция связана с ошибкой предсказания сети во время обучения. Для данного входа x, например конкретного HTML-файла, это разность между заведомо правильной меткой y, например 1,0 для "является вредоносным", и выходом y_hat текущей сети, например 0,7. Ошибку можно считать разностью предсказанной метки y_hat и известной истинной метки y, где `network(x) = y_hat`, а сеть пытается приблизить неизвестную функцию f, такую что `f(x) = y`. Иными словами, `network` приближает f.

Основная идея обучения состоит в том, чтобы передать сети наблюдение x из обучающего набора, получить выход y_hat и определить, как изменение параметров сдвинет y_hat ближе к цели y. Представьте себя в космическом корабле с разными ручками. Вы не знаете назначения каждой, но знаете нужное направление y. Вы нажимаете на газ и отмечаете фактическое направление y_hat. Затем совсем немного поворачиваете ручку и снова нажимаете газ. Разность первого и второго направлений показывает, насколько ручка влияет на направление. Постепенно можно научиться довольно хорошо управлять кораблём.

Обучение нейронной сети похоже. Сначала ей передаётся наблюдение x и получается выход y_hat. Этот шаг называется прямым распространением, потому что вход x проходит вперёд через сеть к окончательному выходу y_hat. Затем определяется влияние каждого параметра на y_hat. Например, если выход сети равен 0,7, а правильный должен быть ближе к 1, можно немного увеличить параметр w и посмотреть, приблизился ли y_hat к y или отдалился и насколько.[2] Это называется частной производной y_hat по w, то есть `∂y_hat/∂w`.

Затем все параметры сети слегка сдвигаются в направлении, приближающем y_hat к y, а следовательно, network к f. Если `∂y_hat/∂w` положительна, w следует немного увеличить, а именно пропорционально `(y - y_hat) * ∂y_hat/∂w`, чтобы новое y_hat немного отошло от 0,7 в сторону 1, то есть y. Иными словами, сеть учится приближать неизвестную функцию f, исправляя ошибки на обучающих данных с известными метками.

Процесс итеративного вычисления частных производных, обновления параметров и повторения называется градиентным спуском. Но в сети с тысячами нейронов, миллионами параметров и зачастую миллионами обучающих наблюдений все эти вычисления требуют огромного объёма работы. Обойти проблему позволяет изящный алгоритм обратного распространения ошибки, делающий вычисления практически осуществимыми. По существу, он позволяет эффективно вычислять частные производные вдоль вычислительных графов вроде нейронной сети.

Оптимизация нейронной сети обратным распространением ошибки

В этом разделе мы построим простую сеть, демонстрирующую обратное распространение. Предположим, есть обучающий пример x = 2 с истинной меткой y = 10. Обычно x представляет собой массив множества значений, но для простоты ограничимся одним. Подставив значения, увидим на рис. 10-11, что при x = 2 сеть выдаёт y_hat = 5.

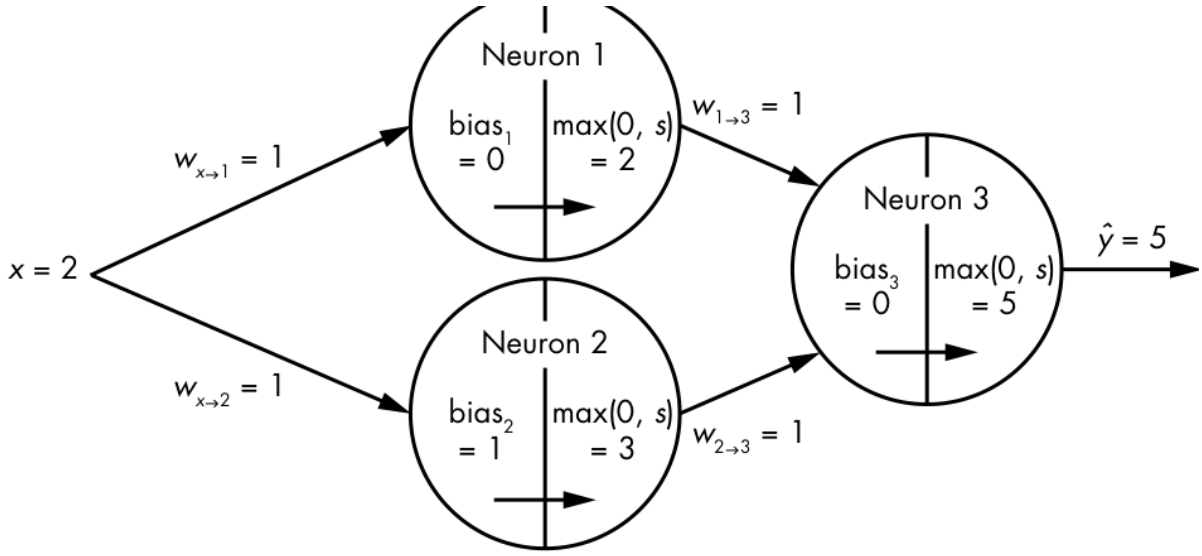

Рисунок 10-11. Трёхнейронная сеть с входом x = 2

Чтобы сдвинуть параметры так, чтобы выход y_hat при x = 2 приблизился к известному y = 10, нужно вычислить влияние w(1→3) на окончательный y_hat. Посмотрим, что произойдёт при небольшом увеличении w(1→3), скажем на 0,01. Взвешенная сумма `neuron3` становится равной `1,01 * 2 + (1 * 3)`, а окончательный y_hat меняется с 5 до 5,02, то есть увеличивается на 0,02. Следовательно, частная производная y_hat по w(1→3) равна 2: изменение w(1→3) вызывает вдвое большее изменение y_hat.

Поскольку y равно 10, а текущий y_hat при нынешних параметрах и x = 2 равен 5, теперь известно, что w(1→3) нужно немного увеличить, чтобы приблизить y_hat к 10.

Это довольно просто. Но нужно знать, в какую сторону сдвигать все параметры сети, а не только параметры нейрона последнего слоя. Что насчёт w(x→1)? Вычислить `∂y_hat/∂w(x→1)` сложнее, поскольку он влияет на y_hat лишь косвенно. Сначала спросим функцию `neuron3`, как выход `neuron1` влияет на y_hat. Если изменить выход `neuron1` с 2 до 2,01, выход `neuron3` изменится с 5 до 5,01, поэтому `∂y_hat/∂neuron1 = 1`. Чтобы узнать влияние w(x→1) на y_hat, достаточно умножить `∂y_hat/∂neuron1` на влияние w(x→1) на выход `neuron1`. Если изменить w(x→1) с 1 до 1,01, выход `neuron1` меняется с 2 до 2,02, поэтому `∂neuron1/∂w(x→1) = 2`. Следовательно:

```
∂y_hat/∂w(x→1) = (∂y_hat/∂neuron1) * (∂neuron1/∂w(x→1))
```

Или:

```
∂y_hat/∂w(x→1) = 1 * 2 = 2
```

Возможно, вы заметили, что мы только что применили правило цепочки.[3]

Иными словами, чтобы выяснить влияние глубоко расположенного параметра вроде w(x→1) на окончательный y_hat, нужно перемножить частные производные в каждой точке пути между параметром и y_hat. Если w(x→1) поступает в нейрон, выход которого передаётся десяти другим нейронам, для вычисления его влияния на y_hat потребуется суммировать все пути от w(x→1) к y_hat, а не один. На рис. 10-12 показаны пути, затрагиваемые учебным весом w(x→2).

by the sample weight parameter $w_{x\to 2}$.

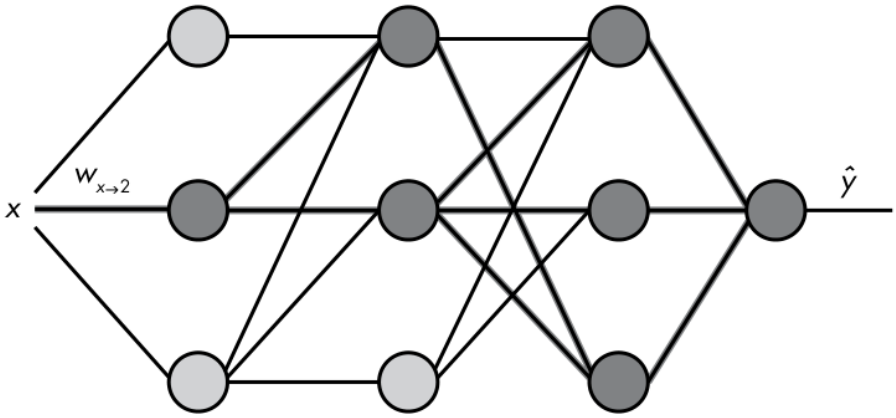

Рисунок 10-12. Пути, затрагиваемые w(x→2), показаны тёмно-серым: это вес соединения входных данных x со средним нейроном первого, крайнего левого, слоя

Обратите внимание: скрытые слои этой сети не являются полносвязными, что объясняет, почему нижний нейрон второго скрытого слоя не выделен.

Взрыв количества путей

Что произойдёт, если сеть станет ещё больше? Количество путей, которые требуется сложить для вычисления частной производной низкоуровневого параметра, растёт экспоненциально. Представьте нейрон, выход которого поступает слою из 1000 нейронов, их выходы поступают ещё 1000 нейронам, а выходы последних - окончательному выходному нейрону.

Получается миллион путей. К счастью, проходить каждый путь и складывать результаты для получения `∂y_hat/∂parameter` не нужно. Здесь помогает обратное распространение. Вместо прохода по каждому пути к окончательным выходам y_hat частные производные вычисляются слой за слоем, начиная сверху вниз, то есть назад.

Используя логику правила цепочки из предыдущего раздела, любую частную производную `∂y_hat/∂w`, где w - параметр, соединяющий выход слоя i-1 с нейроном i в слое i, можно вычислить, суммируя следующее выражение для всех нейронов i+1 слоя i+1, с которыми соединён нейрон i параметра w:

```
(∂y_hat/∂neuron_(i+1)) * (∂neuron_(i+1)/∂neuron_i) * (∂neuron_i/∂w)
```

Послойное вычисление сверху вниз ограничивает взрыв путей за счёт объединения производных на каждом слое. Производные, вычисленные на верхнем слое i+1, например `∂y_hat/∂neuron_(i+1)`, записываются и помогают вычислить производные слоя i. Для производных слоя i-1 используются сохранённые производные слоя i, например `∂y_hat/∂neuron_i`. Затем слой i-2 использует производные слоя i-1 и так далее. Этот трюк значительно сокращает количество повторных вычислений и ускоряет обучение нейронных сетей.

Исчезающий градиент

Одна из проблем очень глубоких сетей - исчезающий градиент. Представьте вес первого слоя сети из десяти слоёв. Сигнал обратного распространения для него равен сумме сигналов всех путей от нейрона этого веса к окончательному выходу.

Проблема в том, что сигнал каждого пути, скорее всего, чрезвычайно мал: он вычисляется перемножением частных производных в каждой точке пути глубиной десять нейронов, а все они обычно меньше 1. Поэтому параметры низкоуровневого нейрона обновляются по сумме огромного количества очень малых чисел, многие из которых взаимно уничтожаются. В результате сети трудно передать сильный сигнал вниз, к параметрам нижних слоёв. С добавлением каждого слоя проблема экспоненциально ухудшается. Как вы узнаете далее, некоторые архитектуры пытаются её обойти.

Виды нейронных сетей

Для простоты во всех предыдущих примерах использовалась сеть прямого распространения. На самом деле существует множество других полезных структур для разных классов задач. Рассмотрим несколько самых распространённых видов и их применение в кибербезопасности.

Нейронная сеть прямого распространения

Простейшая и исторически первая нейронная сеть прямого распространения похожа на куклу Барби без аксессуаров: другие виды обычно лишь вариации этой "стандартной" структуры. Архитектура уже должна быть знакома: она состоит из стопки слоёв нейронов. Каждый слой соединён с некоторыми или всеми нейронами следующего, но соединения никогда не идут назад и не образуют циклов, откуда и название "прямое распространение".

В таких сетях каждое существующее соединение связывает нейрон или исходный вход слоя i с нейроном слоя j, где j > i. Каждый нейрон слоя i необязательно соединён со всеми нейронами слоя i + 1, но все соединения должны вести вперёд, от предыдущих слоёв к последующим.

Обычно сеть прямого распространения - первый вид, который пробуют для задачи, если только уже не известна другая архитектура, особенно хорошо с ней работающая, например свёрточная сеть для распознавания изображений.

Свёрточная нейронная сеть

Свёрточная нейронная сеть, CNN, содержит свёрточные слои, где вход каждого нейрона определяется окном, скользящим по пространству входов. Представьте маленькое квадратное окно, движущееся по большой картинке. Только видимые через окно пиксели соединяются с определённым нейроном следующего слоя. Затем окно сдвигается, и новый набор пикселей соединяется с новым нейроном. Это показано на рис. 10-13.

Структура таких сетей поощряет изучение локальных признаков. Например, нижним слоям полезнее сосредоточиться на отношениях между соседними пикселями изображения, образующими края, фигуры и прочее, чем на отношениях случайно разбросанных пикселей, которые вряд ли что-то означают. Скользящие окна явно навязывают этот фокус, что улучшает и ускоряет обучение в областях, где особенно важно извлечение локальных признаков.

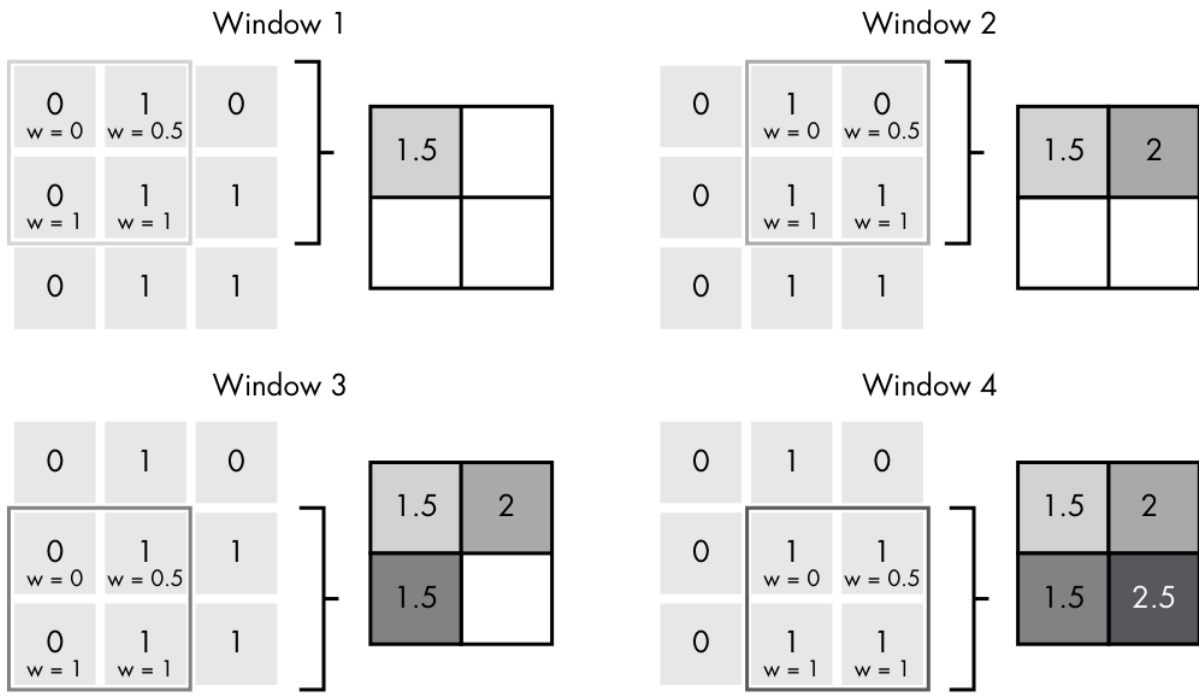

Рисунок 10-13. Свёрточное окно 2 x 2 скользит по пространству входов 3 x 3 с шагом 1 и создаёт выход 2 x 2

Благодаря способности сосредоточиваться на локальных областях входных данных свёрточные сети чрезвычайно эффективны для распознавания и классификации изображений. Они также показали эффективность в некоторых видах обработки естественного языка, что имеет значение для кибербезопасности.

После передачи значений каждого свёрточного окна определённым нейронам свёрточного слоя по выходам этих нейронов снова скользит окно. Но вместо стандартных нейронов, например ReLU, с весами каждого входа данные поступают нейронам без весов, то есть с фиксированным весом 1, и функцией активации `max` или похожей. Иными словами, маленькое окно скользит по выходам свёрточного слоя, а максимальное значение каждого окна передаётся следующему слою. Он называется слоем пулинга. Назначение таких слоёв - "уменьшить масштаб" данных, обычно изображения, сократить размер признаков для ускорения вычислений, сохранив самую важную информацию.

Свёрточная нейронная сеть может содержать один или несколько наборов свёрточных слоёв и слоёв пулинга. Стандартная архитектура может включать свёрточный слой, слой пулинга, ещё одну пару таких слоёв и в конце несколько полносвязных слоёв, как в сети прямого распространения. Цель состоит в том, чтобы последние полносвязные слои получали довольно высокоуровневые признаки, вспомните колёса одноколёсного велосипеда, и благодаря этому точно классифицировали сложные данные вроде изображений.

Нейронная сеть автоэнкодера

Автоэнкодер - нейронная сеть, пытающаяся сжать, а затем распаковать вход с минимальной разностью между исходным обучающим входом и распакованным выходом. Его цель - изучить эффективное представление набора данных. Иными словами, автоэнкодеры действуют как оптимизированные программы сжатия с потерями: сжимают входные данные в меньшее представление, а затем распаковывают до исходного размера.

Вместо оптимизации параметров путём минимизации разности между известными метками y и предсказанными y_hat для данного x сеть минимизирует разность между исходным входом x и восстановленным выходом x_hat.

Структурно автоэнкодеры обычно очень похожи на стандартные сети прямого распространения, но средние слои содержат меньше нейронов, чем ранние и поздние, как показано на рис. 10-14.

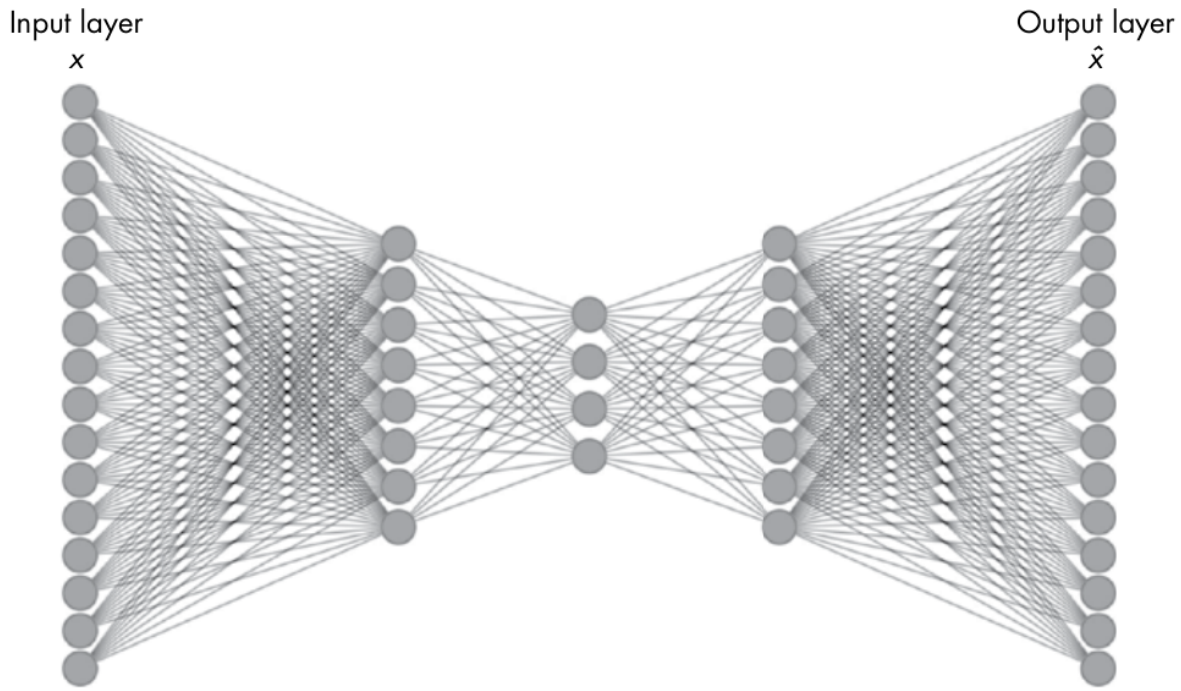

Рисунок 10-14. Визуализация сети автоэнкодера

Средний слой значительно меньше крайнего левого входного и крайнего правого выходного слоёв одинакового размера. Последний слой всегда должен содержать столько же выходов, сколько было исходных входов, чтобы каждый обучающий вход x_i можно было сравнить с его сжатым и восстановленным аналогом x_hat_i.

После обучения автоэнкодер можно применять по-разному. Например, просто как эффективную программу сжатия и распаковки. Автоэнкодеры, обученные сжимать изображения, способны создавать гораздо более чёткие изображения, чем JPEG того же размера.

Генеративно-состязательная сеть

Генеративно-состязательная сеть, GAN, - система из двух нейронных сетей, которые конкурируют и совершенствуются в своих задачах. Обычно генеративная сеть пытается создавать поддельные образцы, например изображения, из случайного шума. Затем вторая сеть-дискриминатор пытается отличить настоящие образцы от поддельных сгенерированных, например реальные изображения спальни от созданных сетью.

Обе сети GAN оптимизируются обратным распространением ошибки. Генератор оптимизирует параметры исходя из того, насколько хорошо обманул дискриминатор на данном раунде, а дискриминатор - исходя из точности различения сгенерированных и настоящих образцов. Иными словами, их функции потерь прямо противоположны.

GAN применяются для создания правдоподобных данных и улучшения низкокачественных или повреждённых данных.

Рекуррентная нейронная сеть

Рекуррентные сети, RRN, - сравнительно широкий класс нейронных сетей, в которых соединения образуют ориентированные циклы, а функции активации зависят от временных шагов. Это позволяет сети развивать память и изучать закономерности в последовательностях данных. В RNN входы, выходы либо и входы, и выходы представляют некоторый временной ряд.

RNN отлично подходят для задач, где важен порядок данных: распознавания связного рукописного текста и речи, перевода языков и анализа временных рядов. В кибербезопасности они применимы к анализу сетевого трафика, поведенческому обнаружению и статическому анализу файлов. Программный код похож на естественный язык тем, что порядок имеет значение, поэтому его можно рассматривать как временной ряд.

Одна проблема RNN связана с исчезающим градиентом: каждый временной шаг подобен целому дополнительному слою сети прямого распространения. При обратном распространении сигналы нижних слоёв, а в этом случае более ранних временных шагов, становятся чрезвычайно слабыми.

Сеть с долгой краткосрочной памятью, LSTM, - особый вид RNN, созданный для решения этой проблемы. LSTM содержит ячейки памяти и специальные нейроны, которые пытаются решить, какую информацию запомнить, а какую забыть. Отбрасывание большей части информации значительно ограничивает исчезающий градиент, поскольку уменьшает взрыв количества путей.

ResNet

ResNet, сокращение от residual network, то есть остаточная сеть, - вид нейронной сети, создающий пропускающие соединения между нейронами ранних, неглубоких слоёв и более глубоких слоёв с пропуском одного или нескольких промежуточных. Слово "остаточная" означает, что такие сети учатся передавать числовую информацию непосредственно между слоями, не пропуская её через функции активации, показанные в табл. 10-1.

Эта структура значительно ослабляет проблему исчезающего градиента и позволяет ResNet быть невероятно глубокими, иногда более 100 слоёв.

Очень глубокие сети превосходно моделируют чрезвычайно сложные и необычные отношения во входных данных. Благодаря большому количеству слоёв ResNet особенно хорошо подходят для сложных задач. Как и сети прямого распространения, ResNet важны скорее общей эффективностью в решении сложных задач, чем специализацией в конкретных предметных областях.

Итоги

В этой главе вы изучили структуру нейронов и их соединение в нейронные сети. Вы также рассмотрели обучение сетей обратным распространением ошибки и узнали о некоторых преимуществах и проблемах, включая универсальность, автоматическое создание признаков и исчезающий градиент. Наконец, вы познакомились со структурами и преимуществами нескольких распространённых видов нейронных сетей.

В следующей главе вы будете создавать нейронные сети для обнаружения вредоносных программ с помощью пакета Python Keras.

Сноски

[1] [назад] R^n можно считать n-мерным евклидовым пространством, где все числа вещественны. Например, R^2 представляет все возможные пары вещественных чисел, такие как (3,5; -5).
[2] [назад] На практике слегка увеличивать параметр и заново вычислять выход сети не требуется. Вся сеть является дифференцируемой функцией, поэтому `∂y_hat/∂w` можно точно и быстрее вычислить средствами математического анализа. Однако рассуждение через небольшое изменение и повторную оценку обычно интуитивнее производных.
[3] [назад] Правило цепочки - формула вычисления производной сложной функции. Например, если f и g являются функциями, а h - сложная функция `h(x) = f(g(x))`, то правило утверждает: `h'(x) = f'(g(x)) * g'(x)`, где `f'(x)` обозначает частную производную f по x.

Глава 11. Создание нейросетевого детектора вредоносных программ с Keras

Десять лет назад создание работоспособной, масштабируемой и быстрой нейронной сети занимало много времени и требовало значительного объёма кода. Однако в последние несколько лет процесс стал гораздо менее мучительным благодаря появлению всё большего количества высокоуровневых интерфейсов проектирования нейронных сетей. Один из них - пакет Python Keras.

В этой главе я пошагово покажу создание учебной нейронной сети средствами Keras. Сначала объясню, как определить архитектуру модели. Затем мы обучим модель отличать безвредные HTML-файлы от вредоносных и научимся сохранять и загружать такие модели. После этого с помощью пакета Python sklearn вы научитесь оценивать точность модели на проверочных данных. Наконец, используем полученные знания, чтобы встроить отчёт о проверочной точности в процесс обучения модели.

Советую читать главу вместе с относящимся к ней кодом из данных, прилагаемых к книге, и редактировать его. Там находится весь обсуждаемый код, организованный в параметризованные функции для упрощения запуска и настройки, а также несколько дополнительных примеров. К концу главы вы будете готовы начать создавать собственные сети.

Для запуска листингов нужно не только установить пакеты из файла `ch11/requirements.txt` командой `pip install -r requirements.txt`, но и следовать инструкциям по установке одного из серверных механизмов Keras: TensorFlow, Theano или CNTK. TensorFlow устанавливается по инструкциям на tensorflow.org.

Определение архитектуры модели

Чтобы построить нейронную сеть, нужно определить её архитектуру: где расположены нейроны, как они соединяются с последующими нейронами и как данные проходят через всю структуру. К счастью, Keras предоставляет простой и гибкий интерфейс для такого определения. Фактически Keras поддерживает два похожих синтаксиса, но мы воспользуемся функциональным API, поскольку он гибче и мощнее другого, "последовательного", синтаксиса.

При проектировании нужны три компонента: вход, обрабатывающая его середина и выход. Иногда модель имеет несколько входов, несколько выходов и очень сложную середину. Но основная идея определения архитектуры состоит в том, чтобы задать, как входные данные, например признаки HTML-файла, проходят через разные нейроны середины, пока последние нейроны не выдадут выход.

Для определения архитектуры Keras использует слои. Слой - группа нейронов, которые применяют один вид функции активации, получают данные от предыдущего слоя и отправляют выходы следующему слою. Обычно входные данные поступают исходному слою нейронов, тот отправляет выходы последующему, далее ещё одному и так далее, пока последний слой не создаст окончательный выход сети.

Листинг 11-1 содержит пример простой модели, определённой синтаксисом функционального API Keras. Советую открыть новый файл Python, вводить код и запускать его самостоятельно по мере построчного разбора. Можно также запускать относящийся к главе код из материалов книги: копировать части `ch11/model_architecture.py` в сеанс IPython или выполнить `python ch11/model_architecture.py` в терминале.

```
from keras import layers  # (1)
```

```
from keras.models import Model  # (2)

input = layers.Input(shape=(1024,), dtype='float32')  # (3) (4)
middle = layers.Dense(units=512, activation='relu')(input)  # (5)
output = layers.Dense(units=1, activation='sigmoid')(middle)  # (6)
model = Model(inputs=input, outputs=output)  # (7)
model.compile(optimizer='adam',  # (8)
              loss='binary_crossentropy',  # (9)
              metrics=['accuracy'])  # (10)
```

Листинг 11-1. Определение простой модели синтаксисом функционального API

Сначала импортируется подмодуль `layers` пакета Keras (1) и класс `Model` из подмодуля `models` (2).

Затем задаётся вид данных одного наблюдения, принимаемого моделью: функции `layers.Input()` передаются форма, кортеж целых чисел (3), и тип данных, строка (4). Здесь объявлено, что вход представляет собой массив из 1024 чисел с плавающей точкой. Если бы входом была, например, матрица целых чисел, первая строка выглядела бы примерно так: `input = Input(shape=(100, 100,), dtype='int32')`.

> **Примечание.** Если в одном измерении модель принимает входы переменного размера, вместо числа можно использовать `None`, например `(100, None,)`.

Далее задаётся слой нейронов, которому будут переданы входные данные. Снова используется импортированный подмодуль `layers`, а именно функция `Dense` (5), задающая плотно, или полносвязный, слой. Каждый выход предыдущего слоя передаётся каждому нейрону данного слоя. `Dense` - самый распространённый вид слоя при разработке моделей Keras. Другие слои позволяют, например, менять форму данных, `Reshape`, или реализовывать собственный слой, `Lambda`.

Функции `Dense` передаются два аргумента: `units=512` задаёт 512 нейронов слоя, а `activation='relu'` сообщает, что они являются выпрямленными линейными нейронами ReLU. Напомним из главы 10: ReLU использует простую функцию активации, выводящую большее из двух значений - 0 или взвешенную сумму входов нейрона. Выражение `layers.Dense(units=512, activation='relu')` определяет слой, а последняя часть строки, `(input)`, объявляет его входом объект `input`. Важно понимать, что поток данных модели определяется именно передачей `input` слою, а не порядком строк кода.

В следующей строке определяется выходной слой, снова функцией `Dense`. Но на этот раз в нём только один нейрон и используется сигмоидальная функция активации (6), отлично объединяющая множество данных в одну оценку от 0 до 1. Выходной слой принимает объект `middle`, то есть выходы всех 512 нейронов среднего слоя передаются этому нейрону.

Определив слои, класс `Model` из подмодуля `models` объединяет их в модель (7). Обратите внимание: нужно указать только входные и выходные слои. Поскольку каждый слой после первого получает предыдущий как вход, окончательный выходной слой содержит всю необходимую модели информацию о предыдущих слоях. Между входом и выходом можно объявить ещё десять средних слоёв, но строка (7) останется прежней.

Компиляция модели

Наконец, модель нужно скомпилировать. Её архитектура и поток данных определены, но ещё не задан способ обучения. Для этого вызывается метод `compile` модели с тремя параметрами:

• Первый параметр `optimizer` (8) задаёт вид алгоритма обратного распространения. Имя алгоритма можно передать строкой, как здесь, либо импортировать алгоритм непосредственно из `keras.optimizers`, чтобы задать конкретные параметры или даже создать собственный.

• Параметр `loss` (9) задаёт величину, минимизируемую во время обучения, то есть обратного распространения. Точнее, это формула разности между истинными обучающими метками и предсказанными моделью метками, выходом. Можно указать имя функции потерь либо передать саму функцию, например `keras.losses.mean_squared_error`.
• Наконец, параметр `metrics` (10) принимает список показателей, которые Keras должен сообщать при анализе работы модели во время и после обучения. Здесь также можно передавать строки или сами функции, например `['categorical_accuracy', keras.metrics.top_k_categorical_accuracy]`.

После выполнения листинга 11-1 запустите `model.summary()`, чтобы вывести структуру модели. Результат должен быть похож на рис. 11-1.

```
In [2]: model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 525,313
Trainable params: 525,313
Non-trainable params: 0
_________________________________________________________________
```

Рисунок 11-1. Вывод `model.summary()`

Рисунок 11-1 показывает вывод `model.summary()`. На экран печатается описание каждого слоя и количество связанных с ним параметров. Например, слой `dense_1` имеет 524 800 параметров, потому что каждый из 512 нейронов получает копию каждого из 1024 входных значений входного слоя, то есть существует 1024 x 512 весов. Прибавив 512 смещений, получим 1024 x 512 + 512 = 524 800.

Хотя модель ещё не обучалась и не проверялась на проверочных данных, это скомпилированная модель Keras, готовая к обучению.

> **Примечание.** В примере `ch11/model_architecture.py` приведена немного более сложная модель.

Обучение модели

Для обучения требуются обучающие данные. Прилагаемая к книге виртуальная машина содержит набор примерно из полумиллиона безвредных и вредоносных HTML-файлов. Они находятся в двух каталогах: безвредные в `ch11/data/html/benign_files/`, а вредоносные в `ch11/data/html/malicious_files/`. Не открывайте эти файлы в браузере! В этом разделе мы обучим сеть предсказывать, является HTML-файл безвредным, 0, или вредоносным, 1.

Извлечение признаков

Сначала нужно решить, как представить данные, то есть какие признаки каждого HTML-файла извлечь для входа модели. Можно, например, просто передавать первые 1000 символов каждого файла, частоты всех букв алфавита или разработать более сложные признаки с помощью анализатора HTML. Чтобы упростить задачу, каждый потенциально очень большой HTML-файл переменной длины будет преобразован в сжатое представление постоянного размера, позволяющее модели быстро обрабатывать и изучать важные закономерности.

В этом примере каждый HTML-файл преобразуется в вектор длины 1024 со счётчиками категорий. Каждый счётчик представляет количество токенов HTML-файла, хеш которых попал в данную категорию. Код извлечения приведён в листинге 11-2.

```
import numpy as np
import murmur
import re
import os

def read_file(sha, dir):
    with open(os.path.join(dir, sha), 'r') as fp:
        file = fp.read()
    return file
def extract_features(sha, path_to_files_dir,
                     hash_dim=1024, split_regex=r"\s+"):  # (1)
    file = read_file(sha=sha, dir=path_to_files_dir)  # (2)
    tokens = re.split(pattern=split_regex, string=file)  # (3)
    # Теперь взять хеш каждого токена по модулю, чтобы каждый токен
    # был заменён корзиной, то есть категорией, от 1 до hash_dim.
    token_hash_buckets = [
        (murmur.string_hash(w) % (hash_dim - 1) + 1) for w in tokens  # (4)
    ]
    # Наконец, подсчитать попадания в каждую корзину, чтобы признаки
    # всегда имели длину hash_dim независимо от размера HTML-файла.
    token_bucket_counts = np.zeros(hash_dim)
    # Возвращает частоты каждого уникального значения в token_hash_buckets.
    buckets, counts = np.unique(token_hash_buckets, return_counts=True)
    # Вставить счётчики в объект token_bucket_counts.
    for bucket, count in zip(buckets, counts):
        token_bucket_counts[bucket] = count  # (5)
    return np.array(token_bucket_counts)
```

Листинг 11-2. Код извлечения признаков

Чтобы понять работу Keras, необязательно разбираться во всех деталях, но советую прочитать комментарии к коду.

Функция `extract_features` начинает с чтения HTML-файла как большой строки (2), а затем разбивает её регулярным выражением на набор токенов (3). Далее вычисляется числовой хеш каждого токена, и хеши распределяются по категориям взятием остатка (4). Окончательные признаки представляют собой количество хешей в каждой категории (5), подобно счётчикам корзин гистограммы. При желании можно изменить регулярное выражение `split_regex` (1), разбивающее HTML-файл на части, и посмотреть, как меняются токены и признаки.

Если вы пропустили объяснение или поняли не всё, ничего страшного. Достаточно знать, что `extract_features` принимает путь к HTML-файлу и преобразует его в массив признаков длины 1024 либо значения `hash_dim`.

Создание генератора данных

Теперь модель Keras должна фактически обучаться на этих признаках. При небольшом объёме данных, уже загруженных в память, модель можно обучить простой строкой из листинга 11-3.

```
# Сначала каким-либо способом загрузить my_data и my_labels, затем:
model.fit(my_data, my_labels, epochs=10, batch_size=32)
```

Листинг 11-3. Обучение модели, когда данные уже загружены в память

Однако при больших объёмах это бесполезно, потому что все обучающие данные невозможно одновременно поместить в память компьютера. Чтобы обойти проблему, используется немного более сложная, но лучше масштабируемая функция `model.fit_generator`. Вместо всех обучающих данных ей передаётся генератор, выдающий данные пакетами, чтобы оперативная память компьютера не захлебнулась.

Генераторы Python работают подобно функциям, но содержат инструкцию `yield`. Вместо возврата одного результата генератор возвращает объект, который можно вызывать снова и снова для получения множества или бесконечного количества наборов результатов. В листинге 11-4 показано создание генератора данных с помощью функции извлечения признаков.

```
def my_generator(benign_files, malicious_files,
                 path_to_benign_files, path_to_malicious_files,
                 batch_size, features_length=1024):
    n_samples_per_class = batch_size / 2
    assert len(benign_files) >= n_samples_per_class  # (1)
    assert len(malicious_files) >= n_samples_per_class
    while True:  # (2)
        ben_features = [
            extract_features(sha, path_to_files_dir=path_to_benign_files,
                             hash_dim=features_length)
            for sha in np.random.choice(benign_files, n_samples_per_class,
                                        replace=False)
        ]
        mal_features = [
            extract_features(sha, path_to_files_dir=path_to_malicious_files,  # (3)
                             hash_dim=features_length)
            for sha in np.random.choice(malicious_files, n_samples_per_class,  # (4)
                                        replace=False)
        ]
        all_features = ben_features + mal_features  # (5)
        labels = [0 for i in range(n_samples_per_class)] + [1 for i in range(
                  n_samples_per_class)]

        idx = np.random.choice(range(batch_size), batch_size)
        all_features = np.array([np.array(all_features[i]) for i in idx])  # (6)
        labels = np.array([labels[i] for i in idx])
        yield all_features, labels  # (7)
```

Листинг 11-4. Написание генератора данных

Сначала две инструкции `assert` проверяют достаточность данных (1). Затем внутри бесконечного цикла `while` (2) выбирается случайная выборка (4) ключей безвредных и вредоносных файлов и для них извлекаются признаки функцией `extract_features` (3). После этого безвредные и вредоносные признаки и связанные метки 0 и 1 объединяются (5) и перемешиваются (6). Наконец, признаки и метки выдаются инструкцией `yield` (7).

После создания при каждом вызове метода `next()` этот генератор должен выдавать модели `batch_size` признаков и меток для обучения: 50 процентов вредоносных и 50 процентов безвредных.

В листинге 11-5 показано, как создать генератор обучающих данных по материалам книги и обучить модель, передав его методу `fit_generator`.

```
import os
```

```
batch_size = 128
features_length = 1024
path_to_training_benign_files = 'data/html/benign_files/training/'
path_to_training_malicious_files = 'data/html/malicious_files/training/'
steps_per_epoch = 1000 # Искусственно мало для скорости примера!

train_benign_files = os.listdir(path_to_training_benign_files)  # (1)
train_malicious_files = os.listdir(path_to_training_malicious_files)  # (2)

# Создать генератор обучающих данных.
training_generator = my_generator(  # (3)
    benign_files=train_benign_files,
    malicious_files=train_malicious_files,
    path_to_benign_files=path_to_training_benign_files,
    path_to_malicious_files=path_to_training_malicious_files,
    batch_size=batch_size,
    features_length=features_length
)

model.fit_generator(  # (4)
    generator=training_generator,  # (5)
    steps_per_epoch=steps_per_epoch,  # (6)
    epochs=10  # (7)
)
```

Листинг 11-5. Создание генератора обучающих данных и его применение для обучения модели

Прочитайте код и постарайтесь понять происходящее. После импорта необходимого пакета и создания переменных параметров имена файлов безвредных (1) и вредоносных (2) обучающих данных считываются в память, но не сами файлы. Эти значения передаются новой функции `my_generator` (3), и получается генератор. Наконец, встроенный метод `fit_generator` модели из листинга 11-1 начинает обучение (4).

Метод принимает три параметра. `generator` (5) задаёт генератор данных для каждого пакета. Во время обучения параметры обновляются один раз на пакет путём усреднения сигналов всех обучающих наблюдений этого пакета. `steps_per_epoch` (6) задаёт количество пакетов, обрабатываемых за одну эпоху. Поэтому общее количество наблюдений за эпоху равно `batch_size*steps_per_epoch`. По соглашению число наблюдений модели за эпоху должно совпадать с размером набора, но здесь и в примере виртуальной машины `steps_per_epoch` уменьшен для ускорения. Параметр `epochs` (7) задаёт количество эпох.

Попробуйте запустить код в прилагаемом каталоге `ch11/`. В зависимости от мощности компьютера каждая эпоха займёт некоторое время. В интерактивном сеансе процесс можно отменить после нескольких эпох сочетанием Ctrl+C, если он идёт долго. Это остановит обучение без потери прогресса. После отмены или завершения кода у вас будет обученная модель. Вывод на экране виртуальной машины должен быть похож на рис. 11-2.

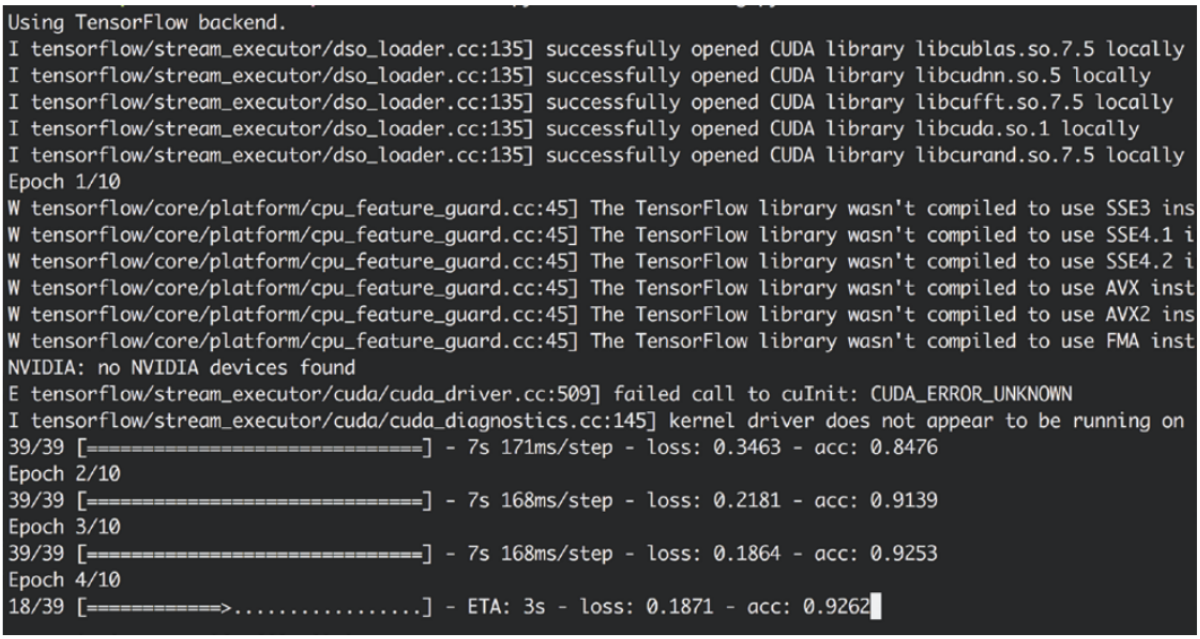

Рисунок 11-2. Консольный вывод обучения модели Keras

В нескольких верхних строках сообщается о загрузке TensorFlow, стандартного серверного механизма Keras. Также показаны предупреждения, как на рис. 11-2. Они лишь означают, что обучение выполняется на центральных, а не графических процессорах. Графические процессоры часто обучают нейронные сети примерно в 2-20 раз быстрее, но для целей книги достаточно CPU. Наконец, для каждой эпохи отображается индикатор хода, оставшееся время, функция потерь и точность.

Включение проверочных данных

В предыдущем разделе вы научились масштабируемо обучать модель Keras на HTML-файлах методом `fit_generator`. Как вы видели, во время обучения модель выводит текущие показатели потерь и точности каждой эпохи. Но в действительности важно, как обученная модель работает на проверочных данных, которых никогда прежде не видела. Они лучше представляют данные будущей производственной среды.

При проектировании более совершенных моделей и выборе продолжительности обучения следует максимизировать проверочную, а не обучающую точность, показанную на рис. 11-2. Ещё лучше использовать проверочные файлы с датами после обучающих данных, точнее имитируя производственную среду.

В листинге 11-6 показана загрузка проверочных признаков в память функцией `my_generator` из листинга 11-4.

```
import os
path_to_validation_benign_files = 'data/html/benign_files/validation/'
path_to_validation_malicious_files = 'data/html/malicious_files/validation/'
# Получить ключи проверочных данных.
val_benign_file_keys = os.listdir(path_to_validation_benign_files)
val_malicious_file_keys = os.listdir(path_to_validation_malicious_files)
# Получить проверочные данные и извлечь признаки.
validation_data = my_generator(  # (1)
    benign_files=val_benign_files,
    malicious_files=val_malicious_files,
    path_to_benign_files=path_to_validation_benign_files,
    path_to_malicious_files=path_to_validation_malicious_files,
```

```
    batch_size=10000,  # (2)
    features_length=features_length
).next()  # (3)
```

Листинг 11-6. Чтение проверочных признаков и меток в память функцией `my_generator`

Код очень похож на создание генератора обучающих данных, но пути изменились и теперь все проверочные данные нужно загрузить в память. Поэтому вместо простого создания генератора создаётся генератор проверочных данных (1) с большим `batch_size` (2), равным количеству файлов, на которых выполняется проверка, и сразу один раз вызывается его метод `.next()` (3).

Загрузив проверочные данные, можно просто передать их `fit_generator()` во время обучения, как в листинге 11-7.

```
model.fit_generator(
    validation_data=validation_data,  # (1)
    generator=training_generator,
    steps_per_epoch=steps_per_epoch,
    epochs=10
)
```

Листинг 11-7. Автоматическое наблюдение за обучением с помощью проверочных данных

Листинг 11-7 почти совпадает с концом листинга 11-5, но теперь `fit_generator` получает `validation_data` (1). Это улучшает наблюдение за моделью: проверочные потери и точность вычисляются вместе с обучающими.

Теперь сообщения обучения должны выглядеть примерно как на рис. 11-3.

Рисунок 11-3. Консольный вывод обучения модели Keras с проверочными данными

Рисунок 11-3 похож на рис. 11-2, но теперь помимо обучающих показателей `loss` и `acc` Keras вычисляет и показывает для каждой эпохи `val_loss`, проверочные потери, и `val_acc`, проверочную точность. Если проверочная точность снижается, это обычно указывает на переобучение на обучающих данных и обучение лучше остановить. Если она растёт, как здесь, модель продолжает улучшаться и обучение следует продолжать.

Сохранение и загрузка модели

Теперь, когда вы умеете строить и обучать нейронную сеть, рассмотрим её сохранение для передачи другим людям.

В листинге 11-8 показано сохранение обученной модели в файл `.h5` (1) и её последующая загрузка (2), возможно позднее.

```
from keras.models import load_model
# Сохранить модель.
model.save('my_model.h5')  # (1)
# Снова загрузить модель из файла в память.
same_model = load_model('my_model.h5')  # (2)
```

Листинг 11-8. Сохранение и загрузка моделей Keras

Оценка модели

В разделе об обучении наблюдались некоторые стандартные показатели: обучающие потери и точность, а также проверочные потери и точность. Теперь рассмотрим более сложные показатели для лучшей оценки моделей.

Один полезный показатель точности двоичного предиктора называется площадью под кривой, AUC. Под кривой понимается ROC-кривая, см. главу 8, которая сопоставляет доли ложноположительных результатов по оси x с долями истинно положительных по оси y для всех возможных порогов оценки.

Например, модель пытается определить вредоносность файла оценкой от 0, безвредный, до 1, вредоносный. При сравнительно высоком пороге для классификации файла как вредоносного будет меньше ложноположительных результатов, что хорошо, но и меньше истинно положительных, что плохо. При низком пороге, напротив, доля ложноположительных, вероятно, будет высокой, что плохо, но доля обнаружения - очень высокой, что хорошо.

Эти две возможности представлены двумя точками ROC-кривой модели: первая находится ближе к левой стороне, вторая - к правой. AUC охватывает все возможности, просто измеряя площадь под ROC-кривой, как на рис. 11-4.

Проще говоря, AUC 0,5 соответствует предсказательной способности подбрасывания монеты, а AUC 1 - безупречному предсказанию.

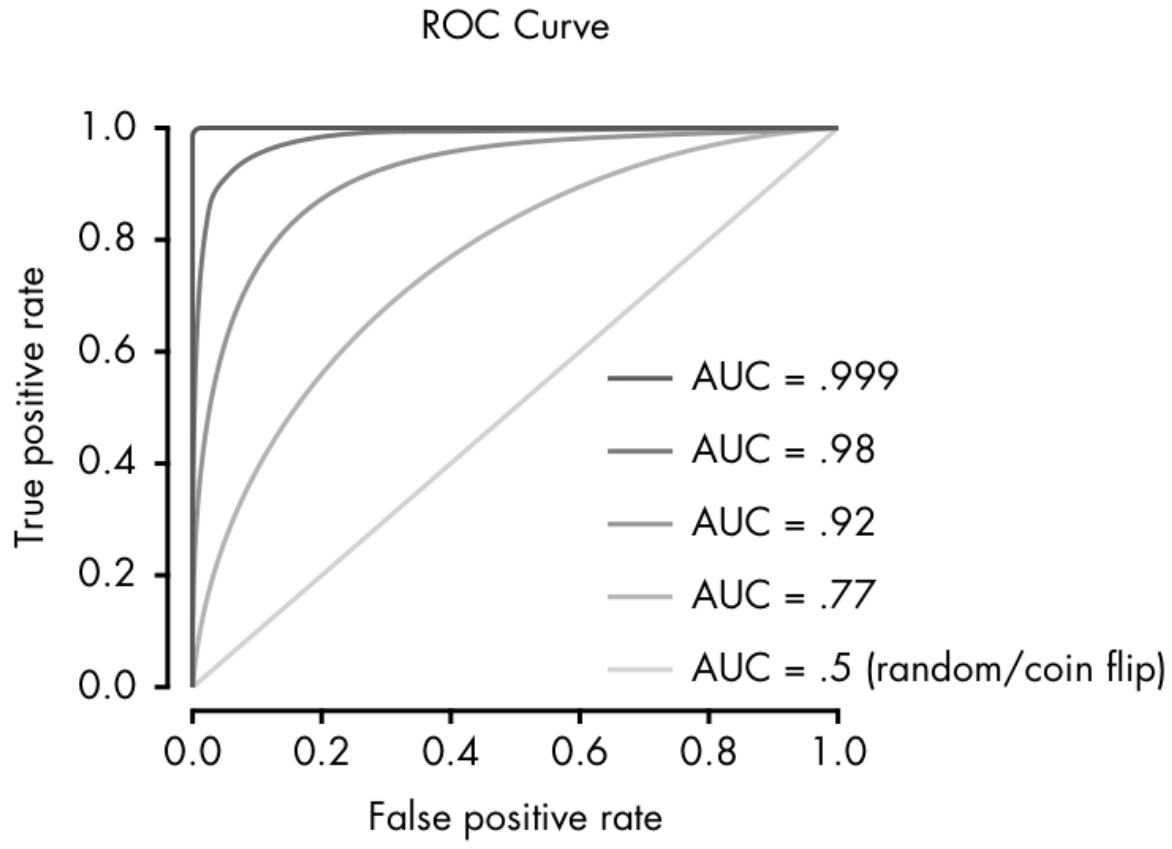

Рисунок 11-4. Примеры ROC-кривых. Каждой линии соответствует своё значение AUC

Вычислим проверочную AUC по проверочным данным с помощью кода листинга 11-9.

```
from sklearn import metrics

validation_labels = validation_data[1]  # (1)
validation_scores = [el[0] for el in model.predict(validation_data[0])]  # (2)
fpr, tpr, thres = metrics.roc_curve(y_true=validation_labels,  # (3)
                                    y_score=validation_scores)
auc = metrics.auc(fpr, tpr)  # (4)
print('Validation AUC = {}'.format(auc))
```

Листинг 11-9. Вычисление проверочной AUC подмодулем `metrics` *библиотеки sklearn*

Здесь кортеж `validation_data` разделяется на два объекта: проверочные метки `validation_labels` (1) и развёрнутые предсказания модели `validation_scores` (2). Затем функция `metrics.roc_curve` библиотеки sklearn вычисляет доли ложноположительных и истинно положительных результатов и связанные пороги для предсказаний модели (3). По ним другой функцией sklearn вычисляется AUC (4).

Хотя код функции здесь не разбирается, для построения самой ROC-кривой можно также использовать `roc_plot()` из файла `ch11/model_evaluation.py` в материалах книги, как показано в листинге 11-10.

```
from ch11.model_evaluation import roc_plot
roc_plot(fpr=fpr, tpr=tpr, path_to_file='roc_curve.png')
```

Листинг 11-10. Создание графика ROC-кривой функцией `roc_plot` *из файла* `ch11/model_evaluation.py` *материалов книги*

Выполнение кода листинга 11-10 должно создать сохранённый в `roc_curve.png` график, похожий на рис. 11-5.

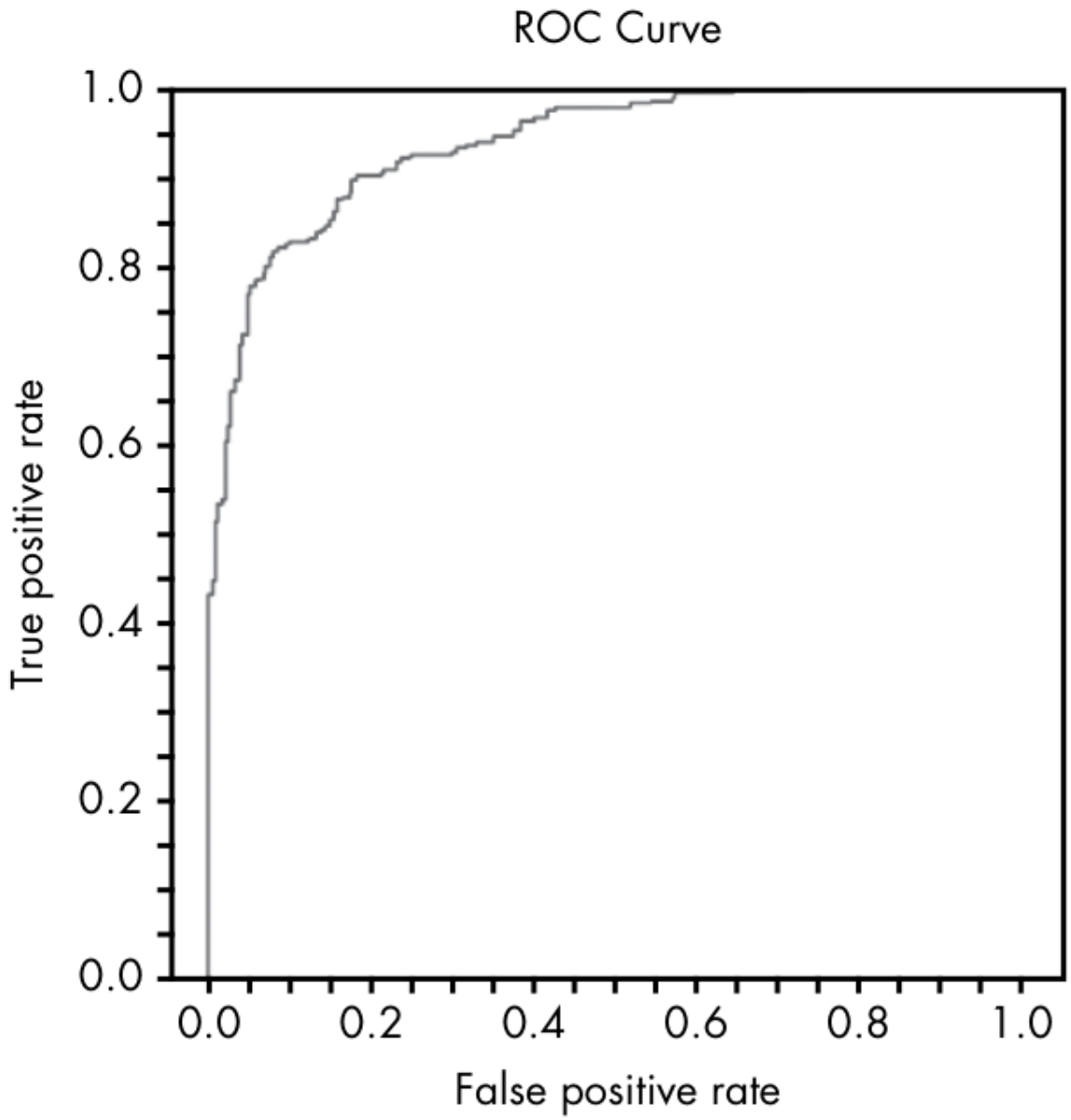

Рисунок 11-5. ROC-кривая!

Каждая точка ROC-кривой на рис. 11-5 представляет конкретную долю ложноположительных результатов по оси x и долю истинно положительных по оси y, связанную с одним из порогов предсказания модели от 0 до 1. С ростом доли ложноположительных растёт доля истинно положительных и наоборот. В производственной среде обычно приходится выбирать один порог, то есть одну точку кривой, если проверочные данные имитируют производственные, и принимать решения с учётом готовности терпеть ложные срабатывания и риска пропустить вредоносный файл.

Улучшение обучения модели с помощью обратных вызовов

До сих пор вы научились проектировать, обучать, сохранять, загружать и оценивать модели Keras. Этого достаточно для неплохого начала, но я также хочу познакомить вас с обратными вызовами Keras, способными ещё больше улучшить обучение.

Обратный вызов Keras представляет набор функций, применяемых на определённых этапах обучения. Например, он может обеспечивать сохранение файла `.h5` в конце каждой эпохи или вывод проверочной AUC после каждой эпохи. Это помогает записывать состояние модели и точнее сообщать о её работе во время обучения.

Сначала воспользуемся встроенным обратным вызовом, а затем напишем собственный.

Применение встроенного обратного вызова

Чтобы использовать встроенный обратный вызов, достаточно передать его экземпляр методу `fit_generator()` во время обучения. Мы применим `callbacks.ModelCheckpoint`, который после каждой эпохи оценивает проверочные потери и сохраняет текущую модель в файл, если потери меньше, чем во всех предыдущих эпохах. Обратному вызову нужен доступ к проверочным данным, поэтому они также передаются `fit_generator()`, как показано в листинге 11-11.

```
from keras import callbacks

model.fit_generator(
    generator=training_generator,
    # Уменьшить steps_per_epoch для быстрого выполнения примера.
    steps_per_epoch=50,
    epochs=5,
    validation_data=validation_data,
    callbacks=[
        callbacks.ModelCheckpoint(save_best_only=True,  # (1)
                                  filepath='results/best_model.h5',  # (2)
                                  monitor='val_loss')  # (3)
    ],
)
```

Листинг 11-11. Добавление обратного вызова `ModelCheckpoint` *в обучение*

Код обеспечивает перезапись модели (1) в единственный файл `results/best_model.h5` (2) всякий раз, когда `val_loss` (3), то есть проверочные потери, достигают нового минимума. Поэтому текущая сохранённая модель всегда представляет лучший по проверочным потерям вариант среди всех завершённых эпох.

Вместо этого код листинга 11-12 может сохранять модель после каждой эпохи в отдельный файл независимо от проверочных потерь.

```
callbacks.ModelCheckpoint(save_best_only=False,  # (4)
    filepath='results/model_epoch_{epoch}.h5',  # (5)
    monitor='val_loss')
```

Листинг 11-12. Добавление обратного вызова `ModelCheckpoint`*, сохраняющего модель после каждой эпохи в новый файл*

Используется тот же код и та же функция `ModelCheckpoint`, что в листинге 11-11, но `save_best_only=False` (4) и путь просит Keras подставить номер эпохи (5). Вместо сохранения только одной "лучшей" версии обратный вызов записывает вариант каждой эпохи в `results/model_epoch_0.h5`, `results/model_epoch_1.h5`, `results/model_epoch_2.h5` и так далее.

Применение собственного обратного вызова

Хотя Keras не поддерживает AUC, можно создать собственный обратный вызов, например для вывода AUC на экран после каждой эпохи.

Для этого нужен класс, наследующий `keras.callbacks.Callback`, абстрактный базовый класс новых обратных вызовов. Можно добавить один или несколько методов из набора, которые будут автоматически выполняться во время обучения в моменты, указанные именами: `on_epoch_begin`, `on_epoch_end`, `on_batch_begin`, `on_batch_end`, `on_train_begin` и `on_train_end`.

В листинге 11-13 создан обратный вызов, вычисляющий и выводящий проверочную AUC в конце каждой эпохи.

```
import numpy as np
from keras import callbacks
from sklearn import metrics

class MyCallback(callbacks.Callback):  # (1)

    def on_epoch_end(self, epoch, logs={}):  # (2)
        validation_labels = self.validation_data[1]  # (3)
        validation_scores = self.model.predict(self.validation_data[0])
        # Развернуть оценки.
        validation_scores = [el[0] for el in validation_scores]
        fpr, tpr, thres = metrics.roc_curve(y_true=validation_labels,
                                            y_score=validation_scores)
        auc = metrics.auc(fpr, tpr)  # (4)
        print('\n\tEpoch {}, Validation AUC = {}'.format(epoch,
                                                        np.round(auc, 6)))

model.fit_generator(
    generator=training_generator,
    # Уменьшить steps_per_epoch для быстрого выполнения примера.
    steps_per_epoch=50,
    epochs=5,
    validation_data=validation_data,  # (5)
    callbacks=[  # (6)
        callbacks.ModelCheckpoint('results/model_epoch_{epoch}.h5',
                                  monitor='val_loss',
                                  save_best_only=False,
                                  save_weights_only=False)
    ]
)
```

Листинг 11-13. Создание и применение собственного обратного вызова, выводящего AUC после каждой эпохи обучения

В примере сначала создаётся класс `MyCallback` (1), наследующий `callbacks.Callbacks`. Для простоты переопределяется единственный метод `on_epoch_end` (2) с двумя ожидаемыми Keras аргументами: `epoch` и `logs`, словарём журнальной информации. Оба Keras передаст при вызове функции во время обучения.

Затем берётся `validation_data` (3), уже сохранённый в объекте `self` благодаря наследованию `callbacks.Callback`, и вычисляется и выводится AUC (4), как в разделе "Оценка модели" на странице 209 оригинала. Для работы кода проверочные данные должны быть переданы `fit_generator()`, чтобы во время обучения обратный вызов имел доступ к `self.validation_data` (5). Наконец, модели приказывается обучаться и указывается новый обратный вызов (6). Результат должен быть похож на рис. 11-6.

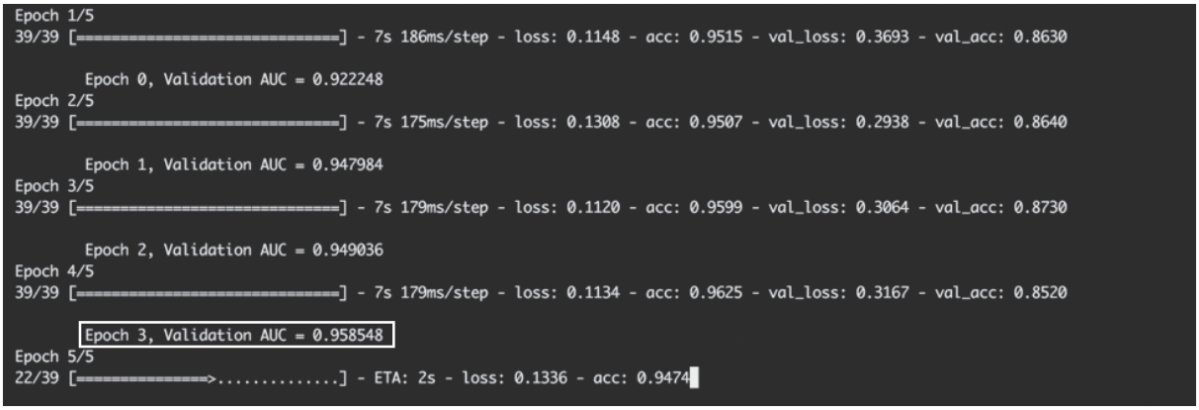

Рисунок 11-6. Консольный вывод обучения модели Keras с собственным обратным вызовом AUC

Если самое важное для вас - минимизация проверочной AUC, этот обратный вызов позволяет легко наблюдать за моделью во время обучения и решить, нужно ли остановить процесс, например если проверочная точность со временем постоянно снижается.

Итоги

В этой главе вы научились создавать собственную нейронную сеть средствами Keras, а также обучать, оценивать, сохранять и загружать её. Затем вы узнали, как улучшить обучение встроенными и собственными обратными вызовами. Советую экспериментировать с кодом из материалов книги и наблюдать, как изменения архитектуры модели и извлечения признаков влияют на точность.

Эта глава предназначена для первого знакомства, а не служит справочником. Самая свежая официальная документация находится на keras.io. Настоятельно советую изучать заинтересовавшие вас аспекты Keras. Надеюсь, эта глава стала хорошей отправной точкой для всех ваших приключений с глубоким обучением в сфере безопасности.

Глава 12. Как стать специалистом по данным

В завершение книги давайте сделаем шаг назад и обсудим, как вы можете построить жизнь и карьеру специалиста по данным о вредоносных программах или, в более общем смысле, специалиста по данным в области безопасности. Хотя эта глава не техническая, она не менее важна, чем технические главы книги, а возможно, даже важнее. Дело в том, что для успеха в профессии специалиста по данным в области безопасности недостаточно просто разбираться в предмете.

В этой главе мы, авторы, расскажем, какими карьерными путями сами пришли к профессиональной работе с данными в области безопасности. Вы получите представление о повседневной жизни такого специалиста и о том, что требуется для эффективной работы с данными. Мы также поделимся советами о том, как подходить к задачам анализа данных и сохранять стойкость перед лицом неизбежных трудностей.

Пути к профессии специалиста по данным в области безопасности

Поскольку наука о данных в области безопасности является новой сферой, путей к этой профессии существует много. Хотя многие специалисты получают формальную подготовку в магистратуре или аспирантуре, многие другие учатся самостоятельно. Например, я вырос в среде компьютерных хакеров 1990-х годов, где научился программировать на C и ассемблере и писать хакерские инструменты для преступных целей. Позднее я получил степень бакалавра, а затем магистра гуманитарных наук, после чего вернулся в мир технологий в качестве разработчика программного обеспечения для безопасности. Попутно по вечерам я самостоятельно изучал визуализацию данных и машинное обучение и в конце концов перешёл на официальную должность специалиста по данным в области безопасности в Sophos, компании, занимающейся исследованиями и разработками в сфере безопасности. Хиллари Сандерс, мой соавтор этой книги, изучала в колледже статистику и экономику, некоторое время работала специалистом по данным, а позднее устроилась специалистом по данным в компанию, занимающуюся безопасностью, и приобрела необходимые знания по безопасности уже в ходе работы.

Наша команда в Sophos столь же разнообразна. У коллег есть степени в самых разных дисциплинах: психологии, науке о данных, математике, биохимии, статистике и информатике. Хотя в науке о данных в области безопасности некоторое преимущество имеют люди с формальной подготовкой по количественным научным методам, в неё входят специалисты с самым разным опытом в этих областях. И хотя научная и количественная подготовка помогает осваивать науку о данных в области безопасности, мой собственный опыт показывает, что прийти в нашу сферу и добиться в ней успеха можно и с нетрадиционным образованием, если вы готовы учиться самостоятельно.

Успех в науке о данных в области безопасности зависит от готовности постоянно узнавать новое. Дело в том, что в нашей сфере практические знания не менее важны, чем теоретические, а практические знания приобретаются в работе, а не во время учёбы.

Готовность узнавать новое важна ещё и потому, что машинное обучение, сетевой анализ и визуализация данных постоянно меняются, поэтому знания, полученные во время учёбы, быстро устаревают. Например, глубокое обучение стало популярным направлением лишь примерно с 2012 года и с тех пор стремительно развивалось, поэтому почти всем специалистам по данным, окончившим учебные заведения раньше, пришлось самостоятельно осваивать эти мощные идеи.

Для тех, кто стремится профессионально заняться наукой о данных в области безопасности, это хорошая новость. Поскольку уже работающие в этой сфере люди вынуждены постоянно самостоятельно приобретать новые навыки, вы можете получить шанс войти в профессию, если такими навыками уже владеете.

Один день из жизни специалиста по данным в области безопасности

Работа специалиста по данным в области безопасности заключается в применении навыков, подобных тем, которым учит эта книга, к сложным задачам безопасности. Но применение этих навыков обычно встроено в более широкий рабочий процесс, требующий и других умений. На рисунке 12-1 показан типичный рабочий процесс специалиста по данным в области безопасности, основанный на нашем опыте и опыте коллег из других компаний и организаций.

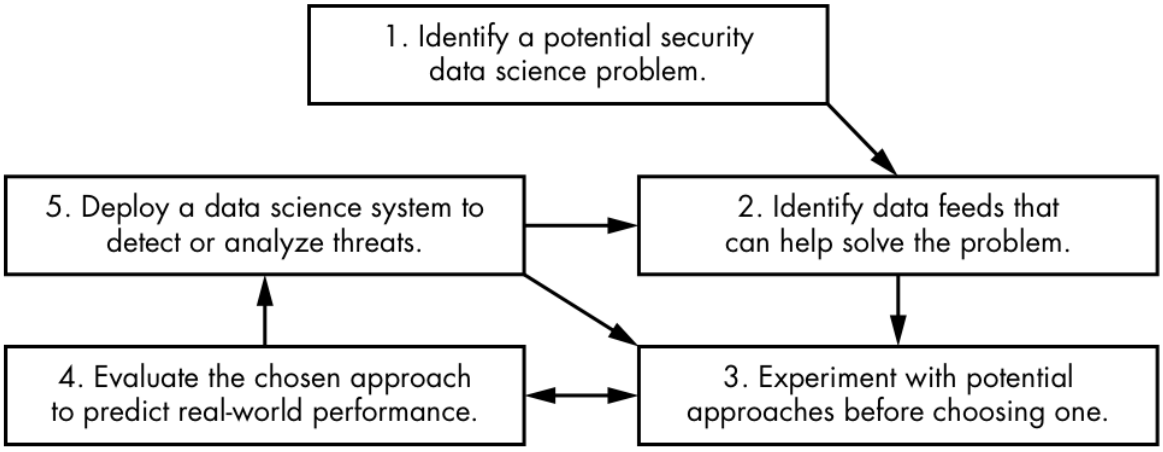

Рисунок 12-1. Модель рабочего процесса науки о данных в области безопасности

Как видно из рисунка 12-1, рабочий процесс науки о данных в области безопасности охватывает пять взаимосвязанных направлений работы. Первое направление, определение проблемы, предполагает выявление проблем безопасности, в решении которых может помочь наука о данных. Например, мы можем выдвинуть гипотезу, что целевые фишинговые письма можно выявлять методами науки о данных, либо решить, что стоит исследовать задачу определения конкретного метода, применённого для обфускации известной вредоносной программы.

На этом этапе любое предположение о том, что данную проблему можно решить с помощью науки о данных, является лишь гипотезой. Когда у вас в руках молоток, то есть наука о данных, каждая проблема может казаться гвоздём, то есть задачей машинного обучения, визуализации данных или сетевого анализа. Нужно обдумать, действительно ли такие проблемы лучше всего решать методами науки о данных, и помнить, что для понимания того, действительно ли наука о данных предлагает наилучшее решение, потребуется создать прототип решения, а затем проверить его.

Когда вы работаете в организации, поиск подходящей проблемы почти всегда требует взаимодействия с заинтересованными сторонами, которые сами не являются специалистами по данным. Например, в нашей компании нам часто приходится общаться с менеджерами по продуктам, руководителями, разработчиками программного обеспечения и сотрудниками отдела продаж, которые считают науку о данных волшебной палочкой, способной решить любую проблему, или приравнивают её к "искусственному интеллекту" и потому приписывают ей некую волшебную способность достигать нереалистичных результатов.

При работе с такими заинтересованными сторонами главное помнить о необходимости честно говорить о возможностях и ограничениях подходов на основе науки о данных, а также сохранять проницательное и взвешенное отношение, чтобы не погнаться за неподходящими проблемами. Следует отвергать проблемы, для которых нет данных, способных обеспечить работу алгоритмов науки о данных, или нет способа оценить, действительно ли ваши подходы работают, а также проблемы, которые явно лучше решаются более ручными методами.

Например, ниже приведены некоторые задачи, за решение которых мы отказались браться, когда нас об этом просили:

• Автоматическое выявление недобросовестных сотрудников, которые могут передавать данные конкурентам. Данных недостаточно для работы алгоритма машинного обучения, но задачу можно было бы исследовать с помощью визуализации данных или сетевого анализа.
• Расшифровка сетевого трафика. Математические основы машинного обучения показывают, что оно просто не способно расшифровывать данные, защищённые шифрованием военного уровня!
• Автоматическое выявление фишинговых писем, вручную составленных для конкретных сотрудников на основе подробных сведений об их образе жизни. И вновь данных недостаточно для работы алгоритма машинного обучения, но задача, возможно, решается посредством визуализации временного ряда или данных электронной почты.

После того как вы успешно определили потенциальную задачу науки о данных в области безопасности, следующим шагом будет поиск потоков данных, которые можно использовать для её решения с помощью методов, изложенных в этой книге. Это показано на шаге 2 рисунка 12-1. В конечном счёте, если у вас нет потоков данных, на которых можно обучать модели машинного обучения, которыми можно наполнять визуализации или которые можно использовать в сетевом анализе для решения выбранной проблемы безопасности, наука о данных, вероятно, вам не поможет.

После выбора проблемы и определения потоков данных, позволяющих построить её решение на основе науки о данных, пора приступать к созданию решения. На деле это происходит в итеративном цикле между шагами 3 и 4 рисунка 12-1: вы что-то создаёте, оцениваете, улучшаете, повторно оцениваете и так далее.

Наконец, когда система готова, вы развёртываете её, как показано на шаге 5 рисунка 12-1. Пока система остаётся развёрнутой, вам придётся возвращаться к ней, подключать новые потоки данных по мере их появления, пробовать новые методы науки о данных и развёртывать новые версии системы.

Качества эффективного специалиста по данным в области безопасности

Успех в науке о данных в области безопасности во многом зависит от вашего отношения к работе. В этом разделе мы перечислим некоторые качества мышления, которые, по нашему опыту, важны для успешной работы с данными в области безопасности.

Непредубеждённость

Данные полны неожиданностей, и они опровергают наши прежние представления о проблеме. Важно сохранять готовность к тому, что данные докажут ошибочность ваших предвзятых суждений. Иначе вы упустите важные знания, которые можно получить из данных, и даже начнёте видеть слишком многое в случайном шуме, убеждая себя в верности ложной теории. К счастью, чем больше вы работаете с данными в области безопасности, тем непредубеждённее относитесь к "обучению" на своих данных и тем спокойнее принимаете то, как мало знаете и как многому вам предстоит научиться при решении каждой новой задачи. Со временем вы начнёте и радоваться неожиданностям в данных, и ожидать их.

Безграничная любознательность

Проекты в области науки о данных сильно отличаются от проектов по разработке программного обеспечения и информационным технологиям, поскольку требуют исследовать данные в поисках закономерностей, выбросов, и тенденций, которые затем используются при создании систем. Выявлять такую динамику нелегко: чтобы получить представление об общей форме данных и скрытых в них историях, часто приходится проводить сотни экспериментов или видов анализа. У одних людей есть почти непреодолимое естественное стремление проводить тщательно продуманные эксперименты и всё глубже исследовать данные, а у других его нет. Именно люди первого типа обычно добиваются успеха в науке о данных. Поэтому любознательность в этой сфере обязательна: именно она определяет, придём ли мы к глубокому пониманию своих данных или лишь к поверхностному. Чем сильнее вам удастся развить любознательное отношение при построении моделей и визуализаций данных, тем полезнее окажутся ваши системы.

Одержимость результатами

После того как вы определили подходящую задачу науки о данных в области безопасности и начали итеративно пробовать решения и оценивать их, вами может овладеть одержимость результатами, особенно в проектах машинного обучения. Это хороший признак. Например, когда я глубоко погружён в проект машинного обучения, у меня круглосуточно и без выходных выполняется несколько экспериментов. Это означает, что я могу просыпаться по нескольку раз за ночь, чтобы проверить их состояние, и нередко должен исправлять ошибки и перезапускать эксперименты в три часа утра. Я обычно проверяю свои эксперименты каждый вечер перед сном и по нескольку раз за выходные.

Такой круглосуточный рабочий процесс часто необходим для создания первоклассных систем науки о данных в области безопасности. Без него легко удовлетвориться посредственными результатами, не сумев выйти из тупика или преодолеть препятствия, возникшие из-за неверных предположений о данных.

Скептическое отношение к результатам

Очень легко обмануть себя, решив, что проект по науке о данных в области безопасности идёт успешно. Например, можно неправильно организовать оценку так, что точность системы будет казаться гораздо выше реальной. Распространённая ошибка - оценивать систему на данных, которые слишком похожи на обучающие или слишком отличаются от реальных. Возможно также, что вы непреднамеренно отобрали только те примеры из сетевой визуализации, которые показались вам полезными, хотя большинство пользователей не видит в них особой ценности. Или вы так много трудились над своим подходом, что убедили себя в хороших результатах оценки, хотя на самом деле они недостаточны, чтобы система приносила пользу в реальном мире. Важно сохранять здоровый скептицизм по отношению к своим результатам, иначе однажды можно оказаться в неловком положении.

Куда двигаться дальше

В этой книге мы рассмотрели многое, но вместе с тем лишь слегка затронули огромный пласт материала. Если книга убедила вас серьёзно заняться наукой о данных в области безопасности, у нас есть две рекомендации. Во-первых, немедленно начните применять освоенные в книге инструменты к волнующим вас проблемам. Во-вторых, читайте больше книг о науке о данных и науке о данных в области безопасности. Вот несколько примеров задач, к которым можно применить приобретённые навыки:

• Выявление вредоносных доменных имён
• Выявление вредоносных URL-адресов
• Выявление вредоносных вложений электронной почты
• Визуализация сетевого трафика для обнаружения аномалий
• Визуализация схем взаимодействия отправителей и получателей электронной почты для выявления фишинговых писем

Для расширения знаний о методах науки о данных мы рекомендуем начать с простого: со статей Википедии об алгоритмах науки о данных, о которых вы хотите узнать больше. Когда речь идёт о науке о данных, Википедия представляет собой на удивление доступный и авторитетный ресурс, к тому же бесплатный. Тем, кто хочет углубиться в предмет, особенно в машинное обучение, мы советуем читать книги по линейной алгебре, теории вероятностей, статистике, анализу графов и многомерному математическому анализу либо проходить бесплатные онлайн-курсы. Изучение этих основ будет приносить плоды на протяжении всей вашей карьеры в науке о данных, поскольку именно на них стоит наша область. Помимо этих фундаментальных дисциплин, рекомендуем также пройти курсы или прочитать более "прикладные" книги о Python, numpy, sklearn, matplotlib, seaborn, Keras и любых других рассмотренных в этой книге инструментах, широко применяемых сообществом специалистов по данным.