Capítulo 2

Aprendizado de Máquina para Segurança: Algoritmos e Aplicações

Fabrício Ceschin[1], Luiz S. Oliveira[1], André Grégio[1]

[1]Universidade Federal do Paraná

Abstract

The massive amount of data produced by security solutions have been creating a strong dependency on automated methods for knowledge discovery. Attacks against computer systems make use of several transmission channels and formats (e.g., network traffic, binary files, text, chained system calls etc.), which difficult their observation among unsuspicious data. Machine learning techniques are a great aid for separating data into classes, but they need to be correctly deployed. In this course, we will show how to adequately apply machine learning algorithms to the security data science process. To do so, we will discuss key concepts about the subject and present practical examples with free, open source tools.

Resumo

A grande quantidade de dados produzidos por soluções de segurança criaram uma grande dependência de métodos automatizados para descoberta de conhecimento. Ataques contra sistemas computacionais utilizam diversos canais de transmissões e formatos (por exemplo, tráfego de rede, arquivos binários, chamadas de sistema encadeadas, etc) que dificultam a sua observação em meio à dados não suspeitos. Técnicas de aprendizado de máquina são uma grande ajuda para separar dados em classes, mas elas precisam ser implantadas corretamente. Neste curso, nós mostraremos como aplicar corretamente algoritmos de aprendizado de máquina ao processo de ciência dos dados em segurança. Para isso, discutiremos conceitos chave sobre o assunto e apresentaremos exemplos práticos com ferramentas grátis e open source.

 © *2019 Sociedade Brasileira de Computação*

2.1. Introdução

Segundo relatório da Symantec [Symantec 2019], cerca de 144 milhões de ataques foram bloqueados na plataforma Windows somente em 2018, uma média de 12 milhões por mês. Essa grande quantidade de ataques cibernéticos em um curto espaço de tempo trás a necessidade de se utilizar métodos inteligentes para análise dos mesmos, tais como sistemas baseados em inteligência artificial, capazes de aprender com os dados recebidos (aprendizado de máquina). O aprendizado de máquina é a ciência de programar computadores para que eles aprendam através de dados para resolver uma determinada tarefa, cuja performance pode ser avaliada através de uma métrica [Gron 2017]. Os dados coletados geralmente pertencem a grupos pré-estabelecidos, cujo agrupamento se dá pela semelhança de seus padrões. No caso de ataques cibernéticos, por exemplo, consideramos a detecção de *malware* como sendo o problema (ou tarefa) composto por duas classes: *malware* (programas malignos) e *goodware* (programas benignos). A semelhança entre os *malware* se dá pelo fato de que esse tipo de programa realiza alguma ação maliciosa cujo objetivo é causar algum dano, alterar ou até mesmo roubar dados. Já os *goodware* são programas cujo comportamento é esperado, sem causar nenhum dos problemas já mencionados. A partir desses dados coletados e das classes definidas, um algoritmo é treinado para identificar se um determinado *software* desconhecido é ou não um *malware*. Para avaliar esse algoritmo treinado (modelo), existem diversas métricas que medem o quão bom é o sistema como um todo, sendo que cada métrica pode apresentar uma perspectiva diferente do problema.

A área de segurança computacional está repleta de resultados de pesquisa e desenvolvimento dependentes de algoritmos de aprendizado de máquina, seja na academia ou na indústria. Entretanto, ao se avaliar os artigos científicos, *white-papers* e até mesmo apresentações de empresas que fazem uso de aprendizado de máquina na detecção ou classificação de ataques, observa-se que não raro as aplicações de tais algoritmos contém problemas, o que acarreta em resultados muitas vezes irreais. A extensão dos problemas varia da coleta e pré-processamento dos dados ao treinamento, classificação e interpretação das informações obtidas. Em problemas de classificação de imagem, por exemplo, os dados possuem distribuição estacionária, isto é, um cachorro sempre será um cachorro e um gato sempre será um gato, isso nunca muda e os modelos assumem que é sempre assim. Entretanto, em segurança, os dados nem sempre possuem essa distribuição: os dados geralmente tem distribuição não-estacionária e mudam conforme o tempo passa, seja para se adaptar a mudanças ou para se aproveitar de falhas desconhecidas (*zero-day*), nos casos de *malware* [Ceschin et al. 2018].

Neste curso, tem-se por objetivo principal promover a aplicação adequada de algoritmos de aprendizado de máquina em dados de segurança, respeitando as etapas básicas necessárias na ciência de dados. Para tanto, visa-se familiarizar o participante com os algoritmos clássicos de aprendizado de máquina e seus usos tradicionais na classificação/detecção e agrupamento de dados que representem instâncias normais/benignas e anômalas/maliciosas. No decorrer do curso, cada etapa do processo de ciência de dados será abordada, desde a coleta, seleção de atributos, escolha do algoritmo, treinamento, avaliação, validação e interpretação dos resultados. Em cada etapa, serão mostradas ferramentas *open source* que auxiliem no cumprimento da tarefa em questão. Todos os códigos fonte e conjunto de dados utilizados neste curso estão disponíveis em um repositório no

github[1].

Este capítulo está organizado da seguinte forma: a Seção 2.2 apresenta como realizar a coleta de dados e rotulá-los de forma correta, destacando os principais problemas relacionados e exemplos de *datasets* (que são utilizados ao longo do curso); na Seção 2.3 destacamos métodos de extração de atributos estáticos e dinâmicos de arquivos desenvolvidos para Windows e Android; a extração de características (utilizando TF-IDF e *Word2Vec*) e normalização das mesmas é o foco da Seção 2.4; a Seção 2.5 apresenta os modelos mais utilizados na literatura de segurança e aprendizado de máquina, dentre eles classificadores, detectores de mudança e agrupadores, incluindo exemplos práticos de como utilizá-los corretamente, bem como as principais bibliotecas; a avaliação dos modelos é o destaque da Seção 2.6, com as principais métricas e modos de validação; na Seção 2.7 apresentamos nossas conclusões.

2.2. Datasets

O conjunto de dados utilizados por um algoritmo de aprendizado de máquina, chamado de *dataset*, é fundamental para o processo de aprendizagem, uma vez que todo algoritmo irá se basear nos exemplares apresentados (conjunto de treino) para tomar uma decisão conforme a tarefa utilizada. Neste minicurso, definimos que um *dataset* pode ser disponibilizado de três formas, cada uma delas relacionada a uma etapa que antecede a criação de um modelo:

1. **Dados Brutos:** os dados brutos são disponibilizados como foram coletados. Um exemplo seriam os executáveis de *malware* em diferentes formatos, tais como PE, ELF e APK, ou tráfego de rede capturado em formato PCAP.

2. **Atributos:** metadados extraídos (geralmente após a coleta e pré-processamento) dos dados brutos são disponibilizados de maneira filtrada, isto é, com menos ruído e mais foco no dado que realmente importa. Um exemplo são *logs* de execução de um software específico ou dados extraídos de seu cabeçalho, ou informações resumidas acerca de um subconjunto do tráfego de rede coletado.

3. **Características:** dados inatos que distinguam amostras, já extraídas dos atributos coletados e processados e prontas para serem utilizadas como entrada no classificador. Um exemplo seria a transformação de *logs* em um vetor para cada software mencionado no passo anterior, cujas posições desse vetor correspondem às características. Outro exemplo seria um arquivo de tráfego que representa um conjunto de origens e/ou destinos de interesse, no qual tenta-se caracterizar cada interação por meio da discretização ou frequência de atributos pré-definidos (ex., protocolos, quantidade de pacotes, bytes, etc.).

Nesta seção focaremos mais em como coletar corretamente os dados para construir um *dataset* e exemplificaremos como rotular um conjunto de dados e quais os problemas comuns que podem acontecer, como a falta de classes para rotulação ou o desbalanceamento das classes. Serão também apresentados *datasets* disponíveis na literatura que

[1] `https://github.com/fabriciojoc/ml-cybersecuritiy-course`

 © 2019 Sociedade Brasileira de Computação

podem ser utilizados para melhor entendimento dos conceitos mostrados nesse cursos, tais como conjuntos de programas maliciosos, mensagens de e-mail não-solicitadas, tráfego de rede e registros de auditoria diversos.

2.2.1. Coletando Dados

A coleta de dados pode ser uma das etapas mais desafiadoras de um projeto, uma vez que todo o problema se baseia nesses dados e pode variar muito de acordo com o domínio e origem dos mesmos. Muitas vezes os dados podem ser provenientes de fontes sensíveis, que não permitem que os dados sejam compartilhados ou utilizados de determinada forma, o que pode acabar comprometendo a reprodutibilidade de uma solução. Se os dados envolvem informações pessoais ou restritas, manter a privacidade dessas informações é necessária (recomenda-se utilizar técnicas de privacidade diferencial nesses casos [Dwork and Roth 2014]). Além disso, recomenda-se coletar o maior número de informações úteis (aquelas que tendem a diferenciar um exemplo de uma classe de outra) possíveis, de forma que ocupe o menor espaço em disco (o que geralmente pode ser uma limitação). Uma informação fundamental que deve-se coletar é o tempo (data de primeira aparição, coleta, desenvolvimento, etc): dados relacionados a segurança geralmente fazem parte de uma distribuição não-estacionária, isto é, não possuem sempre a mesma distribuição e estão em constante mudança, seja para evadir as soluções de segurança ou para se adequar às atualizações de hardware e software [Ceschin et al. 2018]. Nas subseções a seguir trataremos de alguns problemas relacionados a coleta de dados.

2.2.2. Rotulando Dados

Dependendo do domínio do problema, rotular os dados pode ser difícil. Particularmente, os problemas envolvendo *malware* são desafiadores, já que os rótulos são baseados em anti-vírus que usam detecção por assinatura e muitas vezes não possuem o rótulo correto logo quando uma ameaça é encontrada (ela é apenas marcada como uma ameaça para que seja detectada, mas geralmente não é classificada corretamente em sua família desde o início). Tal fato pode complicar a remediação após um ataque, uma vez que dependendo da família do *malware*, diferentes ações podem ser tomadas para voltar o sistema ao estado normal. Por isso, ao rotular os dados de programas maliciosos, recomendamos utilizar a API do VirusTotal[2] em conjunto com a ferramenta AVClass [Sebastián et al. 2016]. Através do VirusTotal, pode-se obter um relatório completo de um determinado arquivo, com o resultado de cada análise dos anti-vírus disponíveis na plataforma. O Código 1 apresenta um exemplo de como obter um relatório utilizando a API (é necessário fornecer sua própria chave de API ao enviar a requisição). Após criar um arquivo JSON (no nosso caso, "samples.json") em que cada linha é um relatório de cada exemplar do **dataset**, deve-ser clonar o repositório do AVClass[3] e executar o comando mostrado no Código 2 (em um terminal), que resultará nos seguintes arquivos:

- **samples.verbose:** criado devido ao parâmetro -v", contém todas as famílias extraídas para cada exemplo, ranqueadas com o número de anti-vírus que o classificaram como determinada família;

[2] `https://developers.virustotal.com/reference`
[3] `https://github.com/malicialab/avclass`

- **samples.families:** criado devido ao parâmetro -fam", contém uma tabela que conta o número de exemplares por família do arquivo de entrada;
- **samples.labels:** criado como saída, contém a família correspondente de cada exemplo de entrada.

```python
import requests
# url de relatorio do virus total
url = 'https://www.virustotal.com/vtapi/v2/file/report'
# chave de api (altere para a sua chave)
API_KEY = "<api_key>"
# parametros: apikey e a chave de api; resource e o md5/sha1/sha256 do arquivo
params = {'apikey': API_KEY, 'resource': '6545c6f328393f9b3168e260ae1b7135c1bfa917'}
# envia requisicao e pega a resposta
response = requests.get(url, params=params)
# salva resposta em um arquivo .json
with open('samples.json', 'w') as out:
    json.dump(response.json(), out)
```

```
{'md5': 'ab243d1fad9ed1af747f313ec88b3fd0',
 ...
 'scans': {
  ...,
  'AVG': {'detected': True,
   'result': 'FileRepMalware',
   'update': '20180325',
   'version': '18.2.3827.0'},
  ...,
  'nProtect': {'detected': False,
   'result': None,
   'update': '20180325',
   'version': '2018-03-25.01'}},
 ...
 'total': 67,
 'verbose_msg': 'Scan finished, information embedded'}
```

Código 1. Obtendo relatório de um exemplar no VirusTotal.

Com o arquivo de saída gerado é possível relacionar os dados originais e obter a família de *malware* correspondente para cada exemplar do *dataset* criado, ”normalizando“ os resultados gerados pelo VirusTotal.

 © 2019 Sociedade Brasileira de Computação

```
1 ./avclass_labeler.py -vt samples.json -v -fam >
    ↪ samples.labels
```

```
1 ab243d1fad9ed1af747f313ec88b3fd0       scriptkd
```

Código 2. Normalizando um arquivo JSON com relatórios do VirusTotal, criando um único arquivo com a respectiva família de cada exemplar.

2.2.3. Problemas

Ao coletar dados e criar um *dataset*, estamos propensos a diversos problemas que podem afetar diretamente a qualidade dos resultados produzidos. Dentre eles, os seguintes:

- **Dados Insuficientes:** as vezes a quantia de dados coletadas pode não ser o suficiente para que o modelo de aprendizado de máquina generalize o problema de forma adequada. Para problemas complexos, tais como reconhecimento de imagens ou fala, podem ser necessárias milhões de imagens [Gron 2017].
- **Dados Não Representativos:** é importante que os dados coletados representem de fato o problema, incluindo novos casos que devem ser generalizados. Caso contrário, a classificação pode apresentar resultados muito ruins, sobretudo ao ser aplicado no mundo real [Gron 2017].
- **Dados com Pouca Qualidade:** dados com erros, *outliers* e ruídos dificultam a identificação de padrões, diminuindo o desempenho da classificação [Gron 2017].
- **Atributos Irrelevantes:** um sistema de aprendizado de máquina será capaz de aprender através de atributos e características relevantes, isto é, aquelas que conseguem diferenciar efetivamente uma classe de outra. Não há segredo: se houver lixo na entrada do sistema, qualquer algoritmo de aprendizado de máquina terá lixo como saída (conhecido como *garbage in, garbage out*) [Gron 2017]. Por isso, a seleção de atributos é tão importante quanto a extração de características e algoritmo de classificação.
- **Falta de Classes:** a falta de classes pode ser um problema dependendo da fonte dos dados coletados: as vezes eles são filtrados (e não representam o problema real) ou até mesmo parte das classes não aparecem onde foram coletadas. Por exemplo, se os dados forem coletados em um país em que determinado ataque não acontece e a solução for aplicada em outro país em que ele acontece, não haverá nenhum exemplo dessa classe para que possa ser feita a classificação.
- **Desbalanceamento de Classes:** muito comum em problemas de segurança, uma vez que determinados tipos de ataques acontecem mais frequentemente que outros, o que é um problema para o classificador, já que pode não generalizar o suficiente as classes menos presentes e ter um viés a favor das classes que aparecem mais.
- ***Delay* de Classes:** a classe de determinado exemplar pode não estar disponível na mesma hora em que o mesmo foi coletado, afetando na atualização do classificador

© 2019 Sociedade Brasileira de Computação

ao obter novos dados, já que o mesmo só poderá conferir se classificou corretamente assim que a classe dele estiver disponível.

2.2.4. Exemplos

Neste curso usaremos dois *datasets* que já foram utilizados em trabalhos anteriores na literatura [Arp et al. 2014, Ceschin et al. 2018] de classificação de *malware*, um de Windows e outro de Android. Ambos estão no formato *CSV* e já contém os atributos extraídos dos dados coletados, o primeiro (de Windows) coletado por nós mesmos e o segundo (Android), melhorado com as informações temporais de cada exemplar, informação crucial para a avaliação correta de um modelo de aprendizado de máquina para segurança.

2.2.4.1. *Brazilian Malware*

O *dataset Brazilian Malware* contém atributos de 50.181 exemplares de *goodware* e *malware* de Windows (do tipo Portable Executable – PE) coletados entre os anos de 2013 e 2019 (21.116 *goodware* e 29.065 *malware*). Os exemplares de *goodware* foram coletados utilizando um *web crawler* em três sites de *download*: *Sourceforge*[4], *Softonic*[5] e *CNET Download*[6]. Os exemplares de *malware* foram obtidos de uma instituição financeira nacional que prefere se manter anônima, advindos de mecanismos de segurança instalados em seus clientes ou coletados de *phishing*. Até o momento deste trabalho, foram coletados $\approx$ 50 GB de binários de *malware*, porém, como a disponibilização de *malware* é proibida no Brasil (e a disponibilização dos mesmos não pode ser feita devido a restrições da instituição financeira), a disponibilidade do *dataset* só pôde ser feita através de atributos (no caso, extraímos atributos estáticos) no formato *CSV*. Além da *hash* SHA1 de cada exemplar, o *dataset* contém as informações dos cabeçalhos dos arquivos *PE* (de forma similar ao apresentado na Seção 2.3.1.1), bem como a entropia do arquivo, bibliotecas, funções importadas e lista de nomes de empacotadores, compiladores ou ferramentas usadas.

2.2.4.2. Drebin

O *dataset* Drebin foi originalmente disponibilizado em 2014, apresentando uma solução para detecção de *malware* no próprio *smartphone*, com taxa de detecção de 94% e baixa taxa de falso positivo, levando cerca de dez segundos em média para analisar um aplicativo [Arp et al. 2014]. O *dataset* contém atributos estáticos extraídos (de forma muito similar ao apresentado na Seção 2.3.1.2, com permissões, funções, URLs, atividades, etc) de 129.013 aplicativos de Android coletados entre 2010 e 2012, sendo 5.560 deles *malware* e 123.453, *goodware*. Como o *dataset* original não possuía informações temporais dos exemplares, nós desenvolvemos uma melhoria do mesmo com esses dados extraídos do Virus Total, considerando que a data de um exemplar é a sua primeira aparição no sistema, o que acabou gerando uma inconsistência, já que alguns exemplares apareceram antes no Virus Total (2009) e alguns, apenas depois (2013 e 2014).

[4] `https://sourceforge.net/`
[5] `https://en.softonic.com/`
[6] `https://download.cnet.com/windows/`

 © 2019 Sociedade Brasileira de Computação

2.3. Obtenção de Atributos

Atributos são fundamentais para o processo de aprendizagem, uma vez que, após terem suas características extraídas e serem rotulados adequadamente, são utilizados como entrada para o treinamento de um classificador. Neste curso, a obtenção de atributos será feita em amostras de programas de Windows e Android por meio da apresentação de ferramentas livres que implementem dois tipos de análise: estática e dinâmica [Gandotra et al. 2014].

2.3.1. Análise Estática

A análise estática consiste em extrair atributos de um *software* sem executá-lo. Geralmente, extraem-se strings de assinaturas, sequência de bytes, chamadas de sistemas, grafo de fluxo de controle, bibliotecas, etc [Gandotra et al. 2014]. Neste curso, abordaremos a extração de atributos estáticos de arquivos *Portable Executable* (PE) e *Android Package Kit* (APK) nas Subseções a seguir.

2.3.1.1. *Portable Executable* (PE)

Para extrair atributos estáticos de arquivos do tipo *Portable Executable* (PE), usaremos a biblioteca *pefile*[7]. A maioria das informações contidas nos cabeçalhos do arquivo PE, seções e seus dados podem ser facilmente acessadas através dessa biblioteca [Yonts 2010, Saxe and Sanders 2018]. O Código 3 apresenta um exemplo de como abrir um arquivo PE utilizando um arquivo executável.

```
1 import pefile
2 # localizacao do arquivo
3 file_location = "./datasets/samples/pe/WinRAR.exe"
4 # abre arquivo
5 pe = pefile.PE(file_location)
```

Código 3. Abrindo um arquivo PE utilizando a biblioteca `pefile`.

Após abrir o arquivo, é possível obter facilmente seus atributos, como mostrado no Código 4, em que as informações do cabeçalho são impressas. No total, são sete atributos que podem ser extraídos: `Machine`, `NumberOfSections`, `TimeDateStamp`, `PointerToSymbolTable`, `NumberOfSymbols`, `SizeOfOptionalHeader` e `Characteristics`. Note que é possível obter cada atributo separadamente ao acessar a variável `FILE_HEADER` do objeto `pe`. Ao mesmo tempo, também é possível obter atributos do cabeçalho opcional do arquivo, basta acessar a variável `OPTIONAL_HEADER` do objeto `pe`.

Além dos atributos do cabeçalho do arquivo, também é possível obter as bibliotecas dinâmicas importadas pelo mesmo. Para isso, basta acessar a variável `DIRECTORY_ENTRY_IMPORT` do objeto `pe`. Esta variável mapeia cada *DLL* utilizada pelo

[7] https://github.com/erocarrera/pefile

```
# obtem atributos do header do arquivo pe
print(pe.FILE_HEADER)
# obtem numero de sessoes do arquivo (para qualquer
 ↪ outro atributo, basta alterar "NumberOfSections")
print(pe.FILE_HEADER.NumberOfSections)
```

```
[IMAGE_FILE_HEADER]
0x114   0x0   Machine:              0x8664
0x116   0x2   NumberOfSections:     0x8
0x118   0x4   TimeDateStamp:        0x5C72EA4B [Sun Feb
 ↪ 24 19:02:35 2019 UTC]
0x11C   0x8   PointerToSymbolTable: 0x0
0x120   0xC   NumberOfSymbols:      0x0
0x124   0x10  SizeOfOptionalHeader: 0xF0
0x126   0x12  Characteristics:      0x22
8
```

Código 4. Obtendo os atributos do cabeçalho do arquivo PE.

programa a um objeto, cujo nome da biblioteca pode ser acessado através da variável `dll`. Um exemplo pode ser visto no Código 5. Através dessa mesma variável, é possível obter todas as funções chamadas por cada biblioteca utilizada pelo programa, como mostrado no Código 6. Mais exemplos de como obter mais dados do arquivo PE podem ser encontrados no repositório do curso. Também recomendamos a leitura deste artigo[8], que também apresenta exemplos interessantes de como utilizar a biblioteca `pefile`.

```
# lista de dlls:
dlls = []
# caminha em DIRECTORY_ENTRY_IMPORT
for d in pe.DIRECTORY_ENTRY_IMPORT:
    # adicionar dll atual na lista
    dlls.append(d.dll)
# imprime lista
print(dlls)
```

```
['KERNEL32.dll', 'USER32.dll', 'GDI32.dll',
 ↪ 'COMDLG32.dll', 'ADVAPI32.dll', 'SHELL32.dll',
 ↪ 'ole32.dll', 'OLEAUT32.dll', 'SHLWAPI.dll',
 ↪ 'POWRPROF.dll', 'COMCTL32.dll', 'UxTheme.dll',
 ↪ 'gdiplus.dll', 'MSIMG32.dll']
```

Código 5. Obtendo bibliotecas dinâmicas do arquivo PE.

[8] https://axcheron.github.io/pe-format-manipulation-with-pefile/

© 2019 Sociedade Brasileira de Computação

```
# lista de simbolos
symbols = []
# caminha em DIRECTORY_ENTRY_IMPORT
for i in pe.DIRECTORY_ENTRY_IMPORT:
    # caminha nas funcoes da biblioca que sao importadas
    for s in i.imports:
        # verifica se o simbolo e valido
        if s.name != None:
            # adiciona a lista de simbolos
            symbols.append(s.name)
# imprime simbolos
print(symbols)
```

```
['DeviceIoControl', 'BackupRead', 'BackupSeek',
 ↪ 'GetShortPathNameW', 'GetLongPathNameW',
 ↪ 'GetFileType', 'GetStdHandle', 'FlushFileBuffers',
 ↪ 'GetFileTime', 'GetDiskFreeSpaceExW',
 ↪ 'GetVersionExW', ... , 'GradientFill']
```

Código 6. Obtendo funções importadas pelo arquivo PE.

2.3.1.2. *Android Package Kit* (APK)

Quando se trata de arquivos do tipo *Android Package Kit* (APK), pode-se utilizar a biblioteca *androguard*[9] para obter atributos estáticos. Com ela é possível obter informações do manifesto do aplicativo, seus recursos, arquivos DEX descompilados e muito mais. Similar ao método utilizado para arquivos PE, primeiramente abre-se o arquivo APK, como mostrado no código 7 (a leitura do APK pode demorar, dependendo do seu tamanho). A diferença aqui se dá ao fato de que a biblioteca nos retorna três objetos: um da classe `APK` (que provê todas as informações do APK), outro um vetor com objetos da classe `DalvikVMFormat` (que corresponde ao arquivo DEX encontrado dentro do APK) e o último da classe `Analysis` (que contém classes especiais para lidar com aplicativos multi-DEX). Neste curso focaremos apenas no primeiro objeto afim de exemplificar a coleta de atributos. Para mais detalhes mais aprofundados sobre a biblioteca, recomenda-se a leitura da documentação completa[10].

[9] `https://github.com/androguard/androguard`
[10] `https://androguard.readthedocs.io/en/latest/index.html`

```
from androguard.misc import AnalyzeAPK
# localizacao do arquivo
file_location =
    "./datasets/samples/apk/com.whatsapp_2.19.203-
    452877_minAPI15(armeabi-
    v7a)(nodpi)_apkmirror.com.apk"
# abre arquivo
a, d, dx = AnalyzeAPK(file_location)
```

Código 7. Abrindo um arquivo APK usando a biblioteca `androguard`.

Após a leitura do arquivo, utilizando os objetos retornados, pode-se obter as permissões e as atividades que estão atrelados ao *AndroidManifest.xml*, como demonstrado nos Códigos 8 e 9. Com o mesmo objeto, é possível também obter as versões do aplicativo (tanto o valor númerico como a string da versão) e do SDK mínimo, máximo e versão alvo, como apresentado no Código 10.

```
print(a.get_permissions())
```

```
['android.permission.AUTHENTICATE_ACCOUNTS',
    'android.permission.CHANGE_WIFI_STATE',
    'android.permission.ACCESS_FINE_LOCATION',
    'android.permission.SEND_SMS', ...,
    'android.permission.MANAGE_OWN_CALLS']
```

Código 8. Obtendo permissões utilizadas pelo APK.

```
print(a.get_activities())
```

```
['org.npci.commonlibrary.GetCredential',
    'com.whatsapp.WebSessionsActivity',
    'com.whatsapp.VoiceMessagingActivity',
    'com.whatsapp.IdentityVerificationActivity', ...,
    'com.google.android.gms.common.api.GoogleApiActivity']
```

Código 9. Obtendo atividades do APK.

2.3.2. Análise Dinâmica

A análise dinâmica consiste em extrair atributos de um *software* executando o mesmo em um ambiente controlado (máquina virtual, simulador, emulador, *sandbox*, etc). Durante a execução, é comum monitorar as chamadas de sistemas e seus parâmetros, fluxo de informação, traços de execução, recursos utilizados, etc [Gandotra et al. 2014]. No Windows, existem diversas estratégias utilizadas para analisar dinamicamente um *software* na

© 2019 Sociedade Brasileira de Computação

```
print(a.get_androidversion_code())
print(a.get_androidversion_name())
print(a.get_min_sdk_version())
print(a.get_max_sdk_version())
print(a.get_target_sdk_version())
```

```
452877
2.19.203
15
None
28
```

Código 10. Obtendo versões do aplicativo (numérico e *string*) e do SDK (mínimo, máximo e versão alvo) utilizados.

literatura [Botacin et al. 2018]. Dentre elas, uma das mais utilizadas é o *Cuckoo Sandbox*[11], uma ferramenta *open source* que automatiza todo o processo de análise dinâmica de diversos arquivos para o Windows, apresentando todo o seu traço de execução (arquivos lidos e alterados, chaves de registros alteradas, etc), tráfego de rede e até mesmo as alterações feitas em memória [Oktavianto and Muhardianto 2013]. Como não é o foco do curso, fica a cargo do leitor ler a documentação referente a obtenção desses dados caso haja interesse, que geralmente precisa de uma extensa lista de passos para ser executada. Também recomendamos a leitura de um minicurso já apresentado no SBSEG de 2011 [Filho et al. 2011]. Já no Android, o tema também é muito extenso e complexo. Por isso recomendamos a leitura dos *surveys* [Hoffmann et al. 2016, Tam et al. 2017] para mais informações sobre os métodos de análise dinâmica desta plataforma.

Relatamos aqui também que é possível obter relatórios de execução dinâmica através do VirusTotal, tanto de Android como de Windows. Entretanto, essa função faz parte da API privada (que é paga). Um exemplo de relatório de *software* do Windows é o Código 11, um *JSON* extraído do VirusTotal (que utiliza uma versão um pouco modificada do *Cuckoo Sandbox*), contendo diversas informações referentes ao comportamento do exemplar, tais como funções executadas (e seus parâmetros) e *hosts* utilizados no tráfego de rede (devido ao espaço ocupado pelo *log* original, compactamos sua representação. Mais informações podem ser obtidas no arquivo original[12]). Um exemplo de análise dinâmica de Android do VirusTotal pode ser visto nesta página[13].

[11] https://cuckoo.sh/docs/
[12] https://tinyurl.com/y36uxamz
[13] https://tinyurl.com/y25wkow6

© 2019 Sociedade Brasileira de Computação

```
1  { "info":{ "started":"2013-02-27 14:44:31",
   ↪ "duration":"15 seconds", "version":"v0.1",
   ↪ "ended":"2013-02-27 14:44:46" },
2    "network":{ "hosts":["0.0.0.0", "255.255.255.255",
     ↪ "10.0.2.2", "10.0.2.15", "239.255.255.250",
     ↪ "224.0.0.22", "65.55.21.14", "10.0.2.255"] },
3    "behavior":{
4       "processes":[
5          {  "parent_id":"1940",
6             "process_id":"2000",
7             "process_name":"6c7a2a4dae13df742a60c0f...",
8             "first_seen":"20130227134444.940",
9             "calls":[
10               {  "category":"device",
                  ↪ "status":"SUCCESS",
11                  "return":"",
12                  "timestamp":"20130227134444.940",
13                  "repeated":6, "api":"DeviceIoControl",
14                  "arguments":[
15                     { "name":"hDevice",
                       ↪ "value":"0x00000044" },
16                     { "name":"dwIoControlCode",
                       ↪ "value":"0x00390008" },
17                     { "name":"lpInBuffer",
                       ↪ "value":"0x77e46318" },
18                     { "name":"nInBufferSize",
                       ↪ "value":"0x00000100" },
19                     { "name":"lpOutBuffer",
                       ↪ "value":"0x0012fbbc" },
20                     { "name":"nOutBufferSize",
                       ↪ "value":"0x00000100" },
21                     { "name":"lpBytesReturned",
                       ↪ "value":"0x0012fbb4" },
22                     { "name":"lpOverlapped",
                       ↪ "value":"0x00000000" } ]
23               } ] } ],
24       "summary":{
25          "files":[ "C:\\6c7a2a4dae13df742a60c0f..." ]
26 } } }
```

Código 11. Exemplo de *log* em *JSON* produzido pelo *Cuckoo Sandbox* do VirusTotal.

 © 2019 Sociedade Brasileira de Computação

2.4. Extração de Características

Muitas vezes os atributos extraídos dos dados não podem ser utilizados diretamente em um modelo de aprendizado de máquina. Nos atributos apresentados na Seção anterior, por exemplo, apenas os atributos numéricos (cujo valor são números reais ou inteiros) poderiam ser utilizados diretamente em um modelo. Já os vetores de palavras extraídas, tais como bibliotecas e funções utilizadas, devem passar por uma etapa a mais chamada de extração de características, cujo objetivo é transformar esses atributos em algo que o classificador consiga "entender", geralmente definindo um mesmo número de características como saída desse processo (já que um programa pode usar mais bibliotecas que outros, por exemplo). Nesta Seção, explicaremos dois métodos de extrair características de textos (conjuntos de palavras, como as extraídas anteriormente). Além disso, também explicaremos como normalizar essas características após a extração.

2.4.1. Métodos

Apresentaremos dois métodos de extração de características de textos, ambos muito utilizados na literatura: TF-IDF e *Word2Vec*. Dado um texto representando um atributo, ambos os métodos produzem como saída um vetor numérico representando de uma forma diferente esse texto. Vale ressaltar que, no caso dos exemplos dados na Seção 2.3, os vetores de palavras extraídos são transformados em textos em que cada palavra é separada por um espaço.

2.4.1.1. TF-IDF

Dado um vocabulário de um conjunto de documentos, isto é, todas as palavras que aparecem neste conjunto, cada documento i é representado por um vetor $\vec{d_i} = (w_{i,1}, w_{i,2}, ..., w_{i,t})$, em que $w_{i,j}$ representa o *TF-IDF* (do inglês *Term Frequency—Inverse Document Frequency*) da *j-ésima* palavra do vocabulário. O *TF-IDF* é uma medida estatística utilizada para avaliar o quão importante uma palavra é para um documento em relação à uma coleção de documentos [Manning et al. 2008a]. Esta medida é obtida através da multiplicação de dois termos:

- *Term Frequency (TF)*: mede com que frequência uma palavra/termo t ocorre em um texto/documento, i.e.,

$$TF(t) = \frac{\text{Número de vezes que } t \text{ aparece no documento}}{\text{Número total de palavras do documento}} \quad (1)$$

- *Inverse Document Frequency (IDF)*: mede o quão importante é uma termo t, i.e.,

$$IDF(t) = \log_e \left(\frac{\text{Número total de documentos}}{\text{Número de documentos com a palavra } t} \right) \quad (2)$$

Resumidamente, cada texto/documento é representado por um vetor esparso que contém suas medidas de *TF-IDF* para cada palavra do vocabulário. O vocabulário pode

ser reduzido a um número V de palavras, sendo essas as que mais estão presentes nos documentos/textos.

Utilizando o *dataset Brazilian Malware* (apresentado na Seção 2.2.4.1), o Código 12 apresenta uma forma de extrair características das listas de bibliotecas utilizadas por cada exemplar com o TF-IDF, através da classe `TfidfVectorizer`[14] da biblioteca scikit-learn [Pedregosa et al. 2011]. Primeiramente, lê-se o *dataset* e inicializamos o extrator de características, no caso com um número máximo de 200 características no total (cada característica representa uma palavra, conforme já mencionado). Então, dividimos os dados de bibliotecas importadas ao meio e utilizamos a primeira metade para treinar o extrator (através do método `fit`), gerando o dicionário de palavras a ser extraído. Finalmente, transformam-se os textos de treino e teste em características (através do método `transform`). Pela saída, pode-se observar que o número de colunas da matriz resultante é 200, por conta do máximo de características que foi configurado ao criar o extrator.

```
1  from sklearn.feature_extraction.text import
     ↪ TfidfVectorizer
2  import pandas as pd
3  # local do dataset
4  data_path = "./datasets/brazilian-malware.csv"
5  # le dataset em CSV
6  data = pd.read_csv(data_path)
7  # obtem lista de dlls
8  texts = data["ImportedDlls"].values
9  # obtem o meio dos dados
10 mid = int((len(texts) + 1)/2)
11 # divide dados em treino e teste
12 train_texts = texts[:mid]
13 test_texts = texts[mid:]
14 # inicializa tfidf com maximo de 200 features
15 vectorizer = TfidfVectorizer(max_features=200)
16 # treina tfidf
17 vectorizer.fit(train_texts)
18 # transforma textos para caracteristicas
19 train_features = vectorizer.transform(train_texts)
20 test_features = vectorizer.transform(test_texts)
21 # imprime a forma das caracateristicas
22 print(train_features.shape, test_features.shape)
```

```
1  (25091, 200) (25090, 200)
```

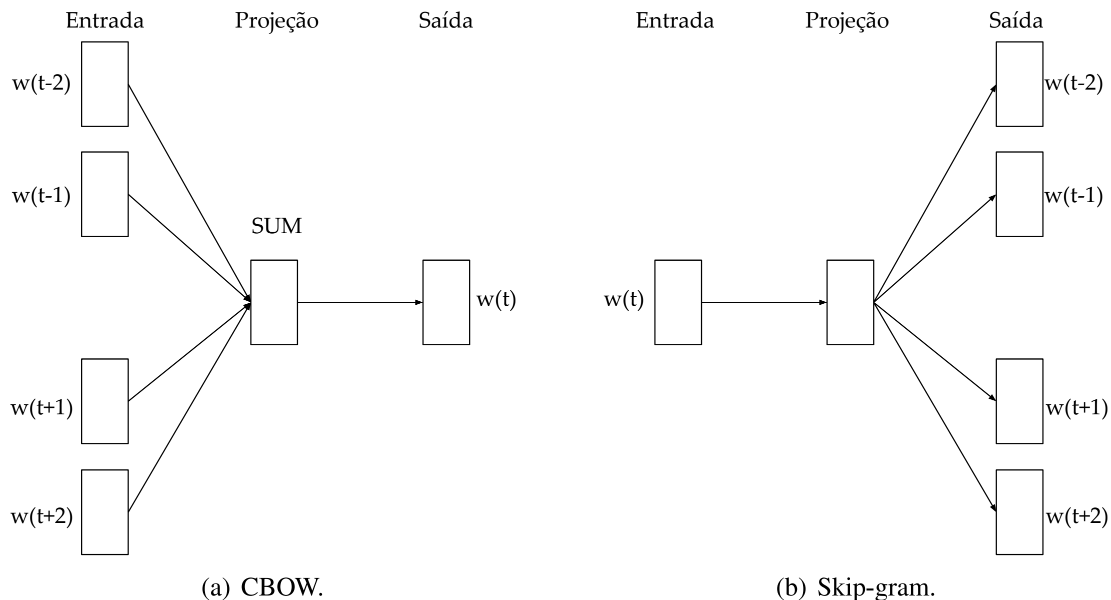
Código 12. Utilizando TF-IDF para extrair características de uma lista de bibliotecas do *dataset Brazilian Malware*.

[14] `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`

 © 2019 Sociedade Brasileira de Computação

2.4.1.2. Word2Vec

Desenvolvido em 2013 por pesquisadores do *Google* [Mikolov et al. 2013a], o *word2vec* é uma forma de criar vetores que representam palavras. A maior inovação desta proposta é a capacidade de representar a semelhança entre palavras e seus significados, diferente do *TF-IDF*, que apenas projeta os textos baseando-se na frequência das mesmas. Além disso, o *word2vec* projeta somente as palavras e não o texto, sendo necessário utilizar alguma estratégia para se extrair características dessas palavras projetadas. O modelo geral do *word2vec* utiliza uma rede neural não supervisionada do tipo *feed-forward* (para mais detalhes sobre rede neural, checar a Seção 2.5.1.4). Existem duas arquiteturas disponíveis neste modelo: *continuous bag-of-words (CBOW)* e *skip-gram*. A primeira (*CBOW*) tenta predizer a palavra central de um conjunto de palavras ao seu redor (entrada), semelhante a um *cluster*. Em contrapartida, a arquitetura *skip-gram* faz o inverso: tenta predizer as palavras ao redor, dado uma palavra central como entrada. A Figura 2.1 apresenta ambos os modelos [Shulman 2016, Mikolov et al. 2013b]. Como apenas a arquitetura *CBOW* foi utilizada neste trabalho, ela será explicada em detalhes a seguir.



(a) CBOW. (b) Skip-gram.

Figura 2.1. Arquiteturas do *word2vec*.

Primeiramente, cada entrada da rede (palavras de um texto) é codificada em um vetor $x_{ck} = \{x_1, x_2, ..., x_V\}$, em que V é o tamanho do vocabulário (conjunto de todas as palavras que aparecem nos textos), c é o identificador da palavra e k, o identificador do texto (como apresentado na Figura 2.2). O item x_i é 1 somente se i corresponde ao identificador da palavra c. Caso contrário, x_i é 0, isto é, apenas uma posição do vetor de cada palavra terá o valor 1 (apenas a que corresponde à palavra). Tal codificação é chamada de *one-hot encoding*. A partir desta entrada, o modelo calcula a média desses vetores e a multiplica pelo vetor de pesos da camada escondida h, gerando a saída y. Detalhes sobre o treinamento da rede podem ser encontrados em [Rong 2014].

© 2019 Sociedade Brasileira de Computação

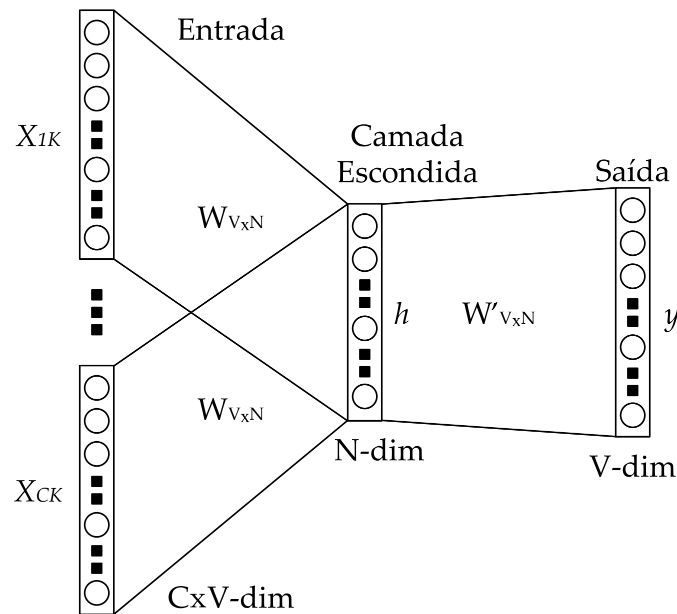


Figura 2.2. Arquitetura *CBOW* do *word2vec*.

O Código 13 apresenta uma forma de se extrair características utilizando o *Word2Vec*. Para isso, implementamos uma classe chamada de `MeanEmbeddingVectorizer`, que utiliza o modelo `Word2Vec`[15] implementado pela biblioteca gensim [Řehůřek and Sojka 2010]. Nesta classe, definimos apenas dois parâmetros (mais parâmetros podem ser adicionados ao modificar essa classe): `size`, que define a dimensão utilizada para projetar as palavras e `min_count`, que define o número mínimo de aparições para que uma palavra seja considerada válida. No método `fit`, treinamos o modelo `Word2Vec` com os dados de treino e obtemos a projeção desses dados. Já no método `transform`, os dados passados como parâmetro utilizam a projeção criada para criar um vetor de características para cada exemplar, com base na média de suas palavras nessa projeção (no caso, caso a palavra não exista, a mesma é considerada como sendo um vetor de zeros). Logo, a classe que implementamos funciona de forma similar a classe implementada para extrair características utilizando o TF-IDF (do *Scikit-Learn*).

2.4.2. Normalização

Um passo importante na extração de características é a normalização já que poucos algoritmos de aprendizado de máquina conseguem ter um bom desempenho quando as características possuem escalas diferentes [Gron 2017]. Alguns algoritmos podem dar mais relevância a valores altos, mesmo que eles representem coisas totalmente diferentes. Por exemplo, se duas características fossem altura, em metros, e peso, em quilos, para classificar o quão saudável uma pessoa é, a diferença de escala entre os dois daria uma significância maior para o peso, já que ele varia em escala muito mais que a altura de uma pessoa. Geralmente utilizam-se dois tipos de normalização [Gron 2017]: *min-max scaling* e *standardization*.

[15] `https://radimrehurek.com/gensim/models/word2vec.html`

 © 2019 Sociedade Brasileira de Computação

```
1  from gensim.models import Word2Vec
2  # classe responsavel por treinar modelo w2v e extrair
     ↪ caracteristicas
3  class MeanEmbeddingVectorizer(object):
4      # salva parametros
5      def __init__(self, size, min_count=1): #word2vec):
6          self.size = size
7          self.min_count = 1
8      # treina w2v
9      def fit(self, X):
10         # treina rede neural w2v
11         w2v =
             ↪ Word2Vec(X,size=self.size,min_count=self.min_count)
12         # obtem projecao das palavras
13         self.word2vec =
             ↪ dict(zip(w2v.wv.index2word,w2v.wv.vectors))
14         # obtem dimensao dos dados
15         self.dim = len(list(word2vec.values())[0])
16         return self
17     # transforma dados (extrai caracteristicas)
18     def transform(self, X):
19         # calcula media das palavras de X
20         return np.array([
21             np.mean([self.word2vec[w] for w in words if
                 ↪ w in self.word2vec]
22                     or [np.zeros(self.dim)], axis=0)
23             for words in X
24         ])
25 ... # leitura e divisao treino e teste
26 # inicializa w2v com maximo de 200 features
27 m = MeanEmbeddingVectorizer(size=200)
28 # treina w2v
29 m.fit(train_texts)
30 # transforma textos para caracteristicas
31 train_features = m.transform(train_texts)
32 test_features = m.transform(test_texts)
33 # imprime forma das caracteristicas
34 print(train_features.shape, test_features.shape)
```

```
1  (25091, 200) (25090, 200)
```

Código 13. Utilizando *Word2Vec* para extrair características de uma lista de bibliotecas (mesma do Código 12) do *dataset Brazilian Malware*.

2.4.2.1. *Min-Max Scaling*

O *MinMax* reescala todas as características em um intervalo entre zero e um, utilizando a Equação 3, em que x_i é o valor da característica x de um exemplar i, $min(x)$ é o menor valor dessa característica no conjunto de treino e $max(x)$, o maior valor dessa característica no mesmo conjunto [Gron 2017].

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{3}$$

A biblioteca *scikit-learn* [Pedregosa et al. 2011] possui uma classe chamada `MinMaxScaler`[16] que efetua a normalização, como demonstrado no Código 14. Primeiramente, inicializa-se o *min-max*, especificando o intervalo em que as características devem ser normalizadas (o que é uma vantagem em relação ao *Standardization*, já que alguns algoritmos funcionam melhor com determinados intervalos, como uma rede neural – intervalo de 0 a 1 – e um SVM com *kernel RBF* – intervalo de -1 a 1). Então, o mesmo utiliza o conjunto de treino para extrair os menores e maiores valores para cada característica (através do método `fit`) para então poder normalizá-las (através do método `transform`).

```python
from sklearn.preprocessing import MinMaxScaler
... # extracao de caracteristicas
# inicializa minmax
scaler = MinMaxScaler(feature_range=(0, 1))
# treina minmax
scaler.fit(train_features.toarray())
# normaliza caracteristicas
train_features_norm =
  scaler.transform(train_features.toarray())
test_features_norm =
  scaler.transform(test_features.toarray())
# imprime forma das caracteristicas
print(train_features_norm.shape,
  test_features_norm.shape)
```

```
(25091, 200) (25090, 200)
```

Código 14. Utilizando *MinMax* para normalizar características extraídas.

2.4.2.2. *Standardization*

O método *standardization* normaliza os dados subtraindo de cada característica seu valor médio (com base nos dados de treino) e dividindo esse valor pela variância, de modo

[16] `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`

© 2019 Sociedade Brasileira de Computação

que a distribuição resultante tenha variância unitária, como demonstrado na Equação 4, em que x_i é o valor da característica x de um exemplar i, u é a média dos exemplares de treino dessa mesma característica e s, a variância dos exemplare de treino, também dessa característica. Uma vantagem desse método com relação ao *min-max* é que ele não é tão afetado por *outliers* como o *min-max*, já que se houver algum *outlier* nos dados de treino do *min-max*, ele levará em consideração e deixará um intervalo grande entre os demais dados "normais".

$$z_i = \frac{x_i - u}{s} \tag{4}$$

De forma similar ao *min-max*, o *Standardization* também já possui implementação no *scikit-learn* [Pedregosa et al. 2011], implementado na classe `StandardScaler`[17], como demonstrado no Código 15. Basta inicializá-lo, treinar com os dados de treino para obter as médias e variância de cada caraterística (através do método `fit`) e, finalmente, normalizar todos os dados (através do método `transform`).

```
1  from sklearn.preprocessing import StandardScaler
2  ... # extracao de caracteristicas
3  # inicializa standardization
4  scaler = StandardScaler()
5  # treina standardization
6  scaler.fit(train_features.toarray())
7  # normaliza caracteristicas
8  train_features_norm =
     ↪ scaler.transform(train_features.toarray())
9  test_features_norm =
     ↪ scaler.transform(test_features.toarray())
10 # imprime forma das caracteristicas
11 print(train_features_norm.shape,
     ↪ test_features_norm.shape)
```

```
1  (25091, 200) (25090, 200)
```

Código 15. Utilizando *Standardization* para normalizar características extraídas.

2.5. Construção de Modelos

Nesta Seção apresentaremos os modelos de aprendizado de máquina mais utilizados nos problemas envolvendo segurança, incluindo suas definições teóricas, algoritmos e códigos para utilizá-los. Dentro os modelos, incluem-se classificadores, detectores e agrupadores, além de um breve comentário sobre as bibliotecas utilizadas para implementá-los.

[17] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

2.5.1. Classificadores

Os classificadores têm como objetivo classificar determinado exemplar de entrada em uma classe previamente conhecida por ele durante o treinamento. O treinamento é a fase em que o classificador aprende os padrões de cada classe com os dados fornecidos (juntamente com seus rótulos), regulando seus parâmetros para o problema em questão. Este tipo de problema é chamado de aprendizado supervisionado [Bishop 2006]. Após o treino, o modelo pode ser utilizado para classificar dados desconhecidos, podendo então ser utilizado. Os classificadores apresentados nessa seção incluem: *K-Nearest Neighbors*, baseado em vizinhança, *Random Forest*, um *ensemble* baseado em árvores de decisão, *Support Vector Machine*, baseado na construção de um hiperplano ótimo, e *Multi-Layer Perceptron*, um tipo de rede neural muito utilizada em *deep learning*.

2.5.1.1. *K-Nearest Neighbors* (KNN)

K-Nearest Neighbors (KNN) é um modelo de aprendizado de máquina baseado em distância, cuja classificação de uma nova instância é baseada na distância dos k exemplares de treino mais próximos. Assim, um novo exemplo desconhecido será classificado como sendo da classe que mais ocorre dentre esses k exemplos, como mostrado na Figura 2.3 [Michie et al. 1994], em que uma nova instância será classificada como vermelha, quando $k = 3$, verde, quando $k = 5$ e desconhecida quando $k = 6$ (o resultado é um empate, por isso números pares não são recomendados para problemas binários).

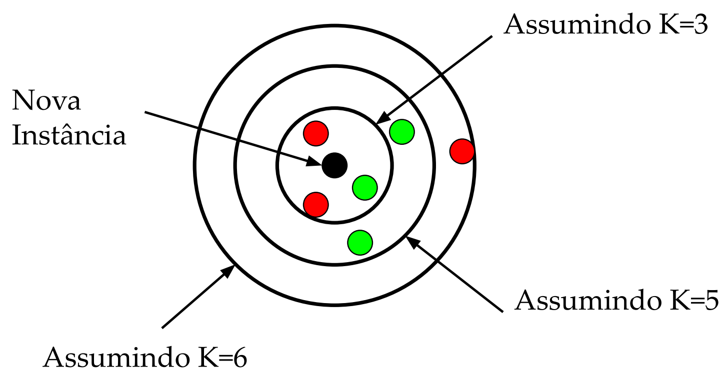


Figura 2.3. Exemplo de funcionamento do KNN.

A distância geralmente utilizada pelo KNN é a distância Euclideana. Dada uma instância x, descrita por (a_1(x),a_2(x),...,a_n(x)), em que a_i(x) é a i-ésima característica, a distância entre duas instâncias x_i and x_j é definida pela Equação 5 [Mellish 2017].

$$d(x_i,x_j) = \sqrt{\sum_{r=1}^{n}(a_r(x_i) - a_r(x_j))^2)} \tag{5}$$

© 2019 Sociedade Brasileira de Computação

2.5.1.2. *Random Forest*

Random forest é um classificador que consiste em uma coleção (chamado de *ensemble*) de classificadores baseados em árvores de decisão, cada um treinado usando seleção aleatória de exemplos e características (*boostrap agreegating* ou *bagging*) e tendo como saída um voto para determinada classe, que será utilizada para tomar a decisão final através de uma votação [Breiman 2001]. A Figura 2.4 apresenta um exemplo de *random forest*, contendo *L* árvores usando *bagging*. Para entender melhor esse ensemble, explicaremos como funciona uma árvore de decisão.

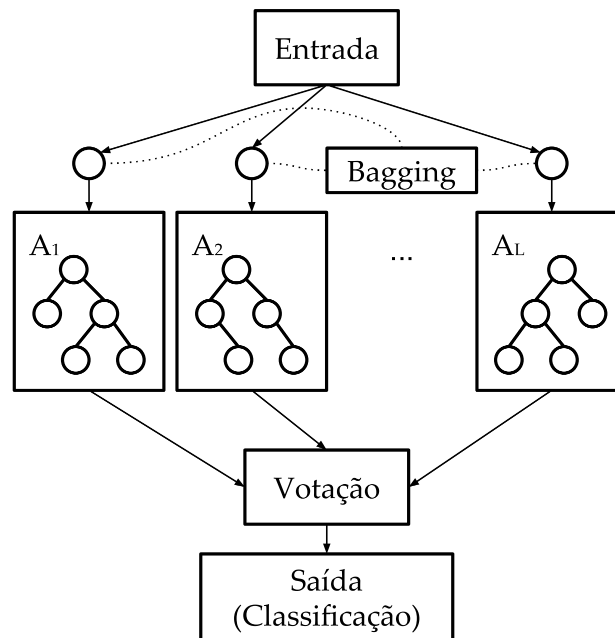


Figura 2.4. Exemplo de *Random Forest*.

Uma árvore de decisão é basicamente um classificador que cria um conjunto de regras *if-then* para classificar novos exemplares. Essas regras melhoram a leitura de resultados por humanos, já que tornam mais fácil de entender e quais características são importantes. A classificação de novas instâncias é feita iniciando-se da raiz da árvore até um nó folha, que provê a classe da instância. Cada nodo interno da árvore representa um teste de alguma características da instância e cada aresta representa possíveis valores que essa característica possui, alterando o caminho de acordo com o mesmo [Mitchell 1997]. A Figura 2.5 mostra um exemplo de árvore de decisão para classificar se um determinado dia é bom para jogar tênis (baseado em condições climáticas).

O treino de uma árvore de decisão é baseado em duas métricas: entropia e ganho de informação. A entropia, representada pela Equação 6, em que *S* é o conjunto de exemplares de treino e *c*, o número de classes, mede a homogeneidade dos exemplos.

$$Entropia(S) = \sum_{i=1}^{c} -p_i \log_2(p_i) \quad (6)$$

O ganho de informação, como mostrado na Equação 7, é usado como métrica

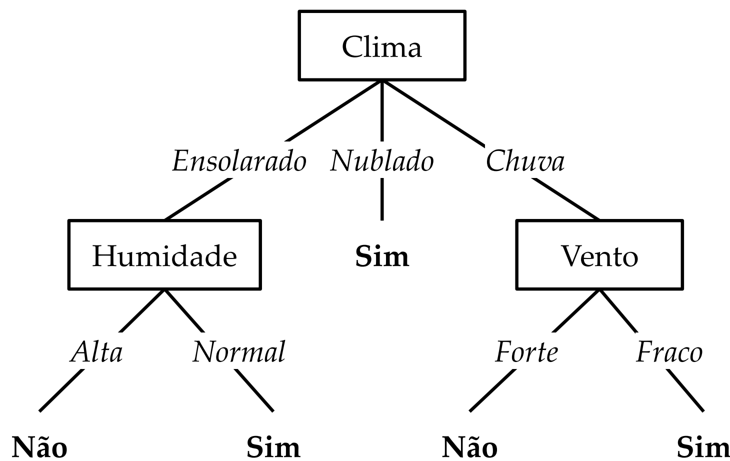


Figura 2.5. Exemplo de árvore de decisão para classificar se um dia é bom para jogar tênis, baseado em condições climáticas.

para seleção de característica. A árvore é construída de acordo com o ganho de informação de uma característica A, já que sempre é utilizado a característica com maior ganho de informação como divisão para criar novas arestas e caminhos para cada valor dessa característica (ou intervalos, no caso de características numéricas) [Mitchell 1997].

$$Ganho(S,A) = Entropia(S) - \sum_{v \in Valores(A)} \frac{|S_v|}{|S|} Entropia(S_v) \quad (7)$$

O algoritmo para treinar uma árvore de decisão (o algoritmo básico, chamado ID3), dado um conjunto de exemplos (exemplos de treino), seus rótulos (classe a ser detectada pela árvore) e as características, é o seguinte [Mitchell 1997]:

1. Criar nodo raiz para a árvore.
2. Se todos os exemplos são positivos, retornar um único nodo raiz, com rótulo $= +$.
3. Se todos os exemplos são negativos, retornar um único nodo raiz, com rótulo $= -$.
4. Se o conjunto de características é vazio, retorna um único nodo raiz, com o rótulo igual ao valor mais presente nos rótulos.
5. Caso contrário, fazer o seguinte:
   (a) Seja A a característica que melhor classifica os exemplos de treino, isto é, aquela com maior ganho de informação, como já definido na Equação 7.
   (b) A característica escolhida para a raiz é A.
   (c) Para cada valor possível v_i de A:
      i. Adicionar um novo caminho abaixo da raiz, correspondente ao teste $A = v_i$.
      ii. Seja $exemplos_{v_i}$ o subconjunto de exemplos que possuem valor v_i para A:

 © 2019 Sociedade Brasileira de Computação

A. Se $exemplos_{v_i}$ é vazio, então, abaixo desse ramo, adicionar um nó folha com o rótulo igual ao rótulo mais presente no conjunto total de exemplos.

B. Caso contrário, abaixo desse ramo, criar uma nova sub-árvore, voltando ao passo 1 usando um subconjunto de exemplos ($exemplos_{v_i}$) e características ($caractersticas - \{A\}$).

2.5.1.3. *Support Vector Machine* (SVM)

O *Support Vector Machine* (SVM), originalmente desenvolvido para classificação binária (classificação de apenas duas classes), procura, em problema linearmente separáveis (isto é, quando uma linha separa as classes), a construção de um hiperplano como superfície de decisão (uma fronteira), em que a separação entre os exemplos é máxima. Quando os padrões não são linearmente separáveis, o SVM pode executar uma função de mapeamento que projeta os dados em um espaço em que os dados se tornam linearmente separáveis. A principal ideia desse classificador é maximizar a margem do hiperplano dos dados de treino. Um hiperplano ótimo é aquele cuja distância da margem para a classe positiva é a mesma distância para a classe negativa. A Figura 2.6 ilustra um hiperplano ótimo definido pelos vetores de suporte, os dados de treino mais próximos a ele [Cortes and Vapnik 1995, Fradkin and Muchnik 2006].

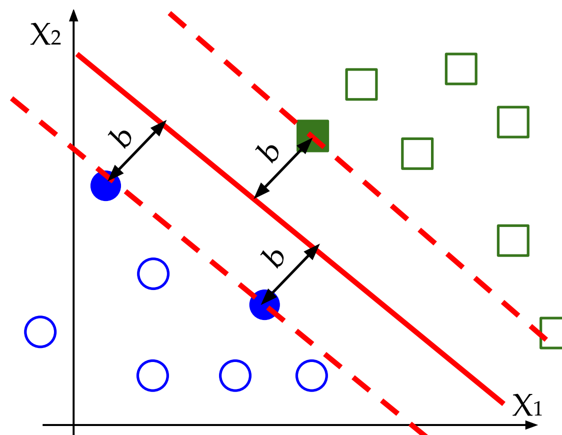


Figura 2.6. Exemplo de hiperplano ótimo de um SVM.

Na maioria das vezes, os problemas não são linearmente separáveis. Por conta disso, é necessário projetar os dados em um espaço em que eles se tornem linearmente separados, chamado de *feature space*. A função de *kernel* é responsável por essa projeção e esse processo é chamado de *kernel trick*. Depois de projetados, é possível encontrar um hiperplano que separa os dados nesse novo espaço. A Figura 2.7 exemplifica o uso do *kernel trick* para projetar os dados em outra dimensão.

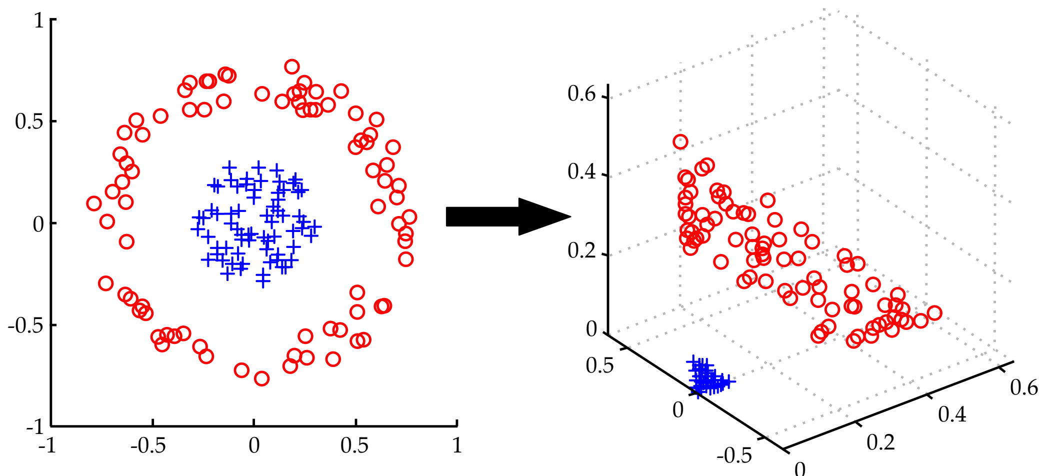


Figura 2.7. Exemplo de projeção de um problema não-linear utilizando *kernel trick*, em que os dados são projetados em outra dimensão (bi-dimensional para tri-dimensional) [Jordan 2017].

A função de decisão responsável por construir o hiperplano é definida pela Equação 8, em que K é a função de *kernel*, α e b são parâmetros encontrados durante o treinamento, x_i são os vetores de características e y_i, seus rótulos.

$$f(x) = \sum_i \alpha_i y_i K(x, x_i) + b \quad (8)$$

Como a maioria dos problemas reais envolvem mais de duas classes e o SVM é um classificador binário, é necessário usar uma abordagem de decisão diferente. A maior comum é a abordagem "um contra todos", também chamada de "um contra o resto" [Manning et al. 2008b]. Nesta abordagem, existem q classificadores (SVMs), em que q é o número de classes. Cada SVM c_i é treinado para a classe i, usando como contra-exemplo os exemplares das outras classes. A decisão final pode ser feita através de uma "contagem de votos" [Milgram et al. 2006].

2.5.1.4. *Multi-Layer Perceptron* (MLP)

Uma rede neural é um modelo computacional de um cérebro humano que pode ser composto de centenas ou milhares de neurônios (unidades de processamento). Em geral, é uma máquina que é desenvolvida para modelar a forma com qual o cérebro realiza uma tarefa em particular. O menor componente de uma rede neural é um *perceptron*, uma unidade de processamento que simula um neurônio [Haykin 2009]. A Figura 2.8 mostra a estrutura de um *perceptron*, em que $x = \{x_1, x_2, \ldots, x_n\}$ são os sinais (entrada), $w = \{w_1, w_2, \ldots, w_n\}$ são os pesos, $\varphi(.)$ é a função de ativação e y é a saída.

A Equação 9 apresenta a saída (y) de um *perceptron*. Os sinais de entrada são multiplicados pelos pesos, que são adaptados a cada iteração durante o treino, usando uma regra de correção de erro (algoritmo de convergência do *perceptron*). Ele também

© 2019 Sociedade Brasileira de Computação

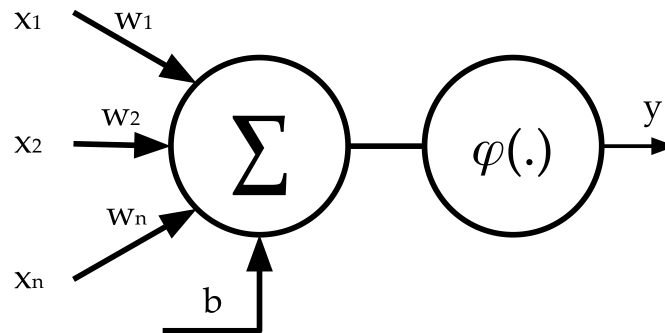


Figura 2.8. Estrutura de um *perceptron*.

inclui um *bias* (b), que tem o efeito de aumentar ou diminuir o sinal da função de ativação. A função de ativação é responsável por determinar a forma e intensidade dos valores transmitidos de um neurônio para outro (ou para a saída), limitando a amplitude da saída [Haykin 2009].

$$y = \varphi(\sum_{i=1}^{n} w_i \times x_i + b) \tag{9}$$

O treino de um único *perceptron* ajuda a entender melhor a operação de uma rede neural. O algoritmo é o seguinte [Haykin 2009]:

1. Inicializar pesos e *bias* com valores aleatórios pequenos.
2. Aplicar padrão de entrada do exemplo corrente e checar a saída da rede (y).
3. Calcular o erro da saída (e), comparando o mesmo (saída y) com o valor esperado (t_j), como mostrado na Equação 10.

$$e = t_j - y \tag{10}$$

4. Se o erro for igual a zero ($e = 0$), significa que a saída está correta. Neste caso, um novo exemplo é apresentado, voltando ao passo 2.
5. Caso contrário, se o erro for diferente de zero ($e \neq 0$), é necessário atualizar os pesos e *bias*, como mostrado nas Equações 11 e 12, respectivamente.

$$w_j = w'_j + e \times x_j \tag{11}$$

$$b = b' + e \tag{12}$$

6. Voltar ao passo 2 e apresentar um novo exemplo a rede. O critério de parada pode ser baseado em número de iterações, erro médio, acurácia, etc.

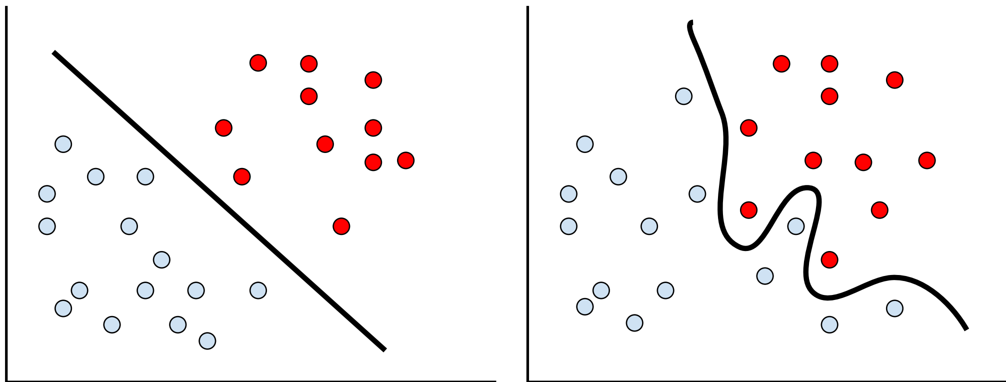


Figura 2.9. Problema linear (esquerda) contra não problema não-linear (direita).

Apesar de ser eficiente, um único *perceptron* pode apenas resolver problemas linearmente separáveis, o que, na maioria das vezes, não acontece no mundo real. Por conta disso, redes neurais geralmente combinam mais que um neurônio, o que torna possível separar problemas não-lineares [Haykin 2009]. A Figura 2.9 mostra dois exemplos de problemas, o primeiro (esquerda), linear e o segundo (direita), não-linear. Como a fronteira de decisão do *perceptron* é definido por uma linha, que resolve problemas como o primeiro caso. Uma rede mais complexa, tal como a *multilayer percetron*, composta por vários neurônios, pode resolver o segundo problema.

Um rede neural do tipo *multilayer perceptron* (MLP) é composta por nós de entrada, uma camada ou mais de *perceptrons*, chamada de camadas escondidas, e os neurônios de saída. Todas as camadas, exceto a entrada, são compostas por neurônios, cada uma delas inicializadas por pesos e *bias* aleatórios, como um *perceptron* qualquer. Esse tipo de rede é progressiva, i.e., os neurônios de uma determinada camada são apenas conectadas com a próxima camada. Assim, a entrada passa por todas as camadas existentes. O número de nós de entrada é a dimensionalidade dos dados de entrada (número de características) e o número de neurônios da saída é geralmente composta pelo número de classes do problema (neste caso, cada neurônio representa uma classe, i.e., o valor de saída do neurônio é diretamente relacionada a sua respectiva classe. Quanto maior o valor, maior as chances de ser daquela classe) [Haykin 2009, Bishop 2006]. A Figura 2.10 apresenta uma rede *multilayer perceptron* com duas camadas escondidas e três neurônios de saída.

Um algoritmo muito importante para redes neurais em geral é o *back propagation*, um método computacional eficiente para o treinamento de múltiplos *perceptrons*. Brevemente, esse algoritmo implementa o grandiente descendente, computando derivadas parciais de uma função aproximada $F(w,x)$ calculada pela rede, ajustando os pesos de acordo com a entrada [Haykin 2009].

2.5.2. Detectores

A detecção de mudança é um tema fundamental para muitos problemas da área de segurança, uma vez que a distribuição dos dados geralmente está em constante mudança para evadir sistemas de segurança. Essas mudanças são chamadas de *Concept drift*, si-

 © 2019 Sociedade Brasileira de Computação

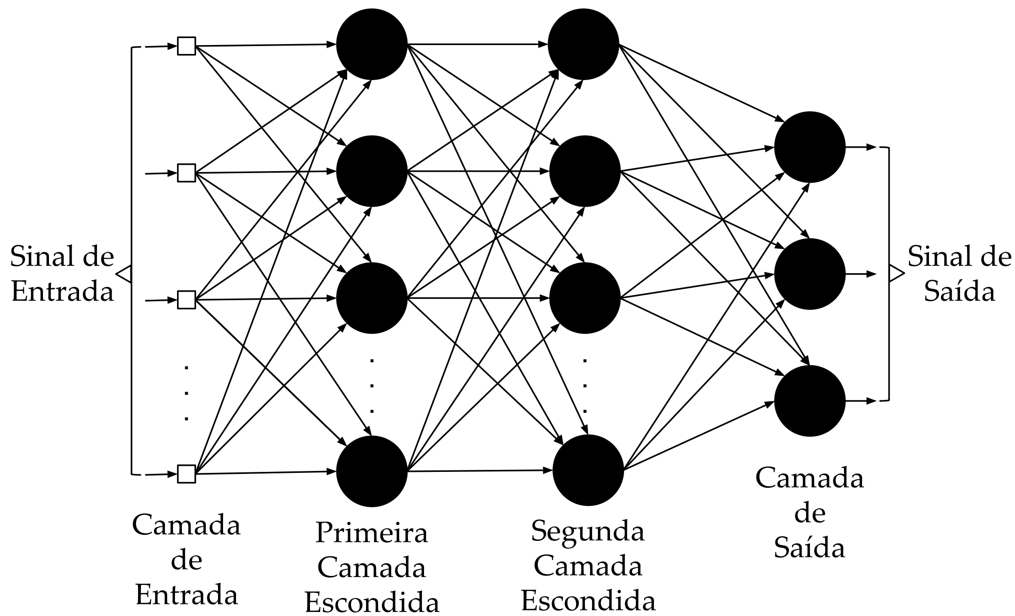
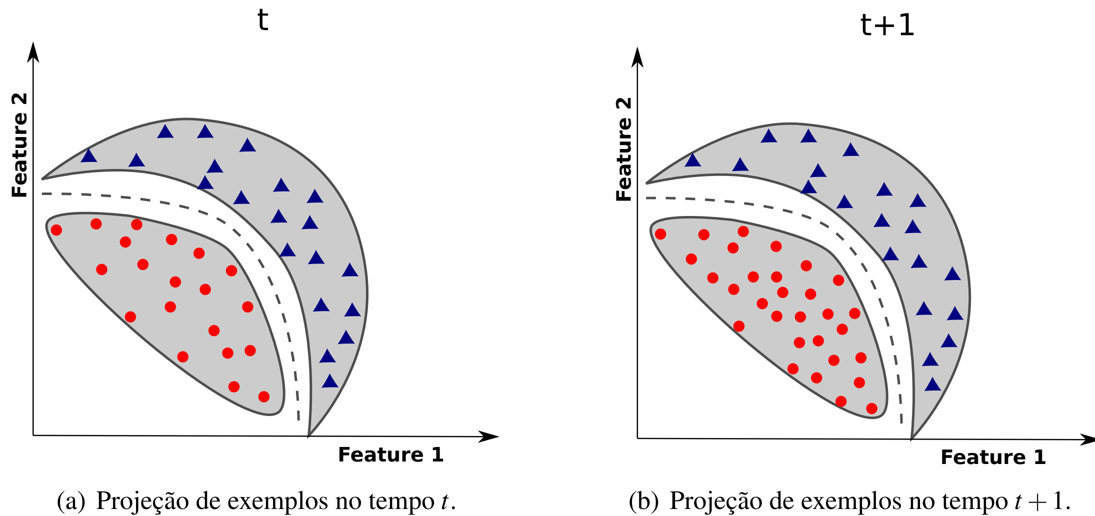


Figura 2.10. Exemplo de rede neural *multilayer perceptron* (MLP).

tuação em que a relação entre os dados de entrada e sua classe mudam com o passar do tempo. Isso geralmente acontece quando há mudanças em um contexto escondido, se tornando complicado de lidar, já que este problema acontece em diferentes campos de pesquisa. Cada exemplo dos dados de entrada é representado por um vetor de características $x = [x_1, x_2, ..., x_L]$, em que L é o número de características que são usadas para determinar sua classe y, de acordo com uma probabilidade *a posteriori* $P(y,x)$. $P(x)$ é definido como distribuição de características e $P(y)$ como probabilidades *a priori*. Na literatura, existem dois tipos de *concept drift*: virtual e real [Wang et al. 2011, Gama et al. 2014].

O *concept drift* virtual acontece quando a distribuição dos dados muda, i.e., $p(x)$ muda, sem alterar $p(y,x)$. A Figura 2.11 mostra um exemplo de *concept drift* virtual com mudanças na distribuição dos dados no tempo t e $t+1$, em que a classe vermelha é mais prevalente mas não leva a mudanças na melhor fronteira de decisão entre as classes [Almeida et al. 2015].

Para ilustrar o problema causado pelo *concept drift* virtual, a Figura 2.12 apresenta a distribuição de uma característica para as classes azul e vermelha (apresentado na Figura 2.11) nos tempos t e $t+1$. A linha vertical pontilhada em ambas as Figuras é o limiar (*threshold*) que separa ambas as classes (o custo de classificação errônea da classe vermelha é maior que da classe azul), que são igualmente distribuídas no tempo t (Figura 2.12(a)), diferente do tempo $t+1$ (Figura 2.12(b)), que possui maior presença da classe vermelha. Apesar do fato da média e desvio padrão de ambas as classes permanecer a mesma, com o *threshold* na mesma posição, i.e., mantendo o classificador sem nenhuma mudança, a probabilidade de encontrar um objeto vermelho como sendo azul aumenta, como mostrado na área vermelha escura na Figura 2.12(b) [Almeida et al. 2015].



(a) Projeção de exemplos no tempo t.

(b) Projeção de exemplos no tempo $t+1$.

Figura 2.11. Exemplo de *concept drift* virtual em que $P_t(y) \neq P_{t+1}(y)$ [Almeida et al. 2015].

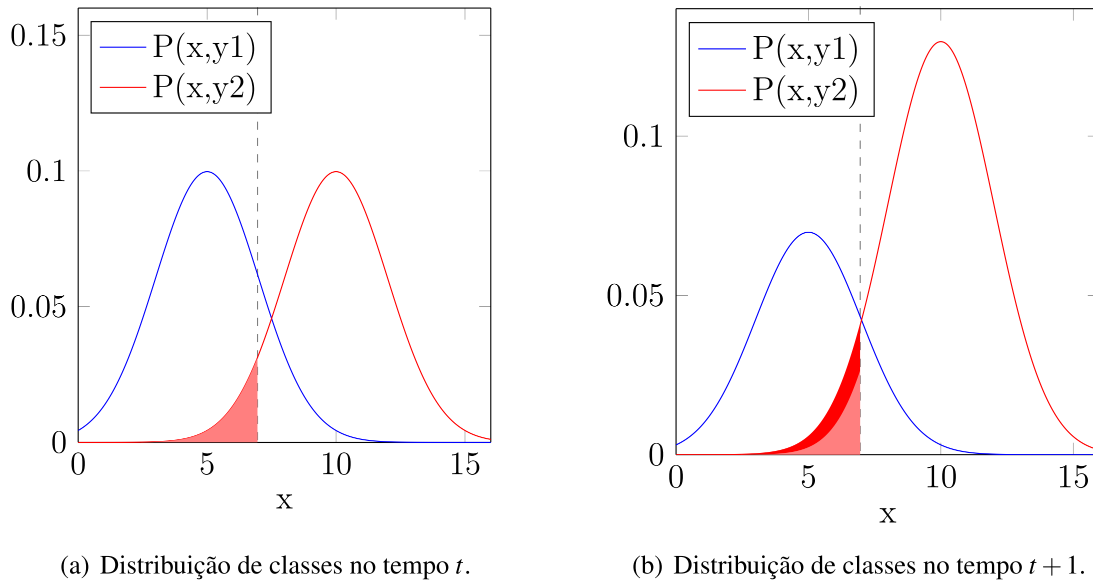
Já o *concept drift* real acontece quando há mudança em $P(y,x)$ com ou sem mudança em $p(x)$, i.e., a relação entre as classes e vetores de características muda com o passar do tempo. Um exemplo clássico é relacionado ao problema de *spam* de e-mails, em que um e-mail representado por um vetor x_e pode ser considerado como spam em determinado tempo t e pode não ser no tempo $t+1$, devido a mudanças de comportamento dos usuários. Como um exemplo de *concept drift* real, a Figura 2.13 mostra um problema de duas classes com um *drift* nas probabilidades *a posteriori*, causando mudanças nas fronteiras do tempo t e $t+1$ (Figuras 2.13(a) e 2.13(b),respectivamente) e forçando uma atualização do classificador [Almeida et al. 2015].

Para ambos os tipos de *concept drift*, existem diversos detectores, cada um deles utiliza uma estratégia diferente para analisar as mudanças ocorridas com o passar do tempo. Neste curso, selecionamos três detectores que são comumente utilizados na literatura: DDM (*Drift Detection Method* [Gama et al. 2004]), EDDM (*Early Drift Detection Method* [Baena-Garćıa et al. 2006]) and ADWIN (*ADaptive WINdowing* [Bifet and Gavaldà 2007]).

2.5.2.1. *Drift Detection Method* (DDM)

A ideia geral do *Drift Detection Method* (DDM) é monitorar a taxa de erro de um modelo, enquanto os exemplos são apresentados ao classificador em sequência, um *data stream* [Gama et al. 2004]. Geralmente, quando a distribuição dos dados é estacionária, o erro do modelo deve cair ou manter-se estável conforme mais dados são utilizados. Quando a taxa de erro aumenta, o DDM utiliza isso como evidência para detectar uma mudança de conceito [Albert Bifet 2018]. Seja p_t a taxa de erro do classificador e $s_t = \sqrt{\frac{p_t(1-p_t)}{t}}$ o desvio padrão, ambos no tempo t, o DDM salva a menor taxa de erro p_{min} e menor desvio padrão s_{min} observados até o tempo t e então realiza as seguintes

 © 2019 Sociedade Brasileira de Computação



(a) Distribuição de classes no tempo t.

(b) Distribuição de classes no tempo $t+1$.

Figura 2.12. Exemplo de *concept drift* virtual em que $P_t(y) \neq P_{t+1}(y)$ [Almeida et al. 2015].

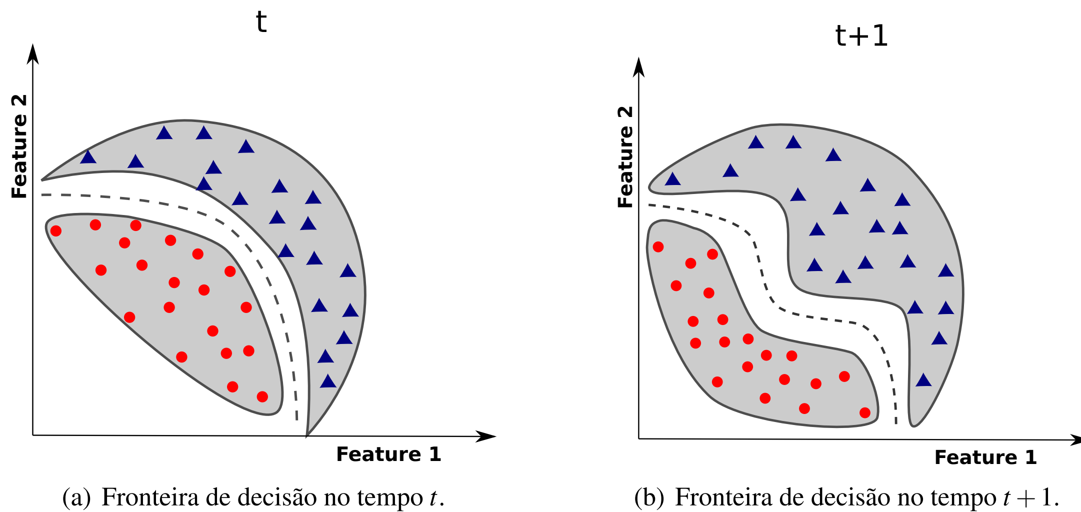
verificações [Albert Bifet 2018]:

- Se $p_t + st \geq p_{min} + 2 * s_{min}$, considera-se que o modelo está em estágio de *warning*, sugerindo que um *drif* está começando. A partir desse ponto, todos os exemplos são armazenados em um *buffer*, já que uma mudança pode estar prestes a acontecer.
- Se $p_t + st \geq p_{min} + 3 * s_{min}$, considera-se que uma mudança aconteceu. O modelo utilizado é descartado e um novo é construído com os exemplos armazenados no *buffer* desde que o *warning* ocorreu. Os valores de p_{min} e s_{min} são resetados.

Embora seja genérico e simples de se utilizar, o DDM pode ser muito lento para responder a mudanças em alguns casos, já que muitos exemplos podem ser observados até que o nível de *drift* seja efetivamente ativado, podendo até armazenar muitos exemplos em memória, deixando o processo custoso [Albert Bifet 2018].

2.5.2.2. *Early Drift Detection Method* (EDDM)

O *Early Drift Detection Method* (EDDM) funciona de forma similar ao DDM. Entretanto, em vez de considerar a taxa de erros, o EDDM considera a distância entre dois erros. Segundo os autores do método, conforme um modelo vai sendo treinado (com novos exemplares), ele melhora suas predições e a distância entre dois erros aumenta. Seja p_i a distância média entre dois erros e s_i seu desvio padrão, o EDDM armazena ambos os valores quando $p_i + 2 * s_i$ atinge seu maior valor (obtendo p_{max} e s_{max}), ou seja, $p_{max} + 2 * s_{max}$ é o ponto em que a distribuição da distância entre erros é máxima [Baena-Garcıa et al. 2006]. Conforme o modelo é atualizado e testado com novas amostras, o EDDM realiza as seguintes verificações, dados dois parâmetros α e β:



(a) Fronteira de decisão no tempo t.

(b) Fronteira de decisão no tempo $t+1$.

Figura 2.13. Exemplo de *concept drift* real [Almeida et al. 2015].

- Se $\frac{p_i + 2*s_i}{p_{max} + 2*s_{max}} < \alpha$, considera-se que o modelo está em estágio de *warning*, armazenando os exemplos vistos a partir desse ponto em um *buffer* e sugerindo o início de possível mudança.
- Se $\frac{p_i + 2*s_i}{p_{max} + 2*s_{max}} < \beta$, considera-se que uma mudança aconteceu, descartando o modelo utilizado e construindo um novo usando os exemplos armazenados no *buffer* desde que o *warning* ocorreu. Os valores de p_{max} e s_{max} são resetados.

No artigo em que o EDDM foi proposto [Baena-Garcıa et al. 2006], o EDDM foi melhor em todos os experimentos em relação ao DDM, mesmo em *datasets* com grande quantidade de ruídos. Entretanto, o problema relacionado ao uso de memória continua o mesmo do DDM, já que ao entrar em estágio de *warning*, o modelo armazena todos os exemplares até que um *drift* ocorra.

2.5.2.3. *ADaptive WINdowing* (ADWIN)

O detector *ADaptive WINdowing* (ADWIN) age de forma diferente do DDM e EDDM, mantendo estatísticas de janelas deslizantes de tamanhos variáveis, que são utilizadas para computar a média da mudança observada "cortando" essas janelas em diferentes pontos. A propriedade dessas janelas deve ser estatisticamente consistente com a hipótese "não houve mudança no valor médio dentro da janela", i.e., uma parte antiga da janela é jogada fora apenas se existir evidência suficiente de que o valor médio difere do resto da janela [Bifet and Gavaldà 2007]. O ADWIN possui como parâmetro um valor de confiança $\delta \in (0,1)$ e um teste $T(W_0, W_1, \delta)$ que compara a média de duas janelas W_0 e W_1, que decide se ambas pertencem a mesma distribuição, com base nas seguintes verificações [Albert Bifet 2018]:

- Se W_0 e W_1 pertencem a mesma distribuição, então, com probabilidade de pelo menos $1-\delta$, o teste responde que não houve mudança.

 © 2019 Sociedade Brasileira de Computação

- Se W_0 e W_1 são de duas distribuições diferentes cuja média difere por uma quantidade maior que $\varepsilon(W_0, W_1, \delta)$, então, com probabilidade de pelo menos $1 - \delta$, o teste responde que houve mudança.

Diferente do DDM e EDDM, o ADWIN pode ser mais eficiente em relação ao uso de memória, já que não é necessário criar um *buffer* para salvar os exemplares, basta utilizar os tamanhos da janelas "cortadas" ao detectar uma mudança.

2.5.3. Agrupadores

Muitas vezes os dados são disponibilizados sem nenhum rótulo, dificultando o uso de classificadores convencionais. Para esses tipos de problemas, utiliza-se o aprendizado não supervisionado [Bishop 2006], normalmente utilizando algoritmos agrupadores (*clustering*), cujo objetivo é agrupar padrões parecidos em grupos que podem ou não ser conhecidos, dependendo do domínio do problema. Neste curso, apresentaremos dois algoritmos muito utilizados para agrupamento de dados: *K-Means* e *DBScan*.

2.5.3.1. K-Means

O algoritmo *K-Means* consiste basicamente em agrupar um conjunto de dados em K grupos, que são definidos com base na média de seus dados – um determinado exemplo pertence ao grupo que está mais próximo a ele [Michie et al. 1994]. O número de grupos K é definido pelo usuário do algoritmo e corresponde ao número de classes gerados após a execução do mesmo. O algoritmo do *K-Means* funciona da seguinte forma, dado um número de grupos K [Rogers and Girolami 2011]:

1. Escolher um número K de grupos. Para cada grupo, um valor aleatório é escolhido no início, considerado o centroide de cada um deles.
2. Para cada exemplar do *dataset*, atribui-se um dado exemplar ao grupo que está mais próximo a ele. .
3. Após rotular todos os dados, um novo centroide é calculado para cada grupo (através da média dos exemplares de cada um).
4. Se algum critério de parada não for atingido (número de iterações, variação média de cada centroide, etc), voltar ao passo 2 até que o mesmo seja atingido.
5. Ao final do algoritmo, serão criados K grupos diferentes para o conjunto de dados.

2.5.3.2. *Density-Based Spatial Clustering of Applications with Noise* (*DBScan*)

Density-based spatial clustering of applications with noise (*DBSCAN*) é um algoritmo que agrupa pontos que estão próximos um dos outros baseado em uma medida de distância (geralmente distância Euclidiana) e um número mínimo de pontos, marcando como *outliers* pontos que estão em regiões de baixa densidade. O *DBSCAN* é composto basicamente por dois parâmetros: *eps* (ε), que especifica o quão perto os pontos devem estar

um dos outros para ser considerado parte de um grupo, isto é, se a distância entre dois pontos for menor ou igual a esse valor, esses pontos são considerados vizinhos (do mesmo grupo); e *minPoints*, o número mínimo de pontos para formar uma região densa, isto é, são necessários pelo menos *minPoints* para que uma região seja considerada densa. Em geral, valores menores de *eps* (ε) são preferíveis para vários *datasets*,e quanto maior o *dataset*, maior o valor de *minPoints* a ser escolhido [Ester et al. 1996].

O *DBSCAN* funciona basicamente da seguinte forma [Schubert et al. 2017]:

1. Encontrar os pontos que estão a uma distância *eps* (ε) de cada ponto e encontrar pontos centrais com mais de *minPoints* vizinhos.

2. Encontrar todos os componentes conectados aos pontos centrais no grafo de vizinhança, ignorando todos os pontos centrais.

3. Atribuir cada ponto não-central ao *cluster* mais próximo se o mesmo está a uma distância *eps* (ε), caso contrário, considera-se que o ponto é um ruído.

Uma das vantagens do *DBSCAN* em relação ao *K-Means* é que o mesmo não precisa definir um número de *clusters* (grupos) a *priori* e o mesmo pode encontrar agrupamentos de diversas formas diferentes [Ester et al. 1996].

2.5.4. Bibliotecas

Neste curso apresentaremos três bibliotecas em que os algoritmos apresentados podem ser utilizados: *Scikit-Learn* [Pedregosa et al. 2011], *Scikit-Multiflow* [Montiel et al. 2018] e *Keras* [Chollet et al. 2015]. Todas elas são *open source*, possuem uma comunidade ativa e são complementares na grande maioria das suas funções. Nesta Subseção, focaremos em como utilizar os classificadores de cada biblioteca mencionada, deixando de lado apenas a avaliação, que será discutida depois, na Seção 2.6. Considera-se que os dados utilizados em cada código apresentado já tiveram suas características extraídas e normalizadas, além de estarem separados em dois conjuntos, treino (`train_features_norm`) e teste (`test_features_norm`), como demonstrado nos Códigos 14 e 15.

2.5.4.1. *Scikit-Learn*

A *Scikit-Learn*[18] [Pedregosa et al. 2011] é uma das bibliotecas mais completas de aprendizado de máquina, com diversos algoritmos já implementados, tanto de classificação, pré-processamento, extração de características e normalização. Por conta disso, essa biblioteca já foi utilizada no curso na parte de extração de características (Seção 2.4), pois implementa o TF-IDF e os dois métodos de normalização apresentados (*Min-Max* e *Standardization*). De todos os algoritmos apresentados, essa biblioteca implementa os

[18] `https://scikit-learn.org/`

© 2019 Sociedade Brasileira de Computação

classificadores *KNN*[19], *Random Forest*[20] e *SVM*[21], além dos agrupadores *K-Means*[22] e *DBScan*[23]. Vale ressaltar que a maioria dos algoritmos implementados no *Scikit-Learn* são feitos para dados em *batches*, em que são usados apenas lotes coletados durante um tempo determinado (esse tempo geralmente é ignorado, permitindo um "acesso aleatório" dos dados e geralmente assumindo que os dados possuem distribuição estacionária, isto é, que não mudam conforme o tempo passa), sem se preocupar com novos dados a serem coletados (neste caso, caso haja novos lotes de dados, seria necessário treinar um novo modelo do zero para que o mesmo possa aprender com os mesmos).

O uso dos classificadores citados funciona da mesma forma nesta biblioteca, conforme mostrado no Código 16, no qual utilizamos o *Random Forest* e definimos o número de árvores utilizadas como sendo 100 (parâmetro `n_estimators`). Já o Código 17 utiliza o *K-Means* para agrupar os exemplares em dois grupos (parâmetro `n_clusters`), sem se preocupar com os rótulos, já que se trata de aprendizado não-supervisionado. O método `fit_predict` treina o agrupar e já retorna as predições para o conjunto passado como parâmetro (funciona da mesma forma para o *DBScan*). Cada classificador e agrupador citado possui os seus parâmetros (a maioria já explicada nesta mesma Seção), todos eles documentados nas páginas de documentação da biblioteca, já mencionadas.

```
1  from sklearn.ensemble import RandomForestClassifier
2  ... # extracao de caracteristicas e normalizacao
3  # inicializa classificador com seus parametros
4  clf = RandomForestClassifier(n_estimators=100)
5  # treina classificador
6  clf.fit(train_features_norm, train_label)
7  # prediz as classes de teste
8  test_pred = clf.predict(test_features_norm)
9  # imprime forma da predicao e do vetor de rotulos
10 print(test_pred.shape, test_label.shape)
```

```
1  (25090,) (25090,)
```

Código 16. Utilizando o classificador *Random Forest* para classificar um conjunto de testes criado a partir do *dataset Brazilian Malware*.

[19]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

[20]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[21]https://scikit-learn.org/stable/modules/svm.html

[22]https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

[23]https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

© 2019 Sociedade Brasileira de Computação

```
1 from sklearn.cluster import KMeans
2 ... # extracao de caracteristicas e normalizacao
3 # inicializa kmeans com seus parametros
4 clustering = KMeans(n_clusters=2)
5 # treina kmeans (note que os rotulos nao sao utilizados)
6 # e prediz suas classes usando os grupos criados
7 train_pred = clustering.fit_predict(train_features_norm)
8 # imprime forma da predicao e do vetor de rotulos
9 print(train_pred.shape, train_label.shape)
```

```
1 (25091,) (25091,)
```

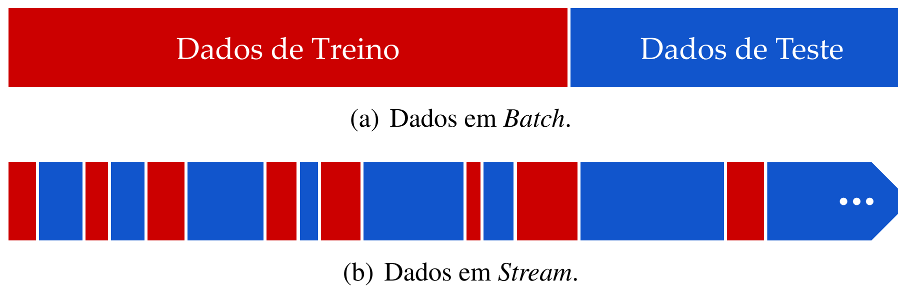
Código 17. Utilizando o agrupador *K-Means* para agrupar conjunto de dados criado a partir do *dataset Brazilian Malware*.

2.5.4.2. *Scikit-Multiflow*

O *Scikit-Multiflow*[24] é uma biblioteca de aprendizado de máquina que segue a "filosofia *Scikits*" focada em problemas para *data streams*. Diferente do formato em *batches*, *data streams* são dados acessados de forma contínua (e sequencial), geralmente em tempo real (o que permite com que sejam utilizado mais facilmente no mundo real), isto é, a qualquer momento um novo exemplar pode chegar, influenciando diretamente o modelo, uma vez que a distribuição normalmente é não-estacionária e possui mudanças ao longo do tempo [Albert Bifet 2018]. A Figura 2.14 compara os dados em *batch* e em *stream*: ao utilizar *batch*, as fases de treino e teste são bem definidas e o conjunto de dados é finito, diferente de uma *stream*, em que os dados podem fazer parte do conjunto de treino e teste ao longo do tempo (dependendo de mudanças de conceito e evolução dos dados), que é, teoricamente, infinito, uma vez que novos dados podem chegar a qualquer momento. Dos classificadores apresentados nesta Seção, a biblioteca *scikit-multiflow* implementa algumas adaptações (com algumas mudanças para problemas envolvendo *data streams*) do *KNN*[25] (esta versão já inclui um detector de mudanças ADWIN embutido no classificador) e *Random Forest*[26] (por padrão, já possui o detector de mudanças ADWIN embutido – é possível alterá-lo e até mesmo desativá-lo). Dos detectores de mudança citados, todos eles são implementados: DDM[27], EDDM[28] e ADWIN[29]. Até o momento deste trabalho, nenhum agrupador citado foi implementado pela biblioteca.

[24] https://scikit-multiflow.github.io/
[25] https://scikit-multiflow.github.io/scikit-multiflow/_autosummary/skmultiflow.lazy.KNNAdwin.html
[26] https://scikit-multiflow.github.io/scikit-multiflow/_autosummary/skmultiflow.meta.AdaptiveRandomForest.html
[27] https://scikit-multiflow.github.io/scikit-multiflow/_autosummary/skmultiflow.drift_detection.DDM.html
[28] https://scikit-multiflow.github.io/scikit-multiflow/_autosummary/skmultiflow.drift_detection.EDDM.html
[29] https://scikit-multiflow.github.io/scikit-multiflow/_autosummary/skmultiflow.drift_detection.ADWIN.html

 © 2019 Sociedade Brasileira de Computação



(a) Dados em *Batch*.

(b) Dados em *Stream*.

Figura 2.14. Comparação: *Batch* VS *Stream*.

O Código 18 apresenta um exemplo de como utilizar um modelo da biblioteca, no caso o `AdaptiveRandomForest`, inicializado com 10 árvores e detector de *drift* ADWIN. A principal diferença para o *Scikit-Learn* é que o método para treino se chama `partial_fit`, que implementa um "treino parcial" e permite que o modelo seja atualizado com novos dados. Além disso, utilizamos a classe `DataStream` para criar uma *stream* com os dados de teste. Então, para cada dado da *stream*, o modelo prediz sua classe e atualiza o modelo com o novo dado (neste mesmo momento, como o classificador possui um detector de *drift* embutido, já é avaliado se houve mudança de conceito nos dados para atualizar o modelo adequadamente – descartando dados antigos se for preciso e mantendo os mais recentes, desde o último *warning*, como já explicado). Nesta implementação, usamos um vetor para salvar todas as predições feitas pelo modelo ao longo da *stream*. Todos os demais classificadores e detectores de *drift* dessa implementação estão documentados nas páginas de documentação da biblioteca, também já mencionadas.

Para utilizar um detector de *drift* em qualquer classificador, pode-se seguir os exemplos dos Códigos 19 e 20. O primeiro deles (Código 19), inicializa um classificador, no caso, uma árvore de decisão *HoeffdingTree*, a mesma utilizada pelo *Random Forest*, e o treinamos com os dados de treino. Um segundo classificador, com a mesma árvore, é criado para ser treinada apenas quando o primeiro modelo entrar em estágio de *warning*. Em seguida, o detector de drift (no caso, o DDM) é inicializado, bem como a *stream* contendo os dados de teste. No Código seguinte (Código 20), cria-se um vetor para armazenar as predições, um contador para contar o número de dados já processados na *stream* e um vetor para armazenar os pontos em que houveram *drift*. Ao iterar sobre a *stream*, obtém-se os dados do próximo exemplar, incrementa-se o contador de exemplares e faz-se a predição do exemplar obtido no classificador corrente. Essa predição é então comparada ao valor real do exemplar pelo detector de *drift*, que checa se está em fase de *warning* ou se realmente houve um *drift*. Se um *warning* é detectado, o classificador dois é habilitado, apresentando o exemplar corrente para ele. Caso um *drift* seja detectado, o ponto de *drift* é salvo e o classificador secundário assume o lugar do primeiro, criando um novo classificador para a próxima fase de *warning* e resetando o detector. De qualquer forma, o classificador principal é atualizado com o novo exemplar. Ao final, os pontos de *drift* são impressos. A parte interessante dessa implementação é que, independente do problema a ser resolvido, a solução não fica presa ao classificador utilizado, já que o detector não é embutido no mesmo (e mesmo no caso do `AdaptiveRandomForest`, é possível desabilitar o detector interno dele), permitindo com que você desenvolva sua própria lógica ao detectar uma mudança de conceito.

2.5.4.3. *Keras*

A biblioteca *Keras* é uma API de redes neurais de alto nível implementada em Python, que permite que seja capaz utilizar bibliotecas de mais baixo nível, tais como *TensorFlow* [Abadi et al. 2015], *CNTK* [Seide and Agarwal 2016] e *Theano*, agilizando o processo de desenvolvimento de pesquisas utilizando redes neurais [Theano Development Team 2016] [Chollet et al. 2015][30]. Por se tratar de uma biblioteca específica de redes neurais, dos modelos citados ela implementa apenas o *Multi-Layer Perceptron*, sendo possível uma vasta variedade de estruturas de rede neurais diferentes.

[30] https://keras.io/

```
1  from skmultiflow.meta import AdaptiveRandomForest
2  from skmultiflow.data import DataStream
3  # inicializa classificador
4  clf = AdaptiveRandomForest(n_estimators=10)
5  # treina classificador
6  clf.partial_fit(train_features_norm, train_label)
7  # cria stream com dados de teste
8  s = DataStream(test_features_norm, test_label)
9  # prepara stream para uso
10 s.prepare_for_use()
11 # cria vetor final de predicao
12 test_pred = []
13 # itera sobre stream
14 while s.has_more_samples():
15     # obtem caracteristicas e rotulo do proximo exemplar
16     sample_features, sample_label = s.next_sample(1)
17     # prediz exemplar
18     sample_pred = clf.predict(sample_features)
19     # adiciona predicao ao vetor de predicao
20     for l in sample_pred:
21         test_pred.append(l)
22     # atualiza modelo com novo exemplar
23     clf.partial_fit(sample_features, sample_label)
24 # transforma vetor de predicao em vetor numpy
25 test_pred = np.array(test_pred)
26 # imprime forma da predicao e do vetor de rotulos
27 print(test_pred.shape, test_label.shape)
```

```
1  (25090,) (25090,)
```

Código 18. Utilizando o classificador *Adaptive Random Forest* (com detector de *drift* ADWIN) para classificar um conjunto de testes criado a partir do *dataset Brazilian Malware*.

 © 2019 Sociedade Brasileira de Computação

```python
from skmultiflow.trees import HoeffdingTree
from skmultiflow.drift_detection import DDM
from skmultiflow.data import DataStream
# inicializa classificador 1
clf1 = HoeffdingTree()
# treina classificador 1
clf1.partial_fit(train_features_norm, train_label)
# inicializa classificador 2
clf2 = HoeffdingTree()
# inicializa detector de drift
drift = DDM()
# cria stream com dados de teste
s = DataStream(test_features_norm, test_label)
# prepara stream para uso
s.prepare_for_use()
```

Código 19. Inicializando dois classificadores (árvore de decisão *Hoeffding-Tree*), um detector de *drift* e a *stream* para classificar um conjunto de testes criado a partir do *dataset Brazilian Malware*.

O Código 21 apresenta um exemplo de como criar uma rede neural *Multi-Layer Perceptron* com duas camadas escondidas. Primeiramente, é necessário transformar o vetor de rótulos no formato *one-hot encoding*, similar ao utilizado pelo *Word2Vec*. Então, inicializá-se o modelo do tipo `Sequential` e, logo em seguida, podemos adicionar camadas a este modelo. A primeira camada nesse exemplo contém 200 neurônios e é necessário especificar o tamanho da entrada (o número de características), assim como a função de ativação (estamos utilizando a função *relu*). Em seguida, adicionamos uma segunda camada, com 100 neurônios e a mesma função de ativação. Finalmente, adicionamos a camada de saída, composta por dois neurônios (um para cada classe do problema) e a função *softmax*, para transformar seus valores em probabilidades. O modelo então é compilado, especificando a função de minimização (no caso *binary_crossentropy*), otimizador (*rmsprop*) e métricas a serem observadas durante o treino, neste caso, a acurácia. Depois de compilado, o modelo pode ser treinado usando os dados de treino e os rótulos de treino transformados em *one-hot encoding*. No exemplo, especificamos um conjunto de validação de 33%, isto é, 33% dos dados de treino serão utilizados para validar o modelo enquanto ele é treinado em 10 épocas e em *batches* de 128 exemplos (o modelo é atualizado durante o treino a cada 128 exemplares). O método `predict_classes` prediz as classes do conjunto de testes. A documentação da biblioteca, assim como as demais, apresenta mais exemplos de redes, bem como todos os seus parâmetros e detalhes.

2.6. Avaliação de Modelos

Tão importante quanto criar uma boa representação para o problema e criar um bom modelo, saber avaliar se o mesmo resolve o problema em questão é fundamental, sobretudo na área de segurança. É comum encontrar na literatura trabalhos que tentam apenas obter

```python
from skmultiflow.core import clone
# cria vetor final de predicao
test_pred = []
# contador de dados
count = 0
# armazena pontos de drift
drifts = []
# itera sobre stream
while s.has_more_samples():
    # obtem caracteristicas e rotulo do proximo exemplar
    sample_features, sample_label = s.next_sample(1)
    # incrementa contador
    count += 1
    # prediz exemplar
    sample_pred = clf1.predict(sample_features)
    # adiciona predicao ao vetor de predicao
    for l in sample_pred:
        test_pred.append(l)
    # adiciona resultado da predicao ao detector
    for e in sample_label == sample_pred:
        drift.add_element(e)
    # detecta se houve warning ou drift
    if drift.detected_warning_zone():
        # atualiza classificador 2
        clf2.partial_fit(sample_features, sample_label)
    if drift.detected_change():
        # salva ponto de drift
        drifts.append(count)
        # muda classificador 1 pelo classificador 2
        clf1 = clone(clf2)
        # reseta classificador 2
        clf2 = HoeffdingTree()
        # reseta detector de drift
        drift.reset()
    # atualiza modelo com novo exemplar
    clf1.partial_fit(sample_features, sample_label)
# imprime pontos de drift
print("Drifts in {}.".format((drifts)))
```

```
Drifts in [19936, 20938, 21126, 21183, 24425, 24493,
  ↪ 24536, 24729].
```

Código 20. Utilizando um detector de *drift* para classificar um conjunto de testes criado a partir do *dataset Brazilian Malware*

 © 2019 Sociedade Brasileira de Computação

uma acurácia próxima de 100% em problemas de classificação de *malware*, isto é, eles avaliam somente se todos os exemplares, independente de sua classe e distribuição de dados, foram corretamente classificados, o que no fundo pode não ser uma forma adequada de avaliar o problema [Ceschin et al. 2018]. Por exemplo, considere que um modelo é avaliado utilizando dez exemplares: oito deles são benignos e dois, malignos. Este modelo tem acurácia de 80%. Seria 80% uma acurácia satisfatória? Supondo que o modelo acerte apenas os oito exemplares benignos, ele não é capaz de identificar nenhum exemplo maligno e mesmo assim a medida de acurácia atinge 80%, passando a falsa impressão de que o modelo funciona bem. É necessário lembrar que, em problemas binários em

```
1  import keras
2  from keras.models import Sequential
3  from keras.layers import Dense
4  # converte rotulos para a codificacao one-hot encoding
5  train_label_onehot =
   ↪ keras.utils.to_categorical(train_label,
   ↪ num_classes=2)
6  test_label_onehot =
   ↪ keras.utils.to_categorical(test_label,
   ↪ num_classes=2)
7  # inicializa rede sequencial
8  model = Sequential()
9  # adiciona camada escondida com 200 neuronios
10 model.add(Dense(200, activation='relu',
   ↪ input_dim=train_features_norm.shape[1]))
11 # adiciona camada escondida com 100 neuronios
12 model.add(Dense(100, activation='relu'))
13 # camada de saida
14 model.add(Dense(2, activation='softmax'))
15 # compila modelo
16 model.compile(loss='binary_crossentropy',
   ↪ optimizer='rmsprop', metrics=['accuracy'])
17 # treina modelo com dados de treino
18 model.fit(train_features_norm, train_label_onehot,
   ↪ validation_split=0.33, epochs=10, batch_size=128,
   ↪ verbose=0)
19 # prediz classes
20 test_pred = model.predict_classes(test_features_norm)
21 # imprime forma da predicao e dos rotulos reais de teste
22 print(test_pred.shape, test_label.shape)
```

```
1  (25090,) (25090,)
```

Código 21. Utilizando o classificador *Multi-Layer Perceptron* para classificar um conjunto de testes criado a partir do *dataset Brazilian Malware*.

geral, um modelo de aprendizado de máquina deve ser robusto o suficiente para identificar padrões de forma genérica, independente da classe utilizada, já que não queremos um modelo que classifique tudo como sendo de apenas uma única classe. Nesta Seção apresentaremos as métricas mais comuns utilizadas em problemas de segurança, bem como métodos de validar as soluções.

2.6.1. Métricas

Para poder avaliar corretamente um modelo, deve-se escolher corretamente uma métrica, já que cada uma pode apresentar uma perspectiva do problema (e até mesmo dar falsas impressões sobre o resultado geral do problema, como no exemplo mencionado anteriormente). Neste curso apresentaremos as seguintes métricas: acurácia, matriz de confusão, *recall*, *precision* e *f1score*.

2.6.1.1. Acurácia

A acurácia é uma medida que mede a porcentagem de exemplares que o classificador conseguiu acertar de determinado conjunto (normalmente do conjunto de testes). Basicamente, corresponde ao número de exemplares corretamente classificados dividido pelo número de exemplares apresentados ao classificador. A biblioteca *Scikit-Learn* implementa a acurácia através do método `accuracy_score`[31], bastando passar como parâmetro as predições feitas pelo classificador e os rótulos do conjunto de teste utilizado (classes reais do conjunto), como mostrado no Código 22. Esta métrica não é recomendada para problemas cujo conjunto de dados é desbalanceado, uma vez que ela pode ter um alto índice, mesmo quando o classificador favorece a classe majoritária e erra todos os exemplares de classes minoritárias [Gron 2017].

```python
from sklearn.metrics import accuracy_score
print(accuracy_score(test_label, test_pred))
```

```
0.8836986847349542
```

Código 22. Acurácia do *Random Forest* para classificar um conjunto de teste criados a partir do *dataset Brazilian Malware*.

2.6.1.2. Matriz de Confusão

A matriz de confusão é uma forma de melhor visualizar os resultados gerados por um classificador. Seja C uma matriz de confusão, $C_{i,j}$ é igual ao número de observações em que o classificador considerou um exemplar da classe i como sendo da classe j. Em classificação binária podemos extrair informações da matriz de confusão: os verdadeiro negativos (*True Negatives – TN*), que são $C_{0,0}$, os falsos negativos (*False Negatives – FN*), que são

[31] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html`

 © 2019 Sociedade Brasileira de Computação

$C_{1,0}$, verdadeiros positivos (*True Positives – TP*), que são $C_{1,1}$ e falsos positivos (*False Positives – FP*), que são $C_{0,1}$ [Pedregosa et al. 2011]. O Código 23 apresenta a matriz de confusão para o nosso problema de exemplo, bastando utilizar o método `confusion_matrix`[32] da *Scikit-Learn*, passando como parâmetro as predições e as classes reais dos dados de teste utilizados. É possível observar que o número de verdadeiros negativos é de 6987 ($C_{0,0}$), o de falsos negativos é de 2672 ($C_{1,0}$), o de verdadeiros positivos é de 15185 ($C_{1,1}$) e o de falsos positivos é de apenas 46 ($C_{0,1}$). A partir dessas informações, duas novas medidas podem ser extraídas (*recall* e *precision*), gerando uma terceira métrica que utiliza ambas (*f1score*).

```
1 from sklearn.metrics import confusion_matrix
2 print(confusion_matrix(test_label, test_pred))
```

```
1 [[ 6987    46]
2  [ 2872 15185]]
```

Código 23. Matriz de confusão do *Random Forest* para classificar um conjunto de teste criados a partir do *dataset Brazilian Malware*.

2.6.1.3. *Recall*

O *recall*, também chamado de *sensitivity* ou *true positive rate* (*TPR*), é a proporção de instâncias positivas que são corretamente classificadas pelo classificador, isto é, a habilidade do classificador encontrar todos os exemplares positivos [Gron 2017, Pedregosa et al. 2011]. O mesmo utiliza o número de verdadeiros positivos (*True Positives – TP*) e falsos negativos (*False Negatives – FN*) e é dado pela Equação 13. O *Scikit-Learn* implementa essa métrica através do método `recall_score`[33], passando como referência o vetor de predição e o vetor de classes reais, como mostrado no Código 24. Em nosso problema exemplo, obtemos cerca de 84% de *recall*, indicando que o classificador confunde alguns *malware* como sendo *goodware* (cerca de 16%).

$$Recall = \frac{TP}{TP+FN} \tag{13}$$

2.6.1.4. *Precision*

A *precision* é uma métrica capaz de medir a acurácia das predições positivas do classificador, isto é, ela mede a habilidade do classificador não rotular como positivo um exemplar que é negativo [Gron 2017, Pedregosa et al. 2011]. Essa métrica utiliza o número de verdadeiros positivos (*True Positives – TP*) e o número de falsos positivos (*False Positives*

[32] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

[33] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html

© 2019 Sociedade Brasileira de Computação

```
from sklearn.metrics import recall_score
print(recall_score(test_label, test_pred))
```

```
0.8409481087666832
```

Código 24. *Recall* do *Random Forest* para classificar um conjunto de teste criados a partir do *dataset Brazilian Malware*.

– *FP*) e é dada pela Equação 14. O *Scikit-Learn* também implementa essa métrica através do método `precision_score`[34], passando como parâmetro o vetor de predição e o vetor de classes reais, como mostrado no Código 25. No exemplo, obtemos cerca de 99% de *precision*, indicando que poucos *goodware* estão sendo confundidos como sendo *malware* (menos de 1% são confundidos).

$$Precision = \frac{TP}{TP + FP} \tag{14}$$

```
from sklearn.metrics import precision_score
print(precision_score(test_label, test_pred))
```

```
0.9969798437397414
```

Código 25. *Precision* do *Random Forest* para classificar um conjunto de teste criados a partir do *dataset Brazilian Malware*.

2.6.1.5. *F1Score*

O *F1Score* utiliza tanto o *recall* como a *precision* para gerar uma única métrica, que nada mais é do que a média harmônica entre essas duas métricas, dado pela Equação 15 [Gron 2017, Pedregosa et al. 2011]. O *Scikit-Learn* implementa essa métrica através do método `f1_score`[35], passando como parâmetro o vetor de predição e o vetor de classes reais, como mostrado no Código 26. Em nosso problema exemplo, obtemos cerca de 91% de *f1score*.

$$F1Score = \frac{2 * (Precision * Recall)}{Precision + Recall} \tag{15}$$

Embora sera interessante, é sempre bom reportar o *recall* e a *precision* individualmente, uma vez que em determinados problemas, um pode ser mais importante que o

[34] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html`
[35] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html`

 © 2019 Sociedade Brasileira de Computação

```
1 from sklearn.metrics import f1_score
2 print(f1_score(test_label, test_pred))
```

```
1 0.9123407834655132
```

Código 26. *F1Score* do *Random Forest* para classificar um conjunto de teste criados a partir do *dataset Brazilian Malware*.

outro e o *F1Score* pode não captar essa necessidade, já que é apenas uma média entre os dois [Gron 2017]. Para um detector de *malware*, por exemplo, muitas vezes pode ser preferível que um *malware* não seja detectado do que um software benigno seja bloqueado (alta *precision*), já que afetaria diretamente na experiência de uso de um usuário. Imagine que um usuário esteja usando seu computador e, ao abrir o *Microsoft Word*, o classificador acredita que ele seja um *malware* e bloqueia a execução do programa. Por mais que o classificador detecte grande maioria dos *malware*, ele torna o computador inutilizável pelo usuário, já que bloquearia grande partes das aplicações benignas do usuário. Em contrapartida, em alguns sistemas mais sensíveis, pode-se querer o contrário (alto *recall*), bloqueando todas as ações malignas, mesmo que algumas benignas sejam "sacrificadas". Essas configurações podem ser feitas utilizando probabilidades de cada classe na saída do classificador, bastando definir um limiar (*threshold*) para que o exemplar seja de uma classe ou de outra, como mostrado no Código 27. Em relação aos resultados anteriores, nota-se que a *precision* aumentou um pouco e o *recall* caiu significativamente, isto é, o número de *malware* classificado como *goodware* aumentou muito em troca de um pequeno número de *goodware* deixando de ser confundido com *malware*. É importante ressaltar que a medida que você aumenta a *precision*, o *recall* diminui e vice versa. Este problema é conhecido como *precision/recall tradeoff* [Gron 2017]. A Figura 2.15 apresenta exatamente esse *tradeoff* entre *recall* e *precision* para o problema exemplo proposto, variando o *threshold* de 0 a 100%.

```
1 # valor do threshold
2 threshold = 0.7
3 # obtem probabilidade de cada exemplar
4 test_proba = clf.predict_proba(test_features_norm)
5 # cria um vetor com as classes, aplicando threshold
6 test_pred = (test_proba[:,1] > threshold).astype('int')
7 # imprime recall e precision
8 print("Rec:", recall_score(test_label, test_pred))
9 print("Pre:", precision_score(test_label, test_pred))
```

```
1 Rec: 0.7345073932546935
2 Pre: 0.9975930801053028
```

Código 27. *Recall* e *precision* utilizando um *threshold* de 70% para classificar um conjunto de teste criados a partir do *dataset Brazilian Malware*.

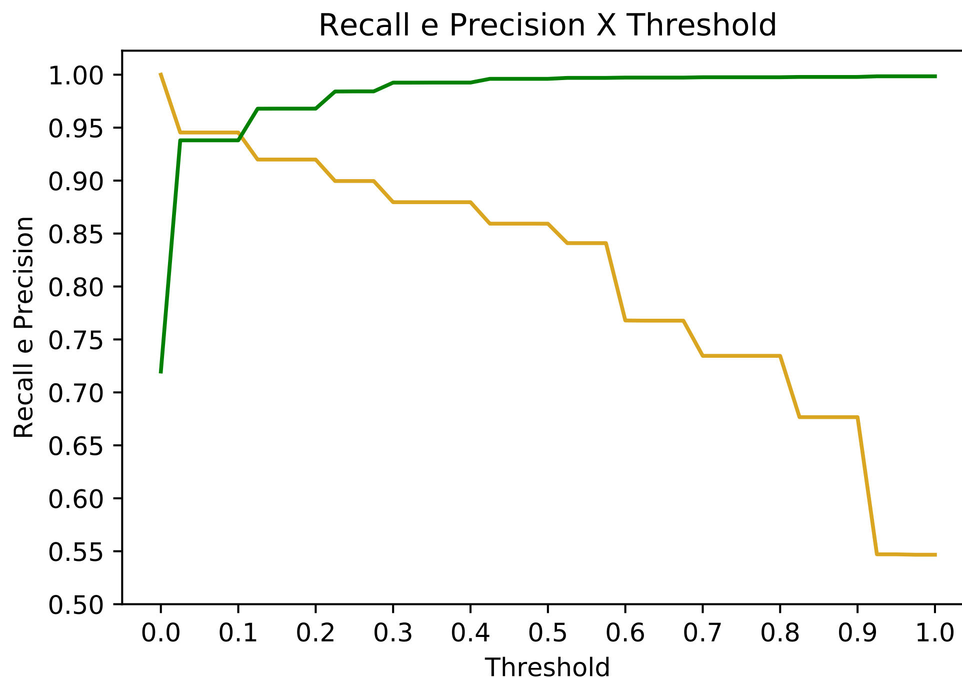


Figura 2.15. *Tradeoff* do *recall* e *precision* para classificar um conjunto de teste criados a partir do *dataset Brazilian Malware*.

2.6.2. Validação

A validação de um modelo é importante pois define o quão genérico o mesmo é, evitando um fenômeno chamado de *overfitting*: quando um classificador "decora" os dados utilizados no treino, obtendo altas taxas de acurácia, porém sem generalizar o problema, isto é, ao ser utilizado de fato no mundo real, o classificador passa a atuar de forma muito diferente do esperado, errando muito mais do que o observado [Gron 2017]. Para problemas em que os dados estão disponíveis em *batches*, é muito comum utilizar o um método chamado de *K-Fold Cross Validation*, que consiste basicamente em dividir aleatoriamente os dados em *K* subconjuntos distintos chamados de *folds* e treinar e avaliar o modelo utilizado *K* vezes, utilizando um *fold* diferente para a avaliação (teste) cada vez e os outros *K-1* para treino. O Código 28 apresenta um exemplo de *K-Fold Cross Validation*, utilizando o método `cross_val_score`[36] da biblioteca *Scikit-Learn*, em que o primeiro parâmetro é o classificador a ser utilizado, o segundo é o conjunto de características, o terceiro é o conjunto de rótulos dessas características, *cv* é o número de *folds* criados e *scoring* é a métrica a ser utilizada. Pelos resultados, observa-se que as acurácias em nosso problema variaram de cerca de 61%, no primeiro fold, até o máximo de quase 99%, atingindo um valor médio de 94% de acurácia.

Para dados em *stream*, já mencionamos a forma correta de se validar um modelo, já que o mesmo simula um classificador que recebe dados em "tempo real", se adaptando à mudanças conforme o tempo passa. Geralmente não é possível utilizar o *K-Fold Cross Validation* em problemas de *stream*, pois o mesmo acaba misturando de forma aleatória

[36] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html`

 © 2019 Sociedade Brasileira de Computação

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
import numpy as np
# inicializa classificador
clf = RandomForestClassifier(n_estimators=10)
# obtem resultados
results = cross_val_score(clf, train_features_norm,
    ↪ train_label, cv=10, scoring="accuracy")
# imprime vetor de acuracia por fold
print(results)
# imprime media dos resultados
print(np.mean(results))
```

```
[0.61195219 0.92151394 0.99561753 0.99282583 0.99521722
    ↪ 0.99163013 0.9920287  0.96811479 0.98883573
    ↪ 0.97886762]
0.9436603674644856
```

Código 28. Resultados do *K-Fold Cross Validation* utilizando o classificador *Random Forest* para classificar um conjunto de dados criados a partir do *dataset Brazilian Malware*.

os exemplares de épocas (e conceitos) diferentes, o que não torna o resultado real: é mais fácil para um classificador "prever o futuro" se ele já viu um exemplo similar ao da época em que vai ser avaliado. Por exemplo, supondo que os *malware* do tipo *Ransomware* ficaram comuns em 2017, avalia-se se um modelo é capaz de detectar esse tipo de ameaça utilizando dados coletados de 2015 a 2017. Ao utilizar o *K-Fold Cross Validation*, é possível que os exemplares de 2017, contendo *RansomWare*, sejam inseridos em algum momento durante o treino, fazendo com que o modelo funcione bem e obtenha uma ótima métrica. Entretanto, na realidade, o classificador não teria nenhum exemplar semelhante até que os mesmos fossem de fato desenvolvidos e conhecidos, tornando o problema seja muito mais difícil.

2.7. Conclusão

O aprendizado de máquina pode ser um poderoso aliado a segurança computacional se utilizado corretamente. Entretanto, ainda é comum encontrar soluções cujos resultados não condizem com a realidade. Por conta disso, é necessário ter certeza de que a solução proposta realmente funciona no mundo real, avaliando-a da forma correta, isto é, levando em consideração o fato de que os dados podem não ter distribuição estacionária e estar em constante mudança.

Neste capítulo, apresentamos diversos conceitos de aprendizado de máquina fundamentais para a área de segurança. Começamos com a coleta dos dados, na Seção 2.2, mostrando como realizá-la e rotular corretamente, os principais problemas relacionados a isso e dois exemplos de *datasets*. Na sequência, apresentamos técnicas de extração de

© 2019 Sociedade Brasileira de Computação

atributos estáticas de Windows e Android e recomendamos ferramentas para a análise dinâmica dos mesmos, na Seção 2.3. Então, na Seção 2.4, mostramos dois algoritmos de extração de características (TF-IDF e *Word2Vec*) e dois métodos de normalização das mesmas (*min-max* e *standardization*). Em seguida, os principais classificadores, detectores e agrupadores foram apresentados, com seus respectivos códigos e bibliotecas, destacando como se utilizar uma *stream* e detectores de *drift*, na Seção 2.5. Para finalizar, mostramos as principais métricas para avaliar corretamente os modelos utilizados, bem como as formas de avaliá-los, incluindo até mesmo uma técnica para poder definir um limiar para os resultados do classificador, na Seção 2.6.1.

Deste modo, esperamos que este trabalho tenha um impacto positivo na comunidade acadêmica e na indústria, fomentando melhorias nas soluções que envolvem ambos os temas (aprendizado de máquina e segurança) e servindo de base para estudos introdutórios ou mesmo confecção de cursos mais avançados.

Referências

[Abadi et al. 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from `tensorflow.org`.

[Albert Bifet 2018] Albert Bifet, Ricard Gavalda, G. H. B. P. (2018). *Machine Learning for Data Streams with Practical Examples in MOA*. MIT Press. `https://moa.cms.waikato.ac.nz/book/`.

[Almeida et al. 2015] Almeida, P., Oliveira, L., Britto, A., and Sabourin, R. (2015). Dealing with concept drifts using dynamic ensembles of classifiers. Tesis presented as partial requirement for the degree of Doctor. Graduate Program in Informatics, Sector of Exact Sciences, Universidade Federal do Paraná.

[Arp et al. 2014] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*.

[Baena-Garcıa et al. 2006] Baena-Garcıa, M., del Campo-Ávila, J., Fidalgo, R., Bifet, A., Gavaldà, R., and Morales-Bueno, R. (2006). Early drift detection method.

[Bifet and Gavaldà 2007] Bifet, A. and Gavaldà, R. (2007). Learning from time-changing data with adaptive windowing. volume 7.

[Bishop 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

 © *2019 Sociedade Brasileira de Computação*

[Botacin et al. 2018] Botacin, M., de Geus, P. L., and Grégio, A. (2018). The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98.

[Breiman 2001] Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.

[Ceschin et al. 2018] Ceschin, F., Pinage, F., Castilho, M., Menotti, D., Oliveira, L. S., and Gregio, A. (2018). The need for speed: An analysis of brazilian malware classifers. *IEEE Security Privacy*, 16(6):31–41.

[Chollet et al. 2015] Chollet, F. et al. (2015). Keras. `https://github.com/fchollet/keras`.

[Cortes and Vapnik 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

[Dwork and Roth 2014] Dwork, C. and Roth, A. (2014). The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407.

[Ester et al. 1996] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*.

[Filho et al. 2011] Filho, D. S. F., Afonso, V. M., Martins, V. F., Grégio, A. R. A., de Geus, P. L., Jino, M., and dos Santos, R. D. C. (2011). Técnicas para análise dinâmica de malware. *Minicurso do SBSEG*. `https://sbseg2011.redes.unb.br/resources/downloads/minicursos/91936.pdf`.

[Fradkin and Muchnik 2006] Fradkin, D. and Muchnik, I. (2006). Support vector machines for classification. *"Discrete Methods in Epidemiology", DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 70:13–20.

[Gama et al. 2004] Gama, J., Medas, P., Castillo, G., and Rodrigues, P. (2004). Learning with drift detection. In Bazzan, A. L. C. and Labidi, S., editors, *Advances in Artificial Intelligence – SBIA 2004*, pages 286–295, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Gama et al. 2014] Gama, J. a., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37.

[Gandotra et al. 2014] Gandotra, E., Bansal, D., and Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security*, 5(2):56–64.

[Gron 2017] Gron, A. (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition.

[Haykin 2009] Haykin, S. (2009). *Neural Networks and Learning Machines*. Number v. 10 in Neural networks and learning machines. Prentice Hall.

[Hoffmann et al. 2016] Hoffmann, J., Rytilahti, T., Maiorca, D., Winandy, M., Giacinto, G., and Holz, T. (2016). Evaluating analysis tools for android apps: Status quo and robustness against obfuscation. pages 139–141.

[Jordan 2017] Jordan, M. I. (2017). The kernel trick, advanced topics in learning & decision making. `https://people.eecs.berkeley.edu/~jordan/courses/281B-spring04/lectures/lec3.pdf`, accessed in July 2017.

[Manning et al. 2008a] Manning, C. D., Raghavan, P., and Schütze, H. (2008a). *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

[Manning et al. 2008b] Manning, C. D., Raghavan, P., and Schütze, H. (2008b). *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK.

[Mellish 2017] Mellish, C. (2017). Machine learning, lecture notes. `http://www.inf.ufpr.br/lesoliveira/aprendizado/machine_learning.pdf`, accessed in July 2017.

[Michie et al. 1994] Michie, D., Spiegelhalter, D. J., Taylor, C. C., and Campbell, J., editors (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, Upper Saddle River, NJ, USA.

[Mikolov et al. 2013a] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

[Mikolov et al. 2013b] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546.

[Milgram et al. 2006] Milgram, J., Cheriet, M., and Sabourin, R. (2006). "One Against One" or "One Against All": Which One is Better for Handwriting Recognition with SVMs? In Lorette, G., editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France). Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[Mitchell 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

[Montiel et al. 2018] Montiel, J., Read, J., Bifet, A., and Abdessalem, T. (2018). Scikit-multiflow: A multi-output streaming framework. *Journal of Machine Learning Research*, 19(72):1–5.

[Oktavianto and Muhardianto 2013] Oktavianto, D. and Muhardianto, I. (2013). *Cuckoo Malware Analysis*. Packt Publishing.

[Pedregosa et al. 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

 © *2019 Sociedade Brasileira de Computação*

[Řehůřek and Sojka 2010] Řehůřek, R. and Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta. ELRA. `http://is.muni.cz/publication/884893/en`.

[Rogers and Girolami 2011] Rogers, S. and Girolami, M. A. (2011). *A First Course in Machine Learning*. Chapman and Hall / CRC machine learning and pattern recognition series. CRC Press.

[Rong 2014] Rong, X. (2014). word2vec parameter learning explained. *CoRR*, abs/1411.2738.

[Saxe and Sanders 2018] Saxe, J. and Sanders, H. (2018). *Malware Data Science: Attack Detection and Attribution*. No Starch Press, San Francisco, CA, USA.

[Schubert et al. 2017] Schubert, E., Sander, J., Ester, M., Kriegel, H. P., and Xu, X. (2017). Dbscan revisited, revisited: Why and how you should (still) use dbscan. *ACM Trans. Database Syst.*, 42(3):19:1–19:21.

[Sebastián et al. 2016] Sebastián, M., Rivera, R., Kotzias, P., and Caballero, J. (2016). Avclass: A tool for massive malware labeling. In Monrose, F., Dacier, M., Blanc, G., and Garcia-Alfaro, J., editors, *Research in Attacks, Intrusions, and Defenses*, pages 230–253, Cham. Springer International Publishing.

[Seide and Agarwal 2016] Seide, F. and Agarwal, A. (2016). Cntk: Microsoft's open-source deep-learning toolkit. pages 2135–2135.

[Shulman 2016] Shulman, B. (2016). A tour of sentiment analysis techniques: Getting a baseline for sunny side up. `https://gab41.lab41.org/a-tour-of-sentiment-analysis-techniques-getting-a-baseline-for-sunny-`

[Symantec 2019] Symantec (2019). 2019 internet security threat report. `https://www.symantec.com/security-center/threat-report`.

[Tam et al. 2017] Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., and Cavallaro, L. (2017). The evolution of android malware and android analysis techniques. *ACM Comput. Surv.*, 49(4):76:1–76:41.

[Theano Development Team 2016] Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.

[Wang et al. 2011] Wang, S., Schlobach, S., and Klein, M. (2011). Concept drift and how to identify it. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(3):247 – 265. Semantic Web Dynamics Semantic Web Challenge, 2010.

[Yonts 2010] Yonts, J. (2010). *Building a Malware Zoo*. The SANS Institute.

© 2019 Sociedade Brasileira de Computação