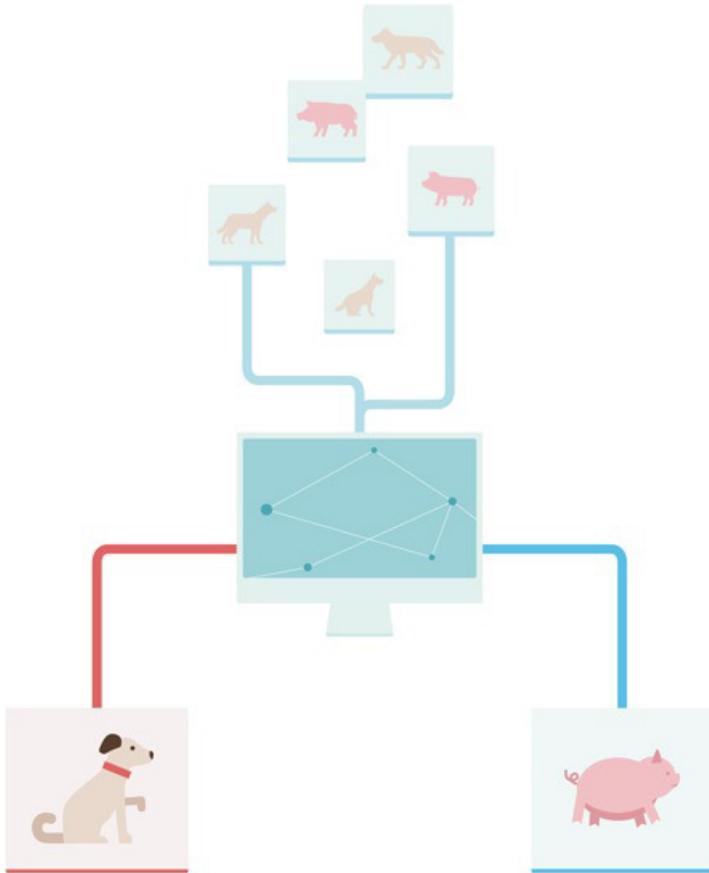


# Machine Learning

Introdução à classificação



Casa do  
Código

— ■ —  
SÉRIE CAELUM

GUILHERME SILVEIRA  
BENNETT BULLOCK

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

*Edição*

Adriano Almeida

Vivian Matsui

*Revisão*

Bianca Hubert

Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

[www.casadocodigo.com.br](http://www.casadocodigo.com.br)



## **SOBRE O GRUPO CAELUM**

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura ([www.alura.com.br](http://www.alura.com.br)), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum ([www.caelum.com.br](http://www.caelum.com.br)), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

# ISBN

Impresso e PDF: 978-85-94188-18-2

EPUB: 978-85-94188-19-9

MOBI: 978-85-94188-20-5

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

## AGRADECIMENTOS

Agradeço ao meu pai e à minha mãe, que aguentaram anos e anos de perguntas, e sempre me instigaram a perguntar mais. Gostaria de um dia ter a paciência que eles tiveram com minha educação.

Gostaria de agradecer ao Alex Felipe, que fez a transposição original do curso da Alura de Machine Learning neste livro.

# SOBRE O AUTOR

## **Guilherme Silveira**

Meu nome é Guilherme de Azevedo Silveira, e aprendi a programar aos 9 anos. Meu irmão foi meu professor particular, me instigando a criar pequenos jogos em Basic. Trabalho com desenvolvimento de software desde meu primeiro estágio, em 1996.

Cresci como profissional utilizando a linguagem Java, apesar de minha jornada ter me levado às mais diversas linguagens, desde C, passando por R, Swift etc.

Por gostar de ensinar computação desde criança, acabei por fundar a Caelum junto ao Paulo Silveira, em 2004. Escrevo livros para a Casa do Código, e hoje sou *head* de educação do Grupo Caelum, atuando principalmente na Alura e Alura Start.

Meu primeiro contato com métodos de análise foi através de aulas que fiz na matemática aplicada do IME-USP. Como todo fiel da matemática e análise, perdi dinheiro na bolsa e errei a previsão de jogos de futebol. Por outro lado, a maturidade no ensino me trouxe a oportunidade de analisar por mais de 14 anos os dados de nossos cursos e alunos, tentando encontrar maneiras diferentes de ajudá-los.

## **Bennett Bullock**

Bennett Bullock é um profissional com 14 anos de experiência no campo de aprendizagem de máquina e NLP. Ele tem

desenvolvido tecnologias de classificação de texto, de pesquisa (na qual obteve uma patente), e de análise financeira para vários clientes no governo americano e na comunidade financeira. Ele é mestre em linguística árabe pela Universidade de Georgetown, e em bioengenharia pelo MIT.

# INTRODUÇÃO

## O caminho com machine learning

Em 1993, eu tinha doze anos quando meu pai dizia que um computador ajudaria muito se fosse capaz de aprender e ensinar, como um adulto faz. Neste livro, abordaremos o aprendizado de máquina, *machine learning*.

Existe uma ilusão de que matemática é teórica e há uma aura mágica ao redor de quem a estuda, como se fosse algo meramente nato. Como qualquer bom jogador de futebol, estudei e pratiquei dia e noite, fazendo chuva ou sol.

Mas não precisava tanto para chegar a pequenas conclusões matemáticas. Quando um apartamento de 2 dormitórios tem um preço médio de R\$ 2.000 reais, e um de 3 dormitórios custa R\$ 3.000 reais, não é necessário um doutorado para aprender que o preço do apartamento é R\$ 1.000 reais vezes o número de dormitórios.

Claro que, no mundo real, o preço de um apartamento é afetado por diversas outras variáveis; influenciam no preço desde o tamanho em metros quadrados, o número de banheiros, a localização geográfica, até as condições de manutenção de um edifício. Quanto cada uma influencia? Aí precisamos de contas mais complexas para aprender suas influências.

No fim, são contas. Contagens como soma, multiplicação e potência formam a base da estatística e matemática computacional, que está por trás dos processos complexos capazes de modelar o

que acontece no mundo real. Esses processos aprendem com os dados que existem aí fora, mostrando para nós uma simplificação que nos permite adivinhar o preço justo de novos apartamentos.

Neste livro, usaremos diversos exemplos de aprendizado de máquina. Veremos, como seres humanos, como distinguimos porcos e cães, e treinaremos um modelo matemático computacional em Python para distinguir um do outro.

Aplicando esse conhecimento básico, veremos como categorizar outras informações importantes do mundo real de negócios. Sabendo qual o comportamento de usuários do meu site e quais compraram meu produto, serei capaz de adivinhar antecipadamente quais abandonarão o site sem comprar nada?

Veremos como trabalhar com variáveis numéricas e categóricas, como testar e validar nossos algoritmos e decisões. Trabalharemos com textos longos, aprendendo a transformá-los em vetores que conseguimos trabalhar, além de passar por todo um processo de padronização e limpeza do texto antes de sua normalização.

Acima de tudo, veremos os cuidados que temos de tomar ao tentar encontrar modelos. Não queremos viciar nossas decisões e acreditar encontrar relações que não existem — algo que é muito fácil de errar.

## O curioso em você

Este livro foca em mostrar algoritmos e processos que podemos usar para responder perguntas do dia a dia baseadas em dados. Quanto mais dados, melhor.

Não estamos preocupados aqui com *buzz words*, como *data scientist* ou *big data*. Nossa foco é conhecer alguns algoritmos, entender como eles podem funcionar por trás, que tipo de perguntas podemos fazer aos dados e como cuidar para que essas perguntas não nos entreguem uma mentira.

Depois do livro, sinta-se à vontade para explorar bibliotecas em outras linguagens (como R), variações de análises de dados (*data analysis*) como séries temporais, classificações de longos textos ou sentimentos etc. Para alguém curioso(a), não existem limites para as perguntas. Ao término do livro, você terá algumas ferramentas novas para respondê-las.

A experiência de ensino também me colocou em contato com pessoas da área que me lembraram de que a inteligência artificial e o machine learning são ferramentas para responder perguntas. Saber quais e quantas perguntas fazer e como interpretar as respostas são habilidades importantíssimas.

## **Matemática ou programação?**

Na prática, raramente um desenvolvedor implementa esses algoritmos a partir das contas matemáticas envolvidas neles. Mas veremos como funciona a base de um dos algoritmos, para entender que machine learning não é mágica, mas sim uma grande sacada. Assim como o fogo, a roda ou a caneta quatro cores, não existe mágica impossível de entender. Se estudarmos, aprenderemos a usar; e se desejarmos, aprenderemos como tudo funciona por trás dos panos.

## **Público-alvo e pré-requisitos**

Este livro é para quem já conhece programação e gostaria de entender e ser introduzido ao mundo de machine learning. Apesar de usarmos Python 3 no livro, quem vem de qualquer outra linguagem de programação não deve ter dificuldades em entender o código escrito aqui. Tentei ser o mais delicado possível ao introduzir conceitos e funções, mesmo quando eles são da própria API da linguagem.

Para quem está começando com programação e não conhece Python, será um pouco mais puxado. Dê uma folheada e veja se a maneira como a linguagem é mostrada o deixa confortável em seus estudos.

Confira qual a versão do Python instalada em sua máquina com `python --version`. Lembre-se de instalar a versão 3 e utilizar o comando `python3` e `pip3`, como no livro.

Professores que ensinam machine learning através de programação ou matemática poderão utilizar este livro como apoio para mostrar aplicações práticas e realistas desses algoritmos.

## Os arquivos

Para acompanhar este livro, usaremos diversos arquivos. Você pode baixar os arquivos de dados e todo o código do livro em:

<https://github.com/guilhermesilveira/machine-learning/archive/master.zip>

O GitHub com todos os arquivos também pode ser acessado em:

<https://github.com/guilhermesilveira/machine-learning>

# Sumário

<b>1 Classificando e-mails, animais e muito mais</b>	<b>1</b>
1.1 Resumindo	39
<b>2 Importando, classificando e validando um modelo</b>	<b>40</b>
2.1 E no mundo real, como classificar dados da web?	40
2.2 Importando, classificando e validando um modelo	49
2.3 Importando os dados	50
2.4 Analisando os valores adicionados	57
2.5 Melhorando a leitura do código	61
2.6 Acertando demasiadamente?	71
2.7 Resumindo	80
<b>3 Classificação de variáveis categóricas</b>	<b>84</b>
3.1 Instalando o Pandas	108
3.2 Resumindo	133
<b>4 O problema do sucesso e o algoritmo burro</b>	<b>135</b>
4.1 Implementando o algoritmo base	143
4.2 Calculando a quantidade de zeros e uns com o data frame	147

4.3 Lidando com sim e não	151
4.4 Utilizando collections do Python	161
4.5 Resumindo	166
<b>5 Naive bayes e maximum a posteriori por trás dos panos</b>	<b>172</b>
5.1 Resumindo	190
<b>6 Testando diferentes modelos e validando o vencedor</b>	<b>193</b>
6.1 Algoritmo AdaBoost	204
6.2 Resumindo	232
<b>7 Novos conceitos de classificação</b>	<b>236</b>
7.1 Classificando um elemento com três categorias	238
7.2 Resumindo	272
<b>8 Utilizando o k-fold</b>	<b>274</b>
8.1 Implementando o k-fold	284
8.2 Implementando o novo fit_and_predict	294
8.3 Resumindo	310
<b>9 Criando um dicionário</b>	<b>311</b>
9.1 Resumindo	319
<b>10 Classificando os textos e ganhando produtividade na empresa</b>	<b>320</b>
10.1 Resumindo	367
<b>11 Quebrando na pontuação adequada</b>	<b>369</b>
<b>12 Conclusão</b>	<b>404</b>
12.1 O caminho	404
12.2 Como continuar os estudos	405

Versão: 22.7.7

## CAPÍTULO 1

# CLASSIFICANDO E-MAILS, ANIMAIS E MUITO MAIS

Neste capítulo, vamos falar de diversos problemas que podemos resolver com **machine learning**, inteligência artificial, aprendizagem de máquina, isto é, com algumas dessas palavras bonitas que estão ligadas de alguma maneira com soluções para resolver problemas um pouco mais difíceis no nosso dia a dia.

Nosso propósito não é mostrar o formalismo, e por isso mesmo não nos apegaremos a definições muito formais. O objetivo é instigar a sua curiosidade de utilizar algoritmos para a solução de diversos tipos de problemas. Ao terminar este livro, você terá a liberdade de escolher seu caminho, e aí sim o formalismo será de extrema importância.

Vamos dar uma olhada em que tipos de problemas podemos resolver, principalmente no começo deste livro?

Recebi e-mails hoje de manhã, vários. Não sei dizer quantos e-mails você recebe, mas eu recebo diversos. Você provavelmente também recebe muitos e nem percebe, mas por que você não percebe? Repare em todos esses e-mails que eu recebi:

<a href="#">Se Compran Cheques</a>	<b>o Facturas Impagas.</b> - COMPRAMOS CHEQUES PROTESTADOS Y FACTURAS IMPAGAS CONTACTAR SOLO INTERI
<a href="#">Hotels-Paris.fr</a>	The Best Design Hotels in Paris with the Best Rates... Up to 65% off ! - This newsletter is in HTML format, if you cannot
<a href="#">Ativo.com</a>	<small>corrida</small> Aproveite! Assine 1 Ano de Revista VO2 e escolha seu Óculos HB! - Caso não esteja visualizando este e-mail,
<a href="#">Charming Events</a>	Dine with Gary Rhodes this Christmas - Fine Dining this Christmas at the Gary Rhodes W1 Restaurant Friday 7th, 14th ar
<a href="#">Casas Bahia - BMC</a>	TV LED em Dobro! TV 32" + TV 22" por apenas 1.699 em 12x sem juros. Aproveite! - Caso não esteja visualizando as :
<a href="#">Cordell Miller</a>	Les réductions d'été sur les meilleures montres les Meilleures marques de 99.99 - Cada fabricador conocido del reloj d
<a href="#">Lepetitjournal.com</a>	Lepetitjournal de Sao Paulo - Edition du 03.08.2012 - Des problèmes pour lire la newsletter? Regardez la sur votre navigi
<a href="#">eFacil</a>	Super presente para o Papai! Até 40% de desconto em TVs, Home Theaters e Mini Systems. - Caso não esteja visuali

Figura 1.1: Meus e-mails

Tem e-mail que parece que não fui eu que pedi para receber, por exemplo, "*Se Compran Cheques?*". E-mail em espanhol? Eu nem sei falar espanhol. **The Best Design Hotels in Paris?** E-mail sobre Paris? Realmente, estou aprendendo francês e por isso assinei um site ou outro, então tudo bem. Mas e **Casas Bahia?** Não assinei e-mails da Casas Bahia, e ela mandou sem eu pedir.

Repare que tem um monte de e-mails aqui que parece ser *spam*. A quantidade de *spam* que vai de um lado para o outro na internet hoje em dia é absurda e, de alguma maneira, alguém precisa classificá-los e dizer: "Este e-mail aqui é *spam*!"

É preciso que alguém na internet faça isso, porque se esse alguém for eu, e eu tiver de olhar os 1.000 e-mails que recebo para dizer que, desses 1.000 e-mails, 990 são *spam*, a minha vida acaba! E eu não quero que a minha vida acabe, e você também não!

Queremos viver alegremente com tempo para outras coisas, e não passar o dia inteiro classificando: "É *spam*. Não é *spam*. É *spam*. Não é *spam*...". Queremos que alguém faça isso por nós: "Por favor, diga para mim se isso aí é *spam* ou se não é *spam*". E esse alguém pode ser uma pessoa!

Eu posso contratar uma pessoa para dizer: "Olha, isso aqui é *spam*", "Esse aqui não é *spam*". Nós a pagamos para ela fazer isso o dia inteiro. Parece trabalhoso e complicado, porém podemos até ter soluções desse gênero para fazer uma classificação, faz sentido.

Outra maneira é eu falar para um programa fazer isso para mim: "Programa, por favor, classifique esse meu e-mail, se ele é *spam* ou se ele não é *spam*". Mas um programa (um computador) sabe o que é um *spam*? Nós sabemos o que é *spam*, aquele e-mail indesejado que eu não pedi para receber e alguém me enviou. Esse é o significado da palavra *spam*. Porém, o computador não faz a mínima ideia do que significa essa palavra. Então, do que precisamos?

Precisamos, de alguma maneira, dizer para o computador o que é *spam* e o que não é *spam*. Mas ele nem conhece a palavra *spam*, então como podemos fazer isso? Como é mesmo a linguagem que o computador conhece? Ele entende apenas 0 ou 1, 0, 1, 2, 3, 4, 5... Binários, numéricos etc.

Se ele só entende esses números, então pedirei para o computador que ele me diga que, se for *spam*, é 1. Se não for *spam*, ele me diz 0, e pronto! Já resolvi o problema da palavra *spam*, ela já não existe mais para o computador, pois consigo falar para ele: "E aí, é 0 ou é 1?". Já é bem melhor, certo?

Já simplifiquei a situação para o computador me entender, pois se eu tentasse explicar para ele o que significa *spam*, ficaria maluco; mas se explicar o que é 0 ou 1, ele já sabe! Só preciso falar: "Olhe esse e-mail 'Facturas Impagas', ele é 1 ou é 0?". Isto é, eu preciso agora passar por cada um desses e-mails, olhar e falar: "Todas 'impagas' (*spam*) é 1"; eu preciso ensinar o computador, treiná-lo!

Como é que vocês sabem se esse e-mail é *spam* ou não? Porque você foi treinado. Você já viu um milhão de e-mails que eram *spam* e que não eram *spam*. Quando você bate o olho, você já sabe. Você já viu tanto e já fez essa classificação tantas vezes, que hoje faz isso facilmente. Então, a mesma coisa aqui:

			<a href="#">Se Compram Cheques</a>	<a href="#">o Facturas Impagadas. - COMPRAMOS CHEQUES PROTESTADOS Y FACTURAS IMPAGADAS CONTACTAR SOLO INTERI</a>
			<a href="#">Hotels-Paris.fr</a>	<a href="#">The Best Design Hotels in Paris with the Best Rates... Up to 65% off ! - This newsletter is in HTML format, if you cannot</a>
			<a href="#">Ativo.com</a>	<a href="#">corrida! Aproveite! Assine 1 Ano de Revista VO2 e escolha seu Óculos HB! - Caso não esteja visualizando este e-mail,</a>
			<a href="#">Charming Events</a>	<a href="#">Dine with Gary Rhodes this Christmas - Fine Dining this Christmas at the Gary Rhodes W1 Restaurant Friday 7th, 14th ar</a>
			<a href="#">Casas Bahia - BMC</a>	<a href="#">TV LED em Dobro! TV 32" + TV 22" por apenas 1.699 em 12x sem juros. Aproveite! - Caso não esteja visualizando as :</a>
			<a href="#">Cordell Miller</a>	<a href="#">Les réductions d'été sur les meilleures montres les Meilleures marques de 99.99 - Cada fabricador conocido del reloj d</a>
			<a href="#">Lepetitjournal.com</a>	<a href="#">Lepetitjournal de São Paulo - Edition du 03.08.2012 - Des problèmes pour lire la newsletter? Regardez la sur votre navig</a>
			<a href="#">eFacil</a>	<a href="#">Super presente para o Papai! Até 40% de desconto em TVs, Home Theaters e Mini Systems. - Caso não esteja visuali</a>

Figura 1.2: Meus e-mails

- 1º e-mail é spam ou não é spam? É **spam!**
- 2º e-mail, hotel em Paris? Realmente, eu estava procurando hotel lá, então **não é spam**.
- 3º e-mail, assinatura da revista **não é spam**, fui eu que realmente assinei a revista de corrida.
- 4º e-mail, jantar com Gary Rhodes... Quem é Gary Rhodes? Eu não tenho a menor ideia, **é spam!** Então 1.
- 5º e-mail, TV LED em Dobro! Não, não assinei Casas Bahia, então 1, **é spam.**
- 6º e 7º e-mails, são 2 e-mails que eu assino então 0, **não é spam.**
- 8º e-mail, Superpresente para o Papai! **É spam.**

Enfim, existe um monte de e-mail aqui que é *spam* e um monte que não é. Batendo o olho, já sei dizer, e eu consigo treinar o computador, treinar um programa, para dizer: se é *spam*, 1; se não é *spam*, 0. Mas como que eu o treino?

Eu dou um conjunto de e-mails que não é *spam* e um conjunto que é, e falo para ele: "Está aí, aprenda!". Aí ele aprende da mesma forma como aprendemos. E o que vai acontecer? Quando chegar um e-mail novo, o que ele faz? Ele vai olhar esse e-mail e falar: "É *spam*". Vai olhar outro e-mail e falar: "Não é *spam*". E é assim que funciona um programa de classificação, ele classifica entre 0 ou 1. Ele responde qualquer pergunta do gênero que vai classificar alguma coisa entre 0 e 1. Mas o que quer dizer 0? O que quer dizer 1? No meu caso, quer dizer se é *spam* ou não. Poderia querer dizer outra coisa? Com certeza!

Podemos usar a classificação entre 0 e 1 para diversas coisas e, como falei, já aqui veremos a classificação de 0 e 1, e também como poderemos utilizar isso em diversas situações. Primeiro, foi esse caso do e-mail: reduzimos o problema para classificar entre 1 ou 0, e ensinamos essa classificação ao programa.

Classificamos elementos entre *spam* e não *spam*, entre 1 ou 0, independente do que signifique 1 ou 0. Vamos ver outros exemplos do dia a dia em que usamos a classificação? Vou dar um exemplo do nosso cotidiano para que possamos entender e visualizar como nós, seres humanos, fazemos a classificação.

Como eu implemento um algoritmo se eu não entendo nem como eu faço uma coisa? "Quero implementar um algoritmo de soma". Ok, como fazemos a soma? "Quero implementar um algoritmo de multiplicação". Certo, como eu faço a multiplicação?

Agora, quero implementar um algoritmo que diz se é *spam* ou não é *spam*, ou se um cliente vai comprar ou não, se um funcionário vai se demitir ou não, então preciso saber como eu faço isso! Isto é, como eu, Guilherme, classifico isso.

Vamos pegar um exemplo do cotidiano: eu estou andando na rua, olho para alguma coisa, vejo e classifico-a, e então implemento um algoritmo similar a essa classificação. Na prática, é isso mesmo que fazemos. Vamos lá?

Queremos classificar entre 1 ou 0. No meu dia a dia, quando eu era criança, ia à fazenda do meu tio no interior de Minas, que tinha diversos animais, e às vezes eu encontrava um animal como este:



Figura 1.3: Qual animal é esse?

Qual animal é esse? Bom, você acabou de classificar; eu classifico esse animal como um porco. Por quê? Olha a perna dele, é perna de porco, uma perninha curtinha. O que mais? Ele é meio gordinho, porquinho é gordinho. Ele faz "oinc, oinc", então, para mim, ele é um porco.

Por essas 3 características, classifico-o como um porco, ou seja, dou uma olhada nelas, vejo se o animal se encaixa, e então concluo: "Sim, ele é um porco!". Mas também em uma fazenda, como é de costume, existem diversos outros animais, e às vezes eu encontrava um animal como este:



Figura 1.4: Qual animal é esse?

No momento em que o via, será que eu me perguntava e falava: "Nossa, que porquinho bonito e fofinho!"? Não! Eu dizia: "Nossa, que cachorrinho bonitinho". Por quê? Pois na minha cabeça tinha ocorrido o processo de classificação. Quando eu olhava o cachorrinho, eu pensava:

- Será que ele tem perninha curtinha que nem o porquinho?  
**Não.**
- Será que ele é gordinho que nem o porquinho? **Alguns sim e outros não.**
- Ele faz *oinc* que nem o porquinho? **Eles não faziam *oinc*.**
- Então, ele não é um porquinho; ele é um cachorrinho.

De acordo com essas três características que citei, eu classifico entre porquinho e cachorrinho. É como eu faço, e é óbvio que cada um faz de uma maneira um pouco diferente, e usam algumas características um pouco diferentes umas das outras. Você tem classificações diferentes, e isso é natural. Porém, vou seguir essa minha classificação como exemplo, ou seja, a que eu faço na minha cabeça, como ser humano, para ensinar o computador a fazer essa mesma classificação. Vamos lá!

Eu tenho aqui fotos de vários animais:

animal	perna curta?	gordinho?	faz oinc?	o que é?
				
				
				
				
				
				

Eu quero saber se eles têm perna curta ou não, se eles são gordinhos ou não, se fazem *oinc* ou não. Também quero dizer se eles são porquinhos ou cachorrinhos. Usarei 1 para dizer se é um porquinho, e 0 para dizer se é cachorrinho. Vamos preencher essa tabela?

Vejamos esse primeiro animal:



Figura 1.6: Primeiro animal

Ele tem perna curta? Sim, então 1. Ele é gordinho e faz *oinc*, então ele é um porquinho. Assim, essa linha fica `[1, 1, 1, 1]`. Vejamos o próximo animal:



Figura 1.7: Segundo animal

Esse outro porquinho também tem perna curta, mas ele não é gordinho, ele é um porquinho mais magrinho. Logo, coloco 0. Ele fazia *oinc*? Sim, ele fazia *oinc* e ele era um porquinho, então 1. Nossa linha fica  $[1, 0, 1, 1]$ . Vamos verificar o próximo:



Figura 1.8: Terceiro animal

Esse outro porquinho tem a perna curta, e ele é gordinho. Só que tem um porém: ele nasceu com um negócio na garganta, por isso ele não faz *oinc* e nem barulho, então vou marcar com 0. Mas, quando eu olhava para ele, eu falava: "você é um porquinho". Marcaremos, portanto, como porquinho. A linha fica  $[1, 1, 0, 1]$ . Vamos dar uma olhada em como está a nossa tabela atualmente:

animal	perna curta?	gordinho?	faz oinc?	o que é?
	1	1	1	1
	1	0	1	1
	1	1	0	1

Para cada elemento da tabela, estou pegando as características, ou seja, se ele tem perna curta, se é gordinho ou se faz *oinc*, e dizendo apenas se sim ou não usando 1 e 0, e no final qual é o tipo de animal, se é porco (1) ou cachorro (0). Ainda faltam mais três animais para analisarmos, vamos lá!



Figura 1.10: Quarto animal

Esse aqui tem perna curta? Esse cachorro não tem perna curta, então 0. Ele também não é gordinho e não faz *oinc*. Adivinha? Um cachorrinho, sim, porque eu olhava para ele e pensava isso. Então, a nossa linha fica  $[0, 0, 0, 0]$ . Agora o próximo animal:



Figura 1.11: Quinto animal

Já esse outro era divertido, pois era um *Yorkshire* pequenino e tinha a perna curta, só que também era gordinho. O mais engraçado é que, quando ele ia fazer *auau* (ou seja, latir), ele fazia *oinc oinc!* Então, eu vou marcar que ele fazia *oinc oinc*.

Só que repare que ele era um cachorro. Apesar de ele ter as três características parecidas com um porco, eu olhava para ele e falava: "Opa! Esse aqui é um cachorro". Então, a linha ficará com esses valores  $[1, 1, 1, 0]$ . Vamos para o último:



Figura 1.12: Sexto animal

Esse aqui não tem perna curta, também não é gordinho e não faz *oinc oinc*. Ele é um cachorrinho mesmo. Então marcamos assim:  $[0, 0, 0, 0]$ .

O que eu estou fazendo? Estou pegando todos os dados dos meus elementos, isto é, desses meus animais, e montando uma tabela. Veja como ela ficou:

animal	perna curta?	gordinho?	faz oinc?	o que é?
	1	1	1	1
	1	0	1	1
	1	1	0	1
	0	0	0	0
	1	1	1	0
	0	0	0	0

Essa tabela define as características dos animais por meio dessas três primeiras colunas (perna curta, gordinho, faz *oinc*) e essa última coluna (o que é) classifica o animal. Estou usando as características para classificar um animal entre dois tipos.

Eu poderia usar essas características ou outras para classificar um e-mail entre *spam* e não *spam*, por exemplo:

- Tamanho do e-mail;
- As palavras que aparecem;
- O horário em que foi enviado;
- Se eu conheço ou não quem enviou;
- Se é a primeira vez que esse e-mail vem.

No meu dia a dia, lá na fazenda do meu tio, eu usava essas três características para classificar os animais, porém, eu poderia usar outras, como:

- "Ele é rosa?"
- "Ele é preto?"

- "Ele é branco?"
- "Ele é azul?"
- "Ele é x?"

A cor influencia para eu dizer se ele é um animal ou outro, pois existem animais de uma cor e animais de outra. Existem diversas características que podemos usar para classificar um animal. Logo, repare que todos os problemas de classificação vão fazer isso: dado um conjunto de características e um monte de dados, eu posso treinar meu algoritmo, meu programa.

Essa tabela de características é o meu treino, então eu falo para o meu programa: "Esta tabela eu já sei, e pode ficar com você". E também mando a classificação, assim o programa ficará todo feliz que foi treinado. Agora, o que acontece? Eu pergunto para ele, assim como eu posso perguntar para mim mesmo: "E esse aqui? Quem é ele?".

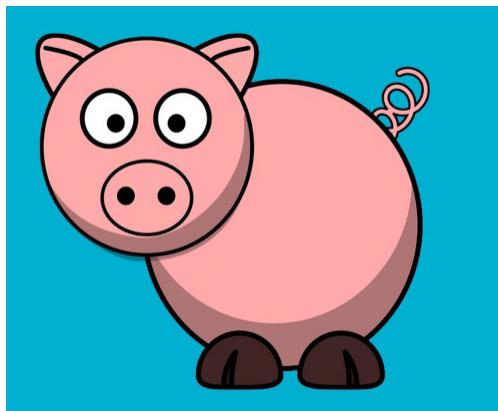


Figura 1.14: "E esse aqui? Quem é ele?"

Ele é um cachorro (0) ou ele é um porco (1)? E aí, ou eu, ou o

programa toma uma decisão baseada em toda a experiência anterior que tivemos e em toda experiência anterior que o programa teve. Com todo o treino que já foi feito, ele vai olhar as características desse animal e vai tomar a decisão se ele é um porco ou um cachorro.

Isto é, esse animal especificamente tem perna curta? Sim (1). Esse animal é gordinho? Sim (1). Ele faz *oinc*? Ele não faz *oinc* (0), já que isso aqui é uma imagem, e não um animal de verdade.

Então, a pergunta se resume a quê? Dadas essas características, por favor, Guilherme e programa, me digam se o resultado é 0 ou 1. Isso significa que eu treinei você com diversas características e classificações e, se eu der uma linha de característica, quero que você me diga se o animal é 0 ou 1, ou seja, quero que você preveja e classifique para mim.

E é assim que vão funcionar os programas de classificação: nós treinamos por meio de características e classificações. Após treinar, pedimos para ele classificar coisas que nós desconhecemos, como, por exemplo, chegou um e-mail novo: "E aí, esse e-mail é um *spam* ou não é?", ou chegou um animal novo: "E aí, eu vou vender esse animal como porco ou como cachorro?".

Resumindo, os algoritmos serão baseados em características e as usaremos entre 0 e 1. É claro que veremos outros tipos depois, mas por enquanto, para classificar se ele é ou não um animal que esperamos, é exatamente isso que faremos com os algoritmos.

Vamos para o código. O que queremos aqui é implementar um sistema de classificação que consiga classificar alguma coisa entre 0 e 1, -1 e 1, 1 e 2, A e B — entre duas categorias diferentes. Eu tenho

duas categorias (porco e cachorro), e quero classificá-las. Para isso, usaremos *Python* e uma biblioteca a ser instalada com o *pip*, o instalador simples de pacotes.

O *pip3* já vem instalado com o Python 3, portanto, podemos instalar as bibliotecas do Python de nosso interesse para trabalhar com as informações científicas que queremos, como o machine learning e a classificação. A biblioteca que usaremos é o *scikit-learn*. Vamos pedir para o *pip* instalá-la para nós. Aproveitaremos para instalar também duas dependências que o *scikit* usará internamente:

```
> pip3 install scikit-learn numpy scipy
```

Caso você esteja usando Linux, será necessário utilizar o *sudo*.

Além da biblioteca do Python *scikit-learn*, nós temos acesso à documentação (<http://scikit-learn.org/stable/documentation.html>) para consultarmos tudo o que ela disponibiliza para nós.

Agora, o que queremos fazer, a princípio, é a classificação entre diversos porcos e cachorros, e depois faremos do nosso mundo web. Vamos abrir um editor de texto para escrevermos o nosso código Python. Utilize um de sua preferência.

Primeiro, vamos criar um arquivo chamado *classificao.py* e salvá-lo. Lembre-se de salvar o arquivo dentro de um diretório de fácil acesso ao terminal. Nesse arquivo, vamos definir todos os

porcos e cachorros que já conhecemos, por exemplo, nós queremos representar o nosso primeiro porquinho, então escrevemos:

```
porco1
```

E o que era esse porquinho? Ele era um *array* de 3 valores, você lembra? Quais são os valores? 0 ou 1, ou seja, se ele tem ou não uma característica. Mas, e quais são as três características que eu, Guilherme, decidi usar agora?

1. Se ele é gordinho.
2. Se ele tem perninha curta.
3. Se ele faz *auau*.

Então, vamos lá. Ele é gordinho? Sim:

```
porco1 = [1]
```

Ele tem perninha curta? Sim:

```
porco1 = [1, 1]
```

Esse porquinho faz *auau*? Não:

```
porco1 = [1, 1, 0]
```

Perceba que eu mudei, agora estou modelando o meu sistema de uma maneira diferente. Estamos dizendo que:

```
[1, 1, 0] -> [tem perna curta, é gordinho, não faz *auau*]
```

Agora eu vou colocar o meu segundo porco. Ele também é gordinho, tem perna curta e não faz *auau*:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
```

Por fim, vou representar o meu terceiro porquinho que também é gordinho, tem perna curta e também não faz *auau*:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
```

Cadastrei os três porquinhos, agora eu preciso cadastrar três cachorros que eu já conheço. O primeiro cachorro é gordinho, tem perninha curta e faz *auau*:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
cachorro1 = [1, 1, 1]
```

Os outros dois não são gordinhos, têm perninhas curtas e fazem *auau*:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
cachorro1 = [1, 1, 1]
cachorro2 = [0, 1, 1]
cachorro3 = [0, 1, 1]
```

Esses são os seis elementos que eu, Guilherme, conheço. Vamos comentar a que cada uma dessas posições se refere, você lembra?

```
# [é gordinho?, tem perninha curta?, faz auau?]
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
cachorro1 = [1, 1, 1]
cachorro2 = [0, 1, 1]
cachorro3 = [0, 1, 1]
```

Dentre esses seis elementos, já sabemos que, do primeiro ao terceiro, são porquinhos e que, do quarto até o sexto, são

cachorros. A grande sacada é dizer o que já sabemos que eles são, ou seja, dizer se eles são 1 ou 0; se são 1 ou -1; se é cachorro ou porco. Precisamos de todos esses elementos, e chamaremos todos eles de dados :

```
# [é gordinho?, tem perninha curta?, faz auau?]
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
cachorro1 = [1, 1, 1]
cachorro2 = [0, 1, 1]
cachorro3 = [0, 1, 1]

dados = [porco1, porco2, porco3,
         cachorro1, cachorro2, cachorro3]
```

Agora que temos os nossos dados agrupados em um *array*, precisamos fazer alguma marcação para indicar o que cada um desses elementos é. Usaremos uma variável chamada `marcacoes` para indicar o que cada um representa:

```
marcacoes
```

Sabemos que, do primeiro ao terceiro, são porcos, logo marcarei com 1.

```
marcacoes = [1, 1, 1]
```

E para marcar como cachorro, eu vou querer marcar com um outro número. Eu posso marcar com 0, -1, 10.000, com o que eu quiser. Para dar ênfase na distinção dos 2 elementos, ou seja, dizer que um é oposto do outro, eu usarei 1 positivo (1) para porco e 1 negativo (-1) para cachorro:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
cachorro4 = [1, 1, 1]
cachorro5 = [0, 1, 1]
```

```
cachorro6 = [0, 1, 1]

dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]

marcacoes = [1, 1, 1, -1, -1, -1]
```

Marcamos todos os nossos elementos, porém, nós temos agora um elemento misterioso: um animal que é gordinho, tem perninha curta e faz *auau*.

```
misterioso = [1, 1, 1]
```

E agora? Esse colega é um porco ou um cachorro? Esse é o nosso elemento misterioso. O que nós queremos saber é: "Será que ele é um cachorro ou um porco?". Então, queremos treinar um algoritmo de classificação baseado nesses dados para que ele me diga se é um cachorro ou é um porco. Vamos lá?

No Python, a mais famosa biblioteca que faz esse trabalho é o `sklearn`. Claro que existem outras e outras linguagens, mas o foco neste livro será o Python e o `sklearn`.

Essa biblioteca faz o treinamento baseado no algoritmo que se chama *bayesiano*, chamado `naive_bayes`. A partir dela, vamos importar o algoritmo `MultinomialNB`, NB de *naive bayes*:

```
from sklearn.naive_bayes import MultinomialNB
```

O `Multinomial` é o algoritmo que usaremos para treinar o nosso modelo que diz se os nossos elementos são cachorros ou porcos. Para treinar esse modelo, nós precisamos dos nossos dados e nossas marcações. E como fazemos para criar um modelo? Basta apenas fazer uma chamada para o `MultinomialNB`:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
```

```
cachorro4 = [1, 1, 1]
cachorro5 = [0, 1, 1]
cachorro6 = [0, 1, 1]

dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]

marcacoes = [1, 1, 1, -1, -1, -1]

misterioso = [1, 1, 1]

from sklearn.naive_bayes import MultinomialNB

modelo = MultinomialNB()
```

Criamos um modelo! Vamos tentar rodar o nosso código? Salve o arquivo, abra o terminal, vá até o diretório onde salvou o arquivo e execute:

```
> python3 classificacao.py
>
```

Não aconteceu nada? Claro que não! Apenas criamos o modelo, nem o mandamos treinar. Como nós, seres humanos, treinamos? Olhamos para um elemento e classificamos como porquinho, olhamos para outro elemento e classificamos como cachorrinho; isto é, baseado nos dados e nas marcações, nós treinamos o modelo do nosso cérebro, e precisamos fazer a mesma coisa para o nosso modelo:

```
modelo = MultinomialNB()
modelo.treinar(dados, marcacoes)
```

Porém, não existe o método `treinar`, então como faremos isso? Observe que o nosso modelo precisa se adequar aos nossos dados e `marcacoes`, ou seja, os dados do `porco1` ao `porco3` são 1; os dados do `cachorro4` ao `cachorro6` são -1. É dessa maneira que precisamos **adequar** o nosso modelo, e para isso usaremos o método *adequar* — no inglês, o método `fit`:

```
modelo.fit(dados, marcacoes)
```

Nesse momento do código, estamos dizendo: "Por favor, adeque-se a essas informações". Vamos testar?

```
> python3 classificacao.py  
>
```

Não aconteceu nada, já que nós apenas treinamos! E o que precisamos é pedir para que o modelo **preveja** quem é o elemento misterioso, utilizando o método `predict`:

```
modelo.predict(misterioso)
```

Esse método vai devolver se é um cachorro ou um porco, então precisamos imprimir o valor com o método `print`:

```
# [é gordinho?, tem perninha curta?, faz auau?]  
porco1 = [1, 1, 0]  
porco2 = [1, 1, 0]  
porco3 = [1, 1, 0]  
cachorro4 = [1, 1, 1]  
cachorro5 = [0, 1, 1]  
cachorro6 = [0, 1, 1]  
  
dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]  
  
marcacoes = [1, 1, 1, -1, -1, -1]  
  
misterioso = [1, 1, 1]  
  
from sklearn.naive_bayes import MultinomialNB  
  
modelo = MultinomialNB()  
modelo.fit(dados, marcacoes)  
print(modelo.predict(misterioso))
```

Testando novamente o nosso algoritmo:

```
> python3 classificacao.py  
> /usr/local/lib/python3.6/dist-packages/sklearn/utils/validation  
.py:386: DeprecationWarning: Passing 1d arrays as data is depreca
```

```
ted in 0.17 and willraise ValueError in 0.19. Reshape your data either using X.reshape(-1, 1) if your data has a single feature or X.reshape(1, -1) if it contains a single sample.  
DeprecationWarning)  
[-1]
```

Ele devolveu um *warning*, porém ele imprimiu o resultado que foi um *array* e, dentro desse *array*, devolveu -1. Logo mais, veremos como resolver esse *warning*. Mas e o que significa o -1 para ele? Significa um cachorro! Como será que ele deduziu que era um cachorro? Se nós olharmos a nossa experiência passada:

```
# [é gordinho?, tem perninha curta?, faz auau?]  
porco1 = [1, 1, 0]  
porco2 = [1, 1, 0]  
porco3 = [1, 1, 0]  
cachorro4 = [1, 1, 1]  
cachorro5 = [0, 1, 1]  
cachorro6 = [0, 1, 1]  
  
dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]  
  
marcacoes = [1, 1, 1, -1, -1, -1]  
  
# restante do código
```

Todos que faziam *auau* eram cachorros. Com base nesses dados, fazendo uma análise pelo nosso cérebro, provavelmente faríamos a mesma classificação. Não sabemos o que aconteceu por trás dos panos, só sabemos que, quando rodamos, ele nos informou que o elemento misterioso é um cachorro.

Repare que nós passamos vários dados para o nosso programa, o treinamos e pedimos para ele verificar um único caso, que foi o elemento misterioso. Mas será que nós queremos prever apenas um caso? Não, queremos prever muitos outros! Então precisamos de outros elementos misteriosos para que o programa classifique para nós. Por exemplo, temos o `misterioso2` que é gordinho,

não tem perninha curta e não faz *auau*:

```
misterioso1 = [1, 1, 1]
misterioso2 = [1, 0, 0]
```

Agora, em vez de prever o `misterioso1`, pediremos para que ele preveja o `misterioso2`:

```
print(modelo.predict(misterioso2))
```

Se rodarmos novamente:

```
> python3 classificacao.py
> /usr/local/lib/python3.6/dist-packages/sklearn/utils/validation
  .py:386: DeprecationWarning: Passing 1d arrays as data is deprecated in 0.17 and will raise ValueError in 0.19. Reshape your data either using X.reshape(-1, 1) if your data has a single feature or X.reshape(1, -1) if it contains a single sample.
  DeprecationWarning)
[1]
```

Na interpretação dele, a maior chance que ele dá é de que esse `misterioso2` seja um porquinho. Se observarmos o `misterioso2`, vemos que ele é gordinho, não tem perna longa e não faz *auau*, logo também chutaria que é um porquinho! Porém nós não prevemos todos de uma vez só! Para isso, precisamos adicionar todos os misteriosos dentro de um array e atribuir a uma variável que vamos chamar de `teste`:

```
misterioso1 = [1, 1, 1]
misterioso2 = [1, 0, 0]

teste = [misterioso1, misterioso2]
```

E então, passamos apenas o `teste` no `predict`:

```
print(modelo.predict(teste))
```

O parâmetro do método `predict` recebe um array com vários

elementos que queremos testar, e é por isso que ele deu aquele warning daquela vez. Antigamente, ele recebia apenas um valor, porém essa solução foi *depreciada*, e logo não haverá mais suporte para ela. No nosso caso, ele deu um *warning*, mas em versões futuras, ele nem funcionará!

Mesmo que seja um *array* de uma única posição, ainda precisaremos passar um array. Vamos testar novamente?

```
> python3 classificacao.py  
> [-1 1]
```

Agora ele imprime dois resultados e diz que o primeiro elemento misterioso é um cachorro e o segundo é um porco. Se verificarmos nossos dois elementos misteriosos:

```
misterioso1 = [1, 1, 1]  
misterioso2 = [1, 0, 0]
```

Realmente, o `misterioso1` é um cachorro e o `misterioso2` é um porco. Então, como funciona o algoritmo Bayesiano Multinomial para classificação?

- Treinamos o nosso modelo pedindo para se adequar aos dados e marcações utilizando o método `fit`.
- Pedimos para prever o que eles são por meio do método `predict`.

Assim, ele vai prevendo se esses elementos são porcos ou cachorros. Mas será que esse código que criamos funciona apenas para porco ou cachorro? E se, em vez de cachorro ou porco, esses elementos fossem *spam* ou não *spam*?

E se fosse um cliente que não paga as dívidas e um cliente que paga as dívidas em dia? E se esses 0 e 1 classificassem qualquer

outra coisa? Eu mudaria alguma coisa no código? O que você acha?

É justamente esse exemplo que eu queria demonstrar para você: como podemos implementar a mesma coisa que vimos com tabelas e números, e traduzir isso para código. É extremamente simples. Quais são os passos mesmo?

- Para cada um dos elementos, definimos as variáveis, por exemplo, `porco1 = [1, 1, 0]`, `porco2 = [1, 1, 0]` e assim por diante.
- Cada variável quer dizer alguma coisa, no nosso exemplo foi: `[é gordinho?, tem perninha curta?, faz auau?]`.
- Marcamos esses elementos para diferenciarmos um do outro, e no nosso exemplo utilizamos 1 para porco e -1 para cachorro: `marcacoes = [1, 1, 1, -1, -1, -1]`.
- Então treinamos o nosso modelo: `modelo.fit(dados, marcacoes)`.
- E, por fim, pedimos para ele imprimir os nossos testes: `print(modelo.predict(teste))`.

Repare que as marcações foram feitas de uma forma diferente da qual vimos na tabela: em vez de usar 0 e 1, usei -1 e 1. Optei por essa escolha, pois essa forma dá uma ênfase melhor de distinção. Poderíamos testar com 0 e 1 sem problema algum:

```
marcacoes = [1, 1, 1, 0, 0, 0]
```

Se tentarmos rodar novamente:

```
> python3 classificacao.py  
> [0 1]
```

Mas vamos deixar -1 e 1, justamente para dar essa ênfase de

um ser o oposto do outro no momento em que eu tenho duas categorias!

Vimos um algoritmo que, dado vários números para ele, devolve um único número. Parece uma coisa de nerd, porém, perceba que é genial poder dizer se um animal é um ou outro, se um e-mail é *spam* ou não, se uma pessoa vai se comportar de uma maneira ou de outra, tudo de acordo com o que aconteceu no passado. Ou seja, de acordo com experiências anteriores, nós podemos classificar grupos para podermos tomar decisões.

Isso não significa que tomaremos decisões negativas ou ruins, pois podemos tomar decisões positivas, por exemplo, poder dizer se um vídeo que está subindo para o YouTube é um vídeo legal — isto é, sem violência ou sem qualquer conteúdo que não pode ser exibido. Provavelmente, eu não terei pessoas assistindo todos os vídeos existentes no YouTube, eu preciso de fato de um programa que consiga verificar se o vídeo contém um conteúdo permitido ou não, para depois ter um filtro por um ser humano. Isso significa que o primeiro filtro precisa ser por uma máquina, para que ela consiga verificar todas as regras que precisam ser aplicadas para um vídeo novo que será publicado.

Um outro exemplo é uma música que está sendo colocada em uma plataforma. Preciso verificar se ela é pirata ou não. Nós precisamos verificar se o usuário está praticando pirataria, e se ele tem os direitos para baixar o conteúdo ou não. Tudo isso significa que precisamos classificar baseado nas características que temos e, para cada caso, usaremos características diferentes.

Mas também existe uma coisa muito importante que é comum em todos esses algoritmos que utilizamos, como também é comum

para nós: **nem sempre temos certeza da nossa classificação.** Vejamos um exemplo. Vamos supor que eu peguei todos os vídeos do YouTube, e agora vou tentar classificar todos eles verificando se aparece um gatinho nele ou não.

Esse algoritmo classifica todos os vídeos que contêm um gatinho. E aí, precisamos parar e pensar que todo algoritmo contém uma margem de erro, logo, é necessário nos atentarmos a toda margem de erro que o nosso algoritmo pode apresentar.

Por exemplo, ele pode classificar alguma coisa como gatinho, porém não era um gatinho; ou então, classificar um gatinho como não gatinho. Existem diversos tipos de erros que podem acontecer em um algoritmo de classificação, e precisamos ficar atentos aos erros para que possamos controlá-los. É necessário sabermos a nossa taxa de erro, e compreender qual é a que vamos aceitar no nosso algoritmo.

Nós vamos nos atentar a essa margem de erro e aprender como calcular para poder dizer se esse erro está sendo bom, ou se nós estamos melhorando a nossa margem atual. Podemos concluir que todos os nossos algoritmos apresentarão erros, porém, o nosso papel é, a partir deles, verificar se são aceitáveis ou não.

Por exemplo, é aceitável que a minha foto seja classificada como gatinho? Não existe uma regra fixa para sabermos se isso é aceitável ou não, pois, dependendo do ponto de vista, pode ser que sim ou pode ser que não. Ou seja, depende muito do contexto para podermos dizer se o nosso erro é aceitável ou não; se eu preciso de uma margem de erro menor, ou se está dentro do padrão que esperamos.

Vimos lá no começo do capítulo que tínhamos três porcos e três cachorros, cada um com suas variáveis, ou seja, suas características que os definiam. Por fim, as representamos como nossos elementos:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
cachorro4 = [1, 1, 1]
cachorro5 = [0, 1, 1]
cachorro6 = [0, 1, 1]
```

E nós tínhamos agrupados todos esses elementos em um *array* para representar os nossos dados:

```
dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]
```

E também fizemos as marcações para indicar quais desses elementos eram porcos ou cachorros, marcando com 1 para porco e -1 para cachorro:

```
marcacoes = [1, 1, 1, -1, -1, -1]
```

Então, treinamos o nosso algoritmo:

```
modelo = MultinomialNB()
modelo.fit(dados, marcacoes)
```

E por fim, testamos dois elementos misteriosos:

```
misterioso1 = [1, 1, 1]
misterioso2 = [1, 0, 0]

teste = [misterioso1, misterioso2]

print(modelo.predict(teste))
```

Mas existe um detalhe no processo que fizemos, pois não adianta rodarmos o nosso algoritmo e não fazermos ideia de quão

bom ele é. Quando treinamos o nosso cérebro, chegamos a ver 300, ou mais, porcos e cachorros na nossa vida, porém nós não temos certeza de que estamos bons o suficiente para distinguir entre um cachorro e um porco, ou então *spam* e não \*spam, ou pessoas que vão me pagar e que não vão me pagar.

E como podemos ter certeza se estamos bons ou não para fazer essas distinções? Precisamos testar! Sim, testar no mundo real, com algum valor que desconhecemos, e ver se vamos funcionar conforme o esperado para verificar quão bons estamos sendo com esse algoritmo. E como podemos testar? Podemos criar um cenário de teste simulando os nossos elementos reais (porcos ou cachorros) por elementos misteriosos:

```
# [é gordinho?, tem perninha curta?, faz auau?]
misterioso1 = [1, 1, 1]
misterioso2 = [1, 0, 0]
misterioso3 = [0, 0, 1]

teste = [misterioso1, misterioso2, misterioso3]
```

Agora que temos o nosso cenário de teste, podemos informar os resultados esperados para cada um desses elementos misteriosos, ou seja, as marcações de teste:

```
marcacoes_teste
```

Agora, precisamos classificar cada elemento misterioso e indicar nas nossas marcações de teste. Sabemos que o `misterioso1` é um cachorro (-1), o `misterioso2` é um porco (1) e o `misterioso3` é um cachorro (-1):

```
marcacoes_teste = [-1, 1, -1]
```

Por fim, para ficar mais claro, vamos atribuir o retorno do método `predict()`, que é o nosso resultado, para uma variável

chamada resultado :

```
resultado = modelo.predict(teste)
print(resultado)
```

Repare no nosso código:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
cachorro4 = [1, 1, 1]
cachorro5 = [0, 1, 1]
cachorro6 = [0, 1, 1]

dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]

marcacoes = [1, 1, 1, -1, -1, -1]

from sklearn.naive_bayes import MultinomialNB

modelo = MultinomialNB()
modelo.fit(dados, marcacoes)

misterioso1 = [1, 1, 1]
misterioso2 = [1, 0, 0]
misterioso3 = [0, 0, 1]

teste = [misterioso1, misterioso2, misterioso3]

marcacoes_teste = [-1, 1, -1]

resultado = modelo.predict(teste)

print(resultado)
```

Será que agora ele vai prever conforme as nossas marcações de teste? Ou seja, falar que os elementos misteriosos são: cachorro, porco e cachorro? Vamos testar.

```
> python3 classificacao.py
> [-1 1 -1]
```

Será que o nosso código se saiu bem? Vamos verificar os

valores que estávamos esperando nas nossas marcações de teste:

```
marcacoes_teste = [-1, 1, -1]
```

De acordo com as nossas marcações, o nosso código funcionou muito bem e conseguiu prever! Mas é importante lembrar que, no mundo real (na prática), esse resultado não costuma ser 100% e o próximo exemplo que veremos logo mais, de compras na web, não vai conseguir acertar sempre.

Porém, observe que estamos testando isso manualmente; estamos verificando se o teste passou a olho nu. Para o nosso caso atual, não tem problema; mas no mundo real, realizaremos testes gigantes, como, por exemplo, com 1.000 elementos. E não faz sentido nós ficarmos olhando um a um para verificar o quanto ele acertou ou o quanto errou.

O que precisamos saber é: quantos elementos da variável `resultado` (resultados que o algoritmo classificou) são diferentes da variável `marcacoes_teste` (resultado que esperamos dos elementos). Mas como podemos fazer para verificar os valores que foram diferentes? Podemos subtrair os dois arrays:

```
print(resultado - marcacoes_teste)
```

Parece estranho, mas essa abordagem funciona da seguinte maneira:

- Resultado e marcação 1; resultará em:  $1 - 1 = 0$ .
- Resultado 1; marcação -1; resultará em:  $1 - (-1) \rightarrow 1 + 1 = 2$ .
- Resultado -1; marcação -1; resultará em:  $-1 + 1 = 0$ .
- Resultado -1; marcação 1; resultará em:  $-1 - 1 = -2$ .

Resumindo, para todas as possibilidades que fizemos, nos casos

em que os valores forem iguais, o resultado será 0. Ou seja, quando resultar em 0, saberemos que o algoritmo acertou. Vamos testar o nosso código?

```
> python3 classificacao.py  
> [0 0 0]
```

O resultado foi tudo 0, então acertou 100% novamente! Mas o ideal seria que o nosso algoritmo mostrasse o percentual em vez de só a diferença entre o `resultado` e a `marcacoes_teste`. Então vamos fazer com que a diferença dessas duas variáveis seja retornada para uma variável chamada `diferencias`:

```
diferencias = resultado - marcacoes_teste
```

Mas como faremos para verificar os acertos? Sabemos que qualquer valor igual a 0 é um acerto, isto é, qualquer valor da variável `diferencias` que seja 0. Isso significa que, para cada diferença dentro da variável `diferencias`:

```
for d in diferencias
```

Se o valor for igual a 0:

```
for d in diferencias if d == 0
```

Ele retornará:

```
acertos = for d in diferencias if d == 0
```

Porém, nós precisamos passar isso para um array, então fazemos:

```
acertos = [d for d in diferencias if d == 0]
```

Se imprimirmos os acertos:

```
acertos = [d for d in diferencias if d == 0]
```

```
print(acertos)
```

O resultado será:

```
> python3 classificacao.py  
> [0, 0, 0]
```

Observe que foi impresso um array com todos os 0 contidos no array `diferencias`, todos os valores. Porém, não queremos os valores, precisamos saber a quantidade de elementos que existem nesse array. Como fazemos para retornar a quantidade de elementos existente de um array? Utilizamos a função `len()`, que retorna o tamanho do array:

```
len(acertos)
```

E esse será o nosso total de acertos, então vamos representar por uma variável chamada `total_de_acertos`:

```
total_de_acertos = len(acertos)
```

Agora vamos imprimir o `total_de_acertos`:

```
total_de_acertos = len(acertos)  
print(total_de_acertos)
```

Se rodarmos novamente:

```
> python3 classificacao.py  
> [0, 0, 0]  
> 3
```

Porém, para verificarmos a porcentagem, nós precisamos também do total de elementos que foram testados. Podemos extrair esse valor a partir do nosso array `teste`, que contém todos os elementos que foram testados:

```
teste = [misterioso1, misterioso2, misterioso3]
```

```
marcacoes_teste = [-1, 1, -1]

resultado = modelo.predict(teste)

diferencias = resultado - marcacoes_teste

print(diferencias)

acertos = [d for d in diferencias if d == 0]

total_de_acertos = len(acertos)

print(total_de_acertos)

total_de_elementos = len(teste)

print(total_de_elementos)
```

Testando o nosso código:

```
> [0 0 0]
> 3
> 3
```

Agora que temos tanto o total de acertos quanto o total de elementos, precisamos apenas fazer a divisão de `total_de_acertos` por `total_de_elementos` para extraímos a nossa taxa de acerto:

```
total_de_acerto / total_de_elementos
```

Então, atribuímos para uma variável chamada `taxa_de_acerto` e a imprimimos:

```
taxa_de_acerto = total_de_acerto / total_de_elementos
print(taxa_de_acerto)
```

Verificando o resultado:

```
> python3 classificacao.py
> [0 0 0]
> 3
```

```
> 3  
> 1
```

Vamos melhorar a nossa impressão e retirar todas as impressões. Primeiro, imprimimos o nosso resultado, `print(resultado)` ; depois a diferença entre o resultado e as marcações de teste, `print(diferenca)` ; e por fim, faremos a impressão da taxa de acerto, `print(taxa_de_acerto)` .

Entretanto, multiplicaremos por 100.0: `taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos` , para que seja apresentado como percentual. Vejamos o resultado do nosso código final:

```
porco1 = [1, 1, 0]  
porco2 = [1, 1, 0]  
porco3 = [1, 1, 0]  
cachorro4 = [1, 1, 1]  
cachorro5 = [0, 1, 1]  
cachorro6 = [0, 1, 1]  
  
dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]  
  
marcacoes = [1, 1, 1, -1, -1, -1]  
  
from sklearn.naive_bayes import MultinomialNB  
  
modelo = MultinomialNB()  
modelo.fit(dados, marcacoes)  
  
misterioso1 = [1, 1, 1]  
misterioso2 = [1, 0, 0]  
misterioso3 = [0, 0, 1]  
  
teste = [misterioso1, misterioso2, misterioso3]  
  
marcacoes_teste = [-1, 1, -1]  
  
resultado = modelo.predict(teste)  
  
diferencias = resultado - marcacoes_teste
```

```
acertos = [d for d in diferencias if d == 0]

total_de_acertos = len(acertos)
total_de_elementos = len(teste)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print(resultado)
print(diferencias)
print(taxa_de_acerto)
```

Rodando o nosso algoritmo, obtemos o seguinte resultado:

```
> python3 classificacao.py
> [-1 1 -1]
> [0 0 0]
> 100.0
```

100.0! Porém, é sempre válido lembrar que 100.0 é um número muito difícil de acontecer no mundo real. Um exemplo bem simples que demonstra o nosso algoritmo errando seria modificar a nossa `marcacoes_teste`, informando que o último elemento misterioso era um porco que fazia *auau*, ou seja, um porquinho que faz *auau*:

```
misterioso1 = [1, 1, 1]
misterioso2 = [1, 0, 0]
misterioso3 = [0, 0, 1]

teste = [misterioso1, misterioso2, misterioso3]

marcacoes_teste = [-1, 1, 1]
```

Vamos verificar se o nosso algoritmo vai adivinhar que esse porquinho que faz *auau* é um porquinho? Vejamos o resultado:

```
> python3 classificacao.py
> [-1 1 -1]
> [ 0 0 -2]
> 66.6666666667
```

Como vimos, ele errou! Perceba que o último valor da diferença foi diferente de 0, ou seja, um erro! Por fim, a nossa taxa de acerto foi de 66%.

## 1.1 RESUMINDO

Repare que a taxa de acerto é fundamental para que o nosso algoritmo, no momento em que recebe novos dados, consiga verificar o quanto bom ele foi em uma situação do mundo real. Como por exemplo, nesse último teste, ele foi capaz de acertar 66%, ao ter de lidar com um novo animal que possui características inesperadas.

Vimos que, em Python, é muito fácil usar uma biblioteca para a qual fornecemos duas informações: as informações que conhecemos dos nossos itens e as categorizações que damos a eles. No nosso caso:

- Os itens eram animais;
- As variáveis que usamos foram três (perna curta, gordinho, faz *auau*);
- A categorização era binária (0 ou 1, cães e porcos).

Daqui a pouco, a nossa situação será diferente e mais próxima ao nosso dia a dia. Usaremos um exemplo real em uma aplicação web, na qual vamos pegar muito mais valores para realizarmos o nosso teste com diversas características e um melhor cálculo para a taxa de acerto. E nesse mesmo exemplo, veremos como a nossa taxa de acerto dificilmente acertará 100%.

## CAPÍTULO 2

# IMPORTANDO, CLASSIFICANDO E VALIDANDO UM MODELO

## 2.1 E NO MUNDO REAL, COMO CLASSIFICAR DADOS DA WEB?

Até agora fizemos uma classificação entre um porco e um cachorro. Porém, o que queremos no nosso dia a dia é realizar uma classificação do mundo web, por exemplo. Vamos tentar fazer essa análise para esse cenário? Veremos um exemplo do mundo web que pode acontecer.

Um dos exemplos seria um usuário que entra em um site. Nesse caso, usaremos o site da Alura (<https://www.alura.com.br/>):

alura

## Coloque seu potencial em prática!

Cursos online de tecnologia que reinventam sua carreira.

[MATRICULE-SE JÁ](#)

Curso de Mobile  
iOS, Android, PhoneGap, e mais...

Curso de Programação  
Java, C#, Ruby, PHP, Python e mais...

Curso de Front-end  
HTML, CSS, Angular, JavaScript e mais...

Curso de Infraestrutura  
Linux, SQL, Git, Docker e mais...

Curso de Design & UX  
Photoshop, Illustrator, Usabilidade e mais...

Curso de Business  
Agile, Data Science, Startups, Marketing, SEO e mais...

<https://www.alura.com.br/cursos-online-infraestrutura>

Então, ele entra em uma segunda página, como <https://www.alura.com.br/cursos-online-programacao>.

Matricule-se já com val... https://www.alura.com.br/planos-cursos-online

alura

MOBILE PROGRAMAÇÃO FRONT-END INFRAESTRUTURA DESIGN & UX BUSINESS

## É hora de decolar sua carreira

Escolha seu plano e comece a estudar

**Premium**

12x R\$ 75  
(à vista R\$ 900)

200 Cursos de todas as trilhas  
7.000 Exercícios  
✓ Estude por 1 ano  
✗ 3 ebooks da Casa do Código

**Premium Plus**

12x R\$ 100  
(à vista R\$ 1.200)

200 Cursos de todas as trilhas  
7.000 Exercícios  
✓ Estude por 1 ano  
✓ 3 ebooks da Casa do Código

Por fim, ele chega nessa terceira página (<https://www.alura.com.br/curso-online-logica-de-programacao>):

The screenshot shows a web browser window with the URL <https://www.alura.com.br/contato>. The page has a blue header bar with the text "Fale Conosco". Below the header, there's a section titled "Crítica, sugestão ou elogio? Envie-nos!" with the subtext "Queremos sempre melhorar nossos serviços.". There are two input fields: one for "Nome (obrigatório)" and one for "Email (obrigatório)". At the top of the page, there's a navigation bar with categories: MOBILE, PROGRAMAÇÃO, FRONT-END, INFRAESTRUTURA, DESIGN & UX, BUSINESS, LOGIN, and MATRICULE-SE.

Fale Conosco

Crítica, sugestão ou elogio? Envie-nos!

Queremos sempre melhorar nossos serviços.

Nome (obrigatório)

Email (obrigatório)

Visto apenas essas três páginas, surge a pergunta: será que o usuário vai ou não comprar? Percebeu que é uma pergunta de classificação? Ou seja, as perguntas de classificação estão em todos os lugares. Já citei alguns exemplos, como: tenho um funcionário que trabalhou X horas no ano passado, teve Y projetos, na avaliação entre pessoas ele ficou com a média Z, o chefe dele deu um aumento, e ele não tirou férias. E agora vem a pergunta: "E aí? Ele vai pedir para sair no próximo ano ou não?".

Tudo isso é um problema de classificação, e são problemas reais usados no mundo afora. Essa situação para detectar se um funcionário vai ou não continuar já foi trabalhada com machine learning anteriormente (leia em <http://bit.ly/human-capital-analytics>).

Um outro exemplo poderia ser este que fizemos, do usuário que acessou as páginas do site: quais serão as chances de ele comprar ou não? Se ele não comprar, não seria melhor eu ajudá-lo para verificar se ele tem interesse pelo produto? Pois, se ele não

tiver, tudo bem, ele vai procurar outra coisa; mas se ele tiver interesse, provavelmente está com dúvida sobre o produto e precisa de ajuda.

Veja o quanto interessante é usar a classificação em diferentes contextos, ou seja, não utilizamos a classificação para classificar apenas cachorros ou porcos, também não usamos apenas para as coisas mais loucas do Google. Queremos vender um produto! Queremos ajudar um cliente, ou verificar se um aluno vai bombar na matéria esse ano.

Em todos esses casos, eu tenho uma classificação, e aqui eu estou querendo classificar se ele vai comprar ou não. O que podemos analisar? Vamos verificar:

- Ele visitou a página 1? Sim (1);
- Ele visitou a página 2? Sim (1);
- Ele visitou a página 3? Sim (1).

Na nossa análise, ele visitou as três páginas, [1, 1, 1], logo, ele vai comprar ou não? Podemos fazer diversas variações, como: ele visitou apenas a página 1 e a 3, [1, 0, 1], será que ele vai comprar? Agora ele visitou a página 2 e a 3 apenas, [0, 1, 1], será que ele vai comprar?

Percebeu que estamos caindo no mesmo problema de sempre? Isto é, dada uma matriz de características de cada usuário do passado e seus comportamentos (suas classificações), como um usuário novo no site vai se comportar? Essa é a classificação que queremos fazer! Nesse caso, a nossa dúvida é: "será que ele vai comprar ou não?". Então, repare que sempre caímos nessa situação de classificação, baseada em características!

No primeiro capítulo, nós escrevemos um código em que temos os nossos elementos, ou seja, os nossos dados:

```
dados = [porco1, porco2, porco3,  
         cachorro4, cachorro5, cachorro6]
```

E as nossas marcações:

```
marcacoes = [1, 1, 1, -1, -1, -1]
```

E da mesma maneira que temos os nossos dados e marcações, também temos os nossos testes que representam os nossos dados com elementos de teste:

```
teste = [misterioso1, misterioso2, misterioso3]
```

E as nossas marcações de teste:

```
marcacoes_teste = [-1, 1, 1]
```

E tudo isso representava o quê? Os animais que estávamos classificando entre -1 e 1, que eram porcos e cachorros. Porém, poderíamos representar alguma outra coisa, e vimos que podemos levar isso para diversos mundos! Vamos trazer isso para o mundo web?

No mundo web, temos diversos usuários acessando meu sistema, então teremos um log de acesso, e vamos utilizar a página da Alura (<https://www.alura.com.br/>) como exemplo, como já vimos anteriormente. A primeira é página principal:

The screenshot shows the Alura website homepage. At the top, there's a navigation bar with a search bar and a 'PESQUISAR' button. Below the header, there's a large banner with the text 'Coloque seu potencial em prática!' and 'Cursos online de tecnologia que reinventam sua carreira.' A green button labeled 'MATRICULE-SE JÁ' is prominently displayed. The main content area features six colored boxes representing different course categories: 'Curso de Mobile' (yellow), 'Curso de Programação' (green), 'Curso de Front-end' (blue), 'Curso de Infraestrutura' (orange), 'Curso de Design & UX' (purple), and 'Curso de Business' (dark blue). Each box contains a small icon and a brief description of the course content.

E então nós temos uma página de uma categoria de cursos:

The screenshot shows a sub-page for mobile courses. The top navigation bar includes links for 'MOBILE', 'PROGRAMAÇÃO', 'FRONT-END', 'INFRAESTRUTURA', 'DESIGN & UX', and 'BUSINESS'. The main heading is 'É hora de decolar sua carreira' with the sub-instruction 'Escolha seu plano e comece a estudar'. Below this, there are two main sections: 'Premium' and 'Premium Plus'. Each section includes a price ('12x R\$ 75' and '12x R\$ 100'), a description of the plan ('Cursos de todas as trilhas', 'Exercícios', 'Estude por 1 ano', and 'ebooks da Casa do Código'), and a small illustration of a rocket launching.

Por fim, chegamos a uma página de um curso:

The screenshot shows a web browser window with the URL <https://www.alura.com.br/contato>. The page has a blue header bar with the text "Fale Conosco". Below the header, there's a section titled "Crítica, sugestão ou elogio? Envie-nos!" with the subtext "Queremos sempre melhorar nossos serviços.". There are two input fields: one for "Nome (obrigatório)" and one for "Email (obrigatório)". The browser interface includes a search bar, a login button, and a matrícula/se button.

Fale Conosco

Crítica, sugestão ou elogio? Envie-nos!

Queremos sempre melhorar nossos serviços.

Nome (obrigatório)

Email (obrigatório)

De alguma forma, sabemos quais foram as páginas que o usuário acessou, ou seja, se acessou as três páginas, se acessou a página 1 e a 3, se foram a 2 e a 3, e assim por diante. É claro que, em um cenário real, a análise é feita em mais 100 páginas ou mais, porém, inicialmente, faremos com três páginas, três características, assim como fizemos com os animais (cachorros e porcos).

Sabemos quais páginas o usuário acessou, então o que precisamos saber? Queremos saber se ele vai comprar ou não, se vai assinar o meu produto ou não vai, se ele vai virar um cliente ou não, se ele vai entrar em contato ou não. Tudo isso se resume a 0 e 1, ou seja, é um tipo de classificação. Estamos classificando um usuário que está acessando o meu site de acordo com quais características?

- Se ele visitou a primeira página, a página principal.
- Se ele visitou a segunda página, de como funciona os planos.
- Se ele visitou a terceira página, a página de contato.

Será que ele entrou nessas páginas? Vamos utilizar 0 para indicar que não entrou, e 1 para indicar que entrou. Então, criaremos um novo arquivo e preencheremos as características do usuário.

O primeiro usuário acessou a página principal (1), porém, não acessou a página de como funciona (0) e nem a de contato (0). Mas esse cara comprou (1):

1, 0, 0, 1

O segundo usuário acessou por um link interno, ou seja, nem passou pela home (0) e foi direto ao como funciona (1). Porém, ele não se interessou, não entrou em contato (0) e não comprou (0):

1, 0, 0, 1  
0, 1, 0, 0

Um terceiro usuário entrou na página principal (1), foi para a página de como funciona (1), não entrou na página de contato (0) e comprou (1).

1, 0, 0, 1  
0, 1, 0, 0  
1, 1, 0, 1

As três primeiras variáveis representam cada acesso que o usuário teve no nosso sistema e, por fim, indicamos se ele comprou ou não. Vamos deixar mais claro o que está acontecendo:

acessou\_home, acessou\_como\_funciona,  
acessou\_contato, comprou  
1, 0, 0, 1  
0, 1, 0, 0  
1, 1, 0, 1

Observe que estamos analisando passo a passo as suas ações

para prever se ele vai comprar ou não no nosso site, mas por que estamos fazendo isso? Digamos que prevemos que o usuário não vai comprar, dessa forma, podemos entrar em contato com ele para entender o motivo dele não ter comprado. De repente, algo do que estamos fornecendo não o agradou ou o que ele procura ainda não está disponível e, então, podemos correr atrás para disponibilizar o conteúdo, por exemplo. Ou pode ser só uma dúvida que ele tem e não conseguiu tirar com ninguém, assim nós mesmos tiramos a dúvida para que ele adquira o produto.

Estou demonstrando um exemplo de como podemos classificar no mundo da web — independentemente se você é um *e-commerce* que vende de tudo, ou um blog que precisa saber se ele acessou determinadas páginas e se vai clicar em uma propaganda ou não. Ou seja, não importa para qual necessidade estamos fazendo a classificação, o importante é: dadas as características do usuário, quero saber se ele comprou ou não.

Observe que, na primeira linha, as três primeiras colunas — `acessou_home` , `acessou_como_funciona` e `acessou_contato` — do nosso novo arquivo são as características que desejamos analisar dos nossos usuários. Já a última, `comprou` , é a nossa marcação. Estamos classificando os usuários como 1 e 0, ou seja, ele comprou ou não?

Temos todos os nossos dados do histórico de acesso dos usuários. Porém, repare que as linhas são separadas por vírgula, ou seja, de acordo com o cabeçalho:

```
acessou_home, acessou_como_funciona,  
acessou_contato, comprou
```

Chamamos esse tipo de arquivo de **comma separated value**

(valor separado por vírgula), mais conhecido por `csv` — um arquivo que separa todos os valores por vírgula. Conseguimos ler este tipo de arquivo pelo Python facilmente por meio de uma função que o carrega, para então tratarmos os dados de uma maneira bem fácil. E é exatamente isso que faremos adiante. Vamos utilizar esse padrão para a leitura dos dados, pois é um padrão bem simples de manter.

## 2.2 IMPORTANDO, CLASSIFICANDO E VALIDANDO UM MODELO

Dentre os arquivos baixados no capítulo de introdução, você encontra um chamado `acesso.csv`. Nele estarão todos os dados referentes ao acesso de páginas do usuário no nosso site, ou seja, se ele acessou a página home, a de como funciona e a de contato. Por fim, informamos se ele comprou ou não. Nesse arquivo, temos diversos dados do nosso usuário no passado, com um total de 100 linhas.

Agora que o temos em mãos, precisamos fazer a leitura dele. Para ler esse arquivo, criaremos um novo arquivo Python chamado `dados.py`, que será o responsável por fazer a sua leitura e colocar os seus dados em um array que tenha as informações dos nossos elementos.

Porém, precisamos apenas de 1 array ou 2 arrays? Lembra que utilizamos um array para os dados:

```
porco1 = [1, 1, 0]
porco2 = [1, 1, 0]
porco3 = [1, 1, 0]
cachorro4 = [1, 1, 1]
cachorro5 = [0, 1, 1]
```

```
cachorro6 = [0, 1, 1]
```

```
dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]
```

E também um outro array para as marcações:

```
marcacoes = [1, 1, 1, -1, -1, -1]
```

Isso significa que precisaremos de 2 arrays também! Então, o nosso arquivo será dividido da seguinte forma:

```
acessou_home, acessou_como_funciona, acessou_contato, comprou  
1, 1, 0, 0
```

Os valores referentes a `acessou_home`, `acessou_como_funciona` e `acessou_contato` (1, 1, 0) serão o nosso primeiro array que representará os nossos dados. Porém, o valor referente ao `comprou` (0) será o nosso segundo array, que representará as nossas marcações.

Observe que os valores dos nossos dados estão em função das marcações, então é comum chamarmos esse tipo de array de X, pois são os dados misteriosos que nós temos, e o que queremos calcular (nesse caso, a marcação) chamamos de Y. É comum usar esses nomes para classificar dados, por isso utilizamos o X para indicar os nossos dados e o Y para os que vamos prever.

## 2.3 IMPORTANDO OS DADOS

Agora que sabemos a forma como representaremos os dados do nosso arquivo `csv`, precisamos importá-lo. Dentro do arquivo `dados.py`, escreveremos o código para importá-lo:

```
import csv
```

Agora precisamos carregar as informações desse `csv`, então

vamos definir a função `carregar_acessos()` :

```
import csv

def carregar_acessos():
```

Mas o que essa função precisa fazer? Ela vai abrir o arquivo `csv` para leitura e ler todas as linhas. A cada linha que for lida, ela associará aos valores para o array de dados e aos valores para o array de marcações:

```
import csv

def carregar_acessos():
    dados = []
    marcacoes = []
```

Agora que definimos os nossos arrays, precisamos abrir o arquivo. Para isso, usamos a função que faz exatamente isso, a `open()`, e enviamos o nome do arquivo por parâmetro:

```
import csv

def carregar_acessos():
    dados = []
    marcacoes = []

    arquivo = open('acesso.csv', 'r')
```

Observe que utilizamos também o parâmetro `r`, indicando que queremos ler o arquivo. Agora precisamos ler esse arquivo `csv`, assim vamos usar a função `reader()` da biblioteca que importamos:

```
import csv

def carregar_acessos():
    dados = []
    marcacoes = []
```

```
arquivo = open('acesso.csv', 'r')
csv.reader(arquivo)
```

A função `reader` devolve um leitor:

```
import csv

def carregar_acessos():
    dados = []
    marcacoes = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)
```

Esse leitor vai passar por cada uma das linhas do arquivo e adicionar os valores dos dados e a marcação:

```
import csv

def carregar_acessos():
    dados = []
    marcacoes = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)
    for linha in leitor:
```

Entretanto, precisamos pegar os valores das colunas, e não uma linha inteira. Então, podemos copiar o nome de cada coluna do nosso arquivo e adicionar no `for` :

```
for acessou_home, acessou_como_funciona,
acessou_contato, comprou in leitor:
```

Nesse instante, estamos dizendo que, para cada uma dessas colunas (`for acessou_home , acessou_como_funciona , acessou_contato , comprou`) dentro do leitor (`in leitor`), vamos fazer alguma coisa (`:` ).

Mas e agora? O que precisamos fazer? Precisamos adicionar ao

nosso array dados as colunas acessou\_home , acessou\_como\_funciona e acessou\_contato , utilizando a função append() :

```
import csv

def carregar_acessos():
    dados = []
    marcacoes = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)
    for acessou_home, acessou_como_funciona,
        acessou_contato, comprou in leitor:

        dados.append([acessou_home,
                      acessou_como_funciona,
                      acessou_contato])

    return dados, marcacoes
```

Por fim, adicionamos as marcações:

```
import csv

def carregar_acessos():
    dados = []
    marcacoes = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)
    for acessou_home, acessou_como_funciona,
        acessou_contato, comprou in leitor:

        dados.append([acessou_home,
                      acessou_como_funciona,
                      acessou_contato])
        marcacoes.append(comprou)

    return dados, marcacoes
```

Agora que adicionamos os dados e as marcações, podemos retorná-los:

```
import csv
```

```
def carregar_acessos():
    dados = []
    marcacoes = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)
    for acessou_home, acessou_como_funciona,
        acessou_contato, comiou in leitor:

        dados.append([acessou_home,
                      acessou_como_funciona,
                      acessou_contato])
        marcacoes.append(comiou)

    return dados, marcacoes
```

Vamos testar o nosso código? Vá ao terminal e abra o interpretador do Python:

```
> python
Python 2.7.10 (default, Oct 14 2015, 16:09:02)
[GCC 5.2.1 20151010] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Primeiro precisamos importar a função `carregar_acessos` do arquivo `dados.py`:

```
>>> from dados import carregar_acessos
>>>
```

Agora podemos usar a nossa função. Vamos retornar seu valor:

```
>>> dados, marcacoes = carregar_acessos()
>>>
```

Será que funcionou? Será que não? Vamos dar uma olhada no valor das nossas marcações:

```
>>> marcacoes
['comiou', '0', '0', '0', '0', '0', '1', '0', '1', '0', '1', '1']
```

```
'1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0'  
, '1', '1', '1', '1', '0', '0', '0', '1', '0', '1', '1', '1', '1', '1'  
, '0', '0', '1', '0', '0', '0', '1', '0', '0', '1', '1', '0', '0', '0'  
, '0', '0', '1', '0', '1', '0', '1', '0', '1', '0', '1', '0', '1', '0'  
, '0', '1', '1', '0', '1', '1', '0', '0', '0', '0', '0', '0', '0', '0'  
, '0', '0', '1', '0', '0', '0', '1', '1', '0', '0', '0', '0', '0', '0'  
, '1', '1', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '0']  
>>>
```

Ele retornou um array com a primeira linha informando ao que se refere, nesse caso, se comprou ou não. Vamos observar as nossas primeiras marcações no arquivo `acesso.csv` :

Ao compararmos os primeiros resultados, funcionou conforme o esperado! Agora vamos dar uma olhada nos nossos dados:

```
'1', '1'], ['0', '0', '1'], ['1', '0', '1'], ['1', '0', '1'], ['1',
    '1', '0'], ['1', '1', '0'], ['0', '0', '1'], ['0', '0', '1'],
    ['0', '0', '1'], ['0', '0', '1'], ['0', '1', '0'], ['0', '0',
    '1'], ['0', '1', '0'], ['1', '1', '0'], ['0', '0', '1'],
    ['0', '0', '1'], ['0', '0', '1'], ['1', '0', '1'], ['1', '0',
    '1'], ['1', '0', '1'], ['1', '1', '0'], ['0', '0', '1'],
    ['0', '0', '1'], ['1', '1', '0'], ['1', '1', '0'], ['0', '0',
    '1'], ['0', '1', '0'], ['0', '0', '1'], ['1', '1', '0'],
    ['1', '1', '0'], ['0', '0', '1'], ['0', '0', '1'], ['1', '1',
    '0'], ['0', '0', '1'], ['0', '0', '1'], ['0', '0', '1']]  
>>>
```

Os dados batem conforme o esperado. Observe que agora separamos os nossos dados das nossas marcações, ou seja, separamos todos os dados que estão em função das marcações. Lembra que, para esses casos de classificação, é tradicional chamarmos os nossos dados (os valores que conhecemos do nosso usuário) de X?

```
def carregar_acessos():
    X = []
    marcacoes = []
    # restante do código
```

Mas e as nossas marcações? Como é mesmo que chamávamos? Lembre-se de que as marcações são os valores que não sabemos, ou seja, os que queremos calcular e prever. Tradicionalmente, as chamamos de Y:

```
def carregar_acessos():
    X = []
    Y = []
    # restante do código
```

Vamos substituir todos os dados para X e marcacoes para Y :

```
import csv

def carregar_acessos():
    X = []
```

```

Y = []

arquivo = open('acesso.csv', 'r')
leitor = csv.reader(arquivo)
for acessou_home, acessou_como_funciona, acessou_contato, compr
ou in leitor:
    X.append([acessou_home, acessou_como_funciona, acessou_cont
ato])
    Y.append(compr)

return X, Y

```

## 2.4 ANALISANDO OS VALORES ADICIONADOS

Vamos dar uma olhada novamente no resultado das nossas marcações:

```

>>> marcacoes
[ 'compr', '0', '0', '0', '0', '0', '1', '0', '1', '0', '1', '1',
, '1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
, '1', '1', '1', '1', '0', '0', '0', '1', '0', '1', '1', '1', '1',
, '0', '0', '1', '0', '0', '1', '0', '0', '1', '1', '1', '0', '0',
, '0', '0', '1', '0', '1', '0', '1', '0', '1', '0', '1', '0', '0',
, '0', '1', '1', '0', '1', '1', '0', '0', '0', '0', '0', '0', '0',
, '0', '0', '0', '1', '0', '0', '0', '1', '1', '0', '0', '0', '0',
, '1', '1', '0', '0', '0', '1', '0', '0', '0', '0', '0', '0', '0'
>>>

```

Rpare que veio também o array com a informação do cabeçalho, ou seja, a palavra 'compr', , porém nós não queremos isso no nosso array. Você se lembra de como eram as nossas marcações no primeiro exemplo?

```
marcacoes = [1, 1, 1, -1, -1, -1]
```

Como podemos ver, só tem números! Em nenhum momento especificamos o que significa por meio de uma palavra na primeira

posição:

```
marcacoes = ['é_um_porco', 1, 1, 1, -1, -1, -1]
```

Então precisamos descartar esses cabeçalhos simplesmente não lendo a primeira linha. Como podemos fazer isso? Simples, basta apenas, após o momento em que criamos o `leitor` (`leitor = csv.reader(arquivo)`), adicionarmos o comando `leitor.next()` que pulará para a linha a seguir:

```
import csv

def carregar_acessos():
    X = []
    Y = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)

    for acessou_home, acessou_como_funciona, acessou_contato, compr
    ou in leitor:
        X.append([acessou_home, acessou_como_funciona, acessou_cont
        ato])
        Y.append(comprou)

    return X, Y
```

Vamos testar o nosso código novamente? Antes de importar a nossa função `carregar_acessos`, feche o interpretador do Python e abra novamente para que ele carregue a nova versão da nossa função. Vejamos o resultado:

```
>>> from dados import carregar_acessos
>>>
```

Importou sem nenhum problema! Porém, em vez de chamar de `dados` e `marcacoes`, chamaremos de `X` e `Y`:

```
>>> from dados import carregar_acessos  
>>> X, Y = carregar_acessos()  
>>>
```

Vamos verificar o valor de `Y` :

```
>>> from dados import carregar_acessos  
>>> X, Y = carregar_acessos()  
>>> Y  
['0', '0', '0', '0', '0', '1', '0', '1', '0', '1', '1', '1', '1', '0',  
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1',  
'1', '1', '0', '0', '0', '1', '0', '1', '1', '1', '1', '1', '0', '0',  
'1', '0', '0', '0', '1', '0', '0', '1', '1', '0', '0', '0', '0', '0',  
'1', '0', '1', '0', '1', '0', '1', '0', '1', '0', '0', '0', '0', '0',  
'1', '0', '1', '1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',  
'0', '1', '0', '0', '1', '1', '0', '0', '0', '0', '0', '0', '0', '1',  
'0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '0', '1', '1',  
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
```

Aparentemente, tudo está funcionando como o esperado! Porém, '0' ? '1' ? Isso é uma *string*! Não queremos *strings*, nós queremos números! Afinal, estamos lendo números. Como resolvemos isso?

Podemos converter as strings para números, mas que tipo de números? Nesse caso, estamos trabalhando com números inteiros, então converteremos para números inteiros! Usaremos a maneira mais tradicional que é enviar, por parâmetro, a coluna desejada na instrução `int()` :

```
import csv  
  
def carregar_acessos():  
    X = []  
    Y = []  
  
    arquivo = open('acesso.csv', 'r')  
    leitor = csv.reader(arquivo)  
  
    next(leitor)  
  
    for acessou_home, acessou_como_funciona, acessou_contato, compr
```

```
    ou in leitor:  
  
        X.append([int(acesou_home),int(acesou_como_funciona)  
                  ,int(acesou_contato)])  
        Y.append(int(comprou))  
  
    return X, Y
```

Testando novamente o nosso código:

```
>>> from dados import carregar_acessos  
>>> X, Y = carregar_acessos()  
>>>
```

Vejamos agora o valor do  $\gamma$ :

E o x?

```
], [0, 0, 1], [0, 1, 0], [0, 0, 0]]
```

Agora sim nós carregamos os nossos números!

## 2.5 MELHORANDO A LEITURA DO CÓDIGO

Observe que o nosso código ficou um pouco extenso e repetitivo:

```
import csv

def carregar_acessos():
    X = []
    Y = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)

    for acessou_home, acessou_como_funciona, acessou_contato, compr
ou in leitor:

        X.append([int(acessou_home), int(acessou_como_funciona)
                  , int(acessou_contato)])
        Y.append(int(comprou))

    return X, Y
```

Veja que usamos os nomes `acessou_home` , `acessou_contato` e `acessou_alguma_coisa` . Vamos retirar esse `acessou` , pois já sabemos que nossos dados se referem ao acesso de página do usuário, então não precisamos repetir a mesma palavra em todos eles! Veja o resultado do nosso código:

```
import csv

def carregar_acessos():
    X = []
    Y = []
```

```
arquivo = open('acesso.csv', 'r')
leitor = csv.reader(arquivo)

next(leitor)

for home, como_funciona, contato, comprou in leitor:

    X.append([int(home),int(como_funciona)
              ,int(contato)])
    Y.append(int(comprou))

return X, Y
```

Também alteraremos no nosso arquivo csv :

```
home, como_funciona, contato, comprou
1,1,0,0
1,1,0,0
1,1,0,0
1,1,0,0
1,1,0,0
1,0,1,1
...
```

Melhoramos uma boa parte do nosso código e deixamos mais limpo, ou seja, com uma leitura mais clara. Porém, perceba que ainda existem alguns detalhes. Veja esse trecho de código:

```
X.append([int(home),int(como_funciona)
          ,int(contato)])
```

Enviamos um monte de informação para o X que, à primeira vista, não dá para saber sobre o que se refere. Nesses casos, podemos extrair todas essas informações para uma variável que deixará mais claro o significado desses valores:

```
dado = [int(home),int(como_funciona)
        ,int(contato)]
```

Observe agora o resultado final do nosso código:

```
import csv

def carregar_acessos():
    X = []
    Y = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)

    for home,como_funciona,contato, comrou in leitor:

        dado = [int(home),int(como_funciona),
                ,int(contato)]
        X.append(dado)
        Y.append(int(comrou))

    return X, Y
```

Bem mais limpo e de fácil compreensão! Esse passo é uma refatoração simples e famosa, uma boa prática chamada de *extract variable*.

Vamos verificar se o nosso código ainda funciona? Porém, chega de ficar executando no *command line* interface diretamente. Vamos passar o conteúdo dessa função para um arquivo mais organizado, chamando-o de `classifica_acessos.py` .

Agora basta importar a função `carregar_acessos()` dentro desse arquivo:

```
from dados import carregar_acessos
X, Y = carregar_acessos()
```

Por fim, vamos imprimir o `X` e o `Y` para verificar se está funcionando corretamente:

```
from dados import carregar_acessos
X, Y = carregar_acessos()
```

```
print(X)
print(Y)
```

Vamos testar o nosso código:

```
> python3 classifica_acessos.py
[[1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 0, 1]
, [1, 1, 0], [1, 0, 1], [1, 1, 0], [1, 0, 1], [1, 1, 0], [1, 0, 1
], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1
], [0, 0, 1], [0, 1, 0], [0, 0, 1], [0, 1, 0], [0, 0, 1], [0, 1, 0
], [1, 0, 1], [1, 1, 1], [1, 1, 1], [1, 0, 1], [0, 1, 0], [0, 0, 1
], [0, 1, 0], [1, 0, 1], [0, 0, 1], [1, 0, 1], [1, 0, 1], [1, 0, 1
], [1, 0, 1], [0, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 0], [0, 0, 1
], [0, 1, 0], [1, 0, 1], [0, 0, 1], [0, 0, 1], [1, 0, 1], [1, 0, 1
], [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 0, 1], [1, 1, 0], [1, 1, 0
], [1, 0, 1], [1, 1, 0], [1, 0, 1], [1, 1, 0], [1, 0, 1], [1, 0, 1
], [1, 0, 1], [0, 0, 1], [0, 1, 0], [1, 0, 1], [0, 0, 1], [1, 0, 1
], [0, 0, 1], [1, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [1, 0, 1
], [1, 0, 1], [1, 1, 0], [1, 1, 0], [0, 0, 1], [0, 0, 1], [1, 0, 1
], [1, 0, 1], [1, 1, 0], [0, 0, 1], [0, 0, 1], [0, 0, 1], [1, 0, 1
], [0, 0, 1], [0, 1, 0], [0, 0, 1], [0, 0, 1], [1, 0, 1], [0, 0, 1
], [0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0
, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0
, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Está funcionando perfeitamente! Agora podemos tirar as impressões do X e Y:

```
from dados import carregar_acessos
X, Y = carregar_acessos()
```

Temos o X (dados) e o Y (marcações), mas e agora? O que faremos com eles? Lembra de que, no exemplo anterior, nós tínhamos os nossos dados e marcacoes?

```
dados = [porco1, porco2, porco3, cachorro4, cachorro5, cachorro6]
marcacoes = [1, 1, 1, -1, -1, -1]
```

O que fazíamos mesmo? Rodávamos o modelo! Então, agora vamos importar o algoritmo `MultinomialNB` :

```
from dados import carregar_acessos  
X, Y = carregar_acessos()  
  
from sklearn.naive_bayes import MultinomialNB
```

E o que fazíamos com esse algoritmo? Pedíamos para ele criar um modelo e adaptá-lo ( `fit` ) com os nossos dados ( `X` ) e marcações ( `Y` ):

```
from dados import carregar_acessos  
X, Y = carregar_acessos()  
  
from sklearn.naive_bayes import MultinomialNB  
modelo = MultinomialNB()  
modelo.fit(X, Y)
```

Agora precisamos testar o nosso modelo. Vamos verificar esse novo usuário:

```
[1, 0, 1]
```

Observe que ele entrou na home, não entrou na página de como funciona e entrou na página de contato. E agora? Ele comprou ou não? Vamos pedir para que o nosso modelo preveja ( `predict` ) para nós:

```
from dados import carregar_acessos  
X, Y = carregar_acessos()  
  
from sklearn.naive_bayes import MultinomialNB  
modelo = MultinomialNB()  
modelo.fit(X, Y)  
  
modelo.predict([1,0,1])
```

Porém, você lembra de que o `predict` espera um array de arrays para não mostrar aquele *warning*? Vamos adicionar um

array:

```
modelo.predict([[1,0,1]])
```

Por fim, vamos imprimir o predict para verificar o resultado:

```
from dados import carregar_acessos
X, Y = carregar_acessos()

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(X, Y)
print(modelo.predict([[1,0,1]]))
```

Testando o nosso algoritmo:

```
> python3 classifica_acessos.py
> [1]
```

Ele imprimiu 1, e isso significa que esse usuário vai comprar. Isto é, estamos acreditando que ele vai comprar de acordo com o nosso modelo.

Mas e se tivéssemos mais um usuário que caiu direto na página de como funciona e não entrou em contato ( [0,1,0] )? Ele simplesmente entrou na página de como funciona e viu apenas quais são os nossos planos, nem chegou a ver quais são os nossos produtos ou serviços. E agora? Esse usuário vai comprar ou não? Vamos adicioná-lo ao nosso modelo e pedir para ele prever para nós!

```
print(modelo.predict([[1,0,1],[0,1,0]]))
```

Verificando o resultado desse novo usuário:

```
> python3 classifica_acessos.py
> [1 0]
```

Observe que o nosso modelo está prevendo para nós que apenas o primeiro usuário vai comprar e o segundo não. Será que apenas esses dois testes já são o suficiente? Com certeza precisamos verificar outros cenários para garantir que está funcionando.

Além de testar apenas dois casos, podemos testar muito mais, então vamos adicionar um terceiro que só acessou apenas a página home ( [1, 0, 0] ):

```
print(modelo.predict([[1,0,1],[0,1,0],[1,0,0]]))
```

E agora? Será que ele compra? Vamos pedir para que o nosso algoritmo preveja:

```
> python3 classifica_acessos.py  
> [1 0 0]
```

Também não compra. Vamos testar mais um caso em que o usuário entra na página home e na página de como funciona, porém não entra na de contato ( [1, 1, 0] ):

```
print(modelo.predict([[1,0,1],[0,1,0],  
[1,0,0], [1,1,0]]))
```

Será que agora ele vai comprar? Vamos verificar o resultado:

```
> python3 classifica_acessos.py  
> [1 0 0 0]
```

Nenhum desses usuários comprará. Provavelmente, existe alguma característica em comum entre eles.

Mas e aquele usuário que entra na página home, entra na página de como funciona e também entra na página de contato ( [1, 1, 1] ), ele compra ou não?

```
print(modelo.predict([[1,0,1],[0,1,0],  
[1,0,0], [1,1,0], [1,1,1]]))
```

Vejamos o resultado:

```
> python3 classifica_acessos.py  
> [1 0 0 0 0]
```

Também não compra. Esse usuário acessou todas as páginas, mas, mesmo assim, provavelmente não comprará.

Observe que o modelo está dizendo apenas o que ele acredita. Essa previsão está boa ou ruim? O algoritmo está chutando bem ou mal? O nível de acerto está alto ou baixo?

Como podemos fazer para saber todas essas informações? Podemos testar o nosso modelo da mesma forma que fizemos quando estávamos classificando os porcos e os cachorros. Lembra como calculávamos a nossa taxa de acerto?

```
# restante do código  
  
resultado = modelo.predict(teste)  
  
diferencas = resultado - marcacoes_teste  
  
acertos = [d for d in diferencas if d == 0]  
  
total_de_acertos = len(acertos)  
total_de_elementos = len(teste)  
  
taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos
```

Temos os dados que representam a variável `teste`, ou seja, o nosso `X`. Vamos pedir para o nosso algoritmo representar o `resultado` prevendo (`predict`) os nossos dados (`X`):

```
from dados import carregar_acessos  
X, Y = carregar_acessos()  
  
from sklearn.naive_bayes import MultinomialNB  
modelo = MultinomialNB()
```

```
modelo.fit(X, Y)  
  
resultado = modelo.predict(X)
```

Na variável `resultado`, teremos vários 0 e 1 que vão prever se cada um desses usuários vai ou não comprar. Se imprimirmos essa variável:

```
print(resultado)
```

E rodarmos o nosso algoritmo:

Além dos dados, temos também as marcações ( Y ) e, por meio delas, sabemos algo a mais sobre os nossos usuários. Nesse caso, saberemos se os que acessaram as páginas do nosso website compraram ou não. Se imprimirmos também o nosso Y :

```
print(resultado)  
print(Y)
```

O resultado será:

Mas e agora? O que faremos com esses 0 e 1? Podemos fazer a mesma comparação de antes. Se eles forem iguais, significa que acertamos, porém, se forem diferentes, significa que erramos. Fazemos isso da mesma forma que fizemos anteriormente, ou seja, a conta de diferença. Então, o valor da variável `diferencias` será o nosso resultado menos o `Y`:

```
diferencias = resultado - Y
```

Agora que calculamos a nossa diferença, precisamos dizer que, para cada valor da `diferencias` que for igual a 0, vamos contabilizar e atribuir para a variável `acertos`:

```
acertos = [d for d in diferencias if d == 0]
```

O total de acertos era justamente o tamanho da nossa variável `acertos`, pois ela representa a quantidade de acertos que tivemos:

```
total_de_acertos = len(acertos)
```

Além do total de acertos, nós também fizemos o total de elementos, que é a quantidade de dados que nós temos (o tamanho do nosso `X`):

```
total_de_elementos = len(X)
```

Por fim, precisamos calcular a nossa taxa de acerto, que é a divisão do total de acertos pela quantidade de elementos, multiplicada por 100.0 para apresentar o ponto flutuante, ou seja, os números após a vírgula para uma melhor precisão do percentual de acerto:

```
taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos
```

E agora? Precisamos imprimir a nossa taxa de acerto e quantos

elementos foram testados!

```
print(taxa_de_acerto)
print(total_de_elementos)
```

Vejamos como ficou o nosso código:

```
from dados import carregar_acessos
X, Y = carregar_acessos()

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(X, Y)

resultado = modelo.predict(X)

diferencias = resultado - Y

acertos = [d for d in diferencias if d == 0]
total_de_acertos = len(acertos)
total_de_elementos = len(X)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print(taxa_de_acerto)
print(total_de_elementos)
```

Vamos testá-lo? Vejamos o resultado:

```
> python3 classifica_acessos.py
> 93.9393939394
> 99
```

## 2.6 ACERTANDO DEMASIADAMENTE?

Acertou 93,93%? Já podemos ficar felizes e declarar vitória! Porém, você se lembra de que eu havia falado que é muito difícil chegarmos a taxas de acerto absurdamente altas, como 100%?

Se isso é verdade, então por que o nosso algoritmo está

acertando tanto? Se verificarmos quem usamos para treinar o nosso algoritmo:

```
modelo.fit(X, Y)
```

Foi o nosso `X`, ou seja, todos os nossos dados. E para testar o nosso algoritmo?

```
resultado = modelo.predict(X)
```

Utilizamos o `X` de novo! Será que estamos testando da maneira correta? Vamos analisar uma situação similar.

Suponhamos que eu dê para você 10 porquinhos e 10 cachorrinhos, diga quais são os porcos e quais são os cachorros, e você aprendeu. Em seguida, dou os **mesmos 10 porquinhos** e os **mesmos 10 cachorros**, e depois pergunto para você: "Quais são os porquinhos e quais são os cachorros?".

Quanto você acha que vai acertar? Eu espero que você acerte muito, pois são os mesmos porquinhos e cachorros que lhe ensinei a poucos instantes atrás.

Um outro exemplo similar seria eu apresentar a minha mão direita e, logo em seguida, a minha mão esquerda. Então, mostro a minha mão direita e pergunto: "Que mão é essa?", e depois mostro a minha mão esquerda e pergunto a mesma coisa. Com certeza você vai acertar, pois eu acabei de falar qual é cada uma das mãos.

Isso significa que, se você estiver utilizando o mesmo elemento com que você treinou o algoritmo para testá-lo, é bem provável que ele acerte! E não é para isso que criamos esse algoritmo!

No mundo real, quando treinamos um algoritmo de classificação, nós o ensinamos com elementos que conhecemos,

para que ele tenha um histórico do que já sabemos sobre aquele determinado elemento. Porém, quando testamos esse tipo de algoritmo, utilizamos apenas elementos desconhecidos, pois são esses que nós queremos que ele classifique para nós.

Não faz sentido algum nós treinarmos o nosso algoritmo com os 99 registros que conhecemos (ou seja, que já classificamos), e pedirmos para ele testar com os mesmos 99 registros. O nosso teste precisa ser feito com elementos que o nosso algoritmo nunca viu.

Mas se nós temos apenas 99 registros anotados, como podemos fazer para que o nosso algoritmo seja treinado e testado por todos eles, sem que aconteça o mesmo caso problemático que vimos anteriormente?

Podemos dividir os nossos registros em duas partes! Por exemplo, podemos deixar os primeiros 50 para treinar o nosso algoritmo e os 49 restantes para testá-lo. Ou então, podemos deixar 80 registros para treiná-lo e os outros 19 para testá-lo.

Uma das maneiras tradicionais para esses casos é quebrar os nossos dados em 90% e 10%, sendo que, os 90% serão os dados que vamos usar para treinar o nosso algoritmo e os 10% serão os dados utilizados para testá-lo! Então, vamos alterar o nosso código. Observe o X e o Y :

```
X, Y = carregar_acessos()
```

Todos os nossos dados estão concentrados nessas duas variáveis, ou seja, todas as características ( X ) e todas as marcações ( Y ) de **treino!** Precisamos separar o X e o Y de treino e o de teste. Começaremos pelos nossos dados e marcações de treino:

```
X, Y = carregar_acessos()
```

```
treino_dados  
treino_marcacoes
```

Precisamos adicionar 90% para ambos, então as 90 **primeiras linhas** para cada um deles:

```
X, Y = carregar_acessos()
```

```
treino_dados = X[:90]  
treino_marcacoes = Y[:90]
```

Agora precisamos adicionar os 10% que restaram para as variáveis de teste, porém, dessa vez, precisamos das 9 **últimas linhas**!

```
X, Y = carregar_acessos()
```

```
treino_dados = X[:90]  
treino_marcacoes = Y[:90]
```

```
teste_dados = X[-9:]  
teste_marcacoes = Y[-9:]
```

Vamos testar o nosso código. Para isso, abriremos o interpretador do Python:

```
> python  
>>>
```

Agora importaremos o nosso método `carregar_acessos()`:

```
>>> from dados import carregar_acessos  
>>>
```

Então, chamamos o método `carregar_acessos()` e retornamos para o `X` e o `Y`:

```
>>> from dados import carregar_acessos  
>>> X,Y = carregar_acessos()
```

Vamos criar as nossas variáveis de treino `treino_dados` e `treino_marcacoes`:

```
>>> from dados import carregar_acessos  
>>> X, Y = carregar_acessos()  
>>> treino_dados = X[:90]  
>>> treino_marcacoes = Y[:90]
```

Será que funcionou? Vamos imprimir os nossos treino\_dados :

```
>>> treino_dados
[[[1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 0, 1],
 , [1, 1, 0], [1, 0, 1], [1, 1, 0], [1, 0, 1], [1, 1, 0], [1, 0, 1],
 ], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 1, 0],
 ], [1, 0, 1], [1, 1, 1], [1, 1, 1], [1, 0, 1], [1, 0, 1], [0, 1, 0],
 ], [0, 1, 0], [1, 0, 1], [0, 0, 1], [1, 0, 1], [1, 0, 1], [1, 0, 1],
 ], [1, 0, 1], [0, 0, 1], [1, 1, 0], [0, 0, 1], [0, 0, 1], [0, 0, 1],
 ], [0, 1, 0], [1, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 0], [0, 0, 1],
 ], [1, 0, 1], [0, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 0], [0, 0, 1],
 ], [0, 1, 0], [1, 0, 1], [0, 0, 1], [1, 0, 1], [1, 0, 1], [1, 0, 1],
 ], [1, 0, 1], [0, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 0], [0, 0, 1],
 ], [0, 0, 1], [1, 0, 1], [1, 0, 1], [1, 1, 0], [1, 1, 0], [0, 0, 1],
 ], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1],
 ], [0, 0, 1], [1, 1, 0], [0, 0, 1], [0, 0, 1], [0, 0, 1], [1, 0, 0],
 ], [0, 0, 1], [1, 0, 1], [1, 1, 0], [0, 0, 1], [0, 0, 1], [1, 0, 0],
 ], [1, 0, 1], [1, 1, 0], [1, 1, 0], [0, 0, 1], [0, 0, 1], [1, 0, 0],
 ], [1, 0, 1], [1, 1, 0], [1, 1, 0], [0, 0, 1], [0, 0, 1], [1, 0, 0],
 ]]
```

Aparentemente foram impressos todos os dados, mas e o tamanho desse array? Será que são 90 dados? Vejamos:

```
>>> len(treino_dados)  
90
```

Funcionou conforme o esperado! Agora faremos o mesmo para os dados e as marcações de teste:

```
>>> teste_dados = X[-9:]
>>> teste_marcacoes = Y[-9:]
>>> len(teste_dados)
9
```

```
>>> len(teste_marcacoes)
9
>>> teste_dados
>>> [[1, 0, 1], [1, 1, 0], [1, 1, 0], [0, 0, 1], [0, 0, 1], [1, 1,
, 1], [0, 0, 1], [0, 1, 0], [0, 0, 0]]
>>> teste_marcacoes
[1, 0, 0, 0, 0, 1, 0, 0, 0]
```

Conseguimos separar todos os nossos dados para treino e para teste. Agora precisamos alterar o nosso código para utilizar esses novos dados. Começaremos pelo treino:

```
from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(X, Y)
```

Em vez de utilizar os nossos dados reais para treinar ( `X` e `Y` ), usaremos os nossos dados de treino, ou seja, `treino_dados` e `treino_marcacoes` :

```
from dados import carregar_acessos
X, Y = carregar_acessos()

treino_dados = X[:90]
treino_marcacoes = Y[:90]

teste_dados = X[-9:]
teste_marcacoes = Y[-9:]

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)
```

Depois de treinar, o que pedíamos para o nosso algoritmo fazer? Prever para nós quais são os novos dados e retornar o resultado! Vejamos como o nosso código está atualmente:

```
resultado = modelo.predict(X)
```

Mas não podemos mais pedir para ele prever com dados que já

conhece. Ou seja, em vez dos dados reais (  $X$  ), pediremos para ele classificar os nossos dados de teste ( `teste_dados` ):

```
resultado = modelo.predict(teste_dados)
```

O nosso algoritmo fez o chute para os nossos dados de teste, agora precisamos verificar a diferença entre o resultado e as nossas marcações de teste. Vejamos como está no nosso código:

```
diferencias = resultado - Y
```

Observe que estamos fazendo a diferença com as marcações reais (  $Y$  ) que se referem aos dados reais. Precisamos mudar para as nossas marcações que representam os nossos dados de teste ( `teste_marcacoes` ):

```
diferencias = resultado - teste_marcacoes
```

Corrigimos o treino e prevenção do algoritmo, agora só precisamos verificar como está sendo feito o cálculo da taxa de acerto:

```
acertos = [d for d in diferencias if d == 0]
total_de_acertos = len(acertos)
total_de_elementos = len(X)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print(taxa_de_acerto)
print(total_de_elementos)
```

Os `acertos` ainda serão todos os valores do array `diferencias` que forem iguais a 0, e a `taxa_de_acertos` será também o tamanho do array `acertos`. Entretanto, o `total_de_elementos` é o tamanho do nosso `X`? Não! Nós estamos fazendo o teste com os nossos dados de teste, então precisamos pegar o tamanho do nosso `teste_dados` para

calcular o total de elementos:

```
acertos = [d for d in diferenças if d == 0]
total_de_acertos = len(acertos)
total_de_elementos = len(teste_dados)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print(taxa_de_acerto)
print(total_de_elementos)
```

Vamos testar o nosso algoritmo? Vejamos o resultado:

```
> python3 classifica_acessos.py
88.8888888889
9
```

Relembrando, nosso código até agora é:

```
from dados import carregar_acessos
X, Y = carregar_acessos()

treino_dados = X[:90]
treino_marcacoes = Y[:90]

teste_dados = X[-9:]
teste_marcacoes = Y[-9:]

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)

resultado = modelo.predict(teste_dados)

diferenças = resultado - teste_marcacoes

acertos = [d for d in diferenças if d == 0]
total_de_acertos = len(acertos)
total_de_elementos = len(teste_dados)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print(taxa_de_acerto)
print(total_de_elementos)
```

Veja que o nosso algoritmo treinou com 90 elementos, testou com 9, e o resultado foi de 89%, ou seja, ele acertou 89% das vezes! Agora sim o nosso teste está sendo mais realista.

Por que dessa vez foi diferente da outra que fizemos? Se levarmos em consideração os porcos e os cachorros, o nosso cenário anterior se resume em:

1. Eu lhe apresento 90 animais entre porcos e cachorros, e ensino quais são os porcos e quais são os cachorros.
2. Então, mostro os mesmos 90 animais que já havia apresentado e peço para que você os classifique entre porcos e cachorros.

Com certeza você acertaria todos ou praticamente quase todos! Mas agora o cenário ficou diferente, estamos agindo da seguinte maneira:

1. Eu lhe apresento 90 animais entre porcos e cachorros, e ensino quais são os porcos e quais são os cachorros.
2. Então, mostro 9 novos animais, que também são porcos e cachorros, mas que você nunca viu, e peço para que você classifique quais são os porcos e os cachorros.

Se você tiver de classificar um novo elemento que você nunca viu, é bem provável que você terá uma taxa de acerto menor. É exatamente por isso que o resultado foi mais baixo do que o anterior, pois agora nosso algoritmo está lidando com elementos que ele nunca viu na vida.

## 2.7 RESUMINDO

Em um processo de classificação, nós temos as características que são todas as informações que utilizamos para poder distinguir um elemento. Por exemplo, quais poderiam ser uma das características de um e-mail? Poderia ser o seu tamanho, número de palavras repetidas, se usa letra maiúscula, se o remetente é conhecido, se já foi marcado como *spam*, entre diversas outras! E se forem animais? Outras características. E se forem alunos que vão repetir de ano? Outras também.

Chamamos essas características de X, que são todos os dados de entrada que nós temos, todos os que nós conhecemos. Estes podem ser de e-mails, animais, funcionários, usuários, alunos ou qualquer coisa que queremos classificar; todos eles são os nossos elementos.

A partir das características dos nossos elementos, nós treinamos e aprendemos o que é o elemento, de acordo com sua característica e marcação, e então tentamos prever um novo elemento. Isto é, se ele se encaixa na classificação 0 ou 1.

Além disso, precisamos sempre lembrar que existe uma taxa de erro, pois nem sempre nós acertaremos. Precisamos sempre verificar a forma como o algoritmo foi treinado, analisando a quantidade de dados e características, para determinar um retorno aceitável do nosso objetivo.

Por exemplo, nosso objetivo pode ser conversar com mais pessoas que provavelmente vão pedir para serem demitidas ou com menos pessoas, ou então se queremos conversar com os alunos que estão com uma chance maior de reprovarem. Talvez

seja mais interessante conversar com mais alunos do que menos alunos, porém, se fosse entre um cachorro e porco, para qual dos dois eu gostaria de errar mais? Depende do caso!

Isso significa que precisaremos verificar o quanto interessante é em cada caso, ou seja, se aceitaremos uma taxa de erro maior ou menor. Temos de nos atentar a todos esses detalhes, pois é dessa forma que o processo de classificação, baseado nas características, funciona.

## Verificando a qualidade dos nossos modelos

E como transformamos essa teoria em código? Primeiro precisamos representar os nossos elementos, suas características e marcações. Podemos representar todos esses dados por meio de um arquivo `csv`, que é uma abordagem muito comum para tratamento de dados, pois podemos pegar uma planilha eletrônica e converter para esse formato.

Vimos que podemos ler um arquivo `csv` por meio de uma função de leitura de arquivo do próprio Python:

```
arquivo = open('acesso.csv', 'r')
leitor = csv.reader(arquivo)
```

Nesse exemplo, só trabalhamos com dados inteiros, mas, mais à frente, veremos como trabalhar com `float`, `string` e outras coisas. Além disso, vimos que, de acordo com os dados que lemos, aprendemos a importância de treinar um modelo e testá-lo:

```
X, Y = carregar_acessos()
```

Entretanto, existe um grande desafio nessa abordagem, porque se treinarmos o nosso modelo com diversos dados:

```
modelo.fit(X,Y)
```

Acabamos viciando o nosso modelo com apenas esses dados. Se testarmos esse modelo perguntando sobre um ou mais elementos que faça parte desses diversos dados:

```
resultado = modelo.predict(X)
```

A chance de ele acertar é muito grande! Será que faz sentido esse teste? Aparentemente não. Mas e se o teste for com um elemento que o nosso modelo nunca viu na vida?

```
X,Y = carregar_acessos()
```

```
treino_dados = X[:90]
treino_marcacoes = Y[:90]
```

```
teste_dados = X[-9:]
teste_marcacoes = Y[-9:]
```

```
from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)
```

```
resultado = modelo.predict(teste_dados)
```

Agora sim estamos verificando se o algoritmo aprendeu de verdade! E é justamente por esse motivo que não usamos os mesmos dados que foram usados para treino, para testar o nosso modelo.

Perceba que utilizamos uma estratégia para separar os dados de treino dos dados de testes, que foi:

- 90% dos dados serão para treino:

```
treino_dados = X[:90]
treino_marcacoes = Y[:90]
```

- 10% restantes dos dados serão para teste:

```
teste_dados = X[-9:]
teste_marcacoes = Y[-9:]
```

O método de treino e teste é tão importante que, na maioria das vezes, formalizamos passo a passo o que utilizamos para chegar a um determinado resultado:

```
# minha abordagem inicial foi
# 1. separar 90% para treino e 10% para teste: 88.89%

from dados import carregar_acessos

X,Y = carregar_acessos()

treino_dados = X[:90]
treino_marcacoes = Y[:90]

teste_dados = X[-9:]
teste_marcacoes = Y[-9:]
```

Podemos ver que, nesse teste, foram 90% para teste e 10% para leitura, e o resultado foi 88,89%. Perceba que, para cada teste, podemos registrá-lo com essas anotações e verificarmos como é o comportamento do nosso algoritmo para cada cenário.

## CAPÍTULO 3

# CLASSIFICAÇÃO DE VARIÁVEIS CATEGÓRICAS

Até agora, já resolvemos alguns problemas de classificação que havíamos visto, porém os dados com que nos deparamos indicavam características dos elementos que estávamos analisando. Por exemplo, se o elemento era um cachorro ou um porco, verificávamos se ele tinha perna curta, se era gordinho ou se fazia *auau*.

Para todas as características, marcamos entre 0 e 1 indicando se o elemento a tinha, ou seja, se ele tinha perna curta, marcávamos como 1; se não, 0; e assim sucessivamente para as demais características.

Perceba que todas as nossas características tiveram apenas dois tipos de valores (0 ou 1). Além disso, nossas marcações também tiveram apenas dois valores, 0 ou 1, ou então, -1 ou 1. Porém, nem sempre nos deparamos com esses tipos de valores para os nossos dados. Vamos verificar um exemplo um pouco diferente do qual vimos até agora.

- Na Alura, um site de cursos online, nós temos um cliente que visitou a home.

De acordo com o que vimos até agora, como que classificariámos essa situação? É fácil, certo? Simplesmente marcamos como 0 se não, e 1 se visitou. Por enquanto, sem nenhuma novidade. Agora esse cliente fez o seguinte:

- Esse cliente já estava logado.

E como fazemos para identificar essa informação? Marcamos com 1 para indicar que ele estava logado. Vejamos a próxima ação dele:

- E então, esse cliente comprou um curso.

Fazemos da mesma forma que fizemos anteriormente, ou seja, marcamos com 1 para indicar que ele comprou. Porém, observe a próxima situação desse mesmo cliente:

- Por fim, esse cliente buscou sobre 'algoritmos'.

Como podemos representar essa situação? Será que podemos também marcar com 1 para dizer que sim e 0 para não? E se ele buscasse outro curso? O que faríamos? Faz sentido marcamos como 0 ou 1? Podemos verificar outro exemplo similar:

- Um cliente que entrou na home (0 ou 1).
- Não estava logado (0 ou 1).
- Não comprou (0 ou 1).
- Porém, buscou sobre 'Java' (e agora?).

Cada cliente poderia buscar sobre diversos tipos de cursos, assuntos ou tecnologias, por exemplo, HTML, CSS, JavaScript, SQL, Ruby ou qualquer outra informação. A questão é: como podemos representar todas essas informações distintas para uma

mesma característica?

Até agora, vimos apenas como distinguir uma característica com apenas 2 valores distintos, ou seja, 0 ou 1. Porém, como valor de busca, podemos ter diversos valores, por exemplo, buscou algoritmos, buscou Java ou buscou Ruby. Vamos verificar algumas possibilidades para esse caso na tabela:

Home	Busca	Estava logado?	Comprou?
1	algoritmos	1	1
1	java	0	0
0	java	1	0
1	algoritmos	1	1
0	java	0	1
1	algoritmos	0	0
1	ruby	1	1

Cada coluna refere-se a:

- **Home:** se o cliente visitou a página home (0 ou 1).
- **Busca:** o que o cliente buscou (algoritmos ou Java ou Ruby etc.).
- **Estava logado?:** se o cliente estava logado (0 ou 1).
- **Comprou?:** se o cliente comprou ou não (0 ou 1).

Analizando cada coluna, qual é a novidade dentre esses conjuntos de dados? É a coluna **busca**, pois é um tipo de dado com não trabalhamos ainda. Observe que essa coluna é uma informação extremamente rica sobre o nosso usuário, pois ela não diz apenas se o cliente buscou ou não, ou seja, ela diz também o

que ele buscou!

Esse tipo de informação é muito importante, pois um cliente que está buscando, por exemplo, "algoritmos", vai ter um comportamento; um outro cliente que busca sobre "Java" vai ter um outro comportamento diferente, e assim sucessivamente. Perceba que podemos conter um conjunto de diversos cursos e que cada um precisa ter o seu comportamento.

Uma situação similar a essa é quando vamos a uma livraria comprar um livro e você quer, por exemplo, um livro de programação. Você vai procurar primeiro a seção de "programação" e, nessa seção, terá apenas um conjunto de livros relacionados a esse tema. E se agora você quiser algum livro de medicina? Naturalmente procuraria a seção de "medicina" primeiro.

Esse tipo de informação da coluna busca influencia se o cliente vai ou não comprar o produto, serviço ou conteúdo que ele buscou. Porém, a grande pergunta é: como podemos trabalhar com uma coluna que não tenha apenas 2 valores?

Como podemos representar esse tipo de informação se até agora trabalhamos com apenas 2 tipos de valores (0 ou 1)? Se nós tentarmos rodar o nosso algoritmo com um texto solto, ele vai ficar doidão e não vai funcionar! E agora?

Precisamos, de alguma forma, transformar essas informações da coluna busca em variáveis numéricas com valores entre 0 e 1, isto é, verificar se o cliente só fez busca de "algoritmos" ou "Java" ou "Ruby". Consegue imaginar como podemos transformar diversos valores diferentes entre 0 e 1?

Mas por que para 0 e 1? Pois era a forma que estávamos acostumados a trabalhar anteriormente. Ou seja, se conseguirmos, resolveremos o nosso problema com um algoritmo que já implementamos!

O nosso desafio agora é reduzir esse nosso problema que contém diversos valores para uma determinada característica em 0 e 1. Será que podemos fazer isso? Vamos começar isolando essa coluna:

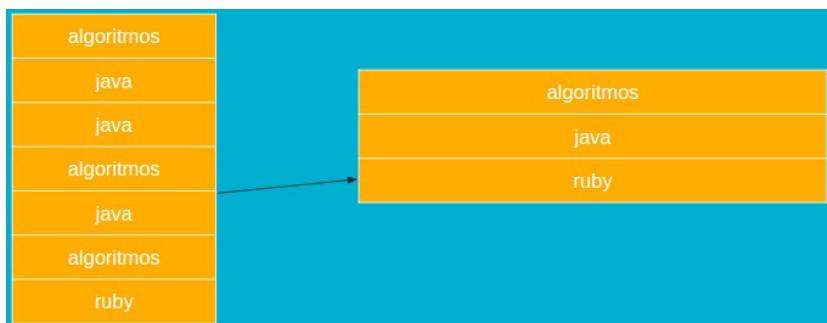


Figura 3.1: Característica "busca"

Observe que identificamos três tipos de características diferentes para essa coluna busca :

- Algoritmos
- Java
- Ruby

Cada um dos nossos clientes fez apenas uma dessas buscas; ele pesquisou ou um, ou outro. Até agora nenhuma novidade, mas, e se fizermos as seguintes perguntas:

- O cliente buscou algoritmos?

- O cliente buscou Java?
- O cliente buscou Ruby?

Para um cliente que buscou sobre "algoritmos" apenas, como podemos marcar essas perguntas?

- O cliente buscou algoritmos? Sim, acessou (1).
- O cliente buscou Java? Não acessou (0).
- O cliente buscou Ruby? Não acessou (0).

Conseguimos descobrir qual foi a busca que ele fez! Repare que convertemos a pergunta "Qual busca o cliente fez?", em três perguntas.

Vamos considerar agora um cliente que buscou por "Java" e fazer novamente essas perguntas:

- O cliente buscou algoritmos? Não acessou (0).
- O cliente buscou Java? Sim, acessou (1).
- O cliente buscou Ruby? Não acessou (0).

Observe que novamente conseguimos utilizar as três perguntas para classificar o que um outro cliente buscou. Então, o que fizemos exatamente? Demos três categorias para a coluna busca :

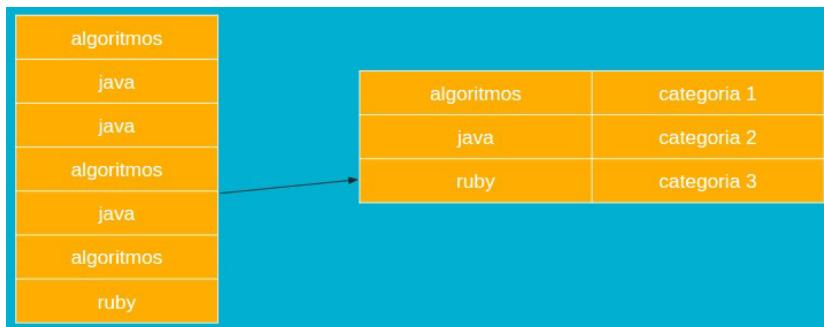


Figura 3.2: Característica "busca"

Observe que cada característica tem a sua própria categoria:

- Algoritmos - categoria 1
- Java - categoria 2
- Ruby - categoria 3

Isso significa que a nossa variável `busca` não é simplesmente uma variável a que atribuímos apenas um valor (0 ou 1), ou seja, podemos atribuir três valores para essa mesma variável. Chamamos esse tipo de **variável categórica**, pois ela possui diversas categorias.

Um outro exemplo no qual poderíamos usar esse tipo de variável seria na seguinte pergunta:

- Em qual estado você está presente neste instante?

Podemos adicionar todos os estados do Brasil como categorias dessa pergunta: São Paulo, Rio de Janeiro, Bahia, Distrito Federal, Pernambuco etc. Agora que sabemos sobre essa variável categórica, podemos separar a nossa tabela da seguinte forma:

algoritmos	
java	
java	
algoritmos	
java	
algoritmos	
ruby	

algoritmos	categoria 0
java	categoria 1
ruby	categoria 2

Figura 3.3: Característica "busca" - variável categórica

Note que agora adicionamos os valores de 0 a 2 para as categorias, pois usaremos arrays para classificar cada uma delas. Mas e na prática? Como poderíamos montar a nossa tabela para cada uma dessas categorias? Vejamos um modelo:

busca	categoria 0	categoria 1	categoria 2	
algoritmos				
java				
java				
algoritmos				
java				
algoritmos				
ruby				

algoritmos	categoria 0
java	categoria 1
ruby	categoria 2

Figura 3.4: Característica "busca" - variável categórica

Já tem uma ideia de como podemos preencher essa tabela? Vamos pegar o primeiro exemplo de um cliente que buscou por

"algoritmos":

busca	categoria 0	categoria 1	categoria 2
algoritmos	1	0	0
java			
java			
algoritmos			
java			
algoritmos			
ruby			

Figura 3.5: Característica "busca" - variável categórica

Já que ele buscou por algoritmos, significa que ele pesquisou pela "categoria 0", ou seja, marcamos a coluna "categoria 0" com 1 e as demais como 0. E para a pessoa que pesquisou sobre Java? Como marcamos?

busca	categoria 0	categoria 1	categoria 2
algoritmos	1	0	0
java	0	1	0
java			
algoritmos			
java			
algoritmos			
ruby			

Figura 3.6: Característica "busca" - variável categórica

Por enquanto, aprendemos como podemos preencher todos os clientes com categorias 0 e 1, ou seja, categorias para busca de "algoritmos" (0) e "Java" (1). Vamos preencher todas as buscas de "algoritmos" e "Java":

busca	categoria 0	categoria 1	categoria 2	
algoritmos	1	0	0	
java	0	1	0	
java	0	1	0	
algoritmos	1	0	0	
java	0	1	0	
algoritmos	1	0	0	
ruby				

Figura 3.7: Característica "busca" - variável categórica

Agora está faltando o último cliente. Este pesquisou sobre "Ruby", e agora? Categoria 0 ou 1? Lembre-se de que as categorias 0 e 1 são referentes a "algoritmos" e "Java", então, precisamos marcar apenas a "categoria 2" que se refere à busca de "Ruby":

busca	categoria 0	categoria 1	categoria 2	
algoritmos	1	0	0	
java	0	1	0	
java	0	1	0	
algoritmos	1	0	0	
java	0	1	0	
algoritmos	1	0	0	
ruby	0	0	1	

Figura 3.8: Característica "busca" - variável categórica

Veja como é fácil traduzir uma coluna categórica em **um conjunto de categorias**, ou seja, novas características baseadas nos valores distintos da variável categórica. Isso significa que a coluna busca é equivalente às três colunas das categorias (0 a 2).

A princípio, a implementação parece difícil, porém, veremos

que será bem mais fácil do que parece. Mas antes de mexermos com o nosso código, vamos modificar a nossa tabela original. Vejamos novamente a nossa tabela original, porém, dessa vez, vamos indicar quais são os tipos de variáveis de cada coluna:

home (bin)	busca (cat)	estava logado? (bin)	comprou? (bin)
1	algoritmos	1	1
1	java	0	0
0	java	1	0
1	algoritmos	1	1
0	java	0	1
1	algoritmos	0	0
1	ruby	1	1

Figura 3.9: Os clientes

Observe que todas as categorias, exceto a `busca`, são do tipo `bin` que significa binário (0 ou 1) e a coluna `busca` é do tipo `cat` que significa uma variável categórica, já que possui categorias!

Esses dados que nós temos atualmente já estão prontos para passarmos para um algoritmo de machine learning, ou existem algumas dessas colunas que precisamos traduzir? Nesse caso, a variável categórica precisa ser trabalhada! O que devemos fazer? Primeiro, precisamos converter essa única coluna para as três possíveis categorias:



busca	categoria 0	categoria 1	categoria 2
algoritmos	1	0	0
java	0	1	0
java	0	1	0
algoritmos	1	0	0
java	0	1	0
algoritmos	1	0	0
ruby	0	0	1

Figura 3.10: Característica "busca" - variável categórica

Encaixamos as novas colunas dentro da nossa tabela original:

home (bin)	busca (cat)	categoria 0	categoria 1	categoria 2	estava logado? (bin)	comprou? (bin)
1	algoritmos				1	1
1	java				0	0
0	java				1	0
1	algoritmos				1	1
0	java				0	1
1	algoritmos				0	0
1	ruby				1	1

Figura 3.11: Os clientes

Agora nós precisamos preencher cada uma dessas categorias. Você lembra que, para busca de algoritmos, adicionávamos na categoria 0? Então, fica 1, 0, 0 :

home (bin)	busca (cat)	categoria 0	categoria 1	categoria 2	estava logado? (bin)	comprou? (bin)
1	algoritmos	1	0	0	1	1
1	java				0	0
0	java				1	0
1	algoritmos	1	0	0	1	1
0	java				0	1
1	algoritmos	1	0	0	0	0
1	ruby				1	1

Figura 3.12: Os clientes

E para Java? Categoria 1! Então, 0, 1, 0 :

home (bin)	busca (cat)	categoria 0	categoria 1	categoria 2	estava logado? (bin)	comprou? (bin)
1	algoritmos	1	0	0	1	1
1	java	0	1	0	0	0
0	java	0	1	0	1	0
1	algoritmos	1	0	0	1	1
0	java	0	1	0	0	1
1	algoritmos	1	0	0	0	0
1	ruby				1	1

E a categoria do Ruby? Categoria 2, logo, 0, 0, 1 :

home (bin)	busca (cat)	categoria 0	categoria 1	categoria 2	estava logado? (bin)	comprou? (bin)
1	algoritmos	1	0	0	1	1
1	java	0	1	0	0	0
0	java	0	1	0	1	0
1	algoritmos	1	0	0	1	1
0	java	0	1	0	0	1
1	algoritmos	1	0	0	0	0
1	ruby	0	0	1	1	1

Figura 3.14: Os clientes

Agora que conseguimos transformar todas as colunas em tipos binários, não precisamos mais da coluna `busca`, ou seja, a variável categórica. Assim, jogamos fora essa coluna, e a nossa tabela atual mantém os seguintes dados:

home (bin)	categoria 0 (bin)	categoria 1 (bin)	categoria 2 (bin)	estava logado? (bin)	comprou? (bin)
1	1	0	0	1	1
1	0	1	0	0	0
0	0	1	0	1	0
1	1	0	0	1	1
0	0	1	0	0	1
1	1	0	0	0	0
1	0	0	1	1	1

Figura 3.15: Os clientes: com variáveis de mentira

home (bin)	categoria 0 (bin)	categoria 1 (bin)	categoria 2 (bin)	estava logado? (bin)	comprou? (bin)
1	1	0	0	1	1

home (bin)	categoria 0 (bin)	categoria 1 (bin)	categoria 2 (bin)	estava logado? (bin)	comprou? (bin)
1	0	1	0	0	0
0	0	1	0	1	0
1	1	0	0	1	1
0	0	1	0	0	1
1	1	0	0	0	0
1	0	0	1	1	1

Agora que todas as nossas colunas são perguntas que esperam apenas 0 ou 1, conseguimos levar para o nosso algoritmo, pois ele sabe lidar com esses valores. Note também que, a partir das cinco variáveis iniciais, vamos prever a sexta, que é saber se o cliente comprou ou não.

Criamos essa tabela com variáveis de "mentira", ou seja, variáveis que possuem um valor real, porém elas não estavam presentes na nossa pesquisa original, pois estavam presentes de uma maneira diferente.

Isso significa que elas foram abordadas e questionadas de uma maneira, porém preenchemos de outra. Chamamos esse tipo de variáveis de *dummies*.

home (bin)	categoria 0	categoria 1	categoria 2	estava logado? (bin)	comprou? (bin)
1	1	0	0	1	1
1	0	1	0	0	0
0	0	1	0	1	0
1	1	0	0	1	1
0	0	1	0	0	1
1	1	0	0	0	0
1	0	0	1	1	1

Figura 3.16: Os clientes: com dummies

O objetivo desse tipo de variável é justamente preencher um espaço de uma outra pergunta que fizemos, ou seja, as variáveis de categorias são as nossas *dummies* do tipo binário. Agora poderemos adicionar todos esses dados em um arquivo e utilizá-los no nosso algoritmo para ele prever quem vai ou não comprar.

Temos aqui uma planilha do Google Spreadsheets ([http://bit.ly/alura\\_analise\\_entrada](http://bit.ly/alura_analise_entrada)) que contém diversos dados:

home	busca	logado	comprou	frequencia	doacoes	buscas_sim_ou_nao	situacao_do_cliente	emails
0	algoritmos	1	1					
0	java	0	1					
1	algoritmos	0	1					
1	ruby	1	0					
1	java	0	1					
0	ruby	1	0					
0	algoritmos	1	1					
0	ruby	0	1					
1	algoritmos	1	1					
1	ruby	1	1					
1	algoritmos	1	1					
1	ruby	1	0					
0	java	1	1					
1	ruby	1	1					
1	algoritmos	0	1					
0	ruby	1	1					
1	algoritmos	1	1					
20	ruby	1	1					
21	1	0	1					
22	algoritmos	0	1					
23	0	1	1					

Cada coluna tem o seguinte significado:

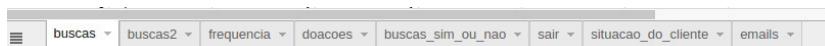
- **Home**: se o usuário acessou a home ou não (0 ou 1).
- **Busca**: qual foi o curso que o usuário buscou (algoritmos, Java ou Ruby).
- **Logado**: se o usuário estava logado ou não (0 ou 1).
- **Comprou**: se o usuário comprou (0 ou 1).

Observe que esses dados são informações que já conhecemos, ou seja, sabemos que as colunas `home`, `logado` e `comprou` são variáveis binárias, e a coluna `busca` uma variável categórica. Mas o que queremos fazer com esses dados? Trabalhar com eles no Python, certo? Porém, essas informações estão em uma planilha. Precisamos transportá-los para um formato com que o nosso algoritmo saiba lidar.

Todas as planilhas eletrônicas, seja o Google Spreadsheets, Excel, Open Office ou outras variações, vão disponibilizar alguma opção para exportarmos esses dados em algum outro formato que nos possibilite trabalhar com eles.

Se entrarmos no menu `Arquivo > Fazer download como` do Google Spreadsheets, veremos que ele nos fornece diversas formas para exportarmos esses arquivos. Para trabalharmos com o Excel, por exemplo, seria no formato `.xlsx`. Porém, estamos trabalhando com o Python. Qual era o tipo de arquivo com que trabalhamos no Python? Lembra do arquivo CSV, aquele que separava cada valor por vírgula? Importaremos a nossa planilha para esse formato utilizando a mesma opção `Arquivo > Fazer download como > Valores separados por vírgula (.csv, página atual)`.

Observe que, nessa opção, ele informa que vai fazer isso para a página atual. Mas o que isso significa exatamente? Se você der uma olhada na planilha:



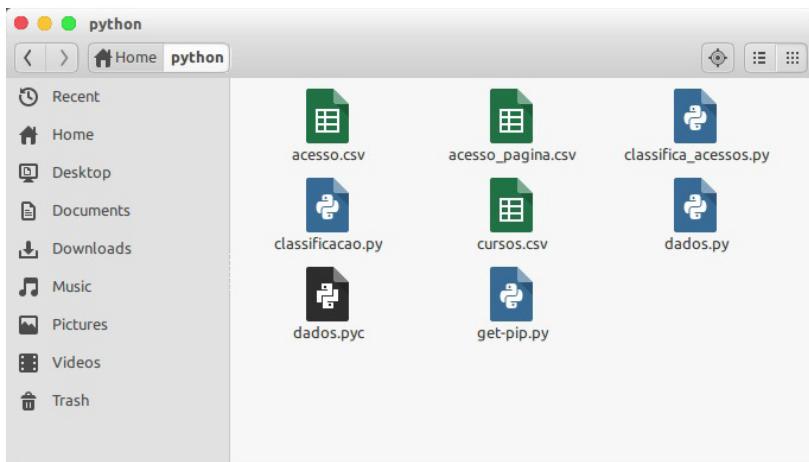
Observará que ela contém outras abas. Isso significa que, para cada aba, há uma página. Ou seja, quando escolhemos essa opção, exportamos um arquivo só com o conteúdo da aba (página) em que estamos atualmente. Ao clicar nessa opção, é realizado o download do arquivo `.csv`.

Renomeie o arquivo para `cursos.csv` para ficar mais fácil de ler. Por fim, coloque-o dentro da pasta onde você salvou os seus arquivos do Python que criamos durante o livro. Dentro do seu editor de texto, abra o arquivo `cursos.csv`:

```
1 home,busca,logado,comprou
2 0,algoritmos,1,1
3 0,java,0,1
4 1,algoritmos,0,1
5 1,ruby,1,0
6 1,ruby,0,1
7 0,ruby,1,0
8 0,algoritmos,1,1
9 0,ruby,0,1
10 1,algoritmos,1,1
11 1,ruby,1,1
12 1,algoritmos,1,1
13 1,ruby,1,0
14 0,ruby,1,1
15 0,java,1,1
```

Veja que ele foi importado sem nenhum problema. Agora já podemos fazer a leitura desse arquivo CSV que representa os acessos e buscas dos clientes aos nossos cursos.

Vamos criar um novo arquivo, porém, qual nome colocaremos? Se estamos classificando cursos, que tal `classifica_acessos.py`? Porém, já existe um arquivo com esse nome:



E agora? Qual nome daremos? Perceba que precisamos criar um padrão para que os nossos arquivos façam sentido. Podemos usar o padrão `classifica_nome_do_arquivo.py`, por exemplo, se o nosso arquivo Python ler o arquivo `acesso.csv`, ele se chamará `classifica_acessos.py`, isto é, se está lendo `cursos.csv`, ele se chamará `classifica_cursos.py`.

Entretanto, é válido pensar se os nomes dos nossos arquivos são coesos, pois o arquivo `cursos.csv` representa as buscas que o nosso cliente realizou, ou seja, é mais coerente chamarmos esse

arquivo de `buscas.csv` , pois não estamos classificando cursos. Consequentemente, o arquivo `classifica_cursos.py` se chamará `classifica_buscas.py` .

Lembre-se de que este arquivo está disponível no GitHub do livro, que pode ser encontrado em <https://github.com/guilhermesilveira/machine-learning>.

Vejamos agora o que precisamos ler do nosso arquivo `buscas.csv` :

```
home,busca,logado,comprou
0,algoritmos,1,1
0,java,0,1
1,algoritmos,0,1
1,ruby,1,0
# restante dos dados
```

Observe que precisamos ler as quatro colunas: `home` , `busca` , `logado` e `comprou` . Já fizemos algo bem similar no arquivo `dados.py` :

```
import csv

def carregar_acessos():
    X = []
    Y = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)

    for home,planos_de_cursos,contato,comprou in leitor:
```

```
dado = ([int(home),int(planos_de_cursos)
         ,int(contato)])
X.append(dado)
Y.append(int(comprou))

return X, Y
```

Se analisarmos a função `carregar_acessos()`, podemos até pensar em reutilizá-la, pois os procedimentos serão idênticos aos dessa função, mesmo que os nomes das colunas sejam diferentes. Porém, existe um pequeno detalhe que impede que reutilizemos essa função para o arquivo `buscas.csv`, que é justamente o tipo dos dados que estão sendo processados:

```
dado = ([int(home),int(planos_de_cursos)
         ,int(contato)])
X.append(dado)
Y.append(int(comprou))
```

Note que **todos os dados** estão sendo convertidos para inteiros.  
E os dados do nosso arquivo `buscas.csv`:

```
home,busca,logado,comprou
0,algoritmos,1,1
0,java,0,1
1,algoritmos,0,1
1,ruby,1,0
# restante dos dados
```

Além de números, ele também possui palavras. Ou seja, não podemos reutilizar essa mesma função. Vamos criar uma nova chamada `carregar_buscas`:

```
def carregar_buscas():
```

Ela será muito similar ao `carregar_acessos`, mas será usada para ler e tratar os dados do arquivo `buscas.csv` da forma correta:

```
def carregar_buscas():

    X = []
    Y = []

    arquivo = open('buscas.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)
```

Agora nós precisamos fazer as iterações, porém, vamos mudar os nomes para as colunas do nosso arquivo `buscas.csv` :

```
def carregar_buscas():

    X = []
    Y = []

    arquivo = open('buscas.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)

    for home, busca, logado, comprou in leitor:
```

Precisamos adicionar esses valores para o nosso `X` e `Y`. Começaremos pelo `X` :

```
# restante do código
for home, busca, logado, comprou in leitor:

    dado = ([int(home), busca, int(logado)])
    X.append(dado)
```

Apenas a coluna `busca` será utilizada como *string*, pois ela é uma palavra. Agora precisamos adicionar o valor do `Y` :

```
# restante do código
for home, busca, logado, comprou in leitor:

    dado = ([int(home), busca, int(logado)])
    X.append(dado)
    Y.append(int(comprou))
```

Por fim, retornamos o X e o Y , e a nossa função carregar\_buscas() ficará da seguinte maneira:

```
def carregar_buscas():

    X = []
    Y = []

    arquivo = open('buscas.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)

    for home, busca, logado, comprou in leitor:

        dado = ([int(home), busca, int(logado)])
        X.append(dado)
        Y.append(int(comprou))

    return X, Y
```

Agora que temos a nossa função que carrega as nossas buscas, podemos ler esses dados a partir do arquivo classifica\_buscas.py . Lembra que fizemos a mesma coisa no classifica\_acessos.py ?

```
from dados import carregar_acessos
X,Y = carregar_acessos()
```

Faremos o mesmo para o arquivo classifica\_buscas.py , porém com a função carregar\_buscas :

```
from dados import carregar_buscas
X,Y = carregar_buscas()
```

Vamos verificar se o X e o Y estão corretos? Começaremos imprimindo o X :

```
from dados import carregar_buscas
X,Y = carregar_buscas()
print(X)
```

Se rodarmos o nosso `classifica_buscas.py` :

```
> python3 classifica_buscas.py
[[0, 'algoritmos', 1], [0, 'java', 0], [1, 'algoritmos', 0], [1,
'ruby', 1], ...]
```

Verificando no arquivo `buscas.csv` :

```
home,busca,logado,comprou
0,algoritmos,1,1
0,java,0,1
1,algoritmos,0,1
1,ruby,1,0
# Restante dos dados
```

Funcionou conforme o esperado! E o nosso `Y`? Será que está correto também? Vejamos:

```
from dados import carregar_buscas
X,Y = carregar_buscas()
print(Y)
```

Rodando o código novamente:

```
> python3 classifica_buscas.py
[1, 1, 1, 0, ...]
```

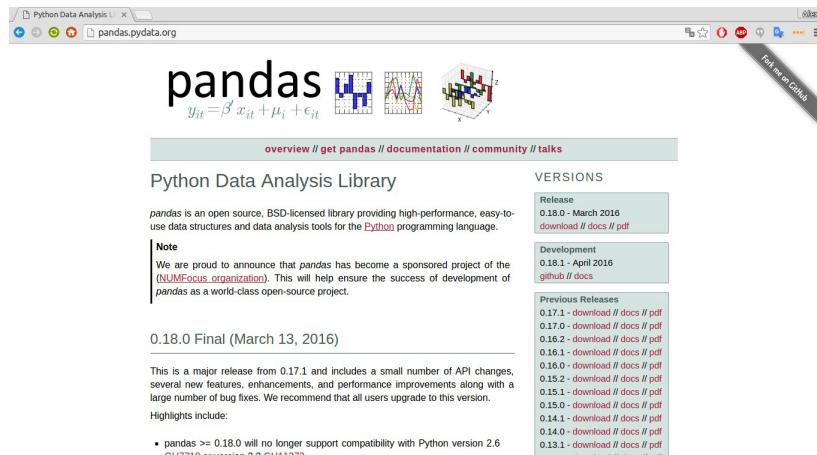
Os valores do `Y` também estão corretos. Porém, todos os nossos dados estão corretos? Observe que ainda estamos lidando com uma variável do tipo *string* e o nosso algoritmo não sabe lidar com esse tipo de variável.

Lembra que a coluna `busca` é uma variável categórica? Isso significa que precisamos convertê-la para três colunas, ou seja, as três categorias possíveis para essa coluna.

Mas como faremos isso no Python? Para fazer essa conversão, tradução de uma variável categórica, usaremos a biblioteca chamada **Pandas** (*Python Data Analysis Library*).

## 3.1 INSTALANDO O PANDAS

Vamos acessar a página do Pandas (<http://pandas.pydata.org/>):



Para instalar o Pandas, poderíamos entrar no link `get pandas` e seguir as instruções. Porém, já que temos o `pip`, então vamos utilizá-lo para instalar o Pandas por meio do comando:

```
> sudo pip3 install pandas
```

Caso você já o tenha, atualize a biblioteca usando o comando:

```
> sudo pip3 install pandas --upgrade
```

É importante manter a versão mais atualizada, pois, dependendo da *API* que usaremos, podem existir variações que podem causar incompatibilidade, ou simplesmente não funcionar conforme o esperado por estar depreciada. **É muito importante se atentar a isso.**

Usaremos o Pandas para fazer a leitura dos nossos dados, pois

ele é uma biblioteca para leitura e análise de dados. Observe as nossas funções que realizam a leitura dos nossos dados:

```
import csv

def carregar_acessos():
    X = []
    Y = []

    arquivo = open('acesso.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)

    for home,planos_de_cursos,contato,comprou in leitor:
        dado = ([int(home),int(planos_de_cursos),
                 int(contato)])
        X.append(dado)
        Y.append(int(comprou))

    return X, Y

import csv

def carregar_buscas():
    X = []
    Y = []

    arquivo = open('buscas.csv', 'r')
    leitor = csv.reader(arquivo)

    next(leitor)

    for home,busca,logado,comprou in leitor:
        dado = ([int(home), busca, int(logado)])
        X.append(dado)
        Y.append(int(comprou))

    return X, Y
```

Podemos notar que, todas as vezes que queremos ler um arquivo, precisamos criar uma função nova e informar qual o tipo de valor que eu estou lendo para cada coluna. Parece repetitivo e, se nos atentarmos mais, perceberemos que isso faz parte de análise de dados. Porém, para analisarmos os dados, a primeira coisa que precisamos fazer é justamente a leitura.

Temos diversas formas para ler dados. Estivemos utilizando a biblioteca padrão `csv`, mas podemos usar uma biblioteca específica para análise de dados, ou seja, usar o Pandas para ler arquivos CSV também! Como fazemos isso? Simplesmente importando o Pandas no nosso arquivo:

```
from dados import carregar_buscas
X,Y = carregar_buscas()
print(Y)

import pandas
```

Podemos também abreviar o Pandas como *python data analysis* utilizando `pd`:

```
from dados import carregar_buscas
X,Y = carregar_buscas()
print(Y)

import pandas as pd
```

E como fazemos para ler um arquivo utilizando essa biblioteca? Podemos usar a função `read_csv()` informando o arquivo que queremos ler por parâmetro:

```
from dados import carregar_buscas
X,Y = carregar_buscas()
print(Y)

import pandas as pd
pd.read_csv('buscas.csv')
```

E como pegamos os dados? Retornando para uma variável chamada `dados` :

```
from dados import carregar_buscas
X,Y = carregar_buscas()
print(Y)

import pandas as pd
dados = pd.read_csv('buscas.csv')
```

Vamos verificar como estão esses dados? Primeiro adicionaremos um `print` :

```
from dados import carregar_buscas
X,Y = carregar_buscas()
print(Y)

import pandas as pd
dados = pd.read_csv('buscas.csv')
print(dados)
```

E agora removeremos o nosso código anterior que fazia a leitura por meio da nossa função `carregar_buscas` :

```
import pandas as pd
dados = pd.read_csv('buscas.csv')
print(dados)
```

Rodando o nosso algoritmo:

```
python classifica_buscas.py
    home      busca  logado  comprou
0       0  algoritmos      1      1
1       0        java      0      1
2       1  algoritmos      0      1
3       1        ruby      1      0
4       1        ruby      0      1
5       0        ruby      1      0
6       0  algoritmos      1      1
...
999     0        ruby      1      0
```

Observe que o Pandas conseguiu ler sem nenhum problema. Além disso, ele conseguiu identificar que a primeira linha refere-se ao cabeçalho. Veja também que ele mostrou de uma forma bem clara o que é cada informação. E mais, ele nos informou quantas linhas de dados nós temos, nesse caso, 1.000 linhas.

Na realidade, em vez de devolver apenas um array puro, bem básico e sem uma estrutura bem elaborada, assim como fizemos, o Pandas nos devolveu um objeto bem completo com todas as informações bem estruturadas. Todos esses objetos que o Pandas nos devolveu são chamados de *data frame*. Logo, em vez de chamarmos o retorno de `dados`, podemos identificá-lo como o nosso *data frame*, ou na abreviação, `df`:

```
import pandas as pd
df = pd.read_csv('buscas.csv')
print(df)
```

É válido lembrar que, na prática, podemos utilizar soluções mais básicas e padrões do próprio Python. Porém, na maioria das situações, é mais interessante usarmos bibliotecas específicas que já resolvem a mesma situação, mas de uma forma mais inteligente.

Perceba que começamos com uma maneira manual, justamente para entender como o processo de leitura de um arquivo de dados (no formato CSV) funciona por trás dos panos. Entretanto, agora que já aprendemos todo o processo, podemos delegar toda essa tarefa para alguém que já faça tudo isso por nós e de uma maneira bem mais inteligente — neste caso, o Pandas.

Lemos o arquivo utilizando o Pandas, porém ainda não resolvemos o grande desafio, que é solucionar a coluna `busca`. Você lembra que, quando nós fazíamos a leitura dos nossos dados,

separávamos todos os que representavam o X e o Y. Mas e agora? Como faremos isso usando o Pandas? Será que se pedirmos para ele a primeira coluna, como se fosse um array, ele funcionaria? Vejamos:

```
import pandas as pd
df = pd.read_csv('buscas.csv')
print(df[0])
```

Vamos testar o nosso código:

```
> python3 classifica_buscas.py
Traceback (most recent call last):
  File "classifica_buscas.py", line 3, in <module>
    print(df[0])
  File "/usr/local/lib/python3.6/dist-packages/pandas/core/frame.py", line 1992, in __getitem__
    return self._getitem_column(key)
  File "/usr/local/lib/python3.6/dist-packages/pandas/core/frame.py", line 1999, in _getitem_column
    return self._get_item_cache(key)
  File "/usr/local/lib/python3.6/dist-packages/pandas/core/generics.py", line 1345, in _get_item_cache
    values = self._data.get(item)
  File "/usr/local/lib/python3.6/dist-packages/pandas/core/internals.py", line 3225, in get
    loc = self.items.get_loc(item)
  File "/usr/local/lib/python3.6/dist-packages/pandas/indexes/base.py", line 1878, in get_loc
    return self._engine.get_loc(self._maybe_cast_indexer(key))
  File "pandas/index.pyx", line 137, in pandas.index.IndexEngine.get_loc (pandas/index.c:4027)
  File "pandas/index.pyx", line 157, in pandas.index.IndexEngine.get_loc (pandas/index.c:3891)
  File "pandas/hashtable.pyx", line 675, in pandas.hashtable.PyObjectHashTable.get_item (pandas/hashtable.c:12408)
  File "pandas/hashtable.pyx", line 683, in pandas.hashtable.PyObjectHashTable.get_item (pandas/hashtable.c:12359)
KeyError: 0
```

Opa, aconteceu algo aqui! Que tal pedirmos para ele usando o nome da coluna? Vamos tentar pedindo a coluna home :

```
import pandas as pd
df = pd.read_csv('buscas.csv')
print(df['home'])
```

Rodando o nosso código:

```
> python3 classifica_buscas.py
0      0
1      0
2      1
3      1
4      1
5      0
6      0
...
999    0
Name: home, dtype: int64
```

Agora sim ele imprimiu todos os dados da coluna. Perceba que, no final, o data frame nos informa o nome e o tipo desses dados. Neste caso, estamos pegando a coluna `home` e o tipo é `int64`.

Mas o valor do nosso `x` é mais de uma coluna, ou seja, são as colunas: `home` , `busca` , `logado` . Vamos tentar separar cada coluna por vírgula:

```
import pandas as pd
df = pd.read_csv('buscas.csv')
print(df['home', 'busca', 'logado'])
```

Se testarmos o nosso código:

```
> python3 classifica_buscas.py
Traceback (most recent call last):
  File "classifica_buscas.py", line 3, in <module>
    print(df['home', 'busca', 'logado'])
  File "/usr/local/lib/python3.6/dist-packages/pandas/core/frame.py", line 1992, in __getitem__
    return self._getitem_column(key)
  File "/usr/local/lib/python3.6/dist-packages/pandas/core/frame.py", line 1999, in _getitem_column
    return self._get_item_cache(key)
```

```
File "/usr/local/lib/python3.6/dist-packages/pandas/core/generi
c.py", line 1345, in _get_item_cache
    values = self._data.get(item)
File "/usr/local/lib/python3.6/dist-packages/pandas/core/intern
als.py", line 3225, in get
    loc = self.items.get_loc(item)
File "/usr/local/lib/python3.6/dist-packages/pandas/indexes/bas
e.py", line 1878, in get_loc
    return self._engine.get_loc(self._maybe_cast_indexer(key))
File "pandas/index.pyx", line 137, in pandas.index.IndexEngine.
get_loc (pandas/index.c:4027)
File "pandas/index.pyx", line 157, in pandas.index.IndexEngine.
get_loc (pandas/index.c:3891)
File "pandas/hashtable.pyx", line 675, in pandas.hashtable.PyOb
jectHashTable.get_item (pandas/hashtable.c:12408)
File "pandas/hashtable.pyx", line 683, in pandas.hashtable.PyOb
jectHashTable.get_item (pandas/hashtable.c:12359)
KeyError: ('home', 'busca', 'logado')
```

Eita, mais um erro. O que será que aconteceu? Será que ele não consegue imprimir mais de uma coluna?

Um detalhe importante quando queremos imprimir mais de uma coluna ao usarmos o Pandas é que devemos colocar todas dentro de um array:

```
import pandas as pd
df = pd.read_csv('buscas.csv')
print(df[['home', 'busca', 'logado']])
```

Agora, se tentarmos novamente rodar o código:

```
> python3 classifica_buscas.py
      home     busca  logado
0        0  algoritmos      1
1        0        java      0
2        1  algoritmos      0
3        1        ruby      1
4        1        ruby      0
5        0        ruby      1
6        0  algoritmos      1
...
...
```

```
999      0      ruby      1
```

```
[1000 rows x 3 columns]
```

As colunas são impressas. Entretanto, ainda existe um pequeno detalhe. Analisando a coluna `busca`, vemos que os valores ainda são as variáveis categóricas. Lembra que precisamos usar as categorias da coluna `busca`? Ou seja, precisamos pegar os *dummies* dela.

Como faremos isso? Podemos pedir para Pandas devolver os *dummies* do nosso `X`:

```
import pandas as pd

df = pd.read_csv('buscas.csv')
X = df[['home', 'busca', 'logado']]
Xdummies = pd.get_dummies(X)
print(Xdummies)
```

Testando o nosso código:

```
> python3 classifica_buscas.py
    home  logado  busca_algoritmos  busca_java  busca_ruby
0      0      1                  1          0          0
1      0      0                  0          1          0
2      1      0                  1          0          0
3      1      1                  0          0          1
4      1      0                  0          0          1
5      0      1                  0          0          1
6      0      1                  1          0          0
...
999     0      1                  0          0          1
```

```
[1000 rows x 5 columns]
```

Conseguimos os *dummies* do `X`. Mas e o `Y`? Precisamos pegar os *dummies* do `Y` sendo que a coluna `comprou` é uma com valores binários? Vamos alterar o código para pegar os *dummies*

do Y e imprimi-los:

```
import pandas as pd
df = pd.read_csv('buscas.csv')
X = df[['home', 'busca', 'logado']]
Y = df['comprou']
Xdummies = pd.get_dummies(X)
Ydummies = pd.get_dummies(Y)

print(Ydummies)
```

Testando o nosso código, temos o seguinte resultado:

```
> python classifica_buscas.py
      0      1
0      0      1
1      0      1
2      0      1
3      1      0
4      0      1
5      1      0
6      0      1
...
999   1      0

[1000 rows x 2 columns]
```

Mas observe que o algoritmo MultinomialNB recebe 2 arrays do Python, ou seja, ele não recebe data frames, mas sim uma lista de dados! Podemos conferir neste instante que temos data frames:

```
> print(type(Xdummies_df))
<class 'pandas.core.frame.DataFrame'>
```

Precisamos, de alguma forma, transformar os nossos data frames e arrays de dados. Se analisarmos melhor as nossas variáveis que representam os data frames:

```
import pandas as pd
df = pd.read_csv('buscas.csv')

X = df[['home', 'busca', 'logado']]
```

```
Y = df['comprou']
Xdummies_df = pd.get_dummies(X)
Ydummies_df = pd.get_dummies(Y)
```

Note que X , Xdummies e Y , Ydummies são nomes ruins, pois não deixam claro se estamos trabalhando com data frames ou arrays. É sempre importante deixarmos bem claro o significado das nossas variáveis, por isso, alteraremos os nomes de ambas para indicar que são data frames:

```
import pandas as pd
df = pd.read_csv('buscas.csv')

X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']
Xdummies_df = pd.get_dummies(X)
Ydummies_df = pd.get_dummies(Y)
```

Repare que, no caso de Y, temos um só *dummie*, então não precisamos do `get_dummies` :

```
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']
Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df
```

Agora nós precisamos pegar os dois data frames e transformá-los em arrays. Para isso, usaremos o `values` do data frame, ou seja, pegaremos os valores dos nossos data frames:

```
Xdummies_df.values
Ydummies_df.values
```

Quem são esses valores? Eles são os valores reais do X e do Y , ou seja, podemos atribuir esses valores para as variáveis X e Y :

```
import pandas as pd
df = pd.read_csv('buscas.csv')
X_df = df[['home', 'busca', 'logado']]
```

```
Y_df = df['comprou']

Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values
```

Vamos imprimir os valores de `X` e `Y`, e verificar o resultado.  
Primeiro imprimiremos o `X`:

```
# restante do código  
print(X)
```

O resultado será:

```
> python3 classifica_buscas.py  
[[0 1 1 0 0]  
 [0 0 0 1 0]  
 [1 0 1 0 0]  
 ...  
 [0 1 0 1 0]  
 [1 0 1 0 0]  
 [0 1 0 0 1]]
```

Agora com o Y:

```
# restante do código  
print(Y)
```

O resultado será:

```
python classifica_buscas.py
[1 1 1 0 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 1 1
 0 1 0 1
1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1
1 1 1 1
1 0 1 1 1 1 1 0 1 0 0 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1
0 0 1 1
1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
1 1 1 1
0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 0
```



Conseguimos pegar os valores do nosso X e Y , isto é, podemos utilizá-los para treinar e testar o nosso algoritmo. Se verificarmos o código do `classifica_acessos.py` :

```
from dados import carregar_acessos
X,Y = carregar_acessos()

treino_dados = X[:90]
treino_marcacoes = Y[:90]

teste_dados = X[-9:]
teste_marcacoes = Y[-9:]

# restante do código
```

Inicialmente separamos os dados de treino e os de teste. Nesse exemplo, usamos os 90 primeiros para treino e os 9 últimos para teste. Faremos o mesmo com os novos dados, porém, vamos verificar a quantidade de dados que temos, por exemplo, no nosso Y :

```
>>> import pandas as pd
>>> df = pd.read_csv('buscas.csv')
>>> X_df = df[['home', 'busca', 'logado']]
>>> Y_df = df['comprou']
>>>
>>> Xdummies_df = pd.get_dummies(X_df)
>>> Ydummies_df = Y_df
>>>
>>> X = Xdummies_df.values
>>> Y = Ydummies_df.values
>>>
>>> len(Y)
1000
```

Espera um pouco! Não temos apenas 99 linhas, mas sim 1.000. Isso significa que não queremos treinar com as 90 primeiras linhas assim como fizemos anteriormente, e sim treinar com os primeiros 90% dos nossos dados e testar com o restante. Então, o que

devemos fazer é criar uma variável que representará a quantidade de dados para treino:

```
# restante do código  
tamanho_de_treino = 0.9 * len(Y)
```

Vamos verificar o valor dessa variável:

```
>>> import pandas as pd  
>>> df = pd.read_csv('buscas.csv')  
>>> X_df = df[['home', 'busca', 'logado']]  
>>> Y_df = df['comprou']  
>>>  
>>> Xdummies_df = pd.get_dummies(X_df)  
>>> Ydummies_df = Y_df  
>>>  
>>> X = Xdummies_df.values  
>>> Y = Ydummies_df.values  
>>>  
>>> tamanho_de_treino = 0.9 * len(Y)  
>>>  
>>> tamanho_de_treino  
900.0  
>>>
```

Conseguimos os nossos primeiros 90% dados, ou seja, os primeiros 900. A partir desse parâmetro, vamos pegar agora os primeiros 90% de X, que são os nossos dados:

```
# restante do código  
tamanho_de_treino = 0.9 * len(Y)  
  
treino_dados = X[:tamanho_de_treino]
```

Agora precisamos pegar as marcações de treino, ou seja, os primeiros 90% das linhas:

```
# restante do código  
tamanho_de_treino = 0.9 * len(Y)  
  
treino_dados = X[:tamanho_de_treino]  
treino_marcacoes = Y[:tamanho_de_treino]
```

Pegamos tanto os dados quanto as marcações de treino. Logo, precisamos apenas pegar os dados e as marcações de teste. Como fazemos isso? Olhando novamente o código do `classifica_acessos.py`:

```
# restante do código
```

```
teste_dados = X[-9:]
teste_marcacoes = Y[-9:]
```

Percebemos que são os últimos 10% dos dados, então precisamos definir o tamanho de teste como 10%. Poderíamos simplesmente fazer:

```
tamanho_de_teste = 0.1 * len(Y)
```

Porém, já que obtivemos o tamanho de treino, podemos simplesmente pegar o tamanho do `Y` e subtrair pela variável `tamanho_de_treino`, pois  $100\% - 90\%$  (`tamanho_de_treino`) resultará nos 10% que precisamos! Além disso, se um dia alterarmos o tamanho de treino para, por exemplo, 80%, ele calculará automaticamente o tamanho de teste, ou seja, modificaremos em apenas um ponto do código.

```
# restante do código
```

```
tamanho_de_treino = 0.9 * len(Y)
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]
```

Para melhorar mais ainda o nosso código, podemos extrair o valor `0.9` para uma variável indicando que se refere à nossa porcentagem:

```
porcentagem_treino = 0.9
```

Dessa forma, podemos reutilizar essa porcentagem em qualquer lugar do código, caso seja necessário. Vamos verificar se está funcionando?

```
>>> import pandas as pd
>>> df = pd.read_csv('buscas.csv')
>>> X_df = df[['home', 'busca', 'logado']]
>>> Y_df = df['comprou']
>>>
>>> Xdummies_df = pd.get_dummies(X_df)
>>> Ydummies_df = Y_df
>>>
>>> X = Xdummies_df.values
>>> Y = Ydummies_df.values
>>>
>>> porcentagem_treino = 0.9
>>>
>>> tamanho_de_treino = porcentagem_treino * len(Y)
>>> tamanho_de_teste = len(Y) - tamanho_de_treino
>>>
>>> tamanho_de_teste
100.0
```

Ele retorna os 10% que precisamos. Podemos retornar os nossos dados de teste, mas como fazemos isso? Lembra de que precisamos dos últimos 10% dos dados? Então, faremos o mesmo, porém utilizando a variável `tamanho_de_teste` :

```
# restante do código

porcentagem_treino = 0.9

tamanho_de_treino = porcentagem_treino * len(Y)
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

teste_dados = X[-tamanho_de_teste:]
teste_marcacoes = Y[-tamanho_de_teste:]
```

Testamos o código anterior, mas temos um erro ao tentar fatiar (*slide*) o nosso `treino_dados` :

```
treino_dados = X[:tamanho_de_treino]  
TypeError: slice indices must be integers or None or have an __in  
dex__ method
```

Uma vez que `tamanho_de_treino` é um número com ponto flutuante, não podemos cortar o array na posição dele (mesmo que ele seja 90.0). Precisamos transformar esse número em inteiro:

```
# restante do código  
  
porcentagem_treino = 0.9  
  
tamanho_de_treino = int(porcentagem_treino * len(Y))  
tamanho_de_teste = len(Y) - tamanho_de_treino  
  
treino_dados = X[:tamanho_de_treino]  
treino_marcacoes = Y[:tamanho_de_treino]  
  
teste_dados = X[-tamanho_de_teste:]  
teste_marcacoes = Y[-tamanho_de_teste:]
```

Agora vamos verificar os nossos dados e marcações. Primeiro, começaremos pelos dados e marcações de treino:

```
>>> import pandas as pd  
>>> df = pd.read_csv('buscas.csv')  
>>> X_df = df[['home', 'busca', 'logado']]  
>>> Y_df = df['comprou']  
>>>  
>>> Xdummies_df = pd.get_dummies(X_df)  
>>> Ydummies_df = Y_df  
>>>  
>>> X = Xdummies_df.values  
>>> Y = Ydummies_df.values  
>>>  
>>> porcentagem_treino = 0.9  
>>>  
>>> tamanho_de_treino = int(porcentagem_treino * len(Y))  
>>> tamanho_de_teste = len(Y) - tamanho_de_treino
```





```
, 1, 1, 1,  
    1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1  
, 1, 1, 1,  
    1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1  
, 1, 1, 1,  
    1, 1, 1])  
>>>
```

Agora com os dados de teste:

```
>>> teste_dados
array([[0, 0, 0, 1, 0],
       [1, 0, 0, 1, 0],
       [1, 1, 1, 0, 0],
       [1, 0, 0, 1, 0],
       [1, 1, 1, 0, 0],
       [1, 1, 1, 0, 0],
       [0, 1, 0, 0, 1],
       [1, 1, 0, 1, 0],
       [0, 0, 1, 0, 0],
       [1, 1, 0, 1, 0],
       [1, 1, 1, 0, 0],
       [1, 0, 1, 0, 0],
       [0, 1, 1, 0, 0],
       [1, 1, 1, 0, 0],
       [0, 0, 0, 0, 1],
       [1, 1, 1, 0, 0],
       [1, 0, 0, 0, 1],
       [0, 0, 0, 1, 0],
       [1, 0, 0, 0, 1],
       [0, 1, 0, 1, 0],
       [0, 1, 0, 0, 1],
       [1, 1, 1, 0, 0],
       [0, 1, 0, 0, 1],
       [1, 1, 1, 0, 0],
       [1, 1, 1, 0, 0],
       [0, 1, 0, 1, 0],
       [1, 0, 0, 1, 0],
       [0, 1, 0, 1, 0],
       [1, 1, 1, 0, 0],
       [1, 1, 1, 0, 0],
       [0, 1, 0, 1, 0],
       [0, 1, 0, 1, 0],
       [1, 0, 0, 0, 1],
       [1, 0, 0, 0, 1],
       [0, 1, 0, 1, 0],
       [1, 1, 1, 0, 0],
       [1, 1, 1, 0, 0],
       [1, 1, 0, 0, 1],
       [0, 1, 0, 1, 0],
       [1, 0, 0, 1, 0],
       [0, 1, 0, 1, 0],
       [1, 0, 0, 0, 1],
       [1, 0, 0, 0, 1],
       [0, 0, 1, 0, 0]]
```

```
[1, 0, 1, 0, 0],  
[1, 0, 0, 0, 1],  
[1, 0, 0, 0, 1],  
[1, 0, 0, 0, 1],  
[1, 1, 0, 1, 0],  
[1, 1, 1, 0, 0],  
[1, 0, 1, 0, 0],  
[0, 1, 0, 1, 0],  
[0, 0, 1, 0, 0],  
[1, 1, 0, 0, 0],  
[1, 1, 0, 0, 1],  
[0, 1, 0, 1, 0],  
[1, 1, 0, 1, 0],  
[1, 0, 1, 0, 0],  
[0, 1, 1, 0, 0],  
[0, 0, 0, 1, 0],  
[0, 1, 1, 0, 0],  
[1, 0, 0, 1, 0],  
[0, 1, 0, 1, 0],  
[1, 1, 0, 0, 1],  
[0, 0, 0, 0, 1],  
[1, 0, 0, 1, 0],  
[0, 0, 0, 1, 0],  
[0, 0, 0, 0, 1],  
[0, 0, 0, 1, 0],  
[0, 0, 1, 0, 0],  
[0, 0, 1, 0, 1],  
[1, 0, 0, 0, 1],  
[0, 0, 1, 0, 0],  
[0, 0, 1, 0, 0],  
[1, 1, 0, 0, 1],  
[1, 1, 1, 0, 0],  
[0, 1, 0, 0, 1],  
[0, 1, 0, 1, 0],  
[1, 1, 0, 0, 1],  
[1, 1, 0, 0, 0],  
[0, 1, 0, 1, 0],  
[1, 1, 0, 0, 1],  
[1, 0, 1, 0, 0],  
[0, 1, 0, 1, 0],  
[1, 0, 0, 0, 1],  
[1, 0, 0, 0, 0],  
[1, 0, 0, 0, 1]
```

Conseguimos pegar tanto os dados de treino quantos os de teste. Qual é o próximo passo? De acordo com os nossos dados de treino e de teste, queremos criar o nosso modelo. Você se lembra de como criamos o modelo? Se verificarmos o código `classifica_acessos.py`:

```
from sklearn.naive_bayes import MultinomialNB
```

```
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)
```

Observe que poderemos reutilizar esse código, ou seja, faremos um *copy and paste*. E o resultado? Será que poderemos fazer o mesmo? Vejamos no `classifica_acessos.py`:

```
resultado = modelo.predict(teste_dados)
```

Também não mudou nada, ou seja, podemos reusá-lo! Por fim, vamos verificar como estão sendo feitos os cálculos para a taxa de acerto:

```
diferencias = resultado - teste_marcacoes

acertos = [d for d in diferencias if d == 0]
total_de_acertos = len(acertos)
total_de_elementos = len(teste_dados)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print(taxa_de_acerto)
print(total_de_elementos)
```

Observe que **todo o código** usado no arquivo `classifica_acessos.py` para treinar e testar o modelo e calcular o resultado e a taxa de acerto não muda. Ou seja, poderemos copiar todo o código e adicioná-lo ao nosso arquivo `classifica_buscas.py`. O nosso código final fica assim:

```
import pandas as pd
df = pd.read_csv('buscas.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']

Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values
```

```
porcentagem_treino = 0.9

tamanho_de_treino = int(porcentagem_treino * len(Y))
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

teste_dados = X[-tamanho_de_teste:]
teste_marcacoes = Y[-tamanho_de_teste:]

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)

resultado = modelo.predict(teste_dados)

diferencias = resultado - teste_marcacoes

acertos = [d for d in diferencias if d == 0]
total_de_acertos = len(acertos)
total_de_elementos = len(teste_dados)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print(taxa_de_acerto)
print(total_de_elementos)
```

Vamos testar e verificar se está funcionando:

```
> python3 classifica_buscas.py
82.0
100
```

O que o nosso algoritmo fez? Dessa vez, ele realizou um treino com 900 elementos e treinou com 100 elementos que ele nunca viu. Dentre esses 100 elementos, ele conseguiu prever 82%!

É válido ressaltar que, de acordo com esses dados, que são de um sistema real, o nosso algoritmo conseguiu acertar 82% das vezes utilizando apenas três características variáveis que descrevem

o que o usuário fez. Poderíamos também adicionar mais variáveis, porém, apenas com essa quantidade de informação o nosso algoritmo foi capaz de acertar 82%. Dessa vez, os dados são reais, então vem aquela mesma pergunta: 82% é bom ou ruim?

Note que essa pergunta é bem difícil de responder, pois não temos nenhum método de classificação **diferente** para que possamos comparar com o nosso código. Se o resultado fosse 100%, com certeza não restaria dúvida de que o algoritmo é perfeito, mas o que precisamos saber agora é quão longe o nosso algoritmo consegue chegar com dados reais.

Temos de levar em consideração que, em cada área, haverá dados diferentes; os resultados provavelmente serão diferentes. Mas precisamos focar justamente em saber o quão bom é o nosso algoritmo para os determinados números com que estamos trabalhando atualmente.

## 3.2 RESUMINDO

Neste capítulo, nós realizamos um teste para verificar quais clientes iriam ou não comprar em um site. Vimos que o nosso algoritmo acertou 82% utilizando dados reais e, a partir deles, podemos entrar em contato com os clientes que não compraram para tentar entender a necessidade deles, ou propor alguma coisa que seja de seu interesse. Podemos concluir que 82% das vezes acertamos se o cliente iria ou não comprar.

Além disso, neste mesmo capítulo, fizemos alguns ajustes no nosso algoritmo. Dessa vez, usamos uma biblioteca nova, o Pandas, para realizarmos a leitura dos nossos arquivos CSV que

continham os nossos dados, de uma forma bem simplificada.

Além de realizar a leitura, ele nos devolveu um conjunto de informações bem mais completo e inteligente do que o array bobo que estávamos fazendo na unha, conhecido como *data frame*, que nos permite rodar algoritmos de *data analysis* (como por exemplo, machine learning). Esses data frames são um conjunto de dados que nos fornece diversos recursos, como pegar uma coluna ou diversas colunas, por meio de um(ns) cabeçalho(s), facilitando, e muito, a nossa vida.

Outro ponto importante visto neste capítulo é que mudamos o método de leitura dos nossos dados. Porém, o método de criação e treino do modelo utilizando o algoritmo `MultinomialNB` e o cálculo da taxa de acerto foram os mesmos que usamos no arquivo `classifica_acessos.py` !

Por fim, nós chegamos a mais uma questão muito importante, que é verificar se o resultado do nosso algoritmo foi bom ou ruim para esse cenário (esses dados). Veremos com mais detalhes essa questão no próximo capítulo.

## CAPÍTULO 4

# O PROBLEMA DO SUCESSO E O ALGORITMO BURRO

Quando rodamos o nosso algoritmo `classifica_buscas.py`, ele conseguiu prever 82% dos casos e, a princípio, ficamos felizes por um número "grande", afinal, podemos considerar que essa porcentagem significa que ele acertará quase sempre. Porém, esse "quase sempre" pode ser que seja apenas uma percepção nossa, ou seja, o que nós achamos.

Perceba que é muito delicado afirmar que um resultado foi ou não satisfatório, pois precisamos de algum critério que avalie o nosso resultado. Em outras palavras, alguma maneira que consiga nos dizer se 82%, para esse algoritmo que utilizamos para analisar e classificar os dados, é bom ou não.

Um exemplo de como poderíamos considerar se o nosso algoritmo é bom ou não seria rodá-lo em um dia e obter o resultado de 50%. Porém, ao rodar no dia seguinte, o resultado fosse 82%. Considerando essa situação, com certeza o nosso algoritmo ficou melhor do que ele era antes. Mas, atualmente, como podemos dizer que o nosso algoritmo é bom ou não? Fizemos algum tipo de comparação? Por enquanto, não. Então, quais seriam as possíveis formas que poderíamos fazer para

compararmos o nosso algoritmo?

Nesse caso, poderíamos comparar com o passado, conforme o exemplo anterior, ou então, com apenas esse valor de 82%. Mas perceba que não é possível realizar uma comparação com apenas um valor, isto é, sem nenhum histórico. De fato, nós precisamos de alguma outra maneira que permita uma comparação mesmo que esse algoritmo tenha sido executado pela primeira vez. Uma das abordagens seria a **forma mais simples possível de classificar elementos**.

Já pensou na maneira de classificação mais simples de todas? Podemos verificar esse método com o seguinte exemplo. Temos dois tipos de características dentro do nosso sistema, que são:

- Usuário que compra;
- Usuário que não compra.

Para esse caso, qual seria o algoritmo mais simples de todos para classificação? Podemos utilizar um outro exemplo bem similar. Temos diversos animais entre porcos e cachorros, e precisamos definir quais destes são porcos ou cachorros. Consegue imaginar no código uma solução mais simples possível? É bem mais fácil do que parece. Simplesmente responda a mesma coisa, ou melhor, entre usuário que comprou ou não, responda apenas que comprou; entre porcos e cachorros, responda apenas porcos!

Uma simulação desse algoritmo seria:

- Qual é o 1º animal? Ele é um porco.
- E o 2º animal? Ele é um porco.
- E agora o 3º animal? Ele é um porco.
- E este outro? Ele é um porco.

Esse é o algoritmo de classificação mais simples que existe e pode ser aplicado para qualquer situação de classificação. E se fosse para classificar entre *spam* e não *spam*? Simples! Responderíamos apenas *spam*. Perceba que nem temos o trabalho de verificar as características. Para implementar o classificador mais simples e burro do mundo, basta responder **sempre** a mesma coisa.

Vimos o quanto simples esse algoritmo é, porém, a questão é: "o quanto eficiente ele é?". Em outras palavras: "o quanto ele erra?". Vamos dar uma olhada? Começaremos pelo nosso `buscas.csv`:

```
home,busca,logado,comprou
0,algoritmos,1,1
0,java,0,1
1,algoritmos,0,1
1,ruby,1,0
1,ruby,0,1
0,ruby,1,0
0,algoritmos,1,1
...
...
```

Se o nosso algoritmo chutasse apenas 1, o que ele acertaria? Apenas todos os elementos que são classificados como 1. Isto é, todos que forem 1, ele vai acertar; e todos que forem 0, ele vai errar. Porém, o quanto bom é esse algoritmo bem simples que chuta apenas 1 valor?

Vamos testá-lo. Abra o interpretador do Python e cole o código do `classifica_buscas.py` até o momento em que pegamos o Y:

```
>>> import pandas as pd
>>> df = pd.read_csv('buscas.csv')
>>> X_df = df[['home', 'busca', 'logado']]
>>> Y_df = df['comprou']
>>>
>>> Xdummies_df = pd.get_dummies(X_df)
```

```
>>> Ydummies_df = Y_df  
>>>  
>>> X = Xdummies_df.values  
>>> Y = Ydummies_df.values  
>>>
```

Lembra do valor do  $\gamma$ ? Vejamos novamente:



```
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0  
, 1, 1, 1,  
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1  
, 1, 1, 1,  
    1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0  
, 1, 1, 1,  
    1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0])  
>>>
```

Vários uns e zeros. O nosso classificado chutará 1 para todos os elementos dele. Quanto será que ele acertará? Podemos verificar a quantidade de números 1 simplesmente somando os valores de `Y` por meio da função `sum()`:

```
>>> sum(Y)  
832  
>>>
```

Isso significa que temos 832 *uns* nos nossos dados, ou seja, 832 pessoas que compraram. Lembra de como verificamos quantos dados nós temos no total? É simples, `len` do `Y`:

```
>>> len(Y)  
1000  
>>>
```

1.000 dados no total. Porém, agora vem a questão: "o quão bom é o nosso algoritmo **extremamente burro**?":

```
>>> sum(Y)  
832  
>>> len(Y)  
1000  
>>>
```

Isso mesmo, o algoritmo que chuta 1 para todos os dados consegue acertar 83,20% dos casos. E o nosso algoritmo anterior? Quanto ele acerta?

```
> python3 classifica_buscas.py
```

82.0  
100

O nosso algoritmo `classifica_buscas.py` acerta 82% com os mesmos dados. Vale a pena rodar um algoritmo tão complexo para acertar 82% sendo que bastava chutarmos "sim" para todos os casos que eu acertaria 83,20% das vezes? Aparentemente, a maneira como estamos rodando esse algoritmo não está tão boa com esses dados.

Os dados foram alterados de propósito para que o algoritmo mais simples e burro possível tivesse um resultado maior do que o nosso algoritmo esperto, que faz todos os procedimentos para entender a lógica que acontece por trás de todos os dados que ele lê. Mas por qual motivo eu forcei os nossos dados para que um algoritmo bem simples, ou seja, que chuta apenas um valor para todos os dados, tivesse um resultado melhor ao qual implementamos?

É justamente para verificarmos que, em situações em que o nosso algoritmo de classificação que implementamos (o algoritmo complexo) tiver um resultado inferior ao algoritmo mais burro possível (que chuta o mesmo valor para todos os dados), o nosso algoritmo é ruim, pois ele obteve um resultado inferior ao que nem ao menos faz uma análise de dados. Isso significa que usar o algoritmo que chuta o mesmo valor para todos os dados é a maneira mais básica de todas que podemos fazer para compararmos o desempenho do nosso algoritmo.

É exatamente por esse motivo que eu alterei os dados, para que o algoritmo sem nenhuma lógica tivesse um resultado melhor sobre o que analisa e tenta se adaptar de acordo com os novos

dados que recebe. Portanto, repare que, no mundo real, podemos cair em duas situações:

- Quando o algoritmo que chuta o mesmo valor para todos os dados obtém um resultado melhor que o algoritmo que implementamos.

Nessa situação, de fato, não podemos utilizar o nosso algoritmo, pois se ele obteve um resultado inferior ao que chuta a mesma coisa, significa que ele é pior, ou seja, use o que chuta o mesmo valor para tudo.

A outra situação seria:

- Quando o algoritmo que implementamos obtém um resultado melhor que o algoritmo que chuta a mesma coisa para todos.

Nesse caso, podemos utilizar esse algoritmo, pois ele obteve um resultado melhor.

Observe que, para afirmarmos se o nosso algoritmo que implementamos é bom ou não, é imprescindível realizar uma comparação com um outro algoritmo. No nosso exemplo, inicialmente, usamos o algoritmo mais simples de todos, pois se ele apresentar um resultado inferior ao que não utiliza nenhum tipo de lógica, significa que o nosso algoritmo é realmente muito ruim.

É válido lembrar que isso depende também dos dados que estão sendo usados, pois, com esses dados manipulados, o resultado chutando apenas 1 foi de 83,20%. E se os nossos chutes fossem apenas 0? Qual resultado teríamos? Podemos pegar a quantidade de zeros por meio do tamanho do  $\gamma$  menos a soma do

Y :

```
>>> len(Y) - sum(Y)  
168
```

168 é a quantidade total de zeros no nosso arquivo de dados. Se para esses dados os nossos chutes fossem apenas 0, acertaríamos 16,80%. Nesse instante, podemos pensar que o classificador que acertou 83% não é tão bom quanto pensamos, pois, se escolhêssemos 0 em vez de 1, compararíamos 82% e 16,80%. Isto é, sabendo que 82% é maior que 16,80%, chegaríamos à conclusão de que o nosso algoritmo é realmente muito bom.

Esse é um pequeno detalhe com que precisamos tomar cuidado, pois, quando usamos esse algoritmo mais básico, precisamos saber ambos os resultados, tanto escolhendo apenas o valor 1 quanto apenas 0. Em outras palavras, dados os resultados entre todos que chutam apenas 1 ou apenas 0, pegamos o maior deles. Nesse caso, quando escolhemos o valor 1, o resultado foi de 83,20% e, quando escolhemos 0, foi de 16,80%, portanto, escolhemos todos que chutam 1 como base, pois o resultado dele foi maior.

## 4.1 IMPLEMENTANDO O ALGORITMO BASE

Mas como podemos implementar isso no nosso código? Primeiro, nós precisamos separar a quantidade total de uns e zeros, então começaremos pelo total de uns:

```
import pandas as pd  
df = pd.read_csv('buscas.csv')  
X_df = df[['home', 'busca', 'logado']]  
Y_df = df['comprou']
```

```
Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

# a eficácia do algoritmo que chuta tudo 0 ou 1
acerto_de_um = sum(Y)
```

Agora precisamos pegar o total de zeros:

```
# restante do código

# a eficácia do algoritmo que chuta tudo 0 ou 1
acerto_de_um = sum(Y)
acerto_de_zero = len(Y) - acerto_de_um
```

Mas e agora? Qual dessas duas variáveis vamos utilizar? A maior delas, pois o nosso algoritmo base precisa usar o maior resultado para o mesmo chute. Como podemos pegar a maior variável entre elas?

Simples, no Python, podemos utilizar a função `max()` enviando duas variáveis como parâmetro, e ele retorna o valor maior:

```
# restante do código

# a eficácia do algoritmo que chuta tudo 0 ou 1
acerto_de_um = sum(Y)
acerto_de_zero = len(Y) - acerto_de_um
max(acerto_de_um, acerto_de_zero)
```

Porém, nós precisamos imprimi-lo. Você se lembra de qual era o nome da variável que usamos para imprimir a taxa de acerto do nosso algoritmo? Vejamos:

```
# restante do código
taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos
```

```
print(taxa_de_acerto)
```

É a `taxa_de_acerto`, então retornaremos o resultado do `max()` para uma variável chamada `taxa_de_acerto_base`, pois se refere à taxa de acerto que o nosso algoritmo base obteve:

```
# restante do código
```

```
# a eficácia do algoritmo que chuta tudo 0 ou 1
acerto_de_um = sum(Y)
acerto_de_zero = len(Y) - acerto_de_um
taxa_de_acerto_base = max(acerto_de_um, acerto_de_zero)
```

Agora, precisamos imprimir a nossa taxa de acerto base:

```
# restante do código
```

```
# a eficácia do algoritmo que chuta tudo 0 ou 1
acerto_de_um = sum(Y)
acerto_de_zero = len(Y) - acerto_de_um
taxa_de_acerto_base = max(acerto_de_um, acerto_de_zero)
print("Taxa de acerto base: %d" % taxa_de_acerto_base)
```

Rodando o nosso arquivo `classifica_buscas.py`:

```
> python3 classifica_buscas.py
Taxa de acerto base: 832
82.0
100
```

Observe que a impressão da taxa de acerto do nosso algoritmo base foi um número inteiro, porém, quando representamos uma taxa, precisamos mostrar em percentual. Nesse caso, deveria nos apresentar 83,20%. Como podemos fazer isso? É simples, basta dividirmos o valor do `max()` pelo total de elementos, ou seja, `len` de `Y`:

```
# restante do código
taxa_de_acerto_base = max(acerto_de_um, acerto_de_zero) / len(Y)
```

Então, multiplicamos por 100.0 para apresentar o ponto flutuante, ou seja, o percentual:

```
# restante do código  
taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zero) /  
len(Y)
```

Ao rodarmos novamente o nosso algoritmo, temos:

```
> python3 classifica_buscas.py  
Taxa de acerto base: 83  
82.0  
100
```

Observe que o resultado foi de 83, porém, não foi impresso o ponto flutuante. Por que será? Repare como estamos imprimindo a variável `taxa_de_acerto_base`:

```
# restante do código  
print("Taxa de acerto base: %d" % taxa_de_acerto_base)
```

Quando imprimimos `%d`, significa que estamos imprimindo variáveis do tipo inteiro, por isso o ponto flutuante não foi exibido. Para imprimirmos uma variável do tipo *float*, basta alterarmos de `%d` para `%f`:

```
# restante do código  
  
# a eficácia do algoritmo que chuta tudo 0 ou 1  
acerto_de_um = sum(Y)  
acerto_de_zero = len(Y) - acerto_de_um  
taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zero) /  
len(Y)  
print("Taxa de acerto base: %f" % taxa_de_acerto_base)
```

Testando novamente o nosso algoritmo:

```
> python3 classifica_buscas.py  
Taxa de acerto base: 83.200000  
82.0  
100
```

Agora a nossa taxa de acerto base foi impressa como o esperado. Além disso, vamos indicar também o que significa os 82.0, isto é, a taxa de acerto do nosso algoritmo:

```
# restante do código
print("Taxa de acerto do algoritmo: %f" % taxa_de_acerto)
```

Agora a impressão é apresentada da seguinte forma:

```
Taxa de acerto base: 83.200000
Taxa de acerto do algoritmo: 82.000000
100
```

## 4.2 CALCULANDO A QUANTIDADE DE ZEROS E UNS COM O DATA FRAME

Repare o cálculo que realizamos para extrair a quantidade de uns e zeros:

```
acerto_de_um = sum(Y)
acerto_de_zero = len(Y) - acerto_de_um
taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zero) / len(Y)
```

A princípio, pode ser que a compreensão não seja tão fácil, pois estamos lidando com o `len` e uma operação de subtração. Além dessa forma que fizemos, podemos também utilizar os próprios data frames para realizar esses cálculos por nós! Vamos fazer um teste por meio do interpretador do Python:

```
>>> import pandas as pd
>>> df = pd.read_csv('buscas.csv')
>>> X_df = df[['home', 'busca', 'logado']]
>>> Y_df = df['comprou']
>>>
>>> Xdummies_df = pd.get_dummies(X_df)
>>> Ydummies_df = Y_df
>>>
```

```

>>> X = Xdummies_df.values
>>> Y = Ydummies_df.values
>>> Y
array([1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1,
       1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 0, 1, 0,
...
       1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 0, 1, 0,
>>>

```

Sabemos que o nosso `Y` é uma sequência de zeros e uns. E se pedirmos todos os valores de `Y` que são iguais a 1? Será que dá certo? Vejamos:

```

>>> Y==1
array([ True,  True,  True, False,  True, False,  True,  True,  True,
       True,  True, False,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True, False,  True,  True,  True,  True,  True,
       True,  True, False], dtype=bool)
>>>

```

Observe que agora ele nos devolveu valores booleanos, ou seja, `True` s e `False` s. Isso significa que todos os `True` s são os valores iguais a 1, e **quaisquer valores diferentes de 1** são os `False` s. E se quiséssemos saber quais são os zeros? Simples, bastaria pedir todos os valores de `Y` que são iguais a 0:

```

>>> Y==0
array([False, False, False,  True, False,  True, False, False,
       False, False,  True, False, False, False, False, False, False,
       False, False,  True, False, False, False, False, False, False]

```

```
lse,  
...  
    False, False, True, False, True, True, False, False, Fa  
lse, True], dtype=bool)  
>>>
```

Observe que agora todos aqueles `True`s que apareceram quando pedimos todos os valores iguais a 1 se tornaram `False`, pois, nesse instante, pedimos todos os valores que são iguais a 0. Logo, todos os `True`s anteriores são os valores iguais a 0, e qualquer valor diferente de 0 é considerado como `False`.

Porém, nós precisamos pegar todos os uns ou então todos os zeros. Como fazemos isso apenas com esse recurso do data frame? Simples, basta pedir a ele:

```
>>> Y[Y==1]  
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
, 1, 1, 1,  
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
, 1, 1, 1,  
...  
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
, 1, 1, 1,  
      1, 1, 1])  
>>>
```

Observe que agora temos todos os valores que são uns. E se quiséssemos os zeros? Faríamos a mesma coisa, porém, pedindo todos os valores iguais a 0:

```
>>> Y[Y==0]  
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0,  
...  
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0,  
      0, 0, 0, 0, 0, 0])  
>>>
```

Note que agora conseguimos separar todos os uns ( `Y[Y==1]` ) e zeros ( `Y[Y==0]` ), porém, o que precisamos de verdade é pegar a quantidade de uns e zeros. Para isso, basta pegarmos o `len` deles, ou seja, a quantidade de cada um:

```
>>> len(Y[Y==1])  
832  
>>> len(Y[Y==0])  
168
```

Como podemos ver, foram retornados os 832 uns e 168 zeros. No nosso código, basta apenas substituirmos a nossa implementação, que calcula a quantidade de zeros e uns, por essa que é uma forma mais comum para esse tipo de cálculo:

```
# restante do código  
acerto_de_um = len(Y[Y==1])  
acerto_de_zero = len(Y[Y==0])
```

É comum utilizarmos essa forma, pois, à primeira vista, é fácil compreender que estamos pegando a quantidade de `Y` para todos os valores que são iguais a 1 ( `len[Y[Y==1]]` ) e a quantidade de `Y` para todos os valores que são iguais a 0 ( `len[Y[Y==0]]` ). Além disso, poderíamos ir mais além.

Por exemplo, suponhamos que queremos saber todas as características dos usuários que não compraram, como poderíamos fazer utilizando esse mesmo recurso? Simples, pedimos ao nosso data frame `X` todos os valores que o `Y` é igual a 0, ou seja, que não compraram:

```
>>> X[Y==0]  
array([[1, 1, 0, 0, 1],  
       [0, 1, 0, 0, 1],  
       [1, 1, 0, 0, 1],  
       [1, 0, 0, 0, 1],  
       ...
```

```
>>> [0, 0, 0, 0, 1],  
[0, 1, 0, 0, 1])
```

Essas são as características de todos os usuários que não compraram. Mas como isso funciona? É exatamente como os exemplos anteriores:

- Primeiro, ele pega todos os valores de `Y` e verifica quais são zeros. Para todos os zeros, ele retorna `True` e, para todos os que não forem, ele retorna `False` (`Y==0`).
- Depois, pedimos todos os valores de `x`, porém, apenas para todos os valores de `Y` que forem `True` (`x[Y==0]`).

É justamente dessa forma que ele consegue retornar todas as características dos usuários que não compraram no nosso site. Repare que o Pandas é muito mais do que o que vimos até agora, já que ele nos dá diversas possibilidades para trabalharmos com dados. Veremos todas as suas funcionalidades de acordo com a necessidade que tivermos.

## 4.3 LIDANDO COM SIM E NÃO

Até agora, quando analisamos os nossos dados, todas as nossas marcações, a coluna `comprou` foi entre zeros e uns:

```
home,busca,logado,comprou  
0,algoritmos,1,1  
0,java,0,1  
1,algoritmos,0,1  
1,ruby,1,0  
1,ruby,0,1  
0,ruby,1,0  
0,algoritmos,1,1  
0,ruby,0,1  
1,algoritmos,1,1
```

Entretanto, nem sempre recebemos as informações com binários, mas sim com valores como: **sim e não** ou **true e false**. Isso porque, quando recebemos esses tipos de dados, é bem provável que seja preenchido com uma **informação diferente de binário**, mas que tem o mesmo significado, como é o caso de **sim e não**. Levando em consideração essa situação, vamos substituir todos os valores de `Y` que são **uns e zeros** por **sim e não**. O arquivo ficaria assim:

```
home,busca,logado,comprou
0,algoritmos,1,sim
0,java,0,sim
1,algoritmos,0,sim
1,ruby,1,nao
1,ruby,0,sim
0,ruby,1,nao
0,algoritmos,1,sim
0,ruby,0,sim
1,algoritmos,1,sim
1,ruby,1,sim
1,algoritmos,1,sim
1,ruby,1,nao
0,ruby,1,sim
...
0,ruby,1,nao
```

Não se preocupe, o arquivo com a coluna `comprou` modificada está disponível para download no diretório `capítulo4` dos arquivos que baixou na *Introdução* (<https://github.com/guilhermesilveira/machine-learning>).

Já sabemos o que fazer quando nos deparamos com uma variável que não era 0 ou 1, como foi o caso da coluna `busca`. Isto é, pegamos os dummies que criavam uma coluna a mais para cada um dos possíveis valores para essa mesma variável (nesse caso, foram três: `algoritmos`, `Java` e `Ruby`).

Mas o que vai acontecer com o Y se temos apenas sim e nao ? Precisamos testar, certo? Executaremos o nosso código parte por parte no interpretador do Python:

```
>>> import pandas as pd
>>> df = pd.read_csv('buscas.csv')
>>> X_df = df[['home', 'busca', 'logado']]
>>> Y_df = df['comprou']
>>>
>>> Xdummies_df = pd.get_dummies(X_df)
>>> Ydummies_df = Y_df
>>>
>>> X = Xdummies_df.values
>>> Y = Ydummies_df.values
>>> Y
array(['sim', 'sim', 'sim', 'nao', 'sim', 'nao', 'sim', 'sim', 'sim',
       'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',
       ...
       'nao', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',
       'sim', 'sim', 'nao', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim',
       'sim', 'nao'], dtype=object)
>>>
```

Os valores do nosso Y são sim s e nao s, e por enquanto não temos nenhum problema. Mas e este trecho de código?

```
# restante do código
acerto_de_um = len(Y[Y==1])
acerto_de_zero = len(Y[Y==0])
taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zero) / len(Y)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)
```

Note que atualmente estamos buscando todos os valores que são iguais a uns e zeros. Será que funciona? Vamos verificar:

```
>>> acerto_de_um = len(Y[Y==1])
>>> acerto_de_zero = len(Y[Y==0])
>>> taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zer
```

```
o) / len(Y)
>>> print("Taxa de acerto base: %f" % taxa_de_acerto_base)
Taxa de acerto base: 0.000000
>>>
```

Como podemos ver, o resultado foi 0, pois não existem mais valores entre zeros e uns, ou seja, agora os valores são sim s e nao s! Isso é bem fácil de corrigir. Basta alterarmos a comparação de uns para sim s:

```
# restante do código
acerto_de_um = len(Y[Y=='sim'])
```

E a comparação de zeros para nao s:

```
# restante do código
acerto_de_um = len(Y[Y=='sim'])
acerto_de_zero = len(Y[Y=='nao'])
```

Vamos verificar se isso funciona? Novamente, no interpretador do Python:

```
>>> acerto_de_um = len(Y[Y=='sim'])
>>> acerto_de_zero = len(Y[Y=='nao'])
>>> taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zero) / len(Y)
>>> print("Taxa de acerto base: %f" % taxa_de_acerto_base)
Taxa de acerto base: 83.200000
>>>
```

Perceba que está funcionando como o esperado. Mas e esta parte, em que pegamos a quantidade para treino e teste?

```
# restante do código
porcentagem_treino = 0.9

tamanho_de_treino = porcentagem_treino * len(Y)
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]
```

```
teste_dados = X[-tamanho_de_teste:]
teste_marcacoes = Y[-tamanho_de_teste:]
```

Vamos verificar:

```
>>> porcentagem_treino = 0.9
>>>
>>> tamanho_de_treino = porcentagem_treino * len(Y)
>>> tamanho_de_teste = len(Y) - tamanho_de_treino
>>>
>>> treino_dados = X[:tamanho_de_treino]
>>> treino_marcacoes = Y[:tamanho_de_treino]
>>>
>>> teste_dados = X[-tamanho_de_teste:]
>>> teste_marcacoes = Y[-tamanho_de_teste:]
>>>
```

Aparentemente, nenhum problema. Como ficou o momento em que utilizamos o nosso algoritmo `MultinomialNB` ?

```
from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)
```

Especificamente no momento em que pedimos para ele treinar, ou seja, o método `fit` , será que as variáveis podem ser `sim` ou `nao` ? Vamos tentar rodar o nosso arquivo `classifica_buscas.py` :

```
python classifica_buscas.py
Taxa de acerto base: 83.200000
Traceback (most recent call last):
  File "classifica_buscas.py", line 34, in <module>
    diferenca = resultado - teste_marcacoes
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Perceba que ele conseguiu treinar com as informações de `sim` e `nao` s, porém deu um erro justamente na linha:

```
# restante do código
```

```
diferencias = resultado - teste_marcacoes
```

Por que isso aconteceu? Vamos dar uma olhada nos valores das variáveis `resultado` e `teste_marcacoes`. Começaremos pela `resultado`:

```
resultado
array(['sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',
       'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim'],
      dtype='|S3')
```

Agora a variável `teste_marcacoes`:

```
>>> teste_marcacoes
array(['sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',
       'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim'],
      dtype=object)
```

Observe que, nesse instante, ambas as variáveis possuem valores do tipo string, e não é possível realizar operações matemáticas entre strings. Para esse caso, ocorreu uma subtração de string, por exemplo: `sim - sim` ou `sim - nao`. Como o

nosso algoritmo faria essa operação entre textos?

Não tem como! Por isso, ele apresentou esse erro. Como poderíamos resolver esse problema? Existem duas abordagens que podemos realizar:

- Transformar todos os sim s e nao s em zeros e uns.
- Comparar as duas variáveis ( resultado e teste\_marcacoes ) e verificar quais possuem os valores iguais.

Vamos verificar como seria a segunda abordagem:

```
>>> resultado==teste_marcacoes
array([ True,  True,  True,  True,  True,  True,  True,  T
rue,
       True,  True,  True,  True,  True,  True, False,  True,  T
rue,
...
       False,  True,  True,  True, False,  True,  True,  True,  T
rue,
       True,  True, False,  True, False,  True,  True,  True,  T
rue, False], dtype=bool)
>>>
```

Veja que o Python nos devolveu um array de True s e False s, isto é, quais são os valores sim e quais não são. E como mudaremos isso no código? Simplesmente alteraremos a subtração para a comparação entre a variável resultado e a variável teste\_marcacoes :

```
# restante do código
diferencas = resultado == teste_marcacoes
```

Se verificarmos novamente o valor da variável diferencas , teremos:

```
>>> diferencas = resultado == teste_marcacoes
```

```
>>> diferencias
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  False,  True,  True,
       True,  False,  True,  True,  True,  True,  True,  True,  True,
       ...,
       True,  False,  True,  True,  False,  False,  False,  True,  Fa
lse,
       False,  True,  True,  True,  False,  True,  True,  True,  True,
       True,  True,  False,  True,  False,  False,  True,  True,  T
rue,  False], dtype=bool)
>>>
```

Agora sim conseguimos pegar todos os valores que são `sim` e os que não são. Mas e o restante do código?

```
# restante do código
acertos = [d for d in diferencias if d == 0]
total_de_acertos = len(acertos)
total_de_elementos = len(teste_dados)
```

A variável `total_de_acertos` é a quantidade de `True`s existentes na variável `diferencias`. Poderíamos iterar em cada uma das linhas e verificar quais valores são `True`, porém, no Python, `True` e `False` são respectivamente 1 e 0. Em outras palavras, se somarmos a variável `diferencias`, teremos a quantidade total de `True`s:

```
>>> sum(diferencias)
82
```

Por meio dessa abordagem, nem precisamos mais da variável `acertos` ou de fazer qualquer tipo de iteração. Basta, apenas, no lugar de `len(acertos)`, adicionarmos a soma da variável `diferencias`:

```
# restante do código
```

```
diferencas = resultado == teste_marcacoes

total_de_acertos = sum(diferencas)
total_de_elementos = len(teste_dados)
```

Perceba que, com essa pequena alteração, o nome da variável `diferencas` já não faz tanto sentido, pois, a partir de agora, ele representa os nossos acertos, então podemos renomeá-la para `acertos`:

```
# restante do código
acertos = resultado == teste_marcacoes

total_de_acertos = sum(acertos)
total_de_elementos = len(teste_dados)
```

Por fim, temos o cálculo de total de acertos:

```
taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos
print("Taxa de acerto do algoritmo: %f" % taxa_de_acerto)
print(total_de_elementos)
```

Aparentemente, sem nenhum detalhe. O arquivo final com as alterações realizadas fica da seguinte forma:

```
import pandas as pd

df = pd.read_csv('buscas.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']
Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

acerto_de_um = len(Y[Y=='sim'])
acerto_de_zero = len(Y[Y=='nao'])
taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zero) /
len(Y)
```

```
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

porcentagem_treino = 0.9

tamanho_de_treino = int(porcentagem_treino * len(Y))
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

teste_dados = X[-tamanho_de_teste:]
teste_marcacoes = Y[-tamanho_de_teste:]

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)

resultado = modelo.predict(teste_dados)

acertos = resultado == teste_marcacoes

total_de_acertos = sum(acertos)
total_de_elementos = len(teste_dados)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print(taxa_de_acerto)
print(total_de_elementos)
```

Vamos executar o arquivo e verificar qual é o resultado:

```
> python3 classifica_buscas.py
Taxa de acerto base: 83.200000
Taxa de acerto do algoritmo: 82.000000
100
```

Como podemos ver, o nosso algoritmo está funcionando como esperado, mas, dessa vez, com valores do tipo string entre sim e nao nas marcacoes (na coluna comprou ).

## 4.4 UTILIZANDO COLLECTIONS DO PYTHON

Atualmente, se recebermos um arquivo que não esteja com valores binários e sim com apenas dois valores distintos que sejam textos (strings) — por exemplo, `sim` e `nao` —, podemos também utilizar o nosso algoritmo que suporta esse tipo de dado. Entretanto, existe um detalhe **muito importante**: para que o nosso algoritmo consiga suportar valores do tipo string, precisaremos **sempre nos atentar** ao momento em que pegamos essas informações:

```
# restante do código
acerto_de_um = len(Y[Y=='sim'])
acerto_de_zero = len(Y[Y=='nao'])
```

Repare que, mesmo dando suporte às variáveis do tipo string, ainda assim estamos vinculados aos valores delas, ou melhor, `sim` e `nao`. Caso os valores sejam distintos de `sim` e `nao`, será necessário modificar esse trecho de código.

Além de utilizar apenas o array do Python, como o nosso `X` e `Y`, podemos também usar uma lista:

```
>>> import pandas as pd
>>> df = pd.read_csv('buscas.csv')
>>> X_df = df[['home', 'busca', 'logado']]
>>> Y_df = df['comprou']
>>>
>>> Xdummies_df = pd.get_dummies(X_df)
>>> Ydummies_df = Y_df
>>>
>>> X = Xdummies_df.values
>>> Y = Ydummies_df.values
>>> list(Y)
['sim', 'sim', 'sim', 'nao', 'sim', 'nao', 'sim', 'sim', 'sim',
 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',
 'sim', ... 'sim', 'sim', 'sim', 'nao', 'sim', 'nao', 'nao', 'sim',
 'sim', 'sim', 'nao']
```

```
>>>
```

A lista é uma estrutura de dados que nos provê diversos recursos, como a função `count` que permite contar a quantidade de elementos existentes dentro da lista por meio de um parâmetro. Em outras palavras, essa função nos devolve a quantidade de `sim`s ou `nao`s existentes dentro dela:

```
>>> list(Y).count('sim')
832
>>> list(Y).count('nao')
168
>>>
```

Em vez de utilizar a função `len` diretamente com os arrays para pegar a quantidade de `sim`s e `nao`s, poderíamos usar as listas:

```
# restante do código
acerto_de_um = list(Y).count('sim')
acerto_de_zero = list(Y).count('nao')
```

Mas perceba que ainda não resolvemos totalmente o nosso problema, pois ainda precisamos informar qual é o tipo de informação existente dentro do arquivo. Para essa situação, precisamos de alguém que seja capaz de contar todos os elementos do nosso array, e também nos informe quais são eles. Como será que podemos fazer para implementar esse tipo de contador?

A partir do Python 2.7, podemos importar do `collections` o `Counter`, um contador que faz justamente o que precisamos:

```
>>> from collections import Counter
```

Então, podemos pedir para ele contar o nosso `Y`:

```
>>> from collections import Counter
>>> Counter(Y)
```

```
Counter({'sim': 832, 'nao': 168})  
>>>
```

Observe que o Counter contou todos os valores de Y e nos informou a quantidade de sim s e nao s como um dicionário. Em outras palavras, para cada valor distinto que ele encontra, ele nos informa a quantidade que encontrou.

Mas e agora? Como podemos usar o Counter no nosso código? Primeiro, precisamos iterar sobre todos os valores do Counter e, para isso, utilizamos a função .values() :

```
>>> Counter(Y).values()  
<dictionary-valueiterator object at 0x7fb26bdccd08>
```

Veja que ele retorna um objeto iterador. Agora, basta pedirmos o maior valor por meio da função max :

```
>>> Counter(Y).values()  
<dictionary-valueiterator object at 0x7fb26bdccd08>  
>>> max(Counter(Y).values())  
832
```

Observe que agora nós conseguimos pegar o maior dos valores entre sim e nao sem ao menos saber se realmente as informações correspondem a sim s e nao s. Em outras palavras, poderiam ser outros textos diferentes de sim s e nao s, como yes ou not , que funcionaria da mesma forma!

Porém, ainda falta modificarmos o nosso código. Começaremos importando o Counter para ele:

```
import pandas as pd  
from collections import Counter  
  
df = pd.read_csv('buscas.csv')  
X_df = df[['home', 'busca', 'logado']]  
Y_df = df['comprou']
```

```
Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values
# restante do código
```

Agora, precisamos criar o nosso contador:

```
# restante do código
Counter(Y)
```

Então, é necessário pegar os valores do Counter , pedindo os .values() :

```
# restante do código
Counter(Y).values()
```

Tendo todos os valores do Counter , pedimos o maior entre eles e atribuímos a uma variável chamada acerto\_base , pois o maior entre eles é justamente o nosso acerto\_base :

```
# restante do código
acerto_base = max(Counter(Y).values())
```

Vejamos agora o restante do código:

```
acerto_base = max(Counter(Y).values())

acerto_de_um = list(Y).count('sim')
acerto_de_zero = list(Y).count('nao')
taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zero) / len(Y)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)
```

Observe que não precisamos mais das variáveis acerto\_de\_um e acerto\_de\_zero , pois já estamos pegando o valor máximo entre os valores do Y ( max(Counter(Y).values()) ) e retornando para a variável

acerto\_base . Isto é, podemos simplesmente excluir as variáveis acerto\_de\_um e acerto\_de\_zero e, no cálculo de maior entre elas, adicionamos a variável acerto\_base . O cálculo da taxa de acerto base fica da seguinte forma:

```
# restante do código
acerto_base = max(Counter(Y).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(Y)
```

O nosso arquivo final fica assim:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('buscas.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']

Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

acerto_base = max(Counter(Y).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(Y)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

porcentagem_treino = 0.9

tamanho_de_treino = porcentagem_treino * len(Y)
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

teste_dados = X[-tamanho_de_teste:]
teste_marcacoes = Y[-tamanho_de_teste:]

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)
```

```
resultado = modelo.predict(teste_dados)

acertos = resultado == teste_marcacoes

total_de_acertos = sum(acertos)
total_de_elementos = len(teste_dados)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print("Taxa de acerto do algoritmo: %f" % taxa_de_acerto)
print(total_de_elementos)
```

Vamos testar novamente o nosso arquivo `classifica_buscas.py`:

```
> python3 classifica_buscas.py
Taxa de acerto base: 83.200000
Taxa de acerto do algoritmo: 82.000000
100
```

Como podemos ver, o nosso algoritmo está funcionando perfeitamente. Além disso, fizemos uma implementação mais genérica. Ele consegue contar a quantidade de dados e nos retorna o maior, independentemente de qual informação esteja contida dentro do arquivo de dados.

## 4.5 RESUMINDO

Neste capítulo, iniciamos a questão do sucesso do algoritmo, isto é, concluir se foi bom ou ruim dado o seu resultado. Vimos que, para sabermos se o nosso resultado foi um sucesso ou um fracasso de verdade, precisamos comparar o nosso algoritmo. Uma das possíveis formas é compararmos com um algoritmo base, ou seja, aquele que chuta o mesmo valor para cada elemento independente de suas características.

Se realizarmos o teste e o nosso algoritmo for pior do que o

algoritmo base, podemos concluir que o nosso não é tão bom quanto imaginamos. Podemos utilizar esse método como o primeiro critério de avaliação para identificar se o algoritmo está bom ou não.

Um segundo critério é verificar se o número obtido é relevante para o seu negócio. Por exemplo, se o seu teste é para detectar terremotos na Terra e o algoritmo acerta em 51% das vezes, e acaba obtendo um resultado melhor do que um algoritmo que chuta apenas sim ou não, perceba que 51% das vezes para gerar um alerta de terremotos parece não fazer muito sentido, pois metade das vezes o alerta será inútil. Será que esse tipo de resultado é bom para esse negócio?

Repare que, além de apenas avaliar o resultado com o algoritmo base, precisamos verificar se o número faz sentido ou não para o negócio também.

Uma outra situação seria descobrir quais serão os alunos que terão dificuldade nas provas no final do ano. A princípio, podemos concluir que nosso algoritmo é bom, porém, pode ser que ele também preveja muitos alunos que não terão dificuldades, logo, provavelmente gastaremos mais tempo dos professores com alunos que não têm dificuldade de verdade.

Perceba que, além de considerar se o resultado do algoritmo foi maior que o algoritmo base, é importante entender se ele faz sentido ou não. Então veja que você também precisa avaliar o resultado e saber dizer se 51%, 82% ou qualquer outro valor faz ou não sentido para o seu objetivo.

Além disso, em todos os outros testes, utilizamos apenas zeros

e uns, mas, dessa vez, vimos que podemos também trabalhar com palavras, isto é, sim s e não s. Porém, tivemos de realizar uma refatoração no nosso código para que ele fosse o mais genérico possível e, assim, aceitasse quaisquer tipos de valores. Para isso, usamos APIs mais genéricas do Python, como no caso do Counter .

Por fim, vimos que, quando queremos comparar dois algoritmos, é de extrema importância **usarmos os mesmos dados**. Ou seja, se foram usados os dados X para treinar e testar o nosso algoritmo, teremos de utilizar esses mesmos dados X para o nosso algoritmo base.

Lembre-se também de que o algoritmo base não exige nenhum tipo de treino, pois ele simplesmente chuta o mesmo valor para todos os elementos. Será que no nosso algoritmo usamos os mesmos dados tanto para o algoritmo base quanto para o algoritmo que implementamos? Vejamos os dados de cada um:

- Algoritmo base:

```
# restante do código
acerto_base = max(Counter(Y).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(Y)
```

- Nossa algoritmo:

```
# restante do código
porcentagem_treino = 0.9

tamanho_de_treino = porcentagem_treino * len(Y)
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

teste_dados = X[-tamanho_de_teste:]
```

```

teste_marcacoes = Y[-tamanho_de_teste:]

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)

resultado = modelo.predict(teste_dados)

acertos = resultado == teste_marcacoes

total_de_acertos = sum(acertos)
total_de_elementos = len(teste_dados)

```

A princípio, parece que são os mesmos dados, porém, no algoritmo base estamos utilizando o `Y`, ou seja, 100% dos dados, enquanto que no algoritmo que implementamos estamos usando uma parte do `Y`, nesse caso, 90%. Será que esse teste está sendo justo?

Com certeza, não! Lembre-se de que, quando é necessário comparar dois algoritmos, **precisamos sempre usar os mesmos dados**. Se usarmos 90% para o nosso algoritmo, precisamos utilizar os mesmos 90% para o algoritmo base. Mas e se forem 90% aleatórios? Os mesmos 90% aleatórios precisarão ser usados no algoritmo base.

Considerando o que acabamos de ver, vamos fazer com que o nosso algoritmo base utilize os mesmos dados que o nosso algoritmo. Primeiro, copiaremos este trecho de código:

```

acerto_base = max(Counter(Y).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(Y)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

```

Então, colaremos no final do arquivo:

```

# restante do código
print("Taxa de acerto do algoritmo: %f" % taxa_de_acerto)

```

```

print(total_de_elementos)

acerto_base = max(Counter(Y).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(Y)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

```

Por fim, alteraremos os valores de `Y` e usaremos a variável `teste_marcacoes` que se refere aos 90% do `Y` usados no nosso algoritmo. O código final fica da seguinte forma:

```

import pandas as pd
from collections import Counter

df = pd.read_csv('buscas.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']

Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_treino = 0.9

tamanho_de_treino = porcentagem_treino * len(Y)
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

teste_dados = X[-tamanho_de_teste:]
teste_marcacoes = Y[-tamanho_de_teste:]

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
modelo.fit(treino_dados, treino_marcacoes)

resultado = modelo.predict(teste_dados)

acertos = resultado == teste_marcacoes

total_de_acertos = sum(acertos)

```

```
total_de_elementos = len(teste_dados)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

print("Taxa de acerto do algoritmo: %f" % taxa_de_acerto)
print(total_de_elementos)

acerto_base = max(Counter(teste_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(teste_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)
```

Ao rodarmos o código com esses ajustes, teremos:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo: 82.000000
100
Taxa de acerto base: 82.000000
```

Repare que, usando exatamente os mesmos dados para ambos os algoritmos, mesmo não tendo um resultado superior, o nosso algoritmo foi tão bom quanto o algoritmo base. Ou seja, é de extrema importância a utilização de exatamente os mesmos dados para a comparação entre dois algoritmos.

## CAPÍTULO 5

# NAIVE BAYES E MAXIMUM A POSTERIORI POR TRÁS DOS PANOS

Até agora, utilizamos apenas o algoritmo `MultinomialNB` (Naive Bayes). Como será que esse algoritmo funciona por de trás dos panos? Em outras palavras, dado um conjunto de dados que separamos entre treino e teste, como será que ele faz para treinar a partir desses dados? Ou então, como o nosso cérebro faz para dizer se é um porco ou um cachorro, ou se o e-mail é *spam* ou não *spam*?

Mas por que precisamos saber como ele funciona? É justamente para obtermos uma base melhor de como um algoritmo de classificação funciona. Existem diversas formas de classificação que envolvem teorias matemáticas, porém, nesse instante, veremos o `MultinomialNB`. Podemos iniciar com um exemplo do dia a dia do brasileiro: a eleição. Pegamos um determinado público e perguntamos quem ganharia, entre o Guilherme e o Paulo, e os resultados foram:

Vai ganhar		Total
Guilherme		70
Paulo		30

Figura 5.1: Naive bayes: a eleição

Observe que, entre 100 pessoas, 70 acreditam que o Guilherme vai ganhar e 30 pessoas acham que é o Paulo. Levando essa informação em consideração, se perguntarmos a uma pessoa quem vai ganhar, o que será que ela responderá? A probabilidade para acharem que será o Guilherme é de 70% e o Paulo, 30%. Perceba que a pergunta não tem o objetivo de saber quem vai ganhar, mas sim explorar a probabilidade para as duas opções.

Se escolhermos uma pessoa aleatória e perguntarmos quem vai ganhar, usando a informação anterior como base, o que você acha que ela vai responder? Lembrando que você só pode falar uma opção, o que você responderia tendo conhecimento dos dados apresentados (tendo treinado com esses dados)? Ela responde Guilherme ou Paulo? Qual foi a regra de decisão que você utilizou para responder entre eles?

Nesse instante, você está tomando uma decisão de acordo com todo o treinamento que você obteve. Uma das possíveis abordagens para tomada de decisão seria escolher entre o maior deles, ou seja, entre 30% ou 70%, eu escolheria o de 70%. Mas por que o de 70%?

Vai ganhar	Total
Guilherme	70%
Paulo	30%

Figura 5.2: Naive bayes: a eleição

Porque 70% é maior, portanto, eu escolheria o maior, nesse caso, o Guilherme. Um outro exemplo seria dizer que 70% das pessoas compram e 30% não compram, novamente eu escolheria o maior, logo, responderia que compram. Note que esse é um exemplo de tomada de decisão.

Mas e se pegássemos o de menor probabilidade? Faz sentido em algum momento? A princípio, parece ruim, pois se perguntarmos para diversas pessoas, uma vez que a chance de erro é muito maior, provavelmente não escolherão o Paulo. Em outras palavras, se levarmos em consideração todos os chutes, em 70% das vezes as pessoas escolherão o Guilherme, por isso não faz muito sentido utilizarmos a menor probabilidade como parâmetro para tomada de decisão.

É importante ressaltar que essa tomada de decisão é válida apenas para os casos em que analisamos somente a quantidade de acertos. Mas será que não existe alguma outra regra que podemos aplicar?

Perceba que apenas a regra de decisão que vimos até agora é muito rígida, pois estamos usando uma só informação para decidir quem será ou não votado. Quando consideramos apenas a maior dentre as porcentagens, não conseguimos distinguir **70% e 30%** de **99% e 1%** ou **51% e 49%**, ou seja, sempre escolheremos o maior

independente de existir alguma chance ou não para o menor. Repare que restringir a nossa decisão não parece uma boa ideia. Podemos melhorar a forma como decidimos qual será o voto escolhido entre o Guilherme e o Paulo.

Uma das possibilidades que é possível abordar seria sortear números entre 1 a 100, então, se o número for menor ou igual a 70, significa que ele faz parte de 70% de chance, logo, o voto seria para o Guilherme. Caso contrário, entre 71 a 100, o voto seria para o Paulo que corresponde aos 30%:

Vai ganhar	Probabilidade	Na sorte...
Guilherme	70%	1 até 70
Paulo	30%	71 até 100

Figura 5.3: Naive bayes: a eleição

Mas o que estamos fazendo nesse momento? Estamos criando uma regra que não decide apenas utilizando a maior ou menor probabilidade. Levando essa nova regra em consideração, se perguntarmos para uma pessoa qualquer em quem ela votará, dessa vez, em vez de escolher o maior para tentar descobrir, sortearemos um número de 1 a 100, pois o número 100 é o número total da probabilidade.

Perceba que agora não estamos lidando apenas com uma possibilidade como estávamos fazendo antes. Dessa vez, existe a possibilidade tanto para o Guilherme quanto para o Paulo, independentemente de um ter mais chance do que o outro.

Observe que essa regra funciona de acordo com as probabilidades. De acordo com o que vimos até agora,

conseguimos definir três tipos de regras de decisão:

- **Preferência ao maior:** é válida quando estamos apenas lidando com quantidade.
- **Preferência ao menor:** não faz sentido, pois o que tiver maiores chances na maioria das vezes será o escolhido.
- **Sortear números aleatórios entre 1 a 100:** é válido quando estamos querendo de fato lidar com as probabilidades.

O nosso treino pode ser baseado em algumas dessas regras de decisões, porém é sempre importante compreender que, para cada um dos casos, uma dessas regras pode ser mais interessante do que a outra. Um outro exemplo que podemos analisar seria conforme a tabela:

Vai ganhar	Total
Guilherme	77
Paulo	42

Figura 5.4: Naive bayes: a eleição

Repare que, dessa vez, em vez de porcentagens, temos a quantidade de votos do Guilherme e do Paulo. O que podemos fazer quando já temos a quantidade de votos em vez do percentual? Simples, calculamos o percentual. Para calcularmos o percentual, precisamos saber a quantidade de votos totais. Nesse caso, basta somar os votos de ambos:

Vai ganhar		Total
Guilherme		77
Paulo		42
Total		119

Figura 5.5: Naive bayes: a eleição

119 é a quantidade total de pessoas que foram entrevistadas e deram os seus votos. Dentre essas 119, 77 votaram no Guilherme e 42 no Paulo. Para calcular a probabilidade, basta pegar a quantidade de votos e dividir pelo total de pessoas.

Por exemplo, para calcular a probabilidade do Guilherme, fazemos  $77/119 = 0,647058824$ , ou seja, ao arredondarmos, obtemos 65%. E para o Paulo? A mesma coisa, porém, agora com 42 votos, então,  $42/119 = 0,352941176$ , e ao arredondar esse número, obtemos 35%. A nossa tabela fica da seguinte maneira:

Vai ganhar	Total	Probabilidade	Na sorte...
Guilherme	77	65%	1 e 65
Paulo	42	35%	66 e 100
Total	119		

Figura 5.6: Naive bayes: a eleição

Se sortearmos um número de 1 a 100 e cair entre 1 e 65, fará parte dos 65%, isto é, o Guilherme será votado. Já caso o número seja entre 66 a 100, estaremos nos 35% do Paulo, logo, o voto vai para o Paulo. Observe que, por meio dessa tabela, podemos utilizar uma das três regras que vimos até agora.

Por exemplo, se utilizarmos a regra do **maior**, apenas diremos que o Guilherme será votado. Caso optarmos pela regra do valor **menor**, falaremos que apenas o Paulo será votado. Por fim, podemos escolher a regra da **probabilidade**; sorteamos um número entre 1 a 100 e, então, dependendo do número que for sorteado, escolhemos.

Perceba que o comportamento do algoritmo que estamos analisando é justamente pegar uma quantidade de dados brutos, calcular as probabilidades e utilizar uma regra de decisão. Esses passos são bem similares ao que fazemos no nosso dia a dia.

Por exemplo, na classificação de e-mails entre *spam* e não *spam*, também fazemos algo bem similar, em outras palavras, também utilizamos um dos três critérios que vimos até agora. Quando vemos os nossos e-mails e nos deparamos com um texto do *subject* que, na maioria das vezes, era um *spam*, imediatamente e sem utilizar nenhum outro critério, concluímos que se tratava de um *spam*.

Ou então, sabemos que na maioria das vezes aquele *subject* se tratava de um *spam*, mas, por meio de uma escolha aleatória, decidimos abrir esse e-mail. Logo, estamos entrando na situação em que sorteamos o número e, então, verificamos em qual probabilidade ele se encaixa.

Um outro exemplo bem comum são as ações da bolsa de valores, em que analisamos uma determinada ação e nos perguntamos se ela vai subir ou descer. Vimos que acontecia um evento comum em que, na maioria das vezes, a ação subia. Se levarmos em consideração o método do maior valor, investiremos nessa ação, pois acreditamos que ela subirá. Porém, pode também

acontecer um evento que, em alguns momentos, ela subiu ou desceu, então usamos o critério da probabilidade: sorteamos um número na nossa cabeça e, a partir dele, escolhemos se vamos ou não investir nessa ação.

Observe que até mesmo nós fazemos passos similares ao do algoritmo de classificação, criando padrões para tomar as nossas decisões, medindo sempre a chance da ocorrência (percentual para que aconteça) e utilizando um método de decisão, seja pelo número maior, menor ou probabilidade. Dentre essas regras de decisão vistas até o momento, a qual usaremos nos exemplos posteriores chama-se *maximum a posteriori*, que é justamente a regra de que, dadas as probabilidades, escolhemos o maior valor dentre as possibilidades.

Vamos para o próximo exemplo, referente a um site de vendas de imóveis. No meu site, tenho diversas informações dos clientes que compraram. Dentre elas, peguei a de qual estado eles são:

Estado	Total	Probabilidade
Rio de Janeiro	31	
São Paulo	56	
Total		

Figura 5.7: Naive bayes: a compra

Podemos ver que 31 dos meus clientes são do Rio de Janeiro, e 56 são de São Paulo. A princípio, podemos calcular a probabilidade de uma pessoa ser de São Paulo ou do Rio de Janeiro, porém, o que fazemos com essa informação? Até então, não temos muita utilidade para ela. O que realmente queremos saber?

Na verdade, o que queremos saber é se, dadas essas informações, a pessoa vai comprar ou não no meu site. Mesmo que a informação que indica em qual região os clientes compram mais tenha alguma relevância, a informação de que o cliente que compra ou não também é de grande importância. Então, observe que teremos de lidar com duas informações, conforme a tabela a seguir:

Rio de Janeiro	Total	Probabilidade
Comprou	37	
Não comprou	17	
Total		

São Paulo	Total	Probabilidade
Comprou	67	
Não comprou	177	
Total		

Figura 5.8: Naive bayes: a compra, dado que...

Perceba que agora sim as informações já começam a fazer sentido, pois, dado o estado dos meus clientes, temos a informação de quantos compraram ou não. Podemos preencher a nossa tabela com as probabilidades, utilizando o mesmo esquema que fizemos no primeiro exemplo, que é somar a quantidade de clientes que compraram e que não compraram de acordo com o seu estado, e dividir os que compraram pelo total para obtermos essa probabilidade. Em seguida, fazemos o mesmo para os que não compraram.

O resultado da nossa tabela fica da seguinte forma:

Rio de Janeiro	Total	Probabilidade
Comprou	37	68,5%
Não comprou	17	31,5%
Total	54	

São Paulo	Total	Probabilidade
Comprou	67	27.5%
Não comprou	177	72.5%
Total	244	

Figura 5.9: Probabilidade condicional

Perceba que agora a nossa tabela possui um aspecto diferente, pois 68,5% dos clientes do Rio de Janeiro compram e 31,5% não compram; e os de São Paulo, 27,5% compram e 72,5% não compram. O que isso significa? Note que agora precisamos lidar com duas variáveis, que são: **localidade** e **compra**.

- **Localidade:** estado em que o cliente mora;
- **Compra:** se comprou ou não.

Nesse instante, precisamos analisar duas informações para poder prever se o cliente comprará ou não. Por exemplo, utilizando o método *maximum a posteriori*, um cliente do Rio de Janeiro comprará ou não? Como podemos ver, 68,5% compram e 31,5% não; em outras palavras, os clientes do Rio de Janeiro vão comprar. Mas e os clientes de São Paulo? 72,5% não compram e 27,5% sim, logo, não comprarão.

Perceba que essas probabilidades possuem condições, pois de acordo com o estado do cliente, temos uma determinada

possibilidade se ele vai comprar ou não. Isto é, para cada estado, teremos uma probabilidade de compra: se o cliente for do Rio de Janeiro, ele tem mais chances de comprar, porém, se for de São Paulo, tem menos. Esse é o tipo de probabilidade que chamamos de **probabilidade condicional**.

Levando em consideração esse tipo de probabilidade, se escolhêssemos uma pessoa qualquer para verificar se ela comprou ou não, qual seria a nossa atitude? Primeiramente, perguntaríamos seu estado. Então, se fosse do Rio de Janeiro, responderíamos que ela compraria, porém, se fosse de São Paulo, falaríamos que não, baseando-se no método do *maximum a posteriori*.

Mas e se usássemos a regra da probabilidade? Como faríamos? Da mesma forma! A única diferença é que, em vez de escolher o maior, sortearíamos um número de 1 a 100, e então verificaríamos em qual probabilidade ela se encaixa. Por exemplo, se fosse o Rio de Janeiro, responderíamos que compraria de 1 a 68,5; mas, de 68,6 a 100, diríamos que não.

Observe que, para esses tipos de dados (com probabilidades condicionais), podemos analisar diversas perguntas, como:

- Dado que ela é do Rio de Janeiro, ela vai comprar?

Qual é a nossa atitude dessa vez? Primeiro, verificamos a qual tabela a pessoa pertence, nesse caso, Rio de Janeiro. Logo, ela tem 68,5% de chance de comprar, então podemos responder que sim. A próxima pergunta seria:

- Dado que ela é do Rio de Janeiro, qual a probabilidade de ela comprar?

E para essa situação? Fazemos a mesma coisa, porém respondemos a chance, ou seja, probabilidade desse cliente comprar, que é de 68,5%. Na matemática, podemos representar essa mesma pergunta da seguinte forma:

$$P(\text{Comprar} \mid \text{Rio de Janeiro})$$

Essa representação matemática só traduz nossa pergunta:  $P$  significa probabilidade e  $(\text{comprar} \mid \text{Rio de Janeiro})$  nos diz a probabilidade de comprar, dado que o cliente é do Rio de Janeiro. E se fosse de São Paulo? Como ficaria? Vejamos a próxima pergunta:

- Dado que ele é de São Paulo, qual a probabilidade de ele comprar?

Da mesma forma que fizemos para o Rio de Janeiro, representamos matematicamente a seguinte fórmula:

$$P(\text{Comprar} \mid \text{São Paulo})$$

Até o momento, tivemos apenas uma única variável condicional que indica qual é o estado. Então, de acordo com o estado, olhamos a probabilidade de comprar ou não. Representamos essa probabilidade condicional conforme a tabela:

Rio de Janeiro	Total	Probabilidade
Comprou	37	68.5%
Não comprou	17	31.5%
Total	54	

São Paulo	Total	Probabilidade
Comprou	67	27.5%
Não comprou	177	72.5%
Total	244	

Figura 5.10: Probabilidade condicional de uma variável

Com apenas uma variável condicional, usamos essas duas

tabelas para nos auxiliar. Porém, e se forem duas variáveis? Por exemplo, sabemos qual é o estado do cliente e, além disso, também temos a informação sobre sua renda, se ela é maior ou menor do que R\$5.000.

Esse tipo de informação influencia se o cliente vai ou não comprar, pois, de acordo com o mercado, os clientes com maiores condições possuem uma chance maior de comprar do que os que possuem menores condições. Tendo o conhecimento das informações apresentadas, nos deparamos com a seguinte situação:

- Temos duas variáveis condicionais, o estado e a faixa salarial do cliente, como podemos fazer a nossa análise?

Observe que agora temos duas variáveis condicionais, ou seja, precisamos analisar duas variáveis distintas. Como podemos resolver isso? Da mesma forma que fizemos para a variável que representa o estado, criaremos uma tabela para a variável que representa a faixa de salário:

Rio de Janeiro	Total	Probabilidade
Comprou	37	68.5%
Não comprou	17	31.5%
Total	54	

São Paulo	Total	Probabilidade
Comprou	67	27.5%
Não comprou	177	72.5%
Total	244	

\$ > 5000	Total	Probabilidade
Comprou		
Não comprou		
Total		

\$ <= 5000	Total	Probabilidade
Comprou		
Não comprou		
Total		

Figura 5.11: Probabilidade condicional de duas variáveis

Agora temos quatro tabelas e, dentre elas, duas representam os

estados em que os clientes moram, e as outras duas representam a sua faixa salarial. O que precisamos fazer agora? Preencher a tabela da faixa salarial:

Rio de Janeiro	Total	Probabilidade
Comprou	37	68.5%
Não comprou	17	31.5%
Total	54	

São Paulo	Total	Probabilidade
Comprou	67	27.5%
Não comprou	177	72.5%
Total	244	

\$ > 5000	Total	Probabilidade
Comprou	32	80%
Não comprou	8	20%
Total	40	

\$ <= 5000	Total	Probabilidade
Comprou	47	18%
Não comprou	211	82%
Total	258	

Figura 5.12: Probabilidade condicional de duas variáveis

Veja que 80% dos clientes com salário acima de R\$5.000 compram e 20% não, porém, dos que possuem salários abaixo de R\$5.000, 18% compram e 82% não. Por meio dessa tabela, podemos concluir que o perfil de uma pessoa que ganha mais que R\$5.000 é totalmente diferente de uma que ganha menos do que isso. Pode ser que os imóveis que estamos vendendo sejam mais caros, ou então, nesse instante, os apartamentos que eram mais caros estão com valores mais acessíveis.

Podem existir diversos pontos para explicar o motivo para alguns clientes com um determinado perfil comprarem mais do que outros. De fato, não sabemos dizer os reais motivos para uns comprarem mais do que outros, porém, sabemos que os clientes com faixa de salário maior que R\$5.000 possuem maiores chances de comprar do que os com faixa abaixo. Também sabemos dizer que, dentre os clientes do Rio de Janeiro ou de São Paulo, os com

maior chance de comprar são os do Rio de Janeiro.

Repare que isolamos as informações de estado e faixa de salário, ou seja, elas são independentes. Mas agora que temos as informações, o que devemos fazer? Lembra de que treinávamos o nosso algoritmo? Agora precisamos treiná-lo também com base nessas informações.

E como treinamos a partir dessas informações? Simplesmente passamos por cada uma das tabelas, verificamos a sua variável condicional e o seu valor, e então coletamos suas respectivas probabilidades de comprar ou não. Vejamos o resultado desse treino:

- 1º tabela: estado: Rio de Janeiro.
  - 68,5% compram e 31,5% não compram. 2º tabela: estado: São Paulo.
    - 27,5% compram e 72,5% não compram.
- 3º tabela: salário:  $\$ > 5000$ .
  - 80% compram e 20% não compram.
- 4º tabela: salário:  $\$ < 5000$ .
  - 18% compram e 82% não compram.

Agora que treinamos, o que precisamos fazer? Testar! E como fazemos o teste com base no nosso treino? Da mesma forma que fizemos antes: vamos calcular a chance e utilizar uma regra de decisão, porém, a única diferença é que, para cada variável condicional que tivermos, precisaremos criar duas tabelas.

Quando tínhamos apenas a variável para o estado, criamos apenas duas tabelas, agora que temos a variável estado e faixa salarial, criamos quatro. Se fossem três variáveis, seriam seis

tabelas e assim por diante. Começaremos com o seguinte exemplo:

- Dado um cliente que é de São Paulo com salário menor do que R\$5.000, qual é a probabilidade de ele comprar?

Lembra de que podemos representar essa pergunta com uma fórmula matemática? Porém, tínhamos feito apenas para uma variável condicional. Como ficaria a mesma fórmula para duas variáveis, ou seja, as variáveis estado e faixa de salário? Vejamos:

$$P(\text{Comprar} \mid \text{São Paulo}, \leq 5000)$$

Perceba que, por meio da fórmula, temos os valores de ambas as variáveis, nesse caso, estado e salário. Além disso, já sabemos quais são os valores, ou seja, São Paulo e menor do que R\$5000, respectivamente. Mas como calcularemos essa probabilidade que envolve tanto um estado como uma faixa de salário a partir dessa fórmula?

Podemos pensar da seguinte maneira: "já que as **variáveis são independentes**, podemos calculá-las individualmente"! Isso significa que, devido ao fato de ambas as variáveis não dependerem uma da outra, podemos quebrar a fórmula matemática mais complexa ( $P(\text{Comprar} \mid \text{São Paulo}, \leq 5000)$ ) em duas mais simples:

$$P(\text{Comprar} \mid \text{São Paulo}, \leq 5000) = P(\text{Comprar} \mid \text{São Paulo}) * P(\text{Comprar} \mid \leq 5000)$$

Observe que criamos duas fórmulas distintas; uma não depende da outra. A partir de agora, estamos calculando a probabilidade separadamente! Começaremos pela fórmula  $P(\text{Comprar} \mid \text{São Paulo})$ :

- $P(\text{Comprar} \mid \text{São Paulo})$  : qual é a probabilidade de compra para os clientes de São Paulo?
  - 27,5% compram.
  - 72,5% não compram.

Agora para a fórmula  $P(\text{Comprar} \mid \leq 5000)$  :

- $P(\text{Comprar} \mid \leq 5000)$  : qual é a probabilidade de compra para os clientes que ganham menos de R\$5.000?
  - 18% compram.
  - 82% não compram.

Por fim, multiplicamos ambas e obtemos a nossa probabilidade final, por exemplo:

- Qual a probabilidade de um cliente de São Paulo comprar?  
◦ 27,5%.
- Qual a probabilidade de um cliente que ganha menos de R\$5.000 comprar?  
◦ 18%.

Temos as porcentagens 27,5% e 18%, então fazemos:  $27,5\% * 18\%$  . Assim, obtemos o seguinte resultado:  $0,275 * 0,18 = 0,0495 \rightarrow 100 * 0.0495 = 4,95 = 5\%$  .

Observe que a probabilidade de um cliente comprar, dado que ele seja de São Paulo e tenha um salário menor que R\$5.000, é de 4,95%. Mas arredondamos para 5% por ser um valor muito próximo.

Agora que calculamos, precisamos utilizar uma regra de decisão. Qual regra seria interessante? A de maior valor? Vejamos:

- 5% compram.
- 95% não compram.

Repare que se utilizarmos a regra do maior, esse cliente não vai comprar, pois existe 95% de chance disso. E se tentarmos a regra de probabilidade? Teremos de sortear um número de 1 a 100 e, se o resultado for de 1 a 5, o cliente vai comprar; caso seja de 6 a 100, ele não vai comprar. Como podemos ver, o resultado final dependerá bastante da regra de decisão que escolhermos. Vejamos um outro exemplo:

$$P(\text{Comprar} \mid \text{Rio de Janeiro}, >5000)$$

Um cliente que é do Rio de Janeiro e tem um salário acima de R\$5.000: qual é a probabilidade de ele comprar? Faremos da mesma forma como anteriormente, ou seja, calcularemos cada probabilidade de acordo com sua variável:

- Qual a probabilidade de ele comprar dado que ele é do Rio de Janeiro?
  - 68,5%
- Qual a probabilidade de ele comprar dado que ele ganha mais de R\$5.000?
  - 80%

Então, multiplicamos as probabilidades:  $0,685 * 0,80 = 0,548$  ->  $100 * 0,548 = 54,80\%$  .

Veja que um cliente que é do Rio de Janeiro e ganha acima de R\$5.000 contém 54,8% de chance de comprar. Um resultado bem diferente comparado ao cliente que era de São Paulo e ganhava menos do que isso, que continha apenas 5% de chance de comprar.

## 5.1 RESUMINDO

Podemos calcular a probabilidade de um evento, como um cliente que é de um estado  $X$  e ganha um salário maior ou menor que  $Y$ . Fizemos isso multiplicando todas as variáveis existentes para obtermos o resultado final, pois assumimos que ambas as variáveis são independentes.

Esse é o processo de treino que precisamos realizar, ou seja, calcular as probabilidades condicionais independentemente se são uma, duas ou três variáveis condicionais. Para cada variável, criamos duas tabelas. Isto é, se for apenas uma variável, então duas tabelas; se forem duas, então quatro tabelas; se forem  $X$  variáveis, então  $X * 2$  tabelas.

Usamos um critério de avaliação, podendo ser tanto o *maximum a posteriori* quanto a probabilidade. Porém, precisamos sempre nos atentar a qual critério escolher, pois, se usarmos o **maximum a posteriori** e as probabilidades de compra forem sempre maiores, ele sempre vai dizer que o cliente vai comprar. Como também, se todas as probabilidades de compra forem menores, ele sempre dirá que o cliente não vai comprar.

Percebe o quanto problemático ele pode ser? É exatamente por esse motivo que, na maioria dos casos, utilizar o critério da probabilidade (que sorteia os números de 1 a 100) acaba sendo mais adequado para treinar o nosso algoritmo. Mas e o nosso teste? Como funciona?

É simples. Multiplicamos todas as variáveis condicionais que foram calculadas e aplicamos uma regra de decisão, então, tomamos uma decisão. Em outras palavras, nós classificamos se o

elemento é da categoria A ou B, se é *spam* ou não, se ele vai comprar ou não.

Repare que utilizamos o algoritmo *Multinomial naive bayes* que se caracteriza por ser um treino extremamente rápido, pois é só calcular essas tabelas que vimos no exemplo. Além disso, ele precisa apenas realizar uma somatória, dado um determinado elemento, quantos deram uns e quantos deram zeros. Esse tipo de algoritmo é chamado de linear ([https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o\\_linear](https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_linear)).

Podemos concluir que ele é um algoritmo eficiente, pois ele faz operações de acordo com a quantidade de elementos. Se forem 100 elementos, ele faz apenas as 100 operações; se forem 1.000 elementos, então 1.000 operações. Logo, ele tem um desempenho muito bom comparado com outros algoritmos existentes.

Vimos também que ele é simples de implementar, pois ele realiza algumas operações matemáticas que são equivalentes, teoricamente, com as tabelas que analisamos. Além disso, ele é bastante usado para classificar texto, como foi o caso do *spam*. Por exemplo, ele pode verificar as palavras mais frequentes e, com base nessas informações, tentar distinguir um e-mail normal de um *spam*.

Perceba que o *naive bayes* não é uma implementação de um outro mundo, pois ele está relacionado a determinados passos e rotinas de que temos uma pequena noção. Ou seja, temos a capacidade de analisar situações, como: dado que 70% das pessoas vão recomendar o produto e 30% não, então perguntamos para uma pessoa qualquer se ela recomenda ou não o produto —

provavelmente ela dirá que sim.

Podemos observar que essa é uma sensação que todos nós, seres humanos, compartilhamos. Por exemplo, levemos em conta essa informação: "A maior parte dos brasileiros gosta de futebol". Se perguntarmos para um brasileiro qualquer se ele gosta de futebol, o que você responderia? Provavelmente que sim, pois é "a maior parte das pessoas". É bem provável que haverá pessoas que não gostam, mas não importa, pois estamos dando preferência a quantidade maior, ou seja, utilizando a regra do maior. Mas também poderíamos utilizar uma outra regra de decisão.

Perceba que o senso de eventos mais comuns (com maior frequência de ocorrência) é uma opção válida para uma regra de decisão e é muito parecido com as regras que estamos usando, ou seja, o que tiver maior frequência provavelmente acontecerá de novo. Esse é justamente o comportamento do *naive bayes*: de acordo com a probabilidade que aconteceu no passado, eu conseguirei classificar o que acontecerá agora.

## CAPÍTULO 6

# TESTANDO DIFERENTES MODELOS E VALIDANDO O VENCEDOR

Até agora, vimos como a implementação do *naive bayes* funciona. Ou seja, levando em consideração as informações do passado, ele dá uma maior preferência ao evento que aconteceu com mais frequência. Portanto, se ele se deparar com um novo cliente que possui determinadas características e, baseado nas informações que ele já conhece, existe um outro cliente com as mesmas características que já comprou, provavelmente ele responderá que este também comprará.

Porém, quando usamos o algoritmo para classificar os dados do arquivo `buscas.csv` :

```
home,busca,logado,comprou
0,algoritmos,1,1
0,java,0,1
1,algoritmos,0,1
1,ruby,1,0
1,ruby,0,1
0,ruby,1,0
0,algoritmos,1,1
0,ruby,0,1
1,algoritmos,1,1
...
```

0, ruby, 1, 0

Tivemos o seguinte resultado:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo: 82.000000
100
Taxa de acerto base: 82.000000
>
```

Podemos observar que a taxa de acerto do nosso algoritmo e a do algoritmo base obtiveram o mesmo resultado. Ou seja, se chutássemos tudo 1 ou se implementássemos um algoritmo bem complexo, cheio de código e que analisa diversas características, o resultado é o mesmo. Faz sentido perdemos tempo implementando um algoritmo tão complexo para obtermos o mesmo resultado de um algoritmo que simplesmente chuta o mesmo valor para tudo?

Em alguns momentos, principalmente para classificação de texto, usaremos o `MultinomialNB`, entretanto, em outras situações pode ser que esse algoritmo não seja tão interessante. Precisamos levar em consideração que, nessa situação, foram usados dados *fakes*, ou seja, informações que foram manipuladas para demonstrar uma comparação entre o nosso algoritmo e o algoritmo base.

Mas e se fossem dados reais, como uma base de dados de uma outra empresa, o nosso algoritmo seria melhor? Qual seria o resultado? Sempre 82%? Provavelmente não, pois depende muito da variedade dos dados e do que estamos tentando classificar.

Dessa vez, vamos mudar um pouco os nossos dados. Você se lembra da planilha do Google Spreadsheets (<http://bit.ly/1T7qjMS>) que contém os nossos dados?

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	home	busca	logado	comprou									
2	0	ruby	1	1									
3	1	algoritmos	0	0									
4	0	algoritmos	0	1									
5	1	java	1	1									
6	1	algoritmos	0	0									
7	0	ruby	0	1									
8	0	java	0	0									
9	1	ruby	1	1									
10	1	ruby	0	0									
11	1	algoritmos	0	0									
12	0	algoritmos	0	1									
13	1	algoritmos	1	0									
14	1	ruby	0	1									
15	0	ruby	0	1									
16	0	ruby	1	1									
17	0	java	0	1									
18	0	ruby	1	1									
19	1	ruby	0	0									
20	1	algoritmos	0	0									
21	1	algoritmos	0	1									
22	1	algoritmos	1	0									
23	1	ruby	0	1									

buscas + buscas2 ~ frequencia ~ doacoes ~ buscas\_sim\_ou\_nao ~ saar ~ situacao\_do\_cliente ~ emails ~ Explorar

Observe que agora temos a aba `buscas2`, que é exatamente o mesmo tipo de informação que temos na aba `buscas`, ou seja, `home` (se acessou a `home`), `busca` (referente ao que ele buscou), `logado` (se ele está logado ou não) e `comprou` (se ele comprou ou não). Porém, nessa aba temos apenas 75 registros.

Agora vamos lidar com muito menos elementos e dados do que anteriormente, e provavelmente teremos um resultado bem diferente. Sempre que tivermos um novo conjunto de dados, precisaremos testar o nosso algoritmo para verificar se ele está bom ou não para o determinado conjunto de dados. Vamos verificar o resultado dele para esses dados?

Primeiro, precisamos salvar esse arquivo. Podemos salvar como `buscas2.csv` no mesmo diretório em que estão os nossos arquivos Python, então alteramos no nosso arquivo `classifica_buscas.py` para que ele leia esse arquivo:

```
import pandas as pd
from collections import Counter
```

```
df = pd.read_csv('buscas2.csv')
# restante do código
```

Ao rodar o nosso código:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo: 75
8
Taxa de acerto base: 62.5
```

Observe que, para esse conjunto de dados, foram usados sete registros para teste, 10% desses dados. Além disso, dessa vez, o nosso algoritmo acertou 85,71% enquanto o algoritmo base acertou 71,42%. Isso comprova que o resultado dos nossos testes varia de acordo com o conjunto de dados. Para afirmarmos que o nosso algoritmo está bom ou não, precisamos testá-lo com o determinado conjunto de dados que temos, e então verificamos se ele foi bom ou não.

É importante ressaltar que a quantidade de dados também é um ponto a se considerar, pois o nosso algoritmo pode ter melhores resultados com quantidades menores do que com maiores, e vice-versa. Claro que um algoritmo que só funciona para um conjunto pequeno de dados se mostra quase sempre inútil por não ter poder de previsão.

Ao mesmo tempo, não podemos esquecer que as características são pontos cruciais para o resultado do nosso algoritmo. Nesse caso, temos três delas: as variáveis ( `home` , `busca` , `logado` ) para esses dados, sendo que, dentre elas, uma é categórica com três valores distintos, ou seja, no total temos cinco variáveis. Podemos nos perguntar:

- Será que essas cinco variáveis são ideais para o nosso teste?

- Será que, se tirarmos uma delas, o nosso algoritmo retorna melhores resultados?

Que tal tirarmos uma variável para verificar o resultado do nosso algoritmo? Vamos tirar a variável `logado`. Para isso, precisamos alterar o nosso arquivo `classifica_buscas.py`:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('buscas2.csv')
X_df = df[['home', 'busca']]
Y_df = df['comprou']
# restante do código
```

Note que agora estamos utilizando apenas as variáveis `home` e `busca`. E agora? Será que conseguiremos prever melhor se o cliente comprou ou não? Vamos testar:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo: 72.5
7
Taxa de acerto base: 62.5
```

Ele obteve a mesma taxa de acerto, ou seja, não teve diferença alguma. Se com a variável `logado` não fez muita diferença, então que tal tirarmos a variável `busca`?

```
import pandas as pd
from collections import Counter

df = pd.read_csv('buscas2.csv')
X_df = df[['home', 'logado']]
Y_df = df['comprou']
# restante do código
```

Nesse exato momento, o nosso algoritmo possui apenas as informações da `home` e do `logado`. Vejamos o que acontece:

```
> python3 classifica_buscas.py  
Taxa de acerto do algoritmo: 62.5  
8  
Taxa de acerto base: 62.5
```

Para esse cenário, o nosso algoritmo não obteve um resultado melhor e não foi tão bem quanto antes, ou seja, tivemos um resultado bem inferior sem a variável de busca . O que podemos concluir?

Isso *pode* nos dizer que a busca é uma informação bem valiosa para o nosso algoritmo, em outras palavras, possui uma grande influência para ele conseguir prever com mais precisão. Por fim, vamos verificar o resultado do nosso algoritmo sem a informação da home :

```
import pandas as pd  
from collections import Counter  
  
df = pd.read_csv('buscas2.csv')  
X_df = df[['busca', 'logado']]  
Y_df = df['comprou']  
# restante do código
```

Rodando o nosso código:

```
> python3 classifica_buscas.py  
Taxa de acerto do algoritmo: 75  
7  
Taxa de acerto base: 62.5
```

Repare que, dentre todas as variáveis, apenas a busca apresentou algum impacto de verdade. Será que, se utilizarmos apenas a variável busca , o nosso algoritmo obterá um resultado tão bom ou melhor ao qual temos agora? Para isso, precisamos testar! Vamos alterar o nosso código:

```
import pandas as pd  
from collections import Counter
```

```
df = pd.read_csv('buscas2.csv')
X_df = df[['busca']]
Y_df = df['comprou']
```

Verificando o resultado novamente:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo: 75
8
Taxa de acerto base: 62.5
```

Como podemos ver, apenas com a variável `busca`, fomos capazes de alcançar o mesmo resultado que estávamos obtendo com todas as variáveis. Perceba que temos muitas coisas em jogo. Para o nosso algoritmo obter um resultado bom ou ruim, tudo depende da quantidade de elementos que estamos utilizando para treinar e testar. Também depende das variáveis, ou melhor, das características que estamos usando.

Como vimos no nosso exemplo, a variável `busca` possui uma grande importância para que o nosso algoritmo atinja um resultado bom, permitindo até o seu uso exclusivo — sem as demais variáveis e, mesmo assim, alcançando o mesmo resultado. Isso significa que, se tivermos 10 variáveis, precisamos testar uma a uma, encontrar as que contêm grande impacto e então usar apenas elas? Na prática, como vimos, parece fazer sentido.

Uma variável do nosso algoritmo é justamente escolher quais serão as variáveis que ele utilizará. Isto é, dado um conjunto de variáveis, o nosso algoritmo obtém um resultado de, por exemplo, 70%. Se tirarmos uma dessas variáveis, o resultado será melhor que 70%? Ou então, pior? Será que, se adicionarmos uma nova variável, fará diferença? Por exemplo, idade, sexo, localização, entre diversas informações que temos.

É bem comum o processo de adicionar e remover variáveis de acordo com as características que os nossos elementos contêm, porém, para cada teste realizado, é um teste a mais. Por exemplo, testamos o nosso código com apenas uma variável, então pensamos em adicionar mais uma e testar novamente. Não felizes com o nosso resultado, fazemos novamente o mesmo processo, e isso vai se repetindo até o momento em que alcançamos um resultado tão bom que parece perfeito. Consegue perceber o quanto problemático essa atitude é?

Existe uma grande chance de o nosso resultado ser tão bom só por **sorte**. Esse mesmo cenário é equivalente a pegarmos diversas variáveis do mundo e fazermos diversas combinações de cruzamento entre elas para teste.

Peguemos, por exemplo, a análise de aumento e queda de preço das bolsas de valores de todos os países, e também a quantidade de filmes feitos por vários atores anualmente. Analisamos essa infinidade de dados e, então, descobrimos que, dentre todos esses testes, existe uma informação em comum que é:

- Todos os anos em que a bolsa de valores dos EUA caiu são os mesmos anos em que o Nicolas Cage filmou um filme de ação.

Assim, concluímos que, cada vez que o Nicolas Cage filmar um filme novo, saberemos que a bolsa de valores cairá. Faz sentido esse tipo de conclusão? Consegue perceber que, nessa atitude de tentar diversas variações, provavelmente teremos alguma coincidência entre essas informações? Estaremos lidando com sorte ou azar.

Um outro exemplo envolve jogar uma moeda para o alto.

Existe 50% de chance de que caia cara, e 50% coroa. Se você jogar 10 vezes a moeda para o alto, a chance é muito pequena de você dar sorte (ou azar) e sair sempre cara (ou sempre coroa). Mas se você pedir para 100 pessoas jogarem a mesma moeda 10 vezes para o alto, provavelmente alguém vai tirar uma série de 10 vezes o mesmo lado! Isso chama-se sorte ou azar, provinda puramente da probabilidade de algo acontecer simplesmente pois tentamos demais.

Então, podemos chegar à conclusão de que testar com muitas variáveis possibilita chegarmos a um resultado muito bom, porém, este poderá ser por sorte. Pior, pode ser por sorte, pode ser de verdade. Mas um resultado bom por sorte não é bom?

Quando usamos um resultado por sorte, somos induzidos às coincidências, ou seja, podemos chegar a conclusões do tipo:

- Sempre que aumenta o consumo de sorvete, aumenta o número de afogamentos, portanto, sorvetes causam afogamentos. Quando na verdade é o verão quem causa afogamentos e consumo de sorvetes.
- Hoje está chovendo, então os meus clientes vão comprar, pois todas as vezes em que choveu os clientes compraram.
- Hoje é o dia 7 do mês, então os meus clientes vão comprar, pois em todos esses dias eles compraram.
- Amanhã é o meu aniversário, então todos os meus clientes comprarão, pois em todo aniversário eles compraram.

Consegue perceber que tudo isso não faz sentido algum? Ou seja, levaremos isso como uma verdade, por causa de sorte e, provavelmente, erraremos a nossa prevenção, pois essas informações não passavam apenas de meras coincidências. Esse é

justamente o problema de treinarmos demais. É justamente por esse motivo que, durante todo o percurso do desenvolvimento do nosso algoritmo, não variamos suas características.

Temos sempre de lembrar que, todas as vezes que alterarmos as variáveis que estamos usando em cada teste, pode ser algo extremamente complicado no mundo real. Isto é, precisamos tomar cuidado em dobro e especificar os passos que foram realizados para chegar a um determinado resultado. Lembra de quando implementamos o `classifica_buscas.py` ?

```
# minha abordagem inicial foi
# 1. separar 90% para treino e 10% para teste: 88.89%

from dados import carregar_acessos

X,Y = carregar_acessos()

treino_dados = X[:90]
treino_marcacoes = Y[:90]

teste_dados = X[-9:]
teste_marcacoes = Y[-9:]
```

Note que fiz uma anotação informando que testamos com 90% dos dados para treino e 10% para teste. Por que será que fizemos isso? É exatamente pela questão de marcarmos o resultado de acordo com a variação que realizamos, pois se criarmos diversos testes com diversas variações e percebermos que todas elas dão errado, e em um determinado teste dentre esse conjunto conseguirmos um resultado satisfatório, existe uma grande possibilidade de esse resultado ter acontecido simplesmente por sorte. E como vimos, se cairmos nesse cenário, nos daremos mal.

Quando apresentarmos um resultado que foi bom dentre os testes que realizamos para aquele algoritmo, é de **extrema**

**importância** demonstrarmos **todos os testes realizados**, ou seja, todos os fracassos que aconteceram para chegar ao resultado satisfatório. Assim, evitamos de cair no cenário em que simplesmente chegamos ali por sorte.

Isso significa que precisamos realizar essas marcações no nosso `classifica_buscas.py` também, então faremos as marcações dos testes realizados:

```
import pandas as pd
from collections import Counter

# teste inicial: home, busca, logado => comprou

df = pd.read_csv('buscas2.csv')
X_df = df[['busca']]
Y_df = df['comprou']
# restante do código
```

Para cada teste do algoritmo, considerarei que realizamos apenas o teste com esse conjunto de variáveis, então quais foram as nossas variações? Fizemos apenas com `home` e `busca` , `home` e `logado` , `busca` e `logado` e, por fim, com a variável de `busca` apenas. Logo, marcamos o nosso código da seguinte maneira:

```
import pandas as pd
from collections import Counter

# teste inicial: home, busca, logado => comprou
# home, busca
# home, logado
# busca, logado
# busca

df = pd.read_csv('buscas2.csv')
X_df = df[['busca']]
Y_df = df['comprou']
# restante do código
```

Para apresentarmos o resultado desse algoritmo, precisamos adicionar todos os passos realizados e, por fim, o resultado final, que foram sete testes com o resultado de 75%.

```
import pandas as pd
from collections import Counter

# teste inicial: home, busca, logado => comprou
# home, busca
# home, logado
# busca, logado
# busca: 75% (8 testes)

df = pd.read_csv('buscas2.csv')
X_df = df[['busca']]
Y_df = df['comprou']
# restante do código
```

Podemos também adicionar o resultado final de cada teste realizado. Porém, o importante é compreendermos nesse exato momento que os nossos testes precisam ser registrados, anotados, para contermos um histórico que comprove toda a nossa trajetória para aquele resultado final.

Dessa forma, provamos que o nosso resultado foi válido, pois demonstramos todo o processo realizado. Por enquanto, vimos apenas variações dos nossos testes entre as características, ou seja, as variáveis disponíveis. Mas e se quisermos variar entre os algoritmos? Será que é possível?

## 6.1 ALGORITMO ADABOOST

Até agora, usamos apenas o `MultinomialNB`, porém existem diversas outras variações para algoritmos de classificação. Uma das variações que vamos utilizar se chama AdaBoost (<https://pt.wikipedia.org/wiki/AdaBoost>).

Esse algoritmo, basicamente, tenta melhorar um algoritmo. Ou seja, ele vai refinando o algoritmo para tentar encontrar a melhor possibilidade, então, quando ele encontra, devolve o resultado.

No mundo real, são diversos os algoritmos que podemos usar, cada um com suas implicações. O objetivo deste livro é introduzirmos ao mundo dos algoritmos de aprendizado de máquina, portanto não nos aprofundaremos em cada um deles.

Mas como podemos utilizar o AdaBoost no nosso algoritmo? Precisamos apenas alterar o algoritmo que estamos usando no nosso arquivo `classifica_buscas.py`, ou seja, no instante em que estamos importando e instanciando o `MultinomialNB`:

```
from sklearn.naive_bayes import MultinomialNB  
modelo = MultinomialNB()
```

Comentamos o `import` e a instância do `MultinomialNB`, e simplesmente importamos o `AdaBoost`:

```
# from sklearn.naive_bayes import MultinomialNB  
# modelo = MultinomialNB()  
  
from sklearn.ensemble import AdaBoostClassifier  
modelo = AdaBoostClassifier()  
  
# restante do código
```

A partir desse momento, o nosso algoritmo está utilizando o `AdaBoost`, porém o restante do código foi usado para o `MultinomialNB`. Será que funcionará para o `AdaBoost` também? Antes de testarmos, agora que estamos usando um algoritmo diferente, precisamos utilizar todas as variáveis que foram usadas com o `MultinomialNB`, então vamos adicionar as variáveis `home` e `logado` novamente:

```
import pandas as pd
from collections import Counter

# teste inicial: home, busca, logado => comprou
# home, busca
# home, logado
# busca, logado
# busca: 85,71% (7 testes)

df = pd.read_csv('buscas2.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']
# restante do código
```

Vamos testar o nosso código.

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo: 75
8
Taxa de acerto base: 62.5
```

Como podemos ver, os métodos de treino e de classificação são os mesmos, tanto para o `MultinomialNB` quanto para o `AdaBoost`. Portanto, não precisaremos alterar o nosso código. Porém, perceba que, para esse conjunto de dados, o resultado manteve-se o mesmo. Será que, para o arquivo `buscas.csv`, o resultado será diferente? Vejamos:

```
import pandas as pd
from collections import Counter

# teste inicial: home, busca, logado => comprou
# home, busca
# home, logado
# busca, logado
# busca: 85,71% (7 testes)

df = pd.read_csv('buscas.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']
# restante do código
```

Rodando novamente o nosso algoritmo:

```
> python3 classifica_buscas.py  
Taxa de acerto do algoritmo: 85.000000  
100  
Taxa de acerto base: 82.000000
```

Observe que o AdaBoost acertou 85%, ou seja, 3% a mais que o MultinomialNB . Isso comprova que, dependendo do conjunto de dados, o nosso teste pode variar. Nesse caso, o AdaBoost obteve um resultado melhor do que o MultinomialNB para esse conjunto, e pode ser que o MultinomialNB tenha um resultado melhor para um outro conjunto de dados.

Repare que, além de variar as variáveis, agora estamos variando o algoritmo também. Logo, além de só utilizar o MultinomialNB , estamos usando o AdaBoost .

Você pode pensar: "Guilherme, as variáveis são variáveis, portanto espero variá-las com o passar do tempo. Mas o algoritmo virou uma variável?". Isso mesmo! O algoritmo também está sendo considerado uma variável da minha análise.

Vimos que, quando variamos o algoritmo, um obtém resultados melhores do que o outro dependendo do conjunto de dados. Considerando essa afirmação, que tal usarmos os dois algoritmos ( MultinomialNB e AdaBoost ) ao mesmo tempo? Mas por que ao mesmo tempo?

Justamente por apresentarem resultados diferentes. Dado que cada algoritmo pode obter um resultado diferente dependendo do conjunto de dados, cada vez que rodarmos o nosso código, escolheremos o de nossa preferência, ou seja, o que apresentar o melhor resultado. Então, vamos implementar os dois algoritmos

no nosso código.

Primeiro, precisamos transformar o código que treina e testa em uma função, pois ambos os algoritmos utilizam o mesmo código. Assim, criaremos a função chamada `fit_and_predict` :

```
def fit_and_predict:  
    modelo.fit(treino_dados, treino_marcacoes)  
  
    resultado = modelo.predict(teste_dados)  
  
    acertos = resultado == teste_marcacoes  
  
    total_de_acertos = sum(acertos)  
    total_de_elementos = len(teste_dados)  
  
    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento  
s  
  
    print("Taxa de acerto do algoritmo: %f" % taxa_de_acerto)
```

Mas observe que as variáveis `modelo` , `treino_dados` , `treino_marcacoes` , `teste_dados` e `teste_marcacoes` precisam ser enviadas para essa função, então vamos adicioná-las na lista de parâmetros da `fit_and_predict` :

```
def fit_and_predict(modelo, treino_dados, treino_marcacoes, teste  
_dados, teste_marcacoes):  
# restante do código da função
```

Além disso, a impressão do total de elementos que está fora da função `fit_and_predict` precisa da variável `total_de_elementos` :

```
# restante do código  
  
print(total_de_elementos)  
  
acerto_base = max(Counter(teste_marcacoes).values())  
taxa_de_acerto_base = 100.0 * acerto_base / len(teste_marcacoes)
```

```
print("Taxa de acerto base: %f" % taxa_de_acerto_base)
```

Entretanto, ela só existe dentro do `fit_and_predict`, ou seja, ele não tem acesso à variável `total_de_elementos`. Então, vamos criar também uma variável `total_de_elemento` que recebe o tamanho da variável `teste_dados`:

```
total_de_elementos = len(teste_dados)
print(total_de_elementos)
# restante do código
```

Vamos aproveitar e deixar esse trecho de código no final do nosso algoritmo, e também descrevê-lo com mais clareza; isto é, vamos deixar uma mensagem explícita sobre o que ele é:

```
acerto_base = max(Counter(teste_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(teste_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(teste_dados)
print("Total de teste: %d" % total_de_elementos)
```

Agora precisamos fazer a chamada do método `fit_and_predict` para cada um dos algoritmos, tanto para o `MultinomialNB` quanto para o `AdaBoost`:

```
from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
fit_and_predict(modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

from sklearn.ensemble import AdaBoostClassifier
modelo = AdaBoostClassifier()
fit_and_predict(modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
```

Aparentemente, todas as alterações foram realizadas, então vamos rodar o nosso algoritmo que utiliza tanto o `MultinomialNB` quanto o `AdaBoost`:

```
> python3 classifica_buscas.py  
Taxa de acerto do algoritmo: 82.000000  
Taxa de acerto do algoritmo: 85.000000  
Taxa de acerto base: 82.000000  
Total de teste: 100
```

Tudo está funcionando como o esperado. Vamos alterar o conjunto de dados e verificar se funciona da mesma forma?

```
# restante do código  
df = pd.read_csv('buscas2.csv')
```

Testando novamente:

```
> python3 classifica_buscas.py  
Taxa de acerto do algoritmo: 75  
Taxa de acerto do algoritmo: 75  
Taxa de acerto base: 62.5  
Total de teste: 8
```

Também está funcionando. Dessa vez, o total de elementos de teste são 8, e ambos os algoritmos são de 75%. Porém, note que, por meio dessa impressão, não conseguimos distinguir quais são os algoritmos que estão sendo usados, e também não sabemos quais são os seus respectivos resultados. E como saberemos qual deles obteve o melhor resultado?

Isso significa que precisamos adicionar o nome de ambos na impressão também, e podemos fazer isso enviando por parâmetro a variável `nome`, que representa o nome do algoritmo que está sendo usado:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes,  
teste_dados, teste_marcacoes):  
    # restante do código
```

Na chamada do método `fit_and_predict`, enviamos o nome do algoritmo:

```
from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
fit_and_predict("MultinomialNB", modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

from sklearn.ensemble import AdaBoostClassifier
modelo = AdaBoostClassifier()
fit_and_predict("AdaBoostClassifier", modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

# restante do código
```

Porém, ainda não estamos exibindo a variável `nome`. Para isso, usaremos o método `format` para agrupar as nossas duas variáveis na mensagem, isto é, `nome` e `taxa_de_acerto`:

```
def fit_and_predict(modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)

    # restante do código

    print("Taxa de acerto do algoritmo {0}: {1}".format(nome, taxa_de_acerto))
```

Enviamos dois argumentos no nosso texto: o `{0}` e o `{1}`, que significam o primeiro e o segundo elemento que foram adicionados no `format` — nesse caso, 0 para `nome` e 1 para `taxa_de_acerto`. Além disso, esse texto refere-se à mensagem que estamos exibindo para o usuário, e uma boa prática é justamente isolarmos string s grandes como essa para uma variável.

Assim, identificamos com mais facilidade com o que estamos lidando. Vamos extrair para uma variável chamada `msg`:

```
def fit_and_predict(modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)
```

```

# restante do código

msg = "Taxa de acerto do algoritmo {0}: {1}".format(nome, tax
a_de_acerto)

print(msg)

```

O nosso código final fica da seguinte forma:

```

import pandas as pd
from collections import Counter

# teste inicial: home, busca, logado => comprou
# home, busca
# home, logado
# busca, logado
# busca: 85,71% (7 testes)

df = pd.read_csv('buscas2.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']

Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_treino = 0.9

tamanho_de_treino = int(porcentagem_treino * len(Y))
tamanho_de_teste = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

teste_dados = X[-tamanho_de_teste:]
teste_marcacoes = Y[-tamanho_de_teste:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes,
                    teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)

    resultado = modelo.predict(teste_dados)

```

```

acertos = resultado == teste_marcacoes

total_de_acertos = sum(acertos)
total_de_elementos = len(teste_dados)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

msg = "Taxa de acerto do algoritmo {0}: {1}".format(nome, taxa_de_acerto)

print(msg)

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
fit_and_predict("MultinomialNB", modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

from sklearn.ensemble import AdaBoostClassifier
modelo = AdaBoostClassifier()
fit_and_predict("AdaBoostClassifier", modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

acerto_base = max(Counter(teste_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(teste_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(teste_dados)
print("Total de teste: %d" % total_de_elementos)

```

Vamos testar o nosso algoritmo? Vejamos o resultado:

```

> python3 classifica_buscas.py
Taxa de acerto do algoritmo MultinomialNB: 75.0
Taxa de acerto do algoritmo AdaBoostClassifier: 75.0
Taxa de acerto base: 62.5
Total de teste: 8

```

Agora está claro o resultado de cada algoritmo. Vamos modificar novamente para o arquivo `buscas.csv` e ver o resultado:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo MultinomialNB: 82.0
Taxa de acerto do algoritmo AdaBoostClassifier: 85.0
Taxa de acerto base: 82
Total de teste: 100
```

Observe que agora é bem fácil de saber qual dos algoritmos obteve o melhor resultado. Agora que implementamos os dois (`MultinomialNB` e `AdaBoostClassifier`) no mesmo código, o que podemos concluir?

A princípio concluímos que, no geral, o algoritmo `AdaBoost` obteve um resultado melhor que o `Multinomial`. Levando em consideração o resultado de ambos, qual a taxa de acerto que esperamos ter no mundo real? Para esse algoritmo, esperamos que ele acerte, teoricamente, 85%. Mas por que 85%? É justamente porque, entre os dois algoritmos, um obteve 82% e o outro 85%, logo, pegamos o de maior valor.

Podemos afirmar que, se rodarmos esse mesmo algoritmo em um conjunto de dados da vida real, ele acertará os 85%? Não! Pois realizamos um teste com 10% de um determinado conjunto de dados, e o escolhemos pois ele apresentou um resultado melhor com esse conjunto. Isto é, essa decisão foi humana, pois escolhemos esse algoritmo como o "melhor", devido a esse teste em específico. É exatamente por esse motivo que não podemos afirmar quantos porcento o nosso algoritmo acertará.

Lembre-se de que é de extrema importância entender que, mesmo que o seu algoritmo acerte 80%, 85%, 90%, 100% ou qualquer outro valor para um conjunto de dados, não significa que ele terá o mesmo resultado para um outro conjunto. Isso porque utilizamos essa porcentagem para decidir qual dos algoritmos

usemos para um conjunto de dados desconhecido, isto é, um que não conhecemos e que queremos que ele classifique para nós.

Repare que o nosso algoritmo terá três fases:

- Treinar os algoritmos;
- Testar os algoritmos;
- Escolher o melhor entre eles e testar com os dados reais.

Realizamos agora esse último passo, pois será dessa forma que poderemos afirmar que, no mundo real, ele obtém um resultado bom ou ruim. Mas e na prática? Como fazemos?

Atualmente, simplesmente usamos 90% para treino e 10% para teste, e os dados do mundo real? Como podemos resolver isso? Existem diversas formas para resolvêrmos esse cenário, e uma das que usaremos será conforme os dados a seguir:

- 80% para treino;
- 10% para teste;
- 10% para teste no mundo real.

Antes tínhamos apenas duas variáveis, isto é, as variáveis para treino e teste. Mas considerando que o nosso algoritmo precisa ser validado com dados do mundo real, precisaremos de uma terceira variável que representará esses dados para validação. Vamos alterar o nosso código. Primeiro, começaremos na separação dos dados. Vejamos como está atualmente:

```
porcentagem_treino = 0.9
```

Veja que atualmente estamos pegando os 90% para treino, mas na verdade, nesse instante, pegaremos 80%:

```
# restante do código  
porcentagem_de_treino = 0.8
```

E agora? Quantos porcento vamos usar para teste? 10%, certo? Assim, criaremos a variável `porcentagem_de_teste` para representar o percentual de teste:

```
# restante do código  
porcentagem_de_treino = 0.8  
porcentagem_de_teste = 0.1
```

Agora podemos calcular ambos os tamanhos de treino e de teste. Como fazemos atualmente? Vejamos:

```
# restante do código  
porcentagem_de_treino = 0.8  
porcentagem_de_teste = 0.1  
  
tamanho_de_treino = int(porcentagem_de_treino * len(Y))  
tamanho_de_teste = len(Y) - tamanho_de_treino
```

Note que o tamanho de treino é justamente multiplicar o total de dados (`len(Y)`) pela variável `porcentagem_de_treino`, ou seja, não mexeremos. Mas e a variável `tamanho_de_teste`?

Atualmente, ela está pegando a diferença do total de dados com o `tamanho_de_treino`, ou seja, os 20% restantes da quantidade total dos dados. Mas queremos pegar esses 20% para teste? Não! Dessa vez, precisamos pegar apenas 10%. Portanto, multiplicaremos a variável `porcentagem_de_teste` com o total de dados, pois ela representa os 10% que precisamos:

```
# restante do código  
porcentagem_de_treino = 0.8  
porcentagem_de_teste = 0.1  
  
tamanho_de_treino = int(porcentagem_de_treino * len(Y))  
tamanho_de_teste = int(porcentagem_de_teste * len(Y))
```

Nesse momento, estamos calculando tanto o tamanho para os dados de treino quanto o para teste. Mas e os dados de validação? Precisamos calculá-lo também, entretanto, não temos nenhuma variável que representa seu percentual. E então? Criamos uma variável para armazenar o seu percentual?

Poderíamos fazer isso também, mas note que esse percentual sempre será a diferença entre a quantidade total pela quantidade de dados para treino e teste. Por exemplo, a quantidade total é 100%, então subtraímos pela quantidade total de treino (80%), logo,  $100\% - 80\% = 20\%$ . Depois, subtraímos esses 20% pela quantidade total de teste (10%), logo,  $20\% - 10\% = 10\%$ .

Dessa forma, se os percentuais de teste ou treino forem alterados, não precisaremos nos preocupar com a quantidade para validação, pois será automaticamente calculada! Mas e no código? Vejamos:

```
# restante do código
porcentagem_de_treino = 0.8
porcentagem_de_teste = 0.1

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_teste = int(porcentagem_de_teste * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino - tamanho_de_te
ste
```

Vamos verificar se os valores dos tamanhos para cada uma das variáveis estão conforme o esperado? Para isso, abriremos o interpretador do Python e colaremos o código até esse ponto:

```
>>> import pandas as pd
>>> from collections import Counter
>>>
>>> # teste inicial: home, busca, logado => comprou
... # home, busca
... # home, logado
```

```
... # busca, logado
... # busca: 85,71% (7 testes)
...
>>> df = pd.read_csv('buscas2.csv')
>>> X_df = df[['home', 'busca', 'logado']]
>>> Y_df = df['comprou']
>>>
>>> Xdummies_df = pd.get_dummies(X_df).astype(int)
>>> Ydummies_df = Y_df
>>>
>>> X = Xdummies_df.values
>>> Y = Ydummies_df.values
>>>
>>> porcentagem_de_treino = 0.8
>>> porcentagem_de_teste = 0.1
>>>
>>> tamanho_de_treino = int(porcentagem_de_treino * len(Y))
>>> tamanho_de_teste = int(porcentagem_de_teste * len(Y))
>>> tamanho_de_validacao = len(Y) - tamanho_de_treino - tamanho_de_teste
>>>
```

Então, vamos verificar cada um dos tamanhos:

```
>>> tamanho_de_treino
800.0
>>> tamanho_de_teste
100.0
>>> tamanho_de_validacao
100.0
>>>
```

Calculamos a quantidade para cada uma das variáveis, porém, dadas essas variáveis, precisamos pegar os dados dos nossos data frames. Será que precisaremos modificar os nossos cálculos também? Vejamos como estão atualmente:

```
# restante do código
treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

teste_dados = X[-tamanho_de_teste:]
teste_marcacoes = Y[-tamanho_de_teste:]
```

Observe que os nossos dados de treino serão novamente os primeiros dados, ou melhor, nesse caso, serão os primeiros 80%. Portanto, não precisamos mexer neles. Mas e os nossos dados de teste? Ainda serão os últimos 10% dos nossos dados? Para esse caso, não!

Lembra que agora precisamos pegar os dados de validação também? É justamente por esse motivo que precisamos pegar os próximos 10% **a partir da quantidade de treino**, nesse caso, os 80%. Mas como fazemos isso no código? Simples, pedimos para ele retornar os dados a partir da variável `tamanho_de_treino`:

```
teste_dados = X[tamanho_de_treino:]  
teste_marcacoes = Y[tamanho_de_treino:]
```

Isso porque ela representa a quantidade de dados para treino, ou seja, os 80%. Então, pedimos para que os dados sejam até a soma das variáveis `tamanho_de_treino` e `tamanho_de_teste`, em outras palavras,  $80\% + 10\% = 90\%$ :

```
teste_dados = X[tamanho_de_treino:tamanho_de_treino + tamanho_de_  
teste]  
teste_marcacoes = Y[tamanho_de_treino:tamanho_de_treino + tamanho  
_de_teste]
```

A soma do tamanho de treino e o teste se tornaram instruções bem grandes, logo, podemos extrair essa operação para uma outra variável. Vamos criar a variável `fim_de_treino`:

```
fim_de_treino = tamanho_de_treino + tamanho_de_teste  
  
teste_dados = X[tamanho_de_treino:fim_de_treino]  
teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]
```

E agora? Precisamos calcular os dados de validação. Mas como podemos calculá-los? Repare que calculamos cada variável por um

determinado intervalo de valores:

```
# 0 até 799
treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

fim_de_treino = tamanho_de_treino + tamanho_de_teste

# 800 até 899
teste_dados = X[tamanho_de_treino:fim_de_treino]
teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]
```

Nesse instante, precisamos nos atentar, pois os dados de validação não podem fazer parte dos dados de treino (0 a 799) e nem dos de teste (800 a 899). Ou seja, precisamos fazer com que ele seja a partir do 900 a 999. E como podemos fazer isso? Da mesma forma que fizemos com os dados de teste.

Em outras palavras, basta pegarmos os valores a partir da variável `fim_de_treino`, pois ela é justamente o momento em que os dados de teste acabam:

```
validacao_dados = X[fim_de_treino:]
validacao_marcacoes = Y[fim_de_treino:]
```

E o final? Lembra que, quando não passamos o argumento no segundo parâmetro, ele pega o restante dos dados? Nesse exato momento, ele está pegando a partir do 900 até o final, o 999. Será mesmo que as nossas variáveis estão pegando os valores corretos? Vamos testar!

Podemos fazer esse teste pelo interpretador do Python. Dentro dele, vamos colar o código:

```
>>> import pandas as pd
>>> from collections import Counter
>>>
>>> # teste inicial: home, busca, logado => comprou
```

```

... # home, busca
... # home, logado
... # busca, logado
... # busca: 85,71% (7 testes)
...
>>> df = pd.read_csv('buscas.csv')
>>> X_df = df[['home', 'busca', 'logado']]
>>> Y_df = df['comprou']
>>>
>>> Xdummies_df = pd.get_dummies(X_df)
>>> Ydummies_df = Y_df
>>>
>>> X = Xdummies_df.values
>>> Y = Ydummies_df.values
>>>
>>> porcentagem_de_treino = 0.8
>>> porcentagem_de_teste = 0.1
>>>
>>> tamanho_de_treino = int(porcentagem_de_treino * len(Y))
>>> tamanho_de_teste = int(porcentagem_de_teste * len(Y))
>>> tamanho_de_validacao = len(Y) - tamanho_de_treino - tamanho_de_teste
>>>
>>> treino_dados = X[:tamanho_de_treino]
>>> treino_marcacoes = Y[:tamanho_de_treino]
>>>
>>> fim_de_treino = tamanho_de_treino + tamanho_de_teste
>>>
>>> teste_dados = X[tamanho_de_treino:fim_de_treino]
>>> teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]
>>>
>>> validacao_dados = X[fim_de_treino:]
>>> validacao_marcacoes = Y[fim_de_treino:]

```

Vamos testar primeiro os nossos dados de treino. Começaremos vendo o seu primeiro valor, isto é, a posição 0:

```

>>> treino_dados[0]
array([0, 1, 1, 0, 0])
>>>

```

Então, vamos verificar também o nosso  $X$  na sua primeira posição:

```
>>> X[0]
array([0, 1, 1, 0, 0])
>>>
```

Por enquanto, funcionou como o esperado, então vejamos o último valor dos nossos dados de treino (nesse caso, o valor 799):

```
>>> treino_dados[799]
array([1, 1, 0, 1, 0])
>>> X[799]
array([1, 1, 0, 1, 0])
>>>
```

Como podemos ver, está funcionando como o esperado. Será que os nossos dados de teste estão corretos também? Da mesma forma que fizemos com os dados de treino, pegaremos o seu primeiro valor e o último. Começaremos pelo primeiro valor:

```
>>> teste_dados[0]
array([1, 1, 0, 0, 1])
>>>
```

Agora vejamos o nosso `X`. Porém, há um detalhe: a qual posição o primeiro valor dos nossos dados de teste refere-se, para o `X`? Lembra que ele vai do 800 ao 899? Ou seja, a primeira posição dos nossos dados de teste no `X` é justamente a 800. Vejamos o resultado:

```
>>> teste_dados[0]
array([1, 1, 0, 0, 1])
>>> X[800]
array([1, 1, 0, 0, 1])
>>>
```

Agora vamos pegar o último valor dos nossos dados de teste e o valor do `X` que representa esse último valor — ou seja, posição 99 dos dados de teste e 899 do `X`.

```
>>> teste_dados[99]
```

```
array([1, 1, 0, 1, 0])
>>> X[899]
array([1, 1, 0, 1, 0])
>>>
```

Por fim, vejamos os nossos dados de validação. Qual é a sua primeira posição? 0, certo? Vejamos:

```
>>> validacao_dados[0]
array([0, 0, 0, 1, 0])
>>>
```

Mas e a posição do `X` que representa esse primeiro valor dos dados de validação? Lembra que os dados de validação vão do 900 ao 999? Portanto, esse valor refere-se ao valor da posição 900 do `X`:

```
>>> validacao_dados[0]
array([0, 0, 0, 1, 0])
>>> X[900]
array([0, 0, 0, 1, 0])
>>>
```

Pegaremos o último valor, que é a posição 99 dos dados de validação, e 999 do `X`:

```
>>> validacao_dados[99]
array([0, 1, 0, 0, 1])
>>> X[999]
array([0, 1, 0, 0, 1])
>>>
```

Calculamos as nossas variáveis que representarão tanto os nossos dados de treino, teste e validação. Porém, ainda temos de isolar o resultado de cada um dos nossos algoritmos para que possamos comparar qual foi o melhor. Isto é, precisamos retornar o resultado da função `fit_and_predict`. Então, retornaremos a variável `taxa_de_acerto`:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes,
teste_dados, teste_marcacoes):
    # restante do código
    return taxa_de_acerto
```

Agora que essa função contém um retorno, podemos atribuí-la a uma variável que identifica o resultado de cada um dos algoritmos, por exemplo, criar duas variáveis, `resultadoMultinomial` e `resultadoAdaBoost`.

```
# restante do código

from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modelo, t
reino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

from sklearn.ensemble import AdaBoostClassifier
modelo = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modelo,
treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
```

Com ambos os resultados, precisamos apenas verificar qual é o maior, qual é a maneira mais simples para isso? Fazendo um único `if` que verifica qual deles é o maior:

```
if resultadoMultinomial > resultadoAdaBoost:
```

Então, retornamos o maior para uma variável chamada `vencedor`. Mas observe que ambos os modelos estão com os mesmos nomes:

```
# restante do código
from sklearn.naive_bayes import MultinomialNB
modelo = MultinomialNB()

from sklearn.ensemble import AdaBoostClassifier
modelo = AdaBoostClassifier()
```

Para resolver isso, basta alterarmos o nome das variáveis para

que cada uma identifique o algoritmo:

```
# restante do código
from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
```

Lembre-se de alterar na chamada do método `fit_and_predict` também:

```
from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMultinomial, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
```

Voltando para o nosso `if`, para identificar o vencedor, simplesmente atribuímos à variável `vencedor` o modelo de acordo com o teste. Ou seja, atribuiremos o `modeloMultinomial` caso o resultado do `Multinomial` seja maior e, se for falso, retornamos o `modeloAdaBoost`:

```
if resultadoMultinomial > resultadoAdaBoost:
    vencedor = modeloMultinomial
else:
    vencedor = modeloAdaBoost
```

Agora que temos o modelo vencedor, precisamos utilizá-lo para realizar o teste real. Mas como faremos esse teste real? Primeiro, vamos criar uma função chamada `teste_real`:

```
def teste_real:
```

Para essa função, precisamos enviar o modelo vencedor e os dados e marcações de validação:

```
def teste_real(modelo, validacao_dados, validacao_marcacoes):
```

Com a função definida, precisamos implementá-la. Sabendo que ela precisa realizar um novo teste para o modelo vencedor, precisamos pedir para esse modelo prever os novos dados, ou seja, chamar a função `predict` com os dados de validação:

```
def teste_real(modelo, validacao_dados, validacao_marcacoes):  
    resultado = modelo.predict(validacao_dados)
```

Realizamos o mesmo cálculo de acertos conforme fizemos antes. Entretanto, a diferença é que, em vez de usar os dados de teste, usaremos os dados de validação:

```
def teste_real(modelo, validacao_dados, validacao_marcacoes):  
    resultado = modelo.predict(validacao_dados)  
    acertos = resultado == validacao_marcacoes  
  
    total_de_acertos = sum(acertos)  
    total_de_elementos = len(validacao_marcacoes)  
  
    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento  
s
```

Por fim, precisamos imprimir a mensagem indicando o resultado do teste real para o algoritmo vencedor:

```
def teste_real(modelo, validacao_dados, validacao_marcacoes):  
    resultado = modelo.predict(validacao_dados)  
    acertos = resultado == validacao_marcacoes  
  
    total_de_acertos = sum(acertos)  
    total_de_elementos = len(validacao_marcacoes)  
  
    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento  
s
```

```
msg = "Taxa de acerto do vencedor entre os dois algoritmos no mundo real: {}".format(taxa_de_acerto)
print(msg)
```

Com a função implementada, podemos chamá-la após acharmos o vencedor:

```
# restante do código

if resultadoMultinomial > resultadoAdaBoost:
    vencedor = modeloMultinomial
else:
    vencedor = modeloAdaBoost

teste_real(vencedor, validacao_dados, validacao_marcacoes)
```

Ao testarmos o nosso código, temos o seguinte resultado:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo MultinomialNB: 82.0
Taxa de acerto do algoritmo AdaBoostClassifier: 84.0
Taxa de acerto base: 82.000000
Total de teste: 100
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 85.0
```

Nosso algoritmo vencedor foi o `AdaBoost` com 84%. Porém, quando usamos para um teste real, isto é, com dados que ele nunca treinou ou havia testado antes, ele conseguiu acertar 85%. Em outras palavras, ele foi melhor do que o esperado, baseado nos nossos testes. Mas ainda existe um detalhe: a taxa de acerto base ainda está baseada nos dados de teste.

```
# restante do código
acerto_base = max(Counter(teste_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(teste_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(teste_dados)
print("Total de teste: %d" % total_de_elementos)
```

Portanto, precisamos alterar para os dados de validação:

```
acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)
```

Além disso, vamos adicioná-lo no final do código, da mesma forma como havíamos feito antes:

```
# restante do código
if resultadoMultinomial > resultadoAdaBoost:
    vencedor = modeloMultinomial
else:
    vencedor = modeloAdaBoost

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)
```

Vamos verificar se tudo está funcionando como o esperado:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo MultinomialNB: 82.0
Taxa de acerto do algoritmo AdaBoostClassifier: 84.0
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 85.0
Taxa de acerto base: 82.000000
Total de teste: 100
```

Como podemos ver, o nosso algoritmo está funcionando da forma correta, pois, além de testá-lo com os dados reais, estamos testando também o algoritmo base com esses mesmos dados reais.

Afinal, o nosso objetivo é comparar o nosso algoritmo com o algoritmo base, dada uma situação do mundo real.

E para os dados do arquivo `buscas2.csv`? Será que o nosso algoritmo funciona? Vejamos:

```
import pandas as pd
from collections import Counter

# teste inicial: home, busca, logado => comprou
# home, busca
# home, logado
# busca, logado
# busca: 85,71% (7 testes)

df = pd.read_csv('buscas2.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']
# restante do código
```

Testando novamente o nosso algoritmo:

```
> python3 classifica_buscas.py
Taxa de acerto do algoritmo MultinomialNB: 85.7142857143
Taxa de acerto do algoritmo AdaBoostClassifier: 85.7142857143
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 62.5
Taxa de acerto base: 62.500000
Total de teste: 8
```

Perceba que tanto o `Multinomial` quanto o `AdaBoost` obtiveram resultados equivalentes para essa situação. Então, qual foi o algoritmo que ele escolheu? O `AdaBoost`, pois a nossa comparação só escolhe o `Multinomial` caso o resultado dele seja maior que o do `AdaBoost`. Em outras palavras, o teste foi falso, por isso o `AdaBoost` foi escolhido. Além disso, podemos ver que o resultado do algoritmo vencedor não fez diferença alguma com o algoritmo base considerando esse conjunto de dados.

A partir desse resultado, o que podemos concluir? Conseguimos contestar que, para esse conjunto de dados, o algoritmo AdaBoost não teve tanta eficiência quanto o conjunto de dados anterior. Também podemos supor que, se utilizássemos o Multinomial , talvez o resultado fosse melhor.

Por fim, o nosso arquivo final fica da seguinte forma:

```
import pandas as pd
from collections import Counter

# teste inicial: home, busca, logado => comprou
# home, busca
# home, logado
# busca, logado
# busca: 85,71% (7 testes)
# comparando [adaboost, multinomial] usando [home, busca,logado]

df = pd.read_csv('buscas2.csv')
X_df = df[['home', 'busca', 'logado']]
Y_df = df['comprou']

Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8
porcentagem_de_teste = 0.1

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_teste = int(porcentagem_de_teste * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino - tamanho_de_te
ste

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

fim_de_treino = tamanho_de_treino + tamanho_de_teste

teste_dados = X[tamanho_de_treino:fim_de_treino]
```

```

teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]

validacao_dados = X[fim_de_treino:]
validacao_marcacoes = Y[fim_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes,
                    teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)

    resultado = modelo.predict(teste_dados)

    acertos = resultado == teste_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(teste_dados)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento
    s

    msg = "Taxa de acerto do algoritmo {0}: {1}".format(nome, tax
a_de_acerto)

    print(msg)
    return taxa_de_acerto

def teste_real(modelo, validacao_dados, validacao_marcacoes):
    resultado = modelo.predict(validacao_dados)
    acertos = resultado == validacao_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(validacao_marcacoes)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento
    s

    msg = "Taxa de acerto do vencedor entre os dois algoritmos no
mundo real: {0}".format(taxa_de_acerto)
    print(msg)

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMul
tinomial, treino_dados, treino_marcacoes, teste_dados, teste_marc
acoes)

```

```

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost,
treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

if resultadoMultinomial > resultadoAdaBoost:
    vencedor = modeloMultinomial
else:
    vencedor = modeloAdaBoost

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)

```

Repare que, na trilha de testes feitos, já adicionei o caso atual e um teste de comparação de Adaboost com MultinomialNB, usando as três variáveis em mãos.

## 6.2 RESUMINDO

Neste capítulo, começamos com o algoritmo `Multinomial`. Sabemos como esse algoritmo funciona por trás, ou seja, que ele utiliza um modelo baseado em probabilidades de os eventos acontecerem. Por exemplo, se em uma determinada região chove com mais frequência, ele vai prever que nos próximos dias vai chover. É uma decisão bem boba, porém tem a capacidade de prever diversas coisas.

Também vimos que as variáveis que usamos para cada classificação podem fazer uma grande diferença, pois dependendo

das variações que utilizamos, o nosso modelo pode atingir um resultado melhor ou pior. Por exemplo, se eu digo apenas que um animal tem quatro patas, você consegue distinguir se ele é um porco ou um cachorro? Por enquanto, não está tão claro, certo?

E se eu falar que ele é meio rosinha? Ainda não temos tanta certeza, correto? Afinal, existem também cachorros rosinhos. Mas e se eu afirmar que esse animal também faz *auau*? Agora sim já temos a capacidade de distinguir com mais precisão se ele é um cachorro ou um porco.

De acordo com as variáveis que damos para o nosso modelo, ele pode apresentar um resultado melhor ou pior. Porém, isso não significa que quanto mais variáveis o nosso algoritmo receber, melhor será o resultado, já que dependendo da quantidade de variáveis com que ele tiver de lidar, em vez de ajudar, isso vai confundi-lo. Serão tantas variáveis que ele já não terá tanta certeza do que ele está classificando.

Além disso, quando efetuamos diversos testes com diversas variáveis, podemos entrar no caso em que encontramos um resultado tão perfeito para os testes que foi simplesmente por sorte. Em outras palavras, esse resultado foi encontrado por coincidência e, no mundo real, esse tipo de resultado pode nos causar diversos problemas, pois acreditamos que o nosso algoritmo está tão bom que, no momento em que vamos utilizá-lo para dados do mundo real, acabamos percebendo que ele não era tão bom quanto imaginávamos. Ou pior ainda, ele não acertará o quanto esperávamos que ele acertasse, e nos daremos mal.

Vimos também que, para cada teste que realizamos, é de extrema importância anotarmos qual foi o teste e o resultado que

obtivemos, pois é uma forma de provar que aquele resultado a que chegamos não foi por acaso. Isto é, com esse histórico de cada teste realizado, temos a capacidade de provar todos os passos que realizamos para chegar ao resultado apresentado.

Além do `Multinomial`, vimos também o `AdaBoost`, que tenta se adaptar de acordo com o algoritmo para achar o melhor resultado. Rodamos tanto o `Multinomial` quanto o `AdaBoost` e vimos que, dependendo do conjunto de dados, cada algoritmo obtém um resultado diferente.

Também implementamos os dois algoritmos no mesmo código para testá-los ao mesmo tempo, e então aquele que retornava o melhor resultado (o maior resultado) escolhíamos como o algoritmo vencedor. Tendo o vencedor em mãos, fazíamos mais um teste, que era justamente um teste do mundo real. Ou seja, pegávamos novos elementos que ele nunca tinha visto antes — nem no seu treino e nem no teste —, então pedíamos para ele prever para nós.

Dessa forma, podemos ter a certeza de como será o seu comportamento, dada uma situação do mundo real. E é justamente esse valor final que vai validar o quão bom o nosso algoritmo é.

Com isso, concluímos que tivemos três fases para o nosso algoritmo, que são:

1. Treinar os algoritmos;
2. Testar os algoritmos;
3. Escolher o melhor entre eles e testar com os dados reais.

Para efetuar esses passos, dividimos o nosso conjunto de dados entre: 80% para treino, 10% para teste e 10% para validação do

mundo real. Dessa forma, conseguimos rodar todos os passos para verificarmos o quanto bom o nosso algoritmo é no mundo real.

# NOVOS CONCEITOS DE CLASSIFICAÇÃO

Até o momento, vimos diversos exemplos de classificação, tais como: verificar se um e-mail é *spam* ou não, se um animal é um cachorro ou um porco, se um cliente vai comprar ou não no nosso site, entre outros. Vimos que podem existir milhares de classificações no nosso dia a dia que conseguimos fazer.

Entretanto, é interessante o computador realizar esse tipo de tarefa para nós, justamente para que consigamos prever resultados esperados, ou seja, evitar de abrir um e-mail que é *spam*, entregar um produto que o nosso cliente espera, conseguir conversar com alunos que provavelmente vão mal na prova, entre diversas ações que podemos tomar para alcançar o nosso objetivo. Todo esse conteúdo foi visto até agora, então vem a pergunta:

- Existe mais alguma coisa que precisamos saber?

No mundo real, poderíamos levar mais adiante. Em vez de querermos saber apenas se o cliente vai comprar ou não, podemos também ter o interesse em uma gama maior de informações, por exemplo, saber sobre sua satisfação — se ele está feliz, neutro, triste, supercontente ou com raiva. Então aparecem as questões:

- Com qual desses clientes temos de interagir?
- Como devemos interagir com cada um desses tipos de clientes?

Um outro exemplo para esse tipo de cenário seria: dado um produto novo que estamos querendo lançar, surgem dúvidas como:

- Será que vai ser um produto de sucesso?
- Será que vai ser um fracasso?
- Será que vai ser um produto neutro?

Perceba que, nesse instante, podemos ter mais de duas categorias, ou seja, três, cinco, 10 ou muitas mais para um serviço, produto ou qualquer tipo de elemento que queremos classificar.

Além disso, atualmente, estamos testando e validando os nossos dados de uma maneira bem simples, que é testar, escolher o melhor entre os dois algoritmos e validar. Porém, e quando tivermos, por exemplo, quatro, 10 ou mais algoritmos? Também queremos ser capazes de validá-los, e eleger o vencedor entre uma grande variação de algoritmos que vamos usar.

Também queremos realizar testes diferentes dos quais vimos até agora. Por exemplo, os nossos testes são realizados apenas uma única vez, mas e se esses dados tiverem algum vício? Ou então, e se tiver um erro que desconhecemos? Como podemos ter a certeza de que, dados os testes que realizamos para os nossos algoritmos, o resultado final que obtivemos é de fato esperado para o mundo real? Esse também será um dos assuntos que vamos abordar no decorrer dos próximos capítulos.

Até agora estávamos trabalhando apenas com números, isto é, categorias numéricas — estas baseadas em apenas números ou em palavras que eram transformadas em números diretamente (por exemplo, 0, 1 ou 2). Porém, no mundo real, também trabalharemos com produção de texto, então, precisamos também classificar textos!

Ou melhor, dado um parágrafo escrito de um livro, queremos saber se ele venderá muito ou não. Consegue perceber que dessa vez a classificação é referente ao conteúdo do texto? Isto é, referente à língua portuguesa, às próprias palavras em si. Mas como podemos processar os nossos textos para classificar as palavras? Seja por algum sentimento, intenção ou qualquer categoria.

Veremos que, para esse tipo de classificação, tomaremos alguns cuidados, como realizar limpeza nos nossos textos para evitar qualquer lixo, ou seja, qualquer informação que não faça sentido algum com o que queremos classificar. Perceba que agora temos diversas novas situações que ainda não vimos, e no decorrer dos próximos capítulos, abordaremos cada uma delas.

## 7.1 CLASSIFICANDO UM ELEMENTO COM TRÊS CATEGORIAS

No mundo real, sabemos que em diversas situações podemos realizar classificações entre duas categorias, como verificar os alunos que vão reprovar ou não, ou os alunos que vão desistir da escola durante o período letivo ou não. Mas podemos nos interessar por classificar mais do que duas categorias, porém, como fazemos para classificar um elemento que possui mais do que duas

categorias diferentes?

Por exemplo, quando recebemos um e-mail, podemos categorizá-lo como *spam* ou não. Logo, estamos classificando apenas duas categorias, mas, além da classificação de *spam*, podemos também classificá-lo como *spam*, *promoção*, *fórum*, *update*, *importante*, *familiar* ou *normal*. Apenas com esse pequeno exemplo que fizemos apareceram sete categorias distintas para classificarmos um e-mail.

Agora, quando um e-mail chegar, não apenas tentaremos classificá-lo entre *spam* ou não, pois teremos sete categorias possíveis para a sua classificação! No nosso primeiro exemplo, usaremos apenas três categorias que demonstram, conceitualmente, a mesma situação, porém, de uma forma reduzida.

Vamos dar uma olhada no nosso primeiro cliente? Ele possui as seguintes características:

- **Último acesso:** visitou ontem (1 dia atrás).
- **Frequência de acesso:** visitou 4 dias.
- **Se inscreveu:** 4 semanas atrás.
- **Está se sentindo:** alegre.

A primeira característica informa a última visita que o cliente fez no nosso site, nesse caso, ontem, um dia atrás. Mas o que ela significa exatamente? Essa informação responde à seguinte pergunta: "o quanto recente foi o último acesso desse cliente?".

Identificamos isso como **recência**. Quando dizemos o último acesso, ou melhor, a recência do nosso cliente ao nosso site, não significa que precisa ser necessariamente em dias, poderia, por

exemplo, ser em horas, semanas, meses ou anos. Ou seja, podemos medir com qualquer unidade, mas nesse exemplo utilizaremos dias.

Mas por que nos interessaria saber a recência desse cliente ao nosso site? Pois, baseado nessa informação, podemos tentar medir como ele está se sentindo em relação a um cliente que acessou há uma semana, ou há um mês ou há um ano.

A segunda característica descreve qual é a frequência de visitas que o cliente teve após sua inscrição. Isso não significa uma frequência seguida. Em outras palavras, estamos respondendo a seguinte pergunta: "o quanto frequente foi o acesso do nosso cliente?", nesse caso, quatro dias distintos.

Da mesma forma que vimos na característica de recência, poderíamos utilizar outras unidades de medida para essa informação, como horas, meses ou anos. Repare que agora estamos usando uma variável diferente, pois estamos medindo quantos **dias distintos** o cliente visitou o nosso site, isto é, a sua frequência. Esse tipo de informação pode ser relevante, pois uma pessoa que acessou quatro dias e outra que não acessou nenhum dia após a inscrição indicam se o meu produto é ou não interessante.

A próxima característica refere-se ao tempo em que o cliente se inscreveu no nosso site. Mas por que é importante saber há quanto tempo um cliente se inscreveu? Pois dependendo do tempo de inscrição, cada cliente poderá conter diferentes tipos de comportamento, ou seja, um cliente que se inscreveu semana passada tem um comportamento totalmente diferente de um que se inscreveu há 3 anos.

Podemos usar o site da Alura como exemplo. No início, havia entre 5 a 10 cursos, porém, atualmente, existem mais de 400. Você consegue perceber que o tempo em que um cliente está dentro de uma plataforma influencia? Porém, isso não significa que quanto mais ou menos tempo o cliente estiver associado a um produto ele vai se sentir melhor ou pior. Por exemplo, existem empresas que, quanto mais tempo ficamos vinculados aos seus produtos, mais contentes ficamos; mas também existem empresas que, quanto menos tempo estamos atrelados ao seu produto, mais chateados ficamos.

Assim, podemos concluir que não existe uma regra geral para dizer se ele ficará contente ou não dependendo do tempo em que ele está inscrito. Portanto, teremos de analisar essa variável para que o nosso algoritmo consiga encontrar um padrão para esse tipo de dado.

Além dessas três variáveis (recência, frequência e tempo de cadastro), poderíamos utilizar diversas outras: quanto dinheiro ele gastou, quantos vídeos ele assistiu (considerando a Alura como exemplo), quantos exercícios ele fez, quantas pessoas ele ajudou, quantas dúvidas ele postou, entre outras. Mas vamos apenas utilizar essas variáveis para o nosso exemplo.

O que queremos classificar com apenas essas variáveis? Para esse exemplo, tentaremos prever se o nosso cliente está alegre, neutro ou chateado com o nosso produto. Nesse primeiro exemplo, vimos as características de um cliente que está alegre. Mas o que significa cada um desses estados? Vejamos:

- **Alegre:** ele está satisfeito, portanto não precisamos nos preocupar tanto.

- **Neutro:** ele não demonstra nem satisfação ou insatisfação com o nosso produto, ou seja, de repente podemos considerar um contato para entender melhor a situação dele.
- **Chateado:** ele está insatisfeito, logo, é de extrema importância entendermos o que está acontecendo, em outras palavras, precisaremos verificar o motivo de ele ficar chateado para tentar ajudá-lo de alguma maneira e recuperar a situação.

Agora, em vez de duas categorias, classificaremos os nossos clientes entre três, isto é, alegre, neutro e chateado. Vejamos um outro exemplo de cliente:

- **Último acesso:** visitou anteontem (2 dias atrás).
- **Frequência de acesso:** acessou 1 dia.
- **Se inscreveu:** 2 semanas atrás.
- **Está se sentido:** neutro.

Como foi o comportamento desse cliente no nosso site? Podemos ver que o último acesso dele foi há dois dias, ou seja, sua recência é 2. Esse cliente acessou em um único dia, porém, isso não significa que ele acessou apenas uma vez, pois nesse mesmo dia ele poderia ter acessado várias vezes. Mesmo assim, note que não estamos analisando a quantidade de vezes que ele acessou, e sim quantos **dias distintos** ele acessou, por isso marcamos apenas como 1 dia. Além disso, esse cliente se inscreveu há duas semanas, então, sabemos que ele está neutro, pois o pessoal do comercial entrou em contato com ele e verificou.

Precisamos entrar em contato justamente para entender como ele está se sentindo, da mesma forma como classificamos se um e-

mail é ou não um *spam*. Ou seja, precisamos ler o e-mail para podermos classificá-lo. Para essa situação, temos esses conjuntos de dados que foram extraídos de uma base de dados de um site de vendas e precisamos entrar em contato para verificar qual é a sensação dele com o nosso produto.

Mas teremos de fazer exatamente a mesma coisa em todos os casos? Não! Tudo depende do conjunto de dados usado e o que queremos classificar. Por exemplo, poderíamos conter um determinado conjunto de dados sobre um produto nosso, então, queremos prever se ele fará sucesso ou não antes mesmo de lançar. Para esse caso, não faz sentido perguntarmos ao produto se ele fará sucesso ou não, ou seja, usaremos apenas o nosso conjunto de dados e, baseado nele, o classificaremos.

Por fim, vejamos mais um possível cliente para o nosso exemplo:

- **Último acesso:** visitou 3 dias atrás.
- **Frequência de acesso:** acessou 1 dia.
- **Se inscreveu:** 7 semanas atrás.
- **Está se sentindo:** chateado.

Observe que esse cliente teve uma recência de três dias atrás, então obteve uma frequência de apenas um dia, porém ele se inscreveu há sete semanas. Por fim, verificamos que ele está chateado.

Mas e agora? O que fazemos com essas informações? Vamos criar e preencher a nossa tabela de dados da mesma forma que fizemos anteriormente. Vejamos a sua estrutura:

recencia	frequencia	semanas	situacao
1	4	4	alegre
2	1	2	neutro
3	1	7	chateado

Figura 7.1: Os clientes

Repare que a nossa tabela contém quatro colunas referentes aos dados que temos de nossos clientes, ou seja, `recencia` , `frequencia` , `semanas` e `situacao` dos clientes que vimos como exemplo. No último campo, preenchemos com palavras, isto é, `alegre` , `neutro` e `chateado` , porém, vamos traduzir essas palavras em números. Nesse caso, usaremos 2 para `alegre` , 1 para `neutro` e 0 para `chateado` :

recencia	frequencia	semanas	situacao
1	4	4	2
2	1	2	1
3	1	7	0

0	chateado
1	neutro
2	alegre

Figura 7.2: Os clientes

Note que caímos em um cenário muito similar ao que já vimos até agora, então qual é a diferença desse caso para os demais que vimos? É justamente o fato de usarmos três categorias para classificar os nossos elementos, em outras palavras, verificar se o

cliente está alegre , neutro ou chateado .

Considerando todo esse contexto que acabamos de ver, precisamos agora implementar o nosso código. Antes mesmo de criar o nosso algoritmo, vamos primeiro pegar os novos dados na planilha do Google Spreadsheets, em <http://bit.ly/1NogiPr>:

The screenshot shows a Google Sheets document with the title "Curso Machine Learning: Introdução à Classificação - entrada". The spreadsheet contains two main sections: "recencia" and "buscas".

**Section "recencia":**

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	recencia	frequencia	semanas_de_int_situacao										
2	1	4	4	2									
3	2	1	2	1									
4	1	4	2	2									
5	1	3	8	1									
6	2	2	1	1									
7	1	4	2	2									
8	1	1	5	1									
9	1	3	8	1									
10	3	1	1	1									
11	3	2	6	1									
12	2	3	6	1									
13	3	1	2	1									
14	3	2	7	1									
15	1	4	2	2									
16	1	3	1	2									
17	1	4	8	1									
18	1	3	7	1									
19	3	1	7	0									
20	2	1	5	1									
21	3	2	6	1									
22	2	2	8	1									
23	3	1	4	1									

**Section "buscas":**

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	buscas	buscas2	frequencia	doacoes	buscas_sim_ou_nao	sair	situacao_do_cliente	emails					
2													

Agora estamos pegando os dados da aba `situacao_do_cliente`. Salve o arquivo como CSV com o nome `situacao_do_cliente.csv` dentro do diretório onde preferir, lembrando que **trabalharemos exatamente no local onde salvaremos esse arquivo**. Agora, crie um arquivo dentro do diretório em que salvou o CSV com o nome `situacao_do_cliente.py`, que será o nosso arquivo Python para escrevermos o nosso código.

Não implementaremos todo o código desde o zero, pois reutilizaremos uma boa parte do algoritmo que implementamos no arquivo `classifica_buscas.py`. Segue o código:

```

import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8
porcentagem_de_teste = 0.1

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_teste = int(porcentagem_de_teste * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino - tamanho_de_te
ste

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

fim_de_treino = tamanho_de_treino + tamanho_de_teste

teste_dados = X[tamanho_de_treino:fim_de_treino]
teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]

validacao_dados = X[fim_de_treino:]
validacao_marcacoes = Y[fim_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes,
                    teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)

    resultado = modelo.predict(teste_dados)

    acertos = resultado == teste_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(teste_dados)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento

```

```

S

    msg = "Taxa de acerto do algoritmo {0}: {1}".format(nome, tax
a_de_acerto)

    print(msg)
    return taxa_de_acerto

def teste_real(modelo, validacao_dados, validacao_marcacoes):
    resultado = modelo.predict(validacao_dados)
    acertos = resultado == validacao_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(validacao_marcacoes)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento
S

    msg = "Taxa de acerto do vencedor entre os dois algoritmos no
mundo real: {0}".format(taxa_de_acerto)
    print(msg)

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMul
tinomial, treino_dados, treino_marcacoes, teste_dados, teste_marca
coes)

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloA
daBoost, treino_dados, treino_marcacoes, teste_dados, teste_marca
coes)

if resultadoMultinomial > resultadoAdaBoost:
    vencedor = modeloMultinomial
else:
    vencedor = modeloAdaBoost

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcaco
es)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

```

```
total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)
```

Copie o código e cole no arquivo `situacao_do_cliente.py`. A única diferença entre esse código e o que usamos no `classifica_buscas.py` é que apagamos as marcações dos testes que realizamos, pois eles eram válidos para aquele outro conjunto de dados que havíamos usado e também para o nome das variáveis:

- **De:**

- Dados: 'home', 'busca', 'logado'
- Marcação: 'comprou' .

- **Para:**

- Dados:  
`'recencia', 'frequencia', 'semanas_de_inscricao'`
- Marcação: 'situacao'

Usaremos os dados para essa nova situação em que estamos, que é avaliar a situação do cliente entre: alegre , neutro ou chateado . Mas e esses novos dados? Qual é a diferença deles com o que tínhamos anteriormente? Leremos o arquivo `situacao_do_cliente.csv` como em:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']
# ...
```

Vejamos o arquivo `situacao_do_cliente.csv` :

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,2
2,1,2,1
1,4,2,2
1,3,8,1
2,2,1,1
1,4,2,2
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Observe que, anteriormente, não havíamos usado dados com marcações com mais de dois valores, porém, dessa vez, estamos variando entre três (0, 1 ou 2).

Será que tanto o algoritmo AdaBoost quanto o Multinomial conseguem trabalhar de forma eficiente com uma variável de marcação que varia entre três valores? Como podemos verificar? Simples! Rodando o nosso arquivo situacao\_do\_cliente.py :

```
> python3 situacao_do_cliente.py
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Nesse primeiro teste que realizamos, podemos concluir que o algoritmo Multinomial obteve um resultado de 72,72% e o AdaBoost de 68,18%, ou seja, o algoritmo vencedor foi o Multinomial. Ao executar o teste do mundo real, ele obteve um resultado superior ao esperado, nesse caso, 82,60%. Porém, mesmo sendo um resultado superior ao dos testes, ele é equivalente ao resultado do algoritmo base, que chuta o mesmo valor para todos.

Podemos até mesmo verificar o resultado do nosso algoritmo Multinomial imprimindo o resultado dentro da função teste\_real :

```
def teste_real(modelo, validacao_dados, validacao_marcacoes):  
    resultado = modelo.predict(validacao_dados)  
  
    print(resultado)  
  
    # restante do código
```

O resultado será:

```
> python3 situacao_do_cliente.py  
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273  
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818  
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]  
Taxa de acerto do vencedor entre os dois algoritmos no mundo real  
: 82.6086956522  
Taxa de acerto base: 82.608696  
Total de teste: 23
```

Nosso algoritmo que faz diversos cálculos para tentar adivinhar o que cada dado é chuta o mesmo valor para todos! Portanto, tanto o Multinomial quanto o AdaBoost não são tão bons para dados com variáveis que possuem três valores distintos. Isso significa que, para esse tipo de cenário com que nos deparamos, não faz sentido usarmos algum desses algoritmos.

Então, o que faremos? Precisamos usar algum outro algoritmo que seja capaz de resolver esse problema de uma maneira mais eficaz, isto é, com três valores diferentes para a variável de classificação.

A grande sacada para essa situação seria encontrar alguma maneira de utilizarmos os mesmos algoritmos, ou seja, alguma forma que os permita classificar elementos com mais de duas

categorias, além de zeros e uns — nesse caso, 0, 1 ou 2, ou então 0, 1, 2, 3 a N. Portanto, classificar com qualquer quantidade acima de 2.

Primeiramente precisamos investigar a fundo os resultados desses algoritmos. Por exemplo, podemos começar verificando o resultado de cada um no momento em que eles realizam os testes, isto é, imprimindo a variável `resultado` da função `fit_and_predict`:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes,
teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)

    resultado = modelo.predict(teste_dados)
    print(resultado)
```

Além disso, retire a impressão que está sendo realizada na função `teste_real`. Rodando novamente o nosso algoritmo, temos o seguinte resultado:

```
> python3 situacao_do_cliente.py
[1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1]
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Repare que tanto o `Multinomial` quanto o `AdaBoost` obtiveram resultados bem similares, porém, o `Multinomial` variou em apenas um dado, prevendo com valor 2 em vez de apenas 1. Quando nos deparamos com um problema grande como este, a grande sacada para resolvê-lo é diminuí-lo para pequenos problemas que já saibamos resolver.

Então, por onde podemos começar? Um bom ponto de partida seria pelos nossos dados, ou seja, o arquivo `situacao_do_cliente.csv`:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,2
2,1,2,1
1,4,2,2
1,3,8,1
2,2,1,1
1,4,2,2
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Observe que atualmente temos os três valores distintos para a coluna `situacao`, então que tal transformarmos todos os valores 2 em 1? Por exemplo, todas as vezes que for 0, reconheceremos como 0, porém, quando o valor for 1, ele pode ser tanto 1 como 2:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,1
2,1,2,1
1,4,2,1
1,3,8,1
2,2,1,1
1,4,2,1
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Assim, o nosso problema vai para:

`0=>0 1=>1, 2`

Supostamente, quando rodamos o `Multinomial`, ele consegue verificar o percentual tanto de zeros quanto de uns

(resto), e obtém o resultado respectivamente de 38% (zeros) e 62% (resto):

```
0=>0 1=>1,2 multinomial 0 ou do resto (38%, resto 62%)
```

Aparentemente, essa redução do problema não foi de grande ajuda, ou seja, vamos retornar aos valores que estavam anteriormente:

```
recencia,frequencia,semanas_de_inscricao,situacao  
1,4,4,2  
2,1,2,1  
1,4,2,2  
1,3,8,1  
2,2,1,1  
1,4,2,2  
1,1,5,1  
1,3,8,1  
3,1,1,1  
...  
4,1,6,0
```

Que tal tentarmos reduzir novamente? Mas, em vez de transformarmos o 2 em 1, vamos transformar o 2 em 0:

```
recencia,frequencia,semanas_de_inscricao,situacao  
1,4,4,0  
2,1,2,1  
1,4,2,0  
1,3,8,1  
2,2,1,1  
1,4,2,0  
1,1,5,1  
1,3,8,1  
3,1,1,1  
...  
4,1,6,0
```

Observe que novamente entramos no caso em que o nosso algoritmo consegue resolver, ou seja, com apenas dois valores. Os nossos dados estão sendo representados da seguinte forma:

```
0=>0, 2 1=>1
```

Quando rodamos novamente o algoritmo `Multinomial` para esses dados, supostamente obtemos o resultado de 44% para uns e o resto é de 56%:

```
0=>0, 2 1=>1 multinomial 1 ou do resto (44%, resto 56%)
```

Como podemos ver, ainda não podemos ter certeza, pois o resto ainda é maior do que os valores de uns. Novamente, vamos retornar aos valores anteriores:

```
recencia,frequencia,semanas_de_inscricao,situacao  
1,4,4,2  
2,1,2,1  
1,4,2,2  
1,3,8,1  
2,2,1,1  
1,4,2,2  
1,1,5,1  
1,3,8,1  
3,1,1,1  
...  
4,1,6,0
```

Vamos fazer a última redução e verificar o resultado que obtemos, isto é, transformar o número 1 em 0:

```
recencia,frequencia,semanas_de_inscricao,situacao  
1,4,4,2  
2,1,2,0  
1,4,2,2  
1,3,8,0  
2,2,1,0  
1,4,2,2  
1,1,5,0  
1,3,8,0  
3,1,1,0  
...  
4,1,6,0
```

Note que novamente temos apenas duas categorias (categoria 0 e categoria 2), e então rodamos o `Multinomial` que supostamente nos devolve o seguinte resultado:

```
0=>0,1 2=>2 multinomial 2 ou do resto (20%, resto 80%)
```

Apenas reduzindo o problema das três possíveis maneiras, aparentemente não resolveu o nosso problema. Porém, se pedirmos para o nosso algoritmo rodar baseado nas três informações distintas que conseguimos ao diminuir o nosso problema maior:

```
0=>0 1=>1,2 multinomial 0 ou do resto (38%, resto 62%)
```

```
0=>0,2 1=>1 multinomial 1 ou do resto (44%, resto 56%)
```

```
0=>0,1 2=>2 multinomial 2 ou do resto (20%, resto 80%)
```

Ele vai analisar cada um dos resultados e provavelmente escolherá o que tiver maior probabilidade. Em outras palavras, ele vai verificar o 0 (38%), o 1 (44%) e o 2 (20%), então, escolherá o 1, pois, dentre eles, o 1 é o que contém maior chance. Você pode estar pensando que isso foi apenas uma manipulação de dados para conseguir chegar a um resultado esperado, e de fato foi, porém, consegue perceber o que acabamos de fazer?

Inicialmente, não sabíamos como resolver o problema para uma classificação de três categorias, então, reduzimos esse problema para um menor que fôssemos capazes de resolver, ou seja, um problema de duas categorias:

- `0=>0 1=>1, 2`
- `0=>0, 2 1=>1`
- `0=>0, 1 2=>2`

O primeiro resultado a que chegamos quando aplicamos a

redução foi a categoria 0 *versus* o resto. O segundo foi a categoria 1 *versus* o resto. Por fim, a categoria 2 *versus* o resto. Em outras palavras, se tivermos 5, 10, 15 ou N categorias, rodaríamos o nosso algoritmo para cada uma delas *versus* o resto. Esse algoritmo é conhecido como "um versus o resto" ou "um versus todos", mas, tecnicamente, é chamado de *One-vs-the-rest* (<http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html>).

Porém, o processo desse algoritmo é mais demorado, pois, em vez de utilizar um único classificador e rodar, estão sendo criados três classificadores, rodando e comparando-os. Ou seja, para esse caso, ficou três vezes mais "lerdo". Esse algoritmo roda a quantidade de categorias diferentes que tivermos, por exemplo, se tivermos: 5 categorias, 5 vezes; 10 categorias, 10 vezes; N categorias, N vezes!

Mas será que precisaremos implementar todos esses passos que fizemos na mão? Felizmente, o pacote **sklearn** já fornece uma implementação do *One-vs-the-rest* para nós, que é o **OneVsRestClassifier**. Então, vamos importar esse classificador no nosso código:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
# 0=>0 1=>1,2 multinomial 0 ou do resto (38%, resto 62%)
# 0=>0,2 1=>1 multinomial 1 ou do resto (44%, resto 56%)
# 0=>0,1 2=>2 multinomial 2 ou do resto (20%, resto 80%)

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMultinomial, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
```

```
from sklearn.ensemble import AdaBoostClassifier  
modeloAdaBoost = AdaBoostClassifier()
```

Quando utilizamos o `OneVsRest`, costumamos usar o algoritmo `LinearSVC`. Em outras palavras, o `OneVsRestClassifier` é basicamente um algoritmo que roda o modelo que damos para ele diversas vezes. Portanto, quando criarmos o modelo do `OneVsRestClassifier`:

```
# restante do código
```

```
from sklearn.multiclass import OneVsRestClassifier  
OneVsRestClassifier  
# 0=>0 1=>1,2 LinearSVC 0 ou do resto (38%, resto 62%)  
# 0=>0,2 1=>1 LinearSVC 1 ou do resto (44%, resto 56%)  
# 0=>0,1 2=>2 LinearSVC 2 ou do resto (20%, resto 80%)
```

Teremos de enviar o modelo que queremos que ele rode várias vezes, nesse caso, o `LinearSVC`. Para isso, precisamos importar o `LinearSVC` e enviar por parâmetro no momento em que criamos o modelo do `OneVsRestClassifier`:

```
# restante do código
```

```
from sklearn.multiclass import OneVsRestClassifier  
from sklearn.svm import LinearSVC  
modelo = OneVsRestClassifier(LinearSVC())
```

Além disso, vamos enviar o parâmetro `random_state = 0`, para rodar o `LinearSVC` de uma maneira fixa em vez de aleatória:

```
# restante do código
```

```
from sklearn.multiclass import OneVsRestClassifier  
from sklearn.svm import LinearSVC  
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))  
# 0=>0 1=>1,2 LinearSVC 0 ou do resto (38%, resto 62%)  
# 0=>0,2 1=>1 LinearSVC 1 ou do resto (44%, resto 56%)  
# 0=>0,1 2=>2 LinearSVC 2 ou do resto (20%, resto 80%)
```

E o restante do código? Faremos da mesma forma como fizemos anteriormente, ou seja, treinamos com o método `fit`:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))
modelo.fit(treino_dados, treino_marcacoes)
```

Após treinar o modelo, pedimos para ele prever e imprimir o resultado do que ele classificou:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))
modelo.fit(treino_dados, treino_marcacoes)
print(modelo.predict(teste_dados))
```

Será que ele vai chutar tudo 1 da mesma forma que o Multinomial e o AdaBoost fizeram? Será que ele vai variar entre zeros, uns e dois? Vamos testar? Antes de executar o código, retiraremos a impressão do resultado dos dois algoritmos dentro da função `fit_and_predict`:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes,
                     teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)

    resultado = modelo.predict(teste_dados)

# restante do código
```

Rodando o nosso código, obtemos o seguinte resultado:

```
> python3 situacao_do_cliente.py
[2 0 1 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 2 1]
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
```

```
Taxa de acerto do vencedor entre os dois algoritmos no mundo real  
: 82.6086956522  
Taxa de acerto base: 82.608696  
Total de teste: 23
```

Rpare que um resultado foi 2, 0 , então uma sequência de 1, depois um 2 e assim por diante. Realmente ele usou valores diferentes em sua classificação. Podemos fazer uma comparação simples, apenas imprimindo a variável teste\_marcacoes a seguir:

```
# restante do código
```

```
from sklearn.multiclass import OneVsRestClassifier  
from sklearn.svm import LinearSVC  
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))  
modelo.fit(treino_dados, treino_marcacoes)  
print(modelo.predict(teste_dados))  
print(teste_marcacoes)
```

Vejamos o resultado:

```
> python3 situacao_do_cliente.py  
[2 0 1 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 2 1]  
[2 0 1 1 1 1 1 2 1 2 1 1 0 1 1 1 1 0 1 2 1]  
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273  
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818  
Taxa de acerto do vencedor entre os dois algoritmos no mundo real  
: 82.6086956522  
Taxa de acerto base: 82.608696  
Total de teste: 23
```

Ele errou apenas dois valores, ou melhor, ele errou menos de 10%, e acertou mais de 90%! Portanto, podemos concluir que ele é o algoritmo ideal para realizar esse tipo de classificação, isto é, classificações com mais de duas categorias.

Os algoritmos capazes de classificar em várias classes (categorias) são conhecidos como *multiclasses*, e utilizamos o

`OneVsRestClassifier`. Porém, precisamos incluir esse algoritmo dentro do nosso processo que envolve as três fases (treino, teste e teste real), certo? Para isso, vamos renomear o modelo da mesma forma como fizemos com os outros algoritmos:

```
# restante do código
```

```
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
```

Em vez de realizar aquele treino e teste manualmente, usaremos a nossa função `fit_and_predict` retornando o resultado do nosso novo modelo:

```
# restante do código
```

```
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
```

Podemos apagar todo aquele código que fizemos para testarmos manualmente o `OneVsRestClassifier`, deixando apenas o seguinte:

```
# restante do código
```

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
```

Vamos verificar se ele está funcionando conforme o esperado:

```
> python3 situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 82.6086956522
```

```
Taxa de acerto base: 82.608696
Total de teste: 23
```

O resultado do `OneVsRestClassifier` foi impresso, mas ainda não o incluímos no terceiro passo, que é justamente eleger o vencedor e utilizá-lo para o teste real. Vejamos como estamos fazendo atualmente:

```
if resultadoMultinomial > resultadoAdaBoost:
    vencedor = modeloMultinomial
else:
    vencedor = modeloAdaBoost
```

Estamos fazendo um `if` e `else` para tentarmos verificar qual é o vencedor entre o `Multinomial` e o `AdaBoost`. Isso significa que teremos de adicionar mais um `if` para o `OneVsRest` também? Você consegue perceber como essa solução não parece boa? Afinal, se adicionarmos mais um algoritmo, precisaremos adicionar mais `if`s.

Como podemos resolver isso? Por meio de um dicionário! Então, vamos criar o nosso dicionário chamado `resultados`:

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
```

Identificamos o resultado do `OneVsRest` como `modeloOneVsRest`:

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[modeloOneVsRest] = resultadoOneVsRest
```

Faremos o mesmo para todos os outros algoritmos:

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[modeloOneVsRest] = resultadoOneVsRest

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMultinomial,
treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[modeloMultinomial] = resultadoMultinomial

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost,
treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[modeloAdaBoost] = resultadoAdaBoost
```

Vamos imprimir o nosso dicionário para verificar os valores que foram inseridos:

```
# restante do código
```

```
from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloA
daBoost, treino_dados, treino_marcacoes, teste_dados, teste_marca
coes)
resultados[modeloAdaBoost] = resultadoAdaBoost

print(resultados)
```

Vejamos o resultado do nosso dicionário:

```
> python3 situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
{MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True): 72.7
2727272727273, OneVsRestClassifier(estimator=LinearSVC(C=1.0, cla
ss_weight=None, dual=True, fit_intercept=True,
    intercept_scaling=1, loss='squared_hinge', max_iter=1000,
    multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
    verbose=0),
n_jobs=1): 90.9090909090909, AdaBoostClassifier(algorit
hm='SAMME.R', base_estimator=None,
    learning_rate=1.0, n_estimators=50, random_state=None):
68.18181818181819}
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Repare que é impressa uma mensagem gigante, onde estão inseridos os modelos que informamos junto com os seus respectivos valores. Logo, agora temos o nosso dicionário que contém tanto o modelo como a porcentagem da taxa de acerto.

Mas e agora? O que faremos com esse dicionário? Precisamos pegar o melhor desses modelos, em outras palavras, o modelo que contém o maior valor para a taxa de acerto. Será que podemos usar a função `max` ?

Até poderíamos, mas em vez de retornar o maior valor, a função `max` retornará a maior chave. Portanto, precisamos alternar as chaves e seus valores, isto é, colocaremos os resultados como chaves e os modelos como os valores do dicionário:

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMultinomial,
treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoMultinomial] = modeloMultinomial

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost,
treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoAdaBoost] = modeloAdaBoost
```

Se rodarmos novamente o nosso código:

```
> python3 situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
{68.181818181819: AdaBoostClassifier(algorithm='SAMME.R', base_
estimator=None,
learning_rate=1.0, n_estimators=50, random_state=None),
72.72727272727273: MultinomialNB(alpha=1.0, class_prior=None, fit_
prior=True), 90.9090909090909: OneVsRestClassifier(estimator=Lin_
earSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
```

```
        intercept_scaling=1, loss='squared_hinge', max_iter=1000,
        multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
        verbose=0),
        n_jobs=1)}
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Veja que agora os resultados são as chaves, e os modelos, os valores. Portanto, podemos usar o `max` enviando o nosso dicionário (`resultados`) e atribuindo para uma variável chamada `maximo`:

```
# restante do código
```

```
maximo = max(resultados)
```

E o vencedor? É justamente o valor do `maximo`. Ou seja, basta apenas pedirmos para o dicionário o modelo por meio da chave, que é justamente a variável `maximo`:

```
# restante do código
```

```
maximo = max(resultados)
vencedor = resultados[maximo]
```

Podemos até imprimir a variável `vencedor` para verificar os valores. Porém, vamos eliminar a impressão da variável `resultados` e também o `if` e `else` que verificavam quem era o vencedor entre o `Multinomial` e o `AdaBoost`:

```
# restante do código
```

```
resultados = {}
```

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
```

```

resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMultinomial
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoMultinomial] = modeloMultinomial

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoAdaBoost] = modeloAdaBoost

maximo = max(resultados)
vencedor = resultados[maximo]

print("Vencedor: ")
print(vencedor)

```

Vejamos o resultado:

```

> python3 situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Vencedor:
OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None,
dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 86.9565217391
Taxa de acerto base: 82.608696
Total de teste: 23

```

Agora estamos imprimindo o algoritmo vencedor e a taxa de acerto no mundo real, que é de 86,95%, superior ao resultado do

algoritmo base. Podemos concluir que, para esses dados, o algoritmo OneVsRest é mais eficaz.

Entretanto, será que o algoritmo base está funcionando bem para esse tipo de dado? Isto é, para dados com mais de duas categorias? Vamos verificar a resposta que ele está chutando, imprimindo o nosso Counter com a variável validacao\_marcacoes que representa os nossos dados de validação:

```
# restante do código

print(Counter(validacao_marcacoes))
acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)
```

Vejamos o resultado:

```
> python3 situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Vencedor:
OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None,
dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 86.9565217391
Counter({1: 19, 0: 3, 2: 1})
Taxa de acerto base: 82.608696
Total de teste: 23
```

De fato, ele funciona como o esperado, então não precisamos

nos preocupar com ele e podemos retirar essa impressão.

Atualmente, vimos três modelos de algoritmos diferentes:

- `MultinomialNB` ;
- `AdaBoostClassifier` ;
- `OneVsRestClassifier` .

Dentre os três algoritmos, acabamos de ver como o `OneVsRest` funciona por trás dos panos, que é justamente comparar um elemento com todos os outros, considerando o conjunto  $\{0, 1, 2\}$  :

- **1º Classificação:**  $0 \text{ versus } 1, 2$ .
- **2º Classificação:**  $1 \text{ versus } 0, 2$ .
- **3º Classificação:**  $2 \text{ versus } 0, 1$ .

Porém, poderíamos fazer uma abordagem um pouco diferente. Em outras palavras, em vez de ficar realizando o teste de um elemento *versus* todos os outros, poderíamos pegar um único elemento e testá-lo *versus* um outro único elemento.

Por exemplo, suponhamos que temos cinco elementos  $\{0, 1, 2, 3, 4\}$  distintos:

- **1º Classificação:**  $0 \text{ versus } 1$ .
- **2º Classificação:**  $0 \text{ versus } 2$ .
- **3º Classificação:**  $0 \text{ versus } 3$ .
- **4º Classificação:**  $0 \text{ versus } 4$ .
- **5º Classificação:**  $1 \text{ versus } 2$ .
- **6º Classificação:**  $1 \text{ versus } 3$ .
- **7º Classificação:**  $1 \text{ versus } 4$ .
- **8º Classificação:**  $2 \text{ versus } 3$ .

- **9º Classificação:** 2 *versus* 4.
- **10º Classificação:** 3 *versus* 4.

Repare que, dessa vez, em vez de realizar um único teste para cada elemento contra todos os outros, estamos criando um teste para cada elemento, isto é, um elemento *versus* apenas um outro elemento, portanto, 0 *vs* 1, 0 *vs* 2 e assim por diante. Esse tipo de algoritmo chamamos de **um versus um** ou, tecnicamente, **OneVsOne**.

Antes mesmo de implementar esse algoritmo no nosso código, precisamos entender um detalhe importante em relação ao **OneVsRest**. Vimos que, para o conjunto de dados  $\{0, 1, 2, 3, 4\}$ , foram realizados 10 testes. Mas, para esse mesmo conjunto de dados, quantos testes seriam realizados com o **OneVsRest**? Vejamos:

- **1º Classificação:** 0 *versus* 1,2,3,4.
- **2º Classificação:** 1 *versus* 0,2,3,4.
- **3º Classificação:** 2 *versus* 0,1,3,4.
- **4º Classificação:** 3 *versus* 0,1,2,4.
- **5º Classificação:** 4 *versus* 0,1,2,3.

Observe que seriam apenas cinco testes! Se fizermos a análise matemática do algoritmo **OneVsOne** (<http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsOneClassifier.html>), veremos que ele é um algoritmo quadrático. Portanto, podemos concluir que sua performance é pior em relação ao **OneVsRest**, que é linear.

Para quem se interessa na fórmula matemática arredondada, o número de comparações do algoritmo **OneVsOne**, em que

`elementos` é o número de elementos, temos algo como:

```
elementos * (elementos - 1) / 2
```

Vejamos algumas simulações da quantidade de testes:

- **3 elementos:**  $3 * (3 - 1) / 2 \rightarrow 3 * 2/2 = 3$  .
- **4 elementos:**  $4 * (4 - 1) / 2 \rightarrow 4 * 3/2 = 6$  .
- **5 elementos:**  $5 * (5 - 1) / 2 \rightarrow 5 * 4/2 = 10$  .

Esse algoritmo cresce quadraticamente de acordo com a quantidade de elementos. Logo, podemos concluir que, quanto mais elementos utilizarmos para testá-lo, cada vez mais lento ele ficará em relação ao `OneVsRest` .

Agora que vimos os pontos principais entre os dois algoritmos, vamos implementar também o `OneVsOne` . Para isso, primeiro precisamos importá-lo:

```
# restante do código

resultados = []

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest
```

```
from sklearn.multiclass import OneVsOneClassifier
```

A sua implementação é praticamente idêntica à do

OneVsRest . A única diferença é que mudaremos o nome das variáveis:

```
resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t
reino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne
```

Note que realizamos exatamente os mesmos passos na implementação do OneVsRest , isto é, importamos, criamos o modelo enviando o algoritmo LinearSVC por parâmetro, pedimos para treinar e testar com a função fit\_and\_predict e, por fim, adicionamos o seu resultado na variável resultados , ou seja, o nosso dicionário. Vamos verificar o resultado do nosso algoritmo agora?

```
> python3 situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo OneVsOne: 100.0
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Vencedor:
OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight=None,
dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 100.0
```

Taxa de acerto base: 82.608696

Total de teste: 23

Repare que o `OneVsOne` acertou 100% dos testes e foi escolhido como o algoritmo vencedor. Além disso, no teste de validação, a taxa de acerto no mundo real foi de 100% também. Lembre-se de que caímos em uma situação bem rara, que é justamente acertar 100% dos dados.

Perceba o quanto importante é rodarmos todos os algoritmos de uma vez para verificar qual deles se sairá melhor, dado um determinado conjunto de dados. Porém, ao usar algoritmos como `OneVsOne` (ou seja, quadráticos), precisamos nos atentar, pois se precisarmos de performance, provavelmente esse algoritmo não poderá ser utilizado, devido ao fato do crescimento de testes para uma grande quantidade de elementos de um conjunto de dado.

## 7.2 RESUMINDO

Repare que, além de dados com duas categorias, podemos também conter dados com mais de duas, como a situação do cliente em relação ao nosso site (se ele está alegre, neutro ou chateado). Levando em consideração apenas esse exemplo, vimos que os algoritmos que usamos anteriormente (`AdaBoost` e `Multinomial`) não conseguiam classificar esses dados de forma eficiente. Em outras palavras, eles obtinham o mesmo resultado que o algoritmo base, portanto, não poderíamos usá-los no mundo real.

Porém, além do `AdaBoost` e `Multinomial`, bons para dados com duas categorias, temos também os algoritmos `OneVsRest` e o `OneVsOne`, que servem justamente para classificar elementos que

podem ter mais de duas categorias. Além da eficiência dos `OneVsRest` e `OneVsOne`, vimos que ambos possuem uma certa diferença quanto à performance, pois o `OneVsRest` cresce constantemente de acordo com a quantidade de elementos dado um conjunto de dados.

Ou seja, se tivermos três elementos, serão três testes; quatro elementos, quatro testes. Mas no `OneVsOne`, a abordagem é bem diferente, pois ele é quadrático; ele cresce muito de acordo com a quantidade de elementos, então, quanto mais elementos tivermos, serão realizados muito mais testes.

Vimos também a importância de rodar diversos algoritmos para o mesmo conjunto de dados, mas precisamos sempre ficar atentos para não viciar o nosso algoritmo. Isto é, não podemos rodar um algoritmo, obter um resultado e, então, de acordo com este, rodar um outro algoritmo. O ideal é que sempre rodemos todos os algoritmos de uma vez, pois dessa forma elegemos o vencedor e rodamos para um conjunto de dados desconhecidos.

## CAPÍTULO 8

# UTILIZANDO O K-FOLD

Todos os processos que realizamos até agora seguiram três passos. Primeiro, pegamos mais de um algoritmo e pedimos para eles treinarem com determinados dados, e então realizamos o mesmo teste para cada um deles. Por fim, verificamos qual dentre eles obteve o melhor resultado, e o escolhemos como algoritmo vencedor, isto é, o algoritmo que realiza a validação (teste do mundo real) e nos diz o resultado final que desejamos.

Porém, vamos analisar melhor esses passos que realizamos para verificar se realmente são eficazes. Começaremos pelo treino. Suponhamos que temos um conjunto de dados e que o dividimos em:

- **Treino:** 60%.
- **Teste:** 20%.
- **Validação:** 20%.

Assim, vamos utilizar um conjunto de dados de 10 elementos para uma demonstração simplificada do processo:



Figura 8.1: Processo

Considerando esse conjunto, faremos sua divisão da seguinte maneira:



Figura 8.2: Processo

Foram usados 60% ( {1, 2, 3, 4, 5, 6} ) dos dados para treino, 20% ( {7, 8} ) para teste e mais 20% ( {9, 10} ) para validação. Quando rodamos esse algoritmo, escolhemos o vencedor e o seu resultado é de 82%.

Aparentemente, não temos nenhuma novidade do que já havíamos visto. Porém, o que aconteceria se o nosso suposto cliente 5 não tivesse acessado naquele exato momento? Em outras palavras, suponhamos que esse cliente ficasse doente e então aparecesse no lugar do cliente 7, o que aconteceria?

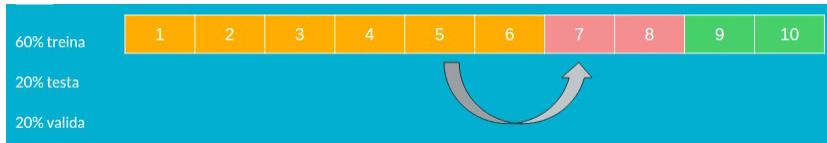


Figura 8.3: Processo

Com essa simples alteração, a nossa análise seria totalmente diferente. Mas por que totalmente diferente? Pois a função `fit` não vai mais treinar com os dados `{1, 2, 3, 4, 5 e 6}`, e sim `{1, 2, 3, 4, 6, 7}`. Qual é o impacto que temos nesse cenário?

Se o nosso treino for diferente, provavelmente o resultado será diferente. Logo, poderemos utilizar um algoritmo diferente e, então, o resultado da nossa avaliação pode ser outro, conforme o exemplo a seguir:



Figura 8.4: Processo

Essa foi apenas uma simulação para verificarmos quão suscetíveis estamos para qualquer alteração dos nossos dados. Com apenas uma simples alteração, temos resultados totalmente diferentes, como por exemplo, enquanto os nossos dados estavam seguindo a ordem de `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`, obtemos 82%, mas, colocando o dado 5 no lugar do dado 7, `{1, 2, 3, 4, 6, 7, 5, 8, 9, 10}`, conseguimos 72%.

Perceba que, além de diferente, obtivemos um resultado bem inferior ao anterior. Mas, e se em vez de 72% fosse 92%? Será que podemos confiar nos resultados obtidos da forma como estamos realizando os testes?

Se for alterado qualquer tipo de variável que os nossos dados

contenham — por exemplo, um cliente que ia comprar em um dia, mas comprou em outro; se alguém ficou doente; ou se em vez de ir hoje, decidiu ir amanhã; ou qualquer tipo de evento que **altere a ordem dos dados** que recebemos — isso fará com o que o nosso algoritmo resulte em determinado valor que ele não deveria obter. Em outras palavras, tanto 72% quanto 82%, ou qualquer outro resultado obtido alterando a ordem dos nossos dados, são válidos, portanto, os nossos resultados estão muito dependentes dessa ordem.

Em vez de treinarmos e testarmos apenas com uma sequência dos nossos dados — como: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ou {1, 2, 3, 4, 6, 7, 5, 8, 9, 10} —, precisamos calcular diversas combinações rodando o `Multinomial` e, a partir de todas elas, tiramos a média. Quando tiramos a média de vários resultados do nosso algoritmo, estamos eliminando esse caso de temporalidade, isto é, considerar um resultado que é válido para uma sequência específica de um conjunto de dados.

Agora, em vez de realizar apenas um teste, vamos fazer vários! Repare que, em todas as nossas abordagens até agora, em vez de realizar apenas uma vez, fizemos várias! Estamos sempre modificando o nosso algoritmo para que ele não seja dependente de um determinado algoritmo, resultado ou sequência de dados, pois queremos que ele funcione para diferentes cenários.

Vejamos como vamos fazer para treinar e testar o nosso algoritmo, variando a sequência de dados. Uma abordagem que podemos fazer é:

60% treina	1	2	3	4	5	6	7	8	
20% testa	1	2	3	4	6	7	5	8	
20% valida	9	10							

Dadas duas sequências distintas, separamos os mesmos dados de validação para ambas, nesse caso, 9 e 10. Lembre-se de que os dados de validação são todos os que o nosso algoritmo nunca viu, portanto, todas as vezes em que iniciarmos esse tipo de teste, o primeiro passo é pegar os dados de validação que serão usados para todas as sequências de dados que escolhemos.

Mas qual é o problema de usar os dados de validação durante os testes? É o fato de viciarmos nos resultados obtidos durante eles. Portanto, vamos separar 20% dos dados para validação desse teste:

60% treina	1	2	3	4	5	6	7	8	
20% testa	1	2	3	4	6	7	5	8	
20% valida	9	10							

Figura 8.6: Processo

Rodamos o Multinomial tanto com a primeira sequência quanto com a segunda e, por exemplo, obtemos os resultados 88% e 74%, respectivamente. Em seguida, tiramos a média entre 88% e 74%:  $(88 + 74) / 2 = 81\%$ .

60% treina	1	2	3	4	5	6	7	8	88%
20% testa	1	2	3	4	6	7	5	8	74%
média: 81%									
20% valida	9	10							

Agora podemos afirmar que, para esse conjunto de dados, independentemente da sua sequência, o Multinomial obtém o resultado de 81%, pois ambos os resultados testados (81% e 74%) são válidos. Por isso precisamos tirar a média de todos os testes realizados e, a partir dela, informar qual é o resultado do algoritmo. Calculamos a média para apenas essas duas sequências, como ficaria o resultado para essas outras duas sequências?

60% treina	1	2	3	4	5	6	7	8	88%
20% testa	1	2	3	4	6	7	5	8	74%
	1	2	3	5	6	7	4	8	?
	1	2	3	4	6	8	5	7	?
+    quantas combinações?									
20% valida	9	10							

Figura 8.8: Processo

Mesmo adicionando mais duas sequências, não estamos cobrindo todas as outras possíveis sequências para esse conjunto de dados  $\{1, 2, 3, 4, 5, 6, 7, 8\}$ . Existem diversas outras para esse conjunto. Além de ser trabalhoso realizar manualmente, é custoso para a CPU, ou seja, precisamos achar alguma forma mais eficaz para realizar esses testes.

Uma abordagem que podemos fazer é quebrar esse conjunto de dados de uma forma que possibilite rodarmos diversas vezes sem que entremos no cenário da temporalidade. Vamos iniciar quebrando o nosso conjunto de dados em 2:

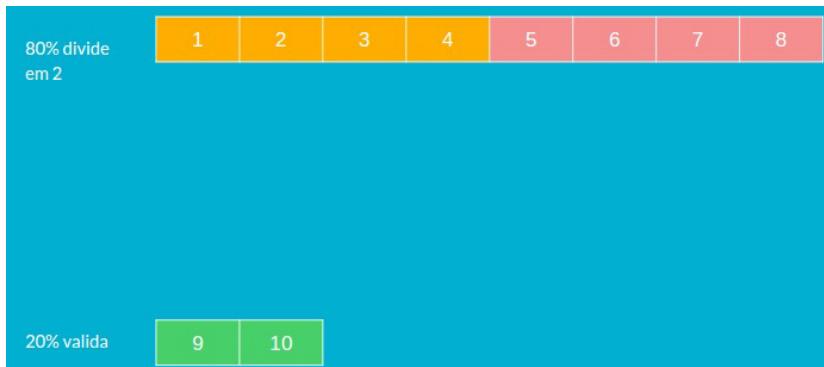


Figura 8.9: Processo

Utilizamos a primeira metade para treinar e a segunda para testar. Agora quebramos novamente em dois:

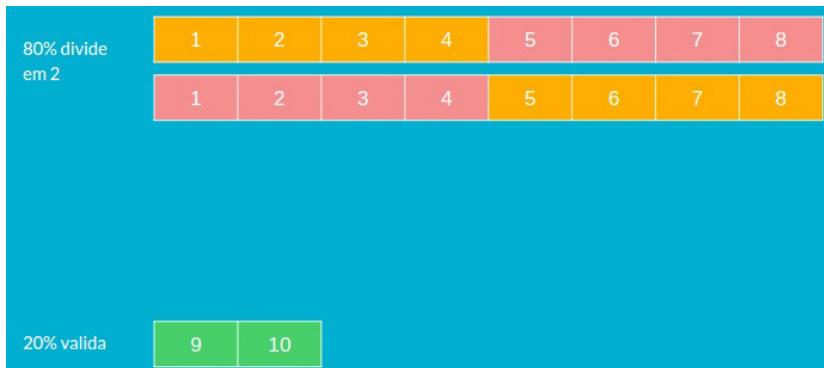


Figura 8.10: Processo

Agora temos duas metades distintas. Vamos tirar a média:

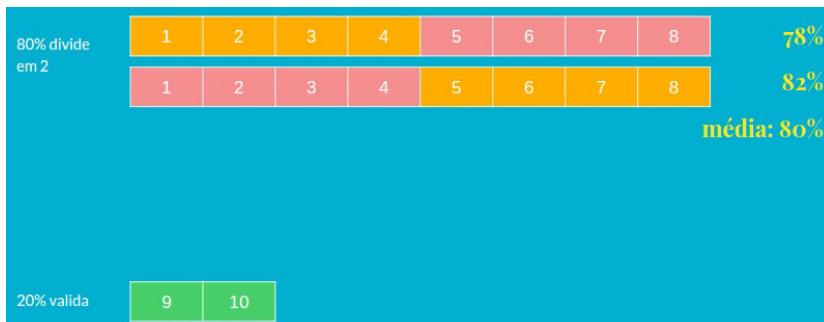


Figura 8.11: Processo

Note que agora o nosso processo de teste mudou. Em vez de fazer o teste com diversas sequências, estamos dividindo os dados em duas metades distintas e inversas. Ou melhor, na primeira vez usamos os dados  $\{1, 2, 3, 4\}$  para treinar e  $\{5, 6, 7, 8\}$  para testar, porém, na segunda, os dados  $\{1, 2, 3, 4\}$  para testar e  $\{5, 6, 7, 8\}$  para treinar.

Quando quebramos um processo de treino e teste em dois pedaços, estamos *foldando* em dois pedaços, nesse caso, *2-fold*. Mas e se quiséssemos quebrar em três pedaços? Como seria essa quebra? Vejamos:

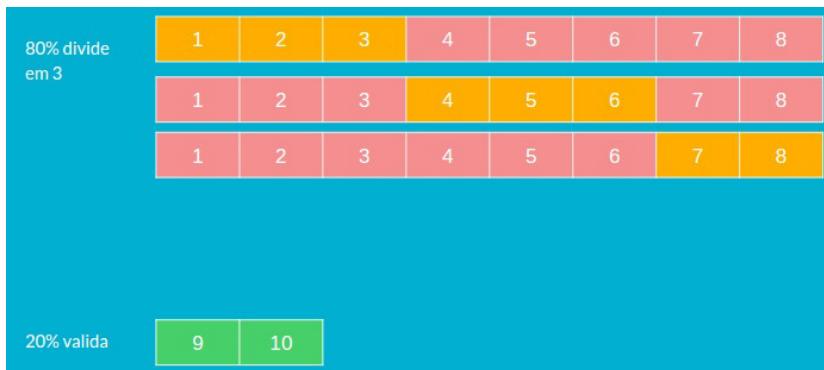


Figura 8.12: Processo: 3-fold

Veja que dividimos em três trechos proporcionais: 3 {1, 2, 3}, 3 {4, 5, 6} e 2 {7, 8}. Então agora, para 3-fold, os nossos treinos e testes ficam da seguinte maneira:

- **treino {4, 5, 6, 7, 8}: teste {1, 2, 3}**
- **treino {1, 2, 3, 7, 8}: teste {4, 5, 6}**
- **treino {1, 2, 3, 4, 5, 6}: teste {7, 8}**

Após separar os dados de treino e teste, o que precisamos fazer? Calcular e tirar a média da mesma forma como fizemos com o 2-fold:  $(88 + 74 + 83) / 3 = 81,66\%$ .

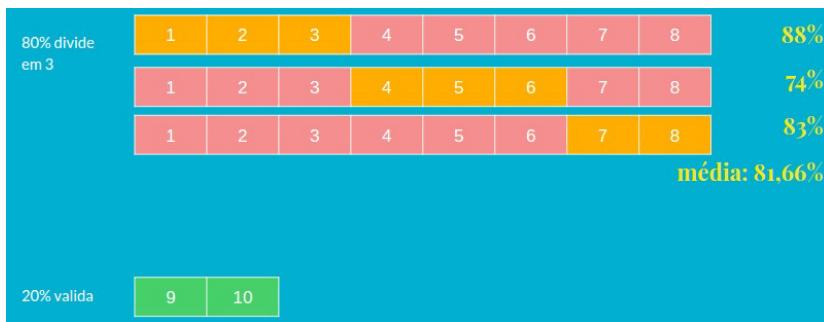


Figura 8.13: Processo: 3-fold

Dessa vez, o nosso Multinomial está acertando em 81,66% das vezes. E se quebrássemos em 4? E se fossem N pedaços? Como faríamos? Exatamente como fizemos com os exemplos anteriores, ou seja, quebraríamos pela quantidade de vezes desejadas, e então usariámos o primeiro pedaço para testar e o resto para treinar; depois, o segundo para testar e o resto para treinar. Em outras palavras, cada vez que quebramos, precisamos treiná-lo com o resto e testá-lo com o pedaço que separamos.

Perceba que agora o nosso processo está ficando cada vez mais lento. Antes testávamos apenas uma única vez, porém, com essa abordagem, estamos rodando o nosso algoritmo de acordo com a quantidade de pedaços. Isto é, se quebramos duas vezes, rodamos duas vezes; se forem três pedaços, três vezes; N pedaços serão N vezes. Porém, precisamos sempre nos atentar ao fato de que, a cada quebra que realizarmos, a média pode variar.

Quebrar os nossos dados em N vezes não significa que podemos quebrar por qualquer quantidade, pois a quantidade de quebras só pode variar entre 2 e a quantidade total de elementos. Ou seja, se temos oito elementos, significa que podemos quebrar os nossos dados de 2 a 8 pedaços. Considerando esse exemplo, como ficaria um 8-fold? Vejamos:

80% divide em 8

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Figura 8.14: Processo: 8-fold

Cada elemento representa uma quebra dos dados, ou seja, cada elemento do conjunto de dados será usado para teste e o restante para treino. Por exemplo, primeiro utilizaremos o dado 1 para teste e o restante para treino; então, o dado 2 para teste e o restante para treino, e assim sucessivamente.

Além disso, temos o caso extremo que é justamente fazer o contrário: utilizar cada pedaço como treino e o resto como teste. Esse algoritmo que quebra, corta e particiona pedaços dos nossos dados é chamado de  $k$ -fold ([https://pt.wikipedia.org/wiki/Valida%C3%A7%C3%A3o\\_cruzada](https://pt.wikipedia.org/wiki/Valida%C3%A7%C3%A3o_cruzada)).

## 8.1 IMPLEMENTANDO O K-FOLD

Vamos implementar o k-fold no nosso algoritmo. Desta vez, vamos criar um novo arquivo Python com o nome

`situacao_do_cliente_kfold.py` , justamente para que não percamos o algoritmo que realizamos no nosso arquivo anterior.

Entretanto, vamos copiar o algoritmo do arquivo `situacao_do_cliente.py` para esse novo arquivo. O arquivo `situacao_do_cliente_kfold.py` fica da seguinte forma:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8
porcentagem_de_teste = 0.1

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_teste = int(porcentagem_de_teste * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino - tamanho_de_te
ste

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

fim_de_treino = tamanho_de_treino + tamanho_de_teste

teste_dados = X[tamanho_de_treino:fim_de_treino]
teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]

validacao_dados = X[fim_de_treino:]
validacao_marcacoes = Y[fim_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes,
teste_dados, teste_marcacoes):
```

```

    modelo.fit(treino_dados, treino_marcacoes)

    resultado = modelo.predict(teste_dados)

    acertos = resultado == teste_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(teste_dados)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento
    s

    msg = "Taxa de acerto do algoritmo {0}: {1}".format(nome, tax
a_de_acerto)

    print(msg)
    return taxa_de_acerto

def teste_real(modelo, validacao_dados, validacao_marcacoes):
    resultado = modelo.predict(validacao_dados)

    acertos = resultado == validacao_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(validacao_marcacoes)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elemento
    s

    msg = "Taxa de acerto do vencedor entre os dois algoritmos no
mundo real: {0}".format(taxa_de_acerto)
    print(msg)

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))

```

```

resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t
reino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMu
ltinomial, treino_dados, treino_marcacoes, teste_dados, teste_marca
coes)
resultados[resultadoMultinomial] = modeloMultinomial

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloA
daBoost, treino_dados, treino_marcacoes, teste_dados, teste_marca
coes)
resultados[resultadoAdaBoost] = modeloAdaBoost

maximo = max(resultados)
vencedor = resultados[maximo]

print("Vencedor: ")
print(vencedor)

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcaco
es)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)

```

Qual será a diferença entre esse novo arquivo e o situacao\_do\_cliente.py ? Será justamente na forma como trabalharemos com os dados. Vejamos esse trecho de código:

```

import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]

```

```
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8
porcentagem_de_teste = 0.1
```

Anteriormente estávamos lendo os dados por meio do Pandas, então pegávamos os dummies e, logo em seguida, separávamos a parte de treino e de teste. Continuaremos com a mesma abordagem, porém, dessa vez, em vez de separar a parte de treino e de teste, vamos usar o mesmo pedaço para ambos os objetivos. Portanto, vamos descartar a variável `porcentagem_de_teste`. Vejamos o restante do código:

```
porcentagem_de_treino = 0.8

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_teste = int(porcentagem_de_teste * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino - tamanho_de_te
ste
```

O tamanho de treino ainda permanece, mas não utilizaremos mais o tamanho de teste, pois, dessa vez, só precisaremos do tamanho de treino que será compreendido como os dados de treino e teste ao mesmo tempo. Então, apagaremos a variável `tamanho_de_teste`.

Mas veja que o tamanho de validação não usa mais a variável `tamanho_de_teste`, ou seja, faremos apenas a diferença com a variável `tamanho_de_treino`:

```
porcentagem_de_treino = 0.8
```

```
tamanho_de_treino = porcentagem_de_treino * len(Y)
tamanho_de_validacao = len(Y) - tamanho_de_treino
```

Vamos continuar verificando o nosso código:

```
treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

fim_de_treino = tamanho_de_treino + tamanho_de_teste

teste_dados = X[tamanho_de_treino:fim_de_treino]
teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]
```

Vamos manter tanto os dados como as marcações de treino, mas removeremos os dados de testes. Portanto, as variáveis `fim_de_treino`, `teste_dados` e `teste_marcacoes` serão apagadas, e ficamos apenas com os dados e as marcações de treino e validação:

```
treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

validacao_dados = X[fim_de_treino:]
validacao_marcacoes = Y[fim_de_treino:]
```

Rpare que ainda estamos usando a variável `fim_de_treino` para calcular os dados e as marcações de validação. Como faremos agora? Em vez de pegar a partir do `fim_de_treino`, usaremos a variável `tamanho_de_treino`:

```
validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]
```

Como estão sendo calculados os nossos dados e marcações nesse instante? Estamos pegando para treino de 0 a 80% e, para validação, de 80% em diante. Entretanto, a variável `tamanho_de_validacao` não está sendo utilizada, então, por enquanto, podemos deixá-la comentada.

Vejamos como ficou o nosso código agora que separamos os dados necessários para esse algoritmo:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
# tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

# restante do código
```

Com os dados de treino e validação em mãos, qual é o nosso próximo passo? É de fato realizar um teste, correto? Porém, dessa vez, precisamos utilizar o algoritmo *k-fold*. Para esse primeiro exemplo, usaremos o algoritmo `OneVsRest`. Vamos implementá-lo logo abaixo:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))
```

E agora? Função `fit`? Não! Lembre-se de que usaremos o *k-fold*, logo, faremos uma outra abordagem. Nossa primeiro passo

será definir o valor do k, isto é, a quantidade de pedaços em que vamos quebrar. Neste exemplo, vamos utilizar o valor 3:

```
from sklearn.multiclass import OneVsRestClassifier  
from sklearn.svm import LinearSVC  
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))  
  
k = 3
```

Qual é o próximo passo? Quebrar os dados em 3 pedaços, e então rodar o algoritmo com o primeiro pedaço e o resto, depois o segundo pedaço e o resto, então o terceiro pedaço e o resto. Logo, vamos pegar os resultados de cada um desses grupos. Para isso, usaremos a função `cross_val_score` do `sklearn.model_selection`, mas antes precisamos importá-la:

```
from sklearn.multiclass import OneVsRestClassifier  
from sklearn.svm import LinearSVC  
from sklearn.model_selection import cross_val_score  
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))  
  
k = 3
```

Agora precisamos chamar a função `cross_val_score` enviando o modelo, os dados de treino e de marcação e, por fim, a quantidade de pedaços em que vamos quebrar:

```
cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
```

Então, retornamos o valor dessa função para uma variável chamada `scores`:

```
scores = cross_val_score(modelo, treino_dados, treino_marcacoes,  
cv = k)
```

Essa variável representará todos os resultados obtidos de acordo com a quantidade de pedaços que enviamos. Vamos imprimi-la também:

```

# restante do código

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
from sklearn.model_selection import cross_val_score
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))

k = 3

scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
cv = k)
print(scores)

```

O restante do código a seguir pode ser comentado ou removido, pois nesse instante não o utilizaremos. Nesse caso, vou removê-lo para uma melhor visibilidade. O nosso código fica da seguinte forma:

```

import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
# tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

```

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
from sklearn.model_selection import cross_val_score
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))

k = 3

scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
cv = k)
print(scores)

Ao      rodarmos      o      nosso      arquivo
situacao_do_cliente_kfold.py , obtemos o seguinte resultado:
```

```
> python3 situacao_do_cliente_kfold.py
[ 0.91803279  0.93333333  0.91525424]
```

Precisamos agora realizar o próximo passo, que é justamente tirar a média a partir dos resultados obtidos. Para isso, vamos usar o `numpy` do Python para calcular a média:

```
# restante do código
```

```
scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
cv = k)
print(scores)
import numpy as np
```

Para extrairmos a média, precisamos usar a função `mean` e, em seguida, retornar para a variável `media`. Então, imprimimos essa variável:

```
# restante do código
```

```
scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
cv = k)
print(scores)
import numpy as np
media = np.mean(scores)
print(media)
```

Rodando novamente o nosso código, obtemos o seguinte resultado:

```
> python3 situacao_do_cliente_kfold.py  
[ 0.91803279  0.93333333  0.91525424]  
0.922206785836
```

Como podemos ver, a nossa média foi de 92,22%. Mas e se o  $k$  fosse, por exemplo, 4? Vejamos o resultado:

```
> python3 situacao_do_cliente_kfold.py  
[ 0.93478261  0.91111111  0.93333333  0.93181818]  
0.92776130874
```

Agora o resultado foi de 92,77%. E se fosse 10? Vejamos:

```
> python3 situacao_do_cliente_kfold.py  
[ 0.94736842  0.89473684  0.84210526  0.89473684  0.94736842  0.9  
4444444  
    0.94444444  0.94117647  0.9375      0.9375      ]  
0.923138114895
```

Para  $k$  igual a 10, temos o resultado de 92,31%. Cada vez que alteramos o valor de  $k$ , estamos viciando a nossa decisão, pois tentaremos sempre nos basear no valor de  $k$  que obtivemos o melhor resultado para aquele conjunto de dados. Ou seja, todas as vezes que fizermos uso do algoritmo k-fold, precisamos decidir primeiro o valor de  $k$  para não ficarmos alterando. Portanto, utilizaremos o valor 10.

## 8.2 IMPLEMENTANDO O NOVO FIT\_AND\_PREDICT

Note que mudamos a forma de treinar e testar o nosso algoritmo. Ou seja, esses passos que realizamos são justamente o `fit_and_predict`. Portanto, vamos implementá-lo:

```
def fit_and_predict():
```

O que enviaremos como parâmetro? O nome do algoritmo, o modelo, os dados e as marcações de treino. Vale lembrar que, para esse algoritmo, não teremos os dados ou marcações de teste:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes)
```

Vamos definir o valor padrão de `k`, ou seja, atribuir o valor 10:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes)
```

```
    k = 10
```

Agora precisamos calcular o `scores`, portanto, vamos usar a função `cross_val_score`:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes)
```

```
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
                             cv = k)
```

Com os `scores` em mãos, qual é o próximo passo? Calcular a taxa de acerto, certo? Como fazemos isso mesmo? Tirando a média a partir dos `scores`. Para tirarmos a média, vamos usar novamente o `numpy`:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes)
```

```
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
                             cv = k)
    taxa_de_acerto = np.mean(scores)
```

Agora construiremos a mensagem que será exibida com o nome do algoritmo e a taxa de acerto. Vamos imprimi-la também:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
    cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
```

Por fim, retornamos a taxa de acerto:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
    cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto
```

Agora o nosso `fit_and_predict` não utiliza a função `fit` ou `predict`, como vimos na outra implementação que fizemos. Para o algoritmo *k-fold*, dentro do `fit_and_predict`, definimos primeiro a quantidade de pedaços em que serão quebrados os nossos dados por meio da variável `k`, então pedimos para ele calcular os `scores`, que é justamente chamar a função `cross_val_score`. Ela, internamente, faz todo o processo para particionar e calcular cada um dos resultados.

Por fim, ele calcula a média com o `numpy`, imprime a mensagem com o nome do algoritmo e a taxa de acerto, e retorna o resultado da taxa de acerto. Portanto, podemos agora retirar o trecho de código que realizava todos esses passos e, a partir de agora, usarmos esse `fit_and_predict`:

```
import pandas as pd
```

```

from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
# tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
    cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto

```

Antes de utilizá-lo, vamos primeiro importar o `numpy` e o `cross_val_score` logo no começo do arquivo:

```

import pandas as pd
from collections import Counter
import numpy as np
from sklearn.model_selection import cross_val_score

# restante do código

```

Para usar o `fit_and_predict`, faremos da mesma forma como fizemos no arquivo `situacao_cliente`, ou seja, criamos o nosso dicionário:

```
resultados = {}
```

Agora importamos o algoritmo que vamos usar e criamos o seu modelo:

```
from sklearn.multiclass import OneVsRestClassifier  
from sklearn.svm import LinearSVC  
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
```

Com o modelo em mãos, chamamos a função `fit_and_predict` enviando o nome do algoritmo, o modelo, os dados e as marcações de treino. Em seguida, atribuímos para a variável que representará o resultado, nesse caso, `resultadoOneVsRest`:

```
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,  
, treino_dados, treino_marcacoes)
```

Por fim, adicionamos o nosso modelo no dicionário `resultados`, enviando o resultado obtido pelo `fit_and_predict` dentro de chaves (nesse caso, o `resultadoOneVsRest`). Depois, imprimimos o dicionário para verificar o resultado que o nosso algoritmo nos devolve:

```
resultados[resultadoOneVsRest] = modeloOneVsRest  
print(resultados)
```

O nosso código fica da seguinte maneira:

```
import pandas as pd  
from collections import Counter  
import numpy as np  
from sklearn.model_validation import cross_val_score
```

```

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
# tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
                             cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
                                      treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

print(resultados)

```

Rodando o nosso algoritmo, obtemos o seguinte resultado:

```
> python3 situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
{0.92313811489508102: OneVsRestClassifier(estimator=LinearSVC(C=1
.0, class_weight=None, dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)}
```

Observe que a média do nosso algoritmo ( `OneVsRest` ) foi de 92,31% para esses dados. Lembra de quanto era o resultado do `OneVsRest` sem utilizar o k-fold? Vejamos:

```
> python3 situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo OneVsOne: 100.0
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Vencedor:
OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight=None,
dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 100.0
Taxa de acerto base: 82.608696
Total de teste: 23
```

Como vimos, esse 90,90% é um resultado que foi obtido por sorte ou azar, pois quando testamos o nosso algoritmo com apenas uma sequência, ficamos suscetíveis a qualquer evento que possa ocorrer.

Atualmente, estamos utilizando o k-fold apenas para o `OneVsRest`, ou seja, precisamos também adicionar os demais algoritmos. Como podemos fazer isso? Simples! Basta copiarmos

da mesma forma como está escrito no arquivo `situacao_do_cliente.py`. Entretanto, vamos copiar apenas um algoritmo para verificar se tudo ocorre como o esperado. Neste caso, copiaremos o `OneVsOne` :

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t
reino_dados, treino_marcacoes, teste_dados, teste_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

print(resultados)
```

O único detalhe sobre copiar o código do arquivo `situacao_do_cliente.py` é que não utilizamos mais os dados e as marcações de teste, portanto, vamos excluí-los:

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest
, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
```

```

modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t
reino_dados, treino_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

print(resultados)

```

Vamos testar o arquivo `situacao_do_cliente_kfold.py` para verificar se tudo ocorre como o esperado:

```

> python3 situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
Taxa de acerto do OneVsOne: 0.99444444444444
{0.92313811489508102: OneVsRestClassifier(estimator=LinearSVC(C=1
.0, class_weight=None, dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1), 0.99444444444444446: OneVsOneClassifier(esti
mator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercep
t=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)}

```

Funcionou sem nenhum problema, mas observe que o resultado obtido pelo algoritmo `OneVsOne` foi de 99,44%. Lembra que, na implementação que fizemos no arquivo `situacao_do_cliente.py`, ele apresentou um resultado de 100%? Em outras palavras, quando usamos o método do k-fold, alcançamos um resultado mais realista do que tínhamos anteriormente. Agora precisamos adicionar os demais algoritmos.

O nosso código fica da seguinte maneira:

```

import pandas as pd
from collections import Counter
import numpy as np
from sklearn.model_selection import cross_val_score

```

```

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
# tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
    cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t

```

```

reino_dados, treino_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMultinomial,
treino_dados, treino_marcacoes)
resultados[resultadoMultinomial] = modeloMultinomial

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost,
treino_dados, treino_marcacoes)
resultados[resultadoAdaBoost] = modeloAdaBoost

print(resultados)

```

Vejamos o resultado obtido por todos os algoritmos utilizando o k-fold:

```

> python3 situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
Taxa de acerto do OneVsOne: 0.9944444444444
Taxa de acerto do MultinomialNB: 0.829977210182
Taxa de acerto do AdaBoostClassifier: 0.762947196422
{0.8299772101823184: MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True),
 0.76294719642242859: AdaBoostClassifier(algorithm='SAMME.R',
 base_estimator=None,
 learning_rate=1.0, n_estimators=50, random_state=None),
 0.92313811489508102: OneVsRestClassifier(estimator=LinearSVC(C=1.0,
 class_weight=None, dual=True, fit_intercept=True,
 intercept_scaling=1, loss='squared_hinge', max_iter=1000,
 multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
 verbose=0),
 n_jobs=1), 0.9944444444444446: OneVsOneClassifier(estimator=LinearSVC(C=1.0,
 class_weight=None, dual=True, fit_intercept=True,
 intercept_scaling=1, loss='squared_hinge', max_iter=1000,
 multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
 verbose=0),
 n_jobs=1)}

```

Ele calculou normalmente a média de todos os algoritmos e

obtivemos os seguintes resultados:

- **OneVsRest**: 92,31%.
- **OneVsOne**: 99,44%.
- **Multinomial**: 82,99%.
- **AdaBoost**: 76,29%.

Mesmo com uma abordagem diferente, o `OneVsOne` continua sendo o vencedor. Entretanto, precisamos verificar qual é o algoritmo vencedor. Como podemos implementar isso no nosso algoritmo? Exatamente da mesma forma como fizemos no `situacao_do_cliente.py`. Em outras palavras, podemos copiar e colar o código novamente:

```
# restante do código

print(resultados)

maximo = max(resultados)
vencedor = resultados[maximo]

print("Vencedor: ")
print(vencedor)

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)
```

Perceba que adicionamos a função `teste_real`, portanto, precisamos adicionar-a ao nosso arquivo `situacao_do_cliente_kfold.py` também:

```

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
    cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}%".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto

def teste_real(modelo, validacao_dados, validacao_marcacoes):
    resultado = modelo.predict(validacao_dados)

    acertos = resultado == validacao_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(validacao_marcacoes)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

    msg = "Taxa de acerto do vencedor entre os dois algoritmos no mundo real: {0}%".format(taxa_de_acerto)
    print(msg)

# restante do código

```

Vamos testar novamente o nosso arquivo situacao\_do\_cliente\_kfold.py :

```

> python3 situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
Taxa de acerto do OneVsOne: 0.994444444444
Taxa de acerto do MultinomialNB: 0.829977210182
Taxa de acerto do AdaBoostClassifier: 0.762947196422
{0.8299772101823184: MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True), 0.76294719642242859: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None, learning_rate=1.0, n_estimators=50, random_state=None), 0.92313811489508102: OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True, intercept_scaling=1, loss='squared_hinge', max_iter=1000, multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,

```

```

    verbose=0),
    n_jobs=1), 0.9944444444444444: OneVsOneClassifier(esti-
mator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercep-
t=True,
    intercept_scaling=1, loss='squared_hinge', max_iter=1000,
    multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
    verbose=0),
    n_jobs=1)}
Vencedor:
OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight=None,
dual=True, fit_intercept=True,
    intercept_scaling=1, loss='squared_hinge', max_iter=1000,
    multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
    verbose=0),
    n_jobs=1)
Traceback (most recent call last):
  File "situacao_do_cliente_kfold.py", line 79, in <module>
    teste_real(vencedor, validacao_dados, validacao_marcacoes)
  File "situacao_do_cliente_kfold.py", line 36, in teste_real
    resultado = modelo.predict(validacao_dados)
  File "/usr/local/lib/python3.6/dist-packages/sklearn/multiclass
.py", line 538, in predict
    Y = self.decision_function(X)
  File "/usr/local/lib/python3.6/dist-packages/sklearn/multiclass
.py", line 557, in decision_function
    check_is_fitted(self, 'estimators_')
  File "/usr/local/lib/python3.6/dist-packages/sklearn/utils/vali-
dation.py", line 678, in check_is_fitted
    raise NotFittedError(msg % {'name': type(estimator).__name__})
)
sklearn.utils.validation.NotFittedError: This OneVsOneClassifier
instance is not fitted yet. Call 'fit' with appropriate arguments
before using this method.

```

O algoritmo apresentou um erro na linha 79, que é justamente a chamada à função `teste_real`. Mas por que isso aconteceu?

O erro nos informa que o `OneVsOneClassifier` (algoritmo vencedor) não foi `fitted`, ou melhor, não chamamos a função `fit`, portanto, ele não consegue prever esses dados. Para resolver esse detalhe, basta adicionarmos o `fit` antes de fazer a chamada à função `teste_real`:

```

# restante do código

print("Vencedor: ")
print(vencedor)

vencedor.fit(treino_dados, treino_marcacoes)

teste_real(vencedor, validacao_dados, validacao_marcacoes)

```

Testando novamente o nosso algoritmo:

```

> python3 situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
Taxa de acerto do OneVsOne: 0.9944444444444
Taxa de acerto do MultinomialNB: 0.829977210182
Taxa de acerto do AdaBoostClassifier: 0.762947196422
{0.8299772101823184: MultinomialNB(alpha=1.0, class_prior=None,
fit_prior=True), 0.76294719642242859: AdaBoostClassifier(algorithm='SAMME.R',
base_estimator=None,
learning_rate=1.0, n_estimators=50, random_state=None),
0.92313811489508102: OneVsRestClassifier(estimator=LinearSVC(C=1.0,
class_weight=None, dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1), 0.99444444444444446: OneVsOneClassifier(estimate
r=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercep
t=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)}
Vencedor:
OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight=None,
dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 100.0
Taxa de acerto base: 75.555556
Total de teste: 45

```

Agora o nosso algoritmo funcionou como o esperado. Note que, no teste do mundo real, o `OneVsOne` acertou 100% das vezes. Além disso, agora, em vez de utilizar 23 elementos para teste, foram usados 45. Mas por que isso aconteceu?

Você se lembra de que, para o k-fold, estamos utilizando 80% para treinar e testar? Então, sobraram 20% para validação, por isso foram 45 testes em vez de 23. E quanto usávamos no arquivo `situacao_do_cliente.py` para atingir 23? Vejamos:

```
porcentagem_de_treino = 0.8  
porcentagem_de_teste = 0.1
```

Em nosso outro algoritmo, usávamos 80% para treino e 10% para teste, ou seja, sobravam apenas 10% para validação. É exatamente por esse motivo que o teste do mundo real era realizado com 23 elementos.

Não podemos também ficar alterando o número de elementos que usaremos tanto para os dados quanto para os testes, pois dessa forma estaremos viciando a nossa decisão humana. Por exemplo, vimos o resultado com 80%, mas quanto será que ele acertaria se fosse 90%? E 60%? Que tal dessa vez X%? Escolheremos o melhor resultado para esse caso, em outras palavras, estaremos viciando a nossa decisão. Esse é um grande problema da ciência, pois existem diversas práticas que precisamos seguir para ter a certeza de que tudo que estamos fazendo de fato é válido.

Entretanto, se não anotarmos todo o processo que foi realizado para chegarmos a um resultado, como uma mudança de valores para um determinado teste, todos vão pensar que o nosso algoritmo acerta sempre e para tudo, já que não apresentamos todos os passos realizados.

## 8.3 RESUMINDO

Vimos como testar e validar nossos modelos de uma maneira mais complexa que os testes de validação que fazíamos até então. Em vez de implementarmos todos os cortes na mão, pedimos à biblioteca para efetuar diversos cortes nos dados, nos devolvendo o resultado médio esperado baseado em nossos dados.

Como sempre, foi fundamental nos atermos ao quanto estamos brincando e testando com as variáveis. Rodar tipos diferentes de testes e validações também é perigoso e pode viciar nossa decisão, uma vez que podemos encontrar um resultado por coincidência após sucessivas falhas. Mesmo que nossa intenção seja positiva, devemos ficar de olho no processo que usamos para definir nosso teste, não só ao código em si. Falhas ao tentar encontrar modelos são tão importantes quanto acertos.

É bem comum as pessoas não publicarem exatamente todos os passos realizados, isto é, todas as falhas e sucessos de um algoritmo, pois existe um "vício humano" que exibe apenas tudo que obteve sucesso. Então, acreditamos que a amostra que nos foi apresentada funciona muito bem para tudo, sendo que, na verdade, existem diversos problemas que não foram apresentados.

Portanto, sempre que tivermos de apresentar os resultados adquiridos, precisaremos apresentar também todo o processo que foi realizado, justamente para comprovar se existe ou não um vício que possa invalidar os nossos resultados.

## CAPÍTULO 9

# CRIANDO UM DICIONÁRIO

Até agora, vimos diversos problemas que podemos ter no nosso dia a dia, e também aprendemos como resolvê-los. Entretanto, nos depararemos com algo novo, uma situação que ainda não vimos. Vejamos o problema que precisamos resolver:

- Recebi alguns e-mails novos dentro da página de contato do meu site, com os seguintes conteúdos:
  - Se eu comprar cinco anos antecipados, eu ganho algum desconto?
  - O exercício 15 do curso de Java 1 está com a resposta errada. Pode conferir, por favor?
  - Existe algum curso para cuidar do marketing da minha empresa?
  - Já trabalho como designer e queria aprender mais de UX, quais cursos devo fazer?

Observe que cada um dos e-mails refere-se a um assunto específico. Por exemplo, o primeiro e-mail refere-se ao setor comercial/ financeiro da minha empresa. Já o segundo a um assunto totalmente diferente do primeiro, pois, no primeiro, o remetente está interessado em comprar, e no segundo, ele está

relatando um problema técnico.

E o terceiro e-mail? Também é diferente, pois o remetente está em dúvida, ou seja, está precisando de sugestões para o seu objetivo. Por fim, o quarto e-mail é bem similar ao terceiro, pois o remetente está em dúvida e precisa de dicas/ sugestões sobre quais cursos ele poderia fazer. Então, podemos classificar esses e-mails da seguinte forma:

- 1º — Venda ou comercial.
- 2º — Problemas técnicos.
- 3º e 4º — Dúvidas e sugestões de carreira.

Dentro da minha empresa, cada um desses tipos de e-mails é tratado e respondido por diferentes setores. Por exemplo, e-mails de venda ou comercial são respondidos pelo pessoal de vendas, porém, os e-mails de problemas técnicos serão respondidos pelo pessoal técnico. Por fim, e-mails relacionados à carreira serão respondidos por pessoas que possuem uma certa experiência em sua carreira.

Portanto, concluímos que, dentro da minha empresa, existem diversas seções, logo, cada e-mail que chega pelo formulário de contato precisa ser redirecionado à seção que vai atendê-lo. Como poderíamos resolver esse problema? Uma das formas bem comum seria adicionar um *combo box* ([https://en.wikipedia.org/wiki/Combo\\_box](https://en.wikipedia.org/wiki/Combo_box)), permitindo ao usuário escolher entre as três opções:

- Comercial;
- Técnico;
- Carreira.

Considerando apenas o cenário que vimos até agora, resolveria. Entretanto, suponhamos que a minha empresa cresceu, ou seja, surgiram seções diferentes, como o financeiro, então adicionaríamos dentro do combo box:

- Comercial;
- Financeiro;
- Técnico;
- Carreira.

Se tivéssemos de mandar um e-mail que tanto o comercial quanto o financeiro resolvessem, qual dentre essas seções escolheríamos? Vejamos outro exemplo adicionando a seção de conteúdo:

- Comercial;
- Financeiro;
- Técnico;
- Conteúdo;
- Carreira.

Se tivéssemos de enviar o segundo e-mail que trata de um problema técnico, porém, também se refere ao conteúdo, para qual seção deveríamos enviar? E para o terceiro que se refere tanto ao conteúdo quanto à carreira, para qual dessas seções enviaríamos? Perceba que está começando a ficar difícil para o usuário final escolher para qual seção ele precisa enviar.

Como podemos lidar com esse problema? Primeiramente, vamos permitir apenas que os nossos e-mails sejam classificados pelas três categorias que vimos anteriormente:

- Comercial;
- Técnico;
- Carreira.

Nesse instante, você pode estar pensando que já nos deparamos com um problema bem similar a esse que, dado um conjunto de dados **numéricos**, classificávamos em três ou mais categorias distintas, como o algoritmo `OneVsRest` e também a validação usando o k-fold. Mas tudo isso estava baseado em conjuntos numéricos, ou seja, mesmo que tivéssemos categorias textuais, transformávamos esses dados em números novamente.

Em outras palavras, desta vez, em vez de rodar um algoritmo com números, queremos rodar o algoritmo com textos. Ou melhor, queremos fazer com que o nosso algoritmo analise e classifique um texto. Como podemos fazer isso?

Lembre-se: da mesma forma que vimos anteriormente, quando temos um problema grande, podemos reduzi-lo em problemas menores que saibamos resolver. Quais são os problemas que já sabemos resolver? Até o momento, sabemos resolver o seguinte problema:

- Classificar números em uma categoria.

Qual é o nosso grande desafio? É justamente transformar essas palavras, ou melhor, essas sequências de palavras, em uma sequência de números. E mais, mesmo os textos possuindo quantidade de caracteres diferentes, como o 1º e o 2º e-mail, eles precisam possuir o mesmo tamanho. Portanto, todas as sequências de palavras precisam conter a mesma quantidade de colunas, como o arquivo `situacao_do_cliente.csv`:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,2
2,1,2,1
1,4,2,2
1,3,8,1
2,2,1,1
...
4,1,6,0
```

Note que as colunas desse arquivo são fixas para todos os dados, e precisamos fazer o mesmo com cada sequência de texto. Consegue imaginar como podemos fazer isso? Podemos começar pela primeira sequência de texto:

- Se eu comprar cinco anos antecipados, eu ganho algum desconto?

Tentaremos primeiro analisar essa frase. Mas como faremos isso? Precisamos realizar essa análise para cada uma dessas palavras. Entretanto, o nosso algoritmo já viu alguma palavra na vida? A resposta é: não! Logo, precisaremos armazenar quaisquer palavras existentes no texto. Vejamos como seria esse armazenamento:

- Se eu comprar cinco anos antecipados, eu ganho algum desconto?

[Se, eu, comprar, cinco, anos, antecipados, ganho, algum, desconto]

Observe que criamos um array adicionando as palavras do texto, mas perceba que existem 10 palavras nesse texto, mas palavras distintas são nove, pois a palavra "eu" se repete uma vez. Esse processo de armazenamento das palavras contidas no texto é justamente a identificação de todas as palavras distintas. Então, qual é o nosso próximo passo?

Atualmente, conhecemos todas as palavras contidas no array, porém, quantas vezes essa mesma palavra aparece nesse texto? Ou seja, quantas vezes a palavra "se" aparece? E a palavra "eu"? Vejamos o resultado:

```
[Se, eu, comprar, cinco, anos, antecipados, ganho, algum, desconto]  
[1, 2, 1, 1, 1, 1, 1, 1]
```

Agora temos 2 arrays: um deles indica todas as palavras distintas e o outro a quantidade em que cada uma dessas palavras se repete dentro do texto. Considerando esse primeiro exemplo, suponhamos que estamos analisando um outro texto que contém as mesmas palavras que armazenamos no nosso array de palavras distintas. O array de quantidade de palavras distintas para esse texto resulta em:

```
[Se, eu, comprar, cinco, anos, antecipados, ganho, algum, desconto]  
[1, 2, 1, 1, 1, 1, 1, 1]  
[2, 1, 0, 3, 1, 3, 4, 2, 1]
```

Mesmo sendo textos diferentes, conseguimos representar todos os nossos textos por meio de arrays que sempre terão o mesmo tamanho! Além disso, no texto que estamos analisando, se de repente aparecerem palavras como: "carreira", "curso", "novos cursos", entre outras referentes à carreira, provavelmente refere-se a um texto destinado à carreira. Caso no texto apareçam as palavras: "preço", "desconto", "valor", "pagamento", provavelmente é do comercial.

Mas somos nós que devemos dizer isso? Não, o próprio algoritmo terá a capacidade de analisar todos esses aspectos e classificar em qual seção o texto se encaixa.

O detalhe importante nesse instante é que conseguimos transformar um texto em um array de tamanho fixo, ou seja, qualquer texto que contenha essas mesmas palavras que armazenamos poderá ser representado com arrays diferentes, porém de tamanhos fixos! Vejamos um exemplo:

- Se eu comprar cinco anos antecipados, eu ganho algum desconto?
- Eu ganho desconto se comprar cinco anos antecipados?

Esse exemplo trata-se de um texto com palavras que vimos até agora. Como ficariam os arrays para cada uma dessas palavras?

[Se, eu, comprar, cinco, anos, antecipados, ganho, algum, desconto]

- Se eu comprar cinco anos antecipados, eu ganho algum desconto?
  - [1, 2, 1, 1, 1, 1, 1, 1, 1]
- Eu ganho desconto se comprar cinco anos antecipados?
  - [1, 1, 1, 1, 1, 1, 1, 0, 1]

Veja que o primeiro texto é representado com o array [1, 2, 1, 1, 1, 1, 1, 1], e o segundo com o seguinte array [1, 1, 1, 1, 1, 1, 1, 0, 1]. Vamos verificar mais uma frase:

- Se eu comprar cinco anos antecipados, eu ganho algum desconto?
- Eu ganho desconto se comprar cinco anos antecipados?
- Ao terminar um curso, eu ganho um certificado?

Dessa vez, nem todas as palavras dessa frase nova estão contidas no nosso array de palavras distintas, isto é, o nosso

dicionário. Precisamos criar um único array que dê suporte para todas as palavras dos textos que temos, ou seja, todas as palavras distintas.

Em seguida, criamos o array para cada uma das frases indicando a quantidade de vezes que uma dessas palavras se repete. Para esse exemplo, o que precisamos fazer? Simplesmente adicionar ao nosso array de palavras distintas todas as que não estão contidas dentro dele. Vejamos como ele fica:

[Se, eu, comprar, cinco, anos, antecipados, ganho, algum, desconto, Ao, terminar, um, curso, certificado]

Veja que agora temos um array de palavras distintas, que dá suporte para cada uma das frases que vimos. Então, qual é o próximo passo? É justamente gerar o array que indica a quantidade de vezes que essas palavras se repetem. Começaremos pela primeira frase:

- Se eu comprar cinco anos antecipados, eu ganho algum desconto?
  - [1, 2, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]

Vejamos agora a próxima frase, neste caso utilizaremos a frase nova:

- Ao terminar um curso, eu ganho um certificado?
  - [0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 2, 1, 1]

Dessa vez, fomos capazes de representar em arrays do mesmo tamanho, com todas as palavras do nosso universo. Portanto, conseguimos trabalhar esses dados dentro dos nossos algoritmos de classificação.

Vale lembrar que precisaremos realizar alguns ajustes para realizar esse tipo de classificação dentro do nosso algoritmo. Porém, aprendemos que, quando temos um conjunto de texto e precisamos transformá-lo em um conjunto de números, é necessário criar um dicionário.

## 9.1 RESUMINDO

Vimos que trabalhar com palavras é difícil, mas podemos de alguma maneira transformar o desafio de usar palavras e trabalhar com números simples. Basta encontrarmos um método (uma função) que reduza o universo dos textos e palavras para uma matriz numérica. No nosso caso, reduzimos qualquer texto a um vetor de tamanho fixo, em que cada posição desse vetor indica o número de ocorrências da palavra no nosso texto.

Essa é uma solução simples que permite trabalharmos de maneira uniforme com muitos tipos de textos, e a usaremos como base para nossas análises.

## CAPÍTULO 10

# CLASSIFICANDO OS TEXTOS E GANHANDO PRODUTIVIDADE NA EMPRESA

Agora precisamos implementar o algoritmo que é capaz de ler os nossos textos, criar um dicionário baseado nas palavras que surgiram durante a leitura deles, e então traduzir todos esses textos em arrays, sempre do mesmo tamanho — que indicam quantas vezes cada palavra aparece. Mas por que precisamos verificar quantas vezes uma palavra aparece em um determinado texto?

É justamente pelo fato de concluirmos que, se uma palavra como "amor" aparece constantemente em um texto, provavelmente ele é relacionado a um assunto pessoal, sobre quem a pessoa ama, por exemplo: seus filhos, seus pais ou qualquer outro assunto pessoal relacionado. Porém, dentre esses textos, se tiver algum assunto sobre "compre uma retroescavadeira", aparentemente é um *spam*, pois eu não utilizo uma retroescavadeira, logo, esse e-mail não faz sentido para mim.

A frequência com que as palavras aparecem pode nos indicar

sobre o que o texto está falando, ou seja, se é relacionado a um problema técnico, comercial, de cobrança ou certificação. Em vez de ficarmos criando a árvore de opções, isto é, nós mesmos ficarmos definindo quais são as palavras-chaves, deixamos o próprio algoritmo realizar essa tarefa.

Vamos implementar esse algoritmo. Começaremos criando um arquivo Python chamado `classificando_emails.py`. Salve-o dentro da pasta onde estão os nossos arquivos Python. Primeiramente, definiremos os textos que vamos usar durante a classificação:

```
texto1 = "Se eu comprar cinco anos antecipados, eu ganho algum de  
sconto?"  
texto2 = "O exercício 15 do curso de Java 1 está com a resposta e  
rrada. Pode conferir pf?"  
texto3 = "Existe algum curso para cuidar do marketing da minha em  
presa?"
```

Existe um pequeno detalhe, que é a quantidade de variáveis que estamos criando para os nossos textos. Por exemplo, atualmente temos três textos, mas se forem 10, ou então, 100? Escreveremos tudo dentro desse arquivo? Não faz muito sentido, portanto, vamos extrair esses textos de um arquivo CSV. Esses dados estarão nas planilhas do Google Spreadsheets (<http://bit.ly/22lUw1z>) na aba `emails`:

	A	B	C	D	E	F	G	H	I	J	K	L
1	email	classificacao										
2	Se eu comprar cinco anos antecipados	1										
3	O exercicio 15 do curso de Java 1 é	2										
4	Existe algum curso para cuidar do n	3										
5	Gostaria de renovar antecipadame	1										
6	O video não está travando as vezet	2										
7	Que trilha vocês recomendam para	3										
8	Quanto custa o plano premium?	1										
9	Como faço para imprimir meus dadd	2										
10	Já tenho o curso designer e queria	3										
11	Coloquei meu nome com letra maiú	1										
12	Onde posso acessar meu certificado	2										
13	Que cursos devo fazer para crescer	3										
14	Se meu acesso expirar ainda tenho	1										
15	Achei um erro no video de Excel. A	2										
16	Que cursos vocês indicam para mei	3										
17	Possuo imprimir meu certificado?	1										
18	Bom, tenho interesse na parte d	2										
19	Qual a carreira indicada para meu fi	3										
20	Gostaria de saber se vocês entendem	1										
21	Qual versão do laravel é o melhor	2										
22	Ola, quero aprender a programar, m	3										
23	Comprei o plano Premium ontem m	1										

Salve esses dados como `emails.csv`, dentro da pasta onde estão os arquivos Python. Vejamos o conteúdo do nosso arquivo CSV:

```
email,classificacao
"Se eu comprar cinco anos antecipados, eu ganho algum desconto?",1
0 exercício 15 do curso de Java 1 está com a resposta errada. Pod e conferir pf?,2
Existe algum curso para cuidar do marketing da minha empresa?,3
"Gostaria de renovar antecipadamente meu plano, como posso fazer?",1
...
gosta de design mas não é fã de programação. Ele deve fazer lógic a?,3
```

Nesse arquivo temos o texto que se refere ao e-mail e um número que corresponde à sua categoria. Precisamos fazer a leitura desse arquivo. Como fazíamos antes? Utilizamos o Pandas, portanto, é o que faremos:

```
texto1 = "Se eu comprar cinco anos antecipados, eu ganho algum de sconto?"
texto2 = "O exercício 15 do curso de Java 1 está com a resposta e
```

```
rrada. Pode conferir pf?"  
texto3 = "Existe algum curso para cuidar do marketing da minha em  
presa?"  
  
import pandas as pd  
classificacoes = pd.read_csv('emails.csv')
```

Vamos imprimir a variável `classificacoes` e verificar o seu valor:

```
print(classificacoes)
```

Testando o nosso arquivo `classificando_emails.py`:

```
> python3 classificando_emails.py  
email  classifica  
cao  
0  Se eu comprar cinco anos antecipados, eu ganho...  
1  
1  O exercício 15 do curso de Java 1 está com a r...  
2  
2  Existe algum curso para cuidar do marketing da...  
3  
3  Gostaria de renovar antecipadamente meu plano,...  
1  
4  O vídeo não está travando as vezes no meu nave...  
2  
5  Que trilha vocês recomendam para quem quer com...  
3  
6          Quanto custa o plano premium?  
1  
...  
41 Estou com dificuldade para fazer o pagamento, ...  
1  
42 Gostaria de saber qual o melhor caminho para m...  
3
```

Ele exibe os e-mails e a classificação, com 42 e-mails no total. Qual é o próximo passo? Precisamos pegar a coluna `email`, que são os dados com os quais vamos trabalhar. Como fazemos mesmo no Pandas? Pedimos a coluna:

```
texto1 = "Se eu comprar cinco anos antecipados, eu ganho algum de sconto?"  
texto2 = "O exercício 15 do curso de Java 1 está com a resposta errada. Pode conferir pf?"  
texto3 = "Existe algum curso para cuidar do marketing da minha empresa?"  
  
import pandas as pd  
classificacoes = pd.read_csv('emails.csv')  
textosPuros = classificacoes['email']
```

Vamos verificar se os e-mails vieram corretamente:

```
print(textosPuros)
```

Executando o nosso arquivo novamente:

```
> python3 classificando_emails.py  
0    Se eu comprar cinco anos antecipados, eu ganho...  
1    O exercício 15 do curso de Java 1 está com a r...  
2    Existe algum curso para cuidar do marketing da...  
3    Gostaria de renovar antecipadamente meu plano,...  
4    O vídeo não está travando as vezes no meu nave...  
...  
41   Estou com dificuldade para fazer o pagamento, ...  
42   Gostaria de saber qual o melhor caminho para m...  
Name: email, dtype: object
```

Conseguimos pegar os textos, porém, precisamos das palavras separadas deles. A abordagem mais simples seria quebrar esses textos em espaços. Como fazer isso? Podemos pedir para o Pandas tratar cada texto como uma *string*, e então separar essas strings por espaço em branco:

```
# restante do código  
  
import pandas as pd  
classificacoes = pd.read_csv('emails.csv')  
textosPuros = classificacoes['email']  
textosQuebrados = textosPuros.str.split(' ')  
print(textosQuebrados)
```

Observe que a variável `textosQuebrados` refere-se aos textos que contêm as palavras separadas por espaço. Vamos verificar novamente como ficou o resultado:

```
> python3 classificando_emails.py
0    [Se, eu, comprar, cinco, anos, antecipados,, e...
1    [0, exercício, 15, do, curso, de, Java, 1, est...
2    [Existe, algum, curso, para, cuidar, do, marke...
3    [Gostaria, de, renovar, antecipadamente, meu, ...
...
41   [Estou, com, dificuldade, para, fazer, o, paga...
42   [Gostaria, de, saber, qual, o, melhor, caminho...
Name: email, dtype: object
```

Agora cada linha se refere a um array com cada uma das palavras separadas. Dado que temos vários arrays que contêm todas as palavras de cada texto, o que precisamos fazer agora? Criar um único array que contenha todas essas palavras, ou seja, o nosso dicionário. Começaremos criando um array chamado `dicionario`:

```
dicionario = []
```

Agora precisamos iterar sobre as listas da variável `textosQuebrados`:

```
for lista in textosQuebrados:
```

Para cada uma dessas listas, pediremos para estender-se ao nosso dicionário:

```
for lista in textosQuebrados:
    dicionario.extend(lista)
```

Agora vamos imprimir esse `dicionario`:

```
print(dicionario)
```

Vejamos o resultado da variável `dicionario`:

```
> python3 classificando_emails.py
['Se', 'eu', 'comprar', 'cinco', 'anos', 'antecipados,', 'eu', 'g
anho', 'algum', 'desconto?', '0', 'exercício', '15', 'do', 'curso
', 'de', 'Java', '1', 'está', 'com', 'a', 'resposta', 'errada.', '
Pode', 'conferir', 'pf?', 'Existe', 'algum', 'curso', 'para', 'cu
idar', 'do', 'marketing', 'da', 'minha', 'empresa?', 'Gostaria',
'de', 'renovar', 'antecipadamente', 'meu', 'plano,', 'como', 'pos
so',
...
'qual', 'o', 'melhor', 'caminho', 'para', 'meu', 'filho,', 'que',
'gosta', 'de', 'design', 'mas', 'não', 'é', 'fã', 'de', 'programa
ção.', 'Ele', 'deve', 'fazer', 'lógica?']
```

Agora temos um superdicionário! Entretanto, note a repetição de palavras, por exemplo, a palavra de se repete! Como vimos anteriormente, o dicionário é um array de palavras **distintas**, não podemos conter palavras iguais dentro dele.

Em vez de usarmos uma lista, precisamos usar um conjunto, pois na matemática, dentro do estudo da teoria dos conjuntos, se temos os dados {1, 2, 3, 4, 5}, não é permitido adicionar novamente qualquer número que já esteja dentro dele. Isto é, podemos apenas **adicionar qualquer valor diferente** de {1, 2, 3, 4, 5}. Tecnicamente chamamos esses conjuntos de set que, no inglês, significa *conjunto*. Modificaremos o nosso dicionário para que ele seja um set :

```
texto1 = "Se eu comprar cinco anos antecipados, eu ganho algum de
sconto?"
texto2 = "O exercício 15 do curso de Java 1 está com a resposta e
rrada. Pode conferir pf?"
texto3 = "Existe algum curso para cuidar do marketing da minha em
presa?"
```

```
import pandas as pd
classificacoes = pd.read_csv('emails.csv')
textosPuros = classificacoes['email']
textosQuebrados = textosPuros.str.split(' ')
dicionario = set()
```

```
for lista in textosQuebrados:  
    dicionario.extend(lista)  
  
print(dicionario)
```

Qual será o comportamento da variável `dicionario`? A cada vez que ele for adicionar um elemento dentro dele, ele verificará se este já existe. Caso não exista, ele adicionará; caso contrário, ele não adicionará. Além disso, precisamos modificar a forma como estamos adicionando os elementos, isto é, em vez do `extends`, usaremos o `update`:

```
# restante do código  
  
dicionario = set()  
  
for lista in textosQuebrados:  
    dicionario.update(lista)  
  
print(dicionario)
```

Vamos rodar novamente e verificar o resultado:

```
> python3 classificando_emails.py  
{'', 'resposta', 'muito', 'foi', 'está', 'travando', 'programar',  
'Django',  
...  
'fã', 'como', 'poderia', 'escolher', 'pais', 'expirar', 'ver', 'P  
osso', 'Comprei', '3', 'avisado', 'virtualização', 'realizados',  
'perto', 'python', 'tenho', 'começar', 'fazer?', 'dia', 'Gostar  
ia', 'a', 'comprar', 'pagar', 'queria', 'favor?', 'criar', 'errad  
a.', 'gosta', 'duvida', 'linguagens', 'mais.', 'isso?', 'Onde',  
'paypal', 'javascrip', 'meu', 'um', 'Terminei', 'estudei', 'começa  
r', 'que', 'curso!!', 'Senti', 'Alura?', 'Obs:', 'aulas', 'todo'  
, 'pouco', 'laravel', 'necessário', 'minutos.', 'Quanto', 'nome',  
'certificado', 'área', 'desse', 'distancia.', 'certificado', 'me  
, 'depoimento', 'algum', 'nosso', 'ótimo', 'web', 'sentiu', 'Como  
, 'enviar', 'IOS', 'crescer',  
...  
'maiúscula', 'se', 'vai', 'tornar', 'cartão'}
```

Agora as palavras não se repetem! Portanto, para pegarmos um conjunto de elementos distintos, utilizamos o `set`. Mas ainda existe um detalhe, pois quando tentamos procurar a palavra "como" dentro do nosso dicionário, ele acha tanto a palavra "como" quanto a "Como". Se uma palavra possui as mesmas letras, mas contém um acento diferente, ou uma pontuação ou letra maiúscula que diferencia uma da outra, ela será considerada uma palavra distinta. Como podemos resolver esse problema?

No caso, como fazemos para que palavras praticamente iguais não sejam adicionadas mais de uma vez? Quando nos deparamos com esse tipo de problema, precisamos fazer uma limpeza no nosso código, ou seja, transformar todas as palavras em minúsculo ou tudo em maiúsculo para que, independentemente de o usuário digitar maiúsculo ou minúsculo, as palavras sejam as mesmas. Então, vem a questão:

- E se fosse um e-mail de *spam*? Faria sentido convertermos tudo para maiúsculo ou minúsculo?

Quando nos deparamos com esse tipo de situação, ter tanto as palavras maiúsculas quanto minúsculas não faz diferença, pois podemos utilizar parâmetros como quantidade de palavras em maiúsculo ou minúsculo para avaliar se o e-mail é um *spam* ou não. Atualmente, não estamos interessados em realizar esse tipo de avaliação.

Mas e se o nosso objetivo for entender se o usuário está bravo? Com certeza quando nos deparamos com textos em maiúsculo, saberemos que esta é a sensação desse usuário. Portanto, palavras em maiúsculo ou minúsculo podem nos indicar alguma coisa, logo, precisamos avaliar se faz sentido ou não mantê-las. Nesse

caso, transformaremos todas em minúsculo.

Mas como fazemos isso no código? No momento em que transformamos a variável `textosPuros` em string, basta pedirmos que transforme todo o texto contido em minúsculo utilizando a função `lower`:

```
textosQuebrados = textosPuros.str.lower().str.split(' ')
```

Testando novamente o nosso arquivo `classificando_emails.py`, temos o seguinte resultado:

```
> python3 classificando_emails.py
{'', 'sou', 'fazer?', 'plataforma', 'vou', 'email', 'moderador?',
'correção', 'sistemas', 'empresa?', 'encontrei', 'todo', 'desejote',
'vertualização', 'nenhum', 'qualquer', 'senti', 'coluna', 'queim',
'linguagens', 'completa.', 'crescer', 'começar', 'emitem',
'só', 'esse', 'não', 'programa', 'plano', 'estou', 'escolher',
'usar', 'paypal', 'sei', 'certificado?', 'independência',
...
'segundo', 'windows?', 'necessário', 'equipe', 'desenvolvedor',
'parabéns!', 'alura?', 'trocar.', 'almejando', 'deseja', 'depoimento',
'práticas', 'mais', 'obs:', 'posso', 'explicação?', 'em',
'como', 'nos'}
```

Se procurarmos pela palavra "como" novamente, teremos apenas uma única palavra! Vejamos também a quantidade de palavras que temos atualmente em nosso dicionário:

```
# restante do código

totalDePalavras = len(dicionario)

print(dicionario)
print(totalDePalavras)
```

Testando novamente:

```
> python3 classificando_emails.py
...
```

364

Podemos concluir que o nosso dicionário contém 364 palavras distintas! O que precisamos fazer agora? Atribuir um número para cada uma das palavras contidas no nosso dicionário, por exemplo, a primeira palavra será o número 0, a segunda o 1, a terceira 2 e assim sucessivamente até chegar à última palavra. Nesse caso, os números vão variar de 0 a 363.

Podemos utilizar a função `zip` do Python. Com ela, podemos dizer o que queremos do lado esquerdo e do lado direito, por exemplo, a primeira palavra do `dicionario` no lado esquerdo e o número 0 no lado direito; a segunda palavra do `dicionario` no lado esquerdo e o número 1 do lado direito.

Primeiro, vamos chamar o `zip` enviando o nosso `dicionario` como primeiro parâmetro para que ele fique no lado esquerdo:

```
zip(dicionario)
```

Agora precisamos dizer o que queremos do lado direito, que são os números de 0 a 364 exclusive, ou seja, 0 a 363. Isso significa que precisamos escrever todos os valores de 0 a 363? Não!

Para resolver esse problema, vamos utilizar a função `range`, que recebe um número como parâmetro e cria um intervalo de 0 até o número enviado por parâmetro **exclusive**. Se enviarmos a quantidade de elementos contidos no `dicionario`, o `range` fará o intervalo de 0 a 363. Então vamos adicionar o `range` como segundo parâmetro, enviando a variável `totalDePalavras` como parâmetro que corresponde ao `len(dicionario)`:

```
zip(dicionario, range(totalDePalavras))
```

Agora, vamos excluir as impressões anteriores e apenas

imprimir o zip :

```
import pandas as pd
classificacoes = pd.read_csv('emails.csv')
textosPuros = classificacoes['email']
textosQuebrados = textosPuros.str.lower().str.split(' ')
dicionario = set()

for lista in textosQuebrados:
    dicionario.update(lista)

totalDePalavras = len(dicionario)

print(zip(dicionario, range(totalDePalavras)))
```

Rodando o nosso arquivo `classificando_emails.py`:

```
> python3 classificando_emails.py
<zip object at 0x1056dbf48>
```

Opa, ele imprimiu um objeto do tipo `zip`. Já sabemos que usamos o `zip`, queremos saber o conteúdo que está dentro dele e mostrá-lo como uma lista:

```
print(list(zip(dicionario, range(totalDePalavras))))
```

O resultado será:

```
[(' ', 0), ('paypal', 1), ('linguagens', 2), ('programação', 3),
 'conheço', 4), ('para', 5), ('sobre', 6), ('favor?', 7), ('alura!', 8),
 ('cadastrei', 9), ('início', 10), ('programar', 11), ('seria', 12),
 ('queria', 13), ('tentar', 14), ('aplicações?', 15), ('c
 urso', 16), ('ios', 17), ('caminho', 18), ('explicação?', 19),
 ('termos', 20), ('todo', 21), ('plano', 22), ('trocar', 23), ('duvida', 24),
 ('conferir', 25), ('gostaria', 26), ('vou', 27), ('existe', 28),
 ('quais', 29), ('eu', 30), ('brasil??', 31), ('plataforma', 32),
 ('pesquisas', 33), ('valido', 34), ('gráfica?', 35),
 'melhor', 36), ('novamente?', 37),
 ...
 ('uma', 348), ('parabéns!', 349), ('14', 350), ('aqui', 351), ('cursos', 352),
 ('java', 353), ('certificado?', 354), ('anos?', 355), ('boa', 356),
 ('muito', 357), ('acesso', 358), ('além', 359), ('maiúscula', 360),
 ('antecipados', 361), ('cartão', 362), ('sent
```

```
iu', 363)]
```

Observe que cada elemento do array representa uma tupla (uma n-upla de tamanho 2, <https://pt.wikipedia.org/wiki/Enupla>), isto é, um par ordenado de uma palavra e um número. Portanto, cada palavra contida no dicionário está associada a um número. Agora, retornaremos o resultado da função zip para uma variável chamada tuplas :

```
# restante do código

for lista in textosQuebrados:
    dicionario.update(lista)

totalDePalavras = len(dicionario)
tuplas = zip(dicionario, range(totalDePalavras))
```

Se quisermos consultar o número de uma palavra, por exemplo, a palavra "pode", como faremos isso? Vamos tentar pedindo para a tuplas . Neste teste, usaremos o interpretador do Python para facilitar os demais testes:

```
> python
>>> import pandas as pd
>>> classificacoes = pd.read_csv('emails.csv')
>>> textosPuros = classificacoes['email']
>>> textosQuebrados = textosPuros.str.lower().str.split(' ')
>>> dicionario = set()
>>>
>>> for lista in textosQuebrados:
...     dicionario.update(lista)
...
>>> totalDePalavras = len(dicionario)
>>> tuplas = zip(dicionario, range(totalDePalavras))
>>>
```

Tentaremos pedir pela palavra "pode":

```
>>> tuplas['pode']
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: 'zip' object is not subscriptable
>>>
```

Um conjunto `zip` não permite a busca por meio dos valores. Lembre-se de que, para buscarmos algum elemento por meio do seu valor chave, precisamos utilizar o dicionário do Python, que é justamente um mapa que permite identificar um valor por meio de uma chave. Declaramos um mapa da seguinte forma:

```
>>> mapa = {}
```

Vamos criar um dicionário chamado `palavrasEIndices` :

```
>>> mapa = {}
>>> palavrasEIndices = {}
```

Agora adicionaremos algumas palavras e suas respectivas posições:

```
>>> palavrasEIndices['pode']=15
>>> palavrasEIndices['poder']=18
```

Vejamos os valores contidos no mapa `palavrasEIndices` :

```
>>> print(palavrasEIndices)
{'pode': 15, 'poder': 18}
```

Note que, se tentarmos buscar a posição do elemento por meio da sua chave, por exemplo a palavra "pode", teremos:

```
>>> print(palavrasEIndices['pode'])
15
```

Considerando o que vimos sobre o dicionário do Python, o que precisamos fazer agora? Transformar a variável `tuplas` em um dicionário. Isto é, cada elemento da variável `tuplas` será transformado em um dicionário. Como fazemos isso? Simples! Faremos um `for` nas `tuplas` :

```
for palavra, indice in tuplas
```

Nesse instante, estamos pegando cada palavra e indice dos elementos da tuplas , ou seja, a palavra e seu número. A partir deles, criaremos o nosso dicionário:

```
{palavra:indice for palavra, indice in tuplas}
```

Então atribuímos esse dicionário para a variável palavrasEIndices :

```
palavrasEIndices = {palavra:indice for palavra, indice in tuplas}
```

Vamos testar esse código. Copie e cole no interpretador, e então imprima o dicionário palavrasEIndices :

```
>>> palavrasEIndices = {palavra:indice for palavra, indice in tuplas}
>>> print(palavrasEIndices)
{': 0, 'pf?': 1, 'convidado': 2, 'parece': 3, 'gráfica?': 4, '3': 5, 'ver': 6, 'anos': 7, 'foi': 8, 'já': 9, 'cobol,' : 10, 'estou': 11, 'filho': 12, 'no': 13, '14': 14, 'vou': 15, 'muito': 16, 'vai': 17, 'exercício': 18, 'envio': 19, 'programar': 20, 'resposta': 21, 'excel.': 22, 'qualquer': 23, 'linguagens,' : 24, 'só': 25, 'como': 26, 'sou': 27, 'vezes': 28, 'pai': 29, 'web': 30, 'gostaria': 31, 'travando': 32, 'pesquisas': 33, 'porem': 34, 'aplicaçõe s': 35, 'proximidade': 36, 'identifiquei': 37, 'serei': 38, 'html ,': 39, 'praticas': 40, 'é': 41, 'termos': 42, 'interesse': 43, 'conteúdo?': 44, 'curso!': 45, 'fica': 46, 'realmente': 47, 'cadas trar,' : 48, 'que': 49, 'framework': 50, 'aprender': 51,
...
'virtualização': 354, 'anos?': 355, 'absorver': 356, 'pagamento,': 357, 'java': 358, 'cinco': 359, 'web?': 360, 'windows?': 361, 'tenho': 362, 'inicialmente.': 363}
>>>
```

Agora temos um dicionário com as nossas tuplas. Será que agora conseguimos encontrar a posição de alguma palavra buscando por ela mesma? Vamos testar o exemplo da palavra "pode"!

```
>>> print(palavrasEIndices['pode'])  
339
```

Temos um mapa que indica se existe uma palavra no nosso dicionário e qual é a posição dela. Vamos tentar a palavra "guilherme":

```
>>> print(palavrasEIndices['guilherme'])  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'guilherme'
```

Perceba que, quando não existir uma palavra, ele nos devolverá essa mensagem, indicando um erro de chave, pois não existe para esse mapa. Porém, se procurarmos palavras que existem, como por exemplo, "com" e "como":

```
>>> print(palavrasEIndices['com'])  
161  
>>> print(palavrasEIndices['como'])  
124
```

Ele nos devolve suas respectivas posições. O que falta agora para o nosso algoritmo? É justamente varrer cada texto e marcar quantas vezes uma determinada palavra se repete. Considere o texto:

- Se eu comprar cinco anos antecipados, eu ganho algum desconto?

O nosso algoritmo deverá verificar primeiro a palavra "Se", então ele vai encontrar a posição dessa palavra e marcar como 1. Depois, ele vai verificar a posição da palavra "eu" e marcará como 1, e assim por diante. Em outras palavras, precisamos pegar os nossos textos e transformá-los em arrays de números por meio do nosso dicionário, que fará o papel de tradutor — isto é, ele vai

traduzir um texto em array de números. Portanto, podemos chamá-lo de tradutor :

```
# restante do código

tuplas = zip(dicionario, range(totalDePalavras))

tradutor = {palavra:indice for palavra, indice in tuplas}
```

Antes de contabilizar a quantidade de vezes que uma palavra se repete nos nossos textos, vamos verificar as palavras contidas no primeiro texto da variável textosQuebrados :

```
>>> print(textosQuebrados[0])
['se', 'eu', 'comprar', 'cinco', 'anos', 'antecipados,', 'eu', 'ganho',
'algum', 'desconto?']
```

Como podemos fazer para analisar cada uma dessas palavras? Precisamos fazer um laço que passe por cada uma delas, ou seja, um for :

```
for palavra in textosQuebrados[0]:
```

E então, vamos imprimir cada uma dessas palavras:

```
>>> for palavra in textosQuebrados[0]:
...     print(palavra)
...
se
eu
comprar
cinco
anos
antecipados,
eu
ganho
algum
desconto?
```

Qual é o próximo passo? Nesse exemplo, precisamos pegar primeiro a palavra "se" e contabilizar 1 no array que representa essa

frase. Como ainda não temos esse array, precisamos criá-lo. Lembra quando tínhamos apenas aquelas três frases de exemplo:

- Se eu comprar cinco anos antecipados, eu ganho algum desconto?
  - Eu ganho desconto se comprar cinco anos antecipados?
  - Ao terminar um curso, eu ganho um certificado?

Usando somente essas frases, o dicionário que as suporta será:

[Se, eu, comprar, cinco, anos, antecipados, ganho, algum, desconto, Ao, terminar, um, curso, certificado]

Somente para as três frases, esse dicionário contém apenas 14 palavras distintas, portanto, para elas, utilizamos um array de tamanho 14. Entretanto, no nosso cenário atual, temos 364 palavras distintas! Um array que representa qualquer frase dos nossos textos precisará ter um tamanho de 364 posições. Assim, criaremos um vetor com 364 posições, isto é, um que tenha o tamanho da variável `totalDePalavras` e que inicie com valor 0:

```
vetor = [0] * totalDePalavras
```

Vamos verificar se o nosso vetor está funcionando como o esperado:

304

Temos um vetor de tamanho 364 com todos os valores iguais a 0. Essa é a representação de uma frase vazia, isto é, que não tenha pelo menos uma posição com valor acima de 0. Agora que temos o nosso array que permite representarmos todas as frases do nosso universo (todas as que são escritas com as palavras contidas no nosso dicionário), precisamos verificar quantas vezes uma palavra se repete. Vamos utilizar o exemplo anterior:

```
>>> for palavra in textosQuebrados[0]:  
...     print(palavra)  
...  
se  
eu  
comprar  
cinco  
anos  
antecipados,  
eu  
ganho  
algum  
desconto?
```

Qual é o primeiro passo? Pegaremos a palavra "se" e, em seguida, precisamos verificar se ela existe no nosso dicionário, na variável `tradutor`. Caso exista, pegamos a sua posição, por exemplo, se a palavra "se" estiver na posição 32, pegamos o array que representa a frase e somamos 1 na posição 32.

Precisamos implementar esses passos dentro do nosso código.

Porém, antes de começarmos com essa implementação, adicionaremos todos os passos realizados no interpretador do Python dentro do nosso arquivo `classificando_emails.py`. O nosso código atual fica da seguinte forma:

```
import pandas as pd
classificacoes = pd.read_csv('emails.csv')
textosPuros = classificacoes['email']
textosQuebrados = textosPuros.str.lower().str.split(' ')
dicionario = set()

for lista in textosQuebrados:
    dicionario.update(lista)

totalDePalavras = len(dicionario)
tuplas = zip(dicionario, range(totalDePalavras))
tradutor = {palavra:indice for palavra, indice in tuplas}

texto = textosQuebrados[0]
vetor = [0] * totalDePalavras

print(texto)
print(vetor)
for palavra in texto:
    print(palavra)
```

Qual é o nosso primeiro passo? Precisamos verificar palavra por palavra, portanto, vamos trabalhar na instrução `for` que, a partir de um texto, extraí uma palavra:

```
for palavra in texto:
    print(palavra)
```

Dada uma palavra do nosso texto, precisamos verificar se ela existe no nosso dicionário. Ou seja, precisamos adicionar um `if` que vai verificar se ela está contida no nosso dicionário, nesse caso, a variável `tradutor`:

```
for palavra in texto:
    if palavra in tradutor:
```

```
print(palavra)
```

Caso a palavra exista no nosso dicionário, o que precisamos fazer? Imprimi-la? Não! Então vamos retirar o código que a imprime:

```
for palavra in texto:  
    if palavra in tradutor:
```

E agora, o que temos de fazer? Precisamos somar 1 no vetor de acordo com a posição da palavra, mas como saberemos a posição? Simples! Basta pedir a posição da palavra desejada para o nosso dicionário `tradutor`, que tem o objetivo de traduzir uma palavra contida no dicionário em número (posição):

```
for palavra in texto:  
    if palavra in tradutor:  
        posicao = tradutor[palavra]
```

Em seguida, somamos 1 na variável `vetor` de acordo com a posição contida na `posicao`:

```
for palavra in texto:  
    if palavra in tradutor:  
        posicao = tradutor[palavra]  
        vetor[posicao] += 1
```

Esse nosso código está fazendo os seguintes passos:

1. Pega uma palavra contida na nossa variável `texto`;
2. Verifica se essa palavra está contida no dicionário `tradutor`;
3. Se sim, pede para o tradutor devolver a posição, e então soma 1 no `vetor`.

Ele vai repetindo esse processo até verificar todas as palavras contidas na variável `texto`, que é um array de strings. Por fim, o

nosso resultado final é justamente a variável `vetor`, pois ela traduz o texto em um array de números.

Em vez de imprimir a variável `vetor` antes do `for`, vamos imprimi-la por último, pois queremos verificar a nossa palavra traduzida:

```
for palavra in texto:  
    if palavra in tradutor:  
        posicao = tradutor[palavra]  
        vetor[posicao] += 1  
  
print(vetor)
```

Vamos testar o nosso arquivo `classificando_emails.py` e verificar o resultado:

```
> python3 classificando_emails.py  
['se', 'eu', 'comprar', 'cinco', 'anos', 'antecipados', 'eu', 'g  
anho', 'algum', 'desconto?']  
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

O array foi impresso com algumas posições preenchidas. As palavras contidas nesse texto apareceram apenas uma vez — com exceção de uma, que foram duas vezes (nesse caso, a palavra "eu").

Isso ocorreu porque a palavra que pedimos para ele traduzir contém palavras que o nosso dicionário (a variável `tradutor`) conhece. Caso o texto não tivesse nenhuma das palavras do nosso `tradutor`, ele simplesmente devolveria um array preenchido com zeros apenas, pois ignoraria palavra por palavra dentro desse texto.

Aparentemente terminamos os passos que precisávamos realizar, porém, temos de reutilizar esse processo. Ou seja, precisamos transformar esse `for` — que faz todo esse trabalho de, a partir de um texto, pegar uma palavra e transformá-la em um array de números — em uma função para que possamos traduzir todos os nossos textos. Criaremos uma função que se chamará `vetorizar_texto()`:

```
# restante do código

def vetorizar_texto():

    texto = textosQuebrados[0]
    vetor = [0] * totalDePalavras

    print(texto)

    for palavra in texto:
        if palavra in tradutor:
            posicao = tradutor[palavra]
            vetor[posicao] += 1

    print(vetor)
```

Precisamos enviar tanto o texto quanto o tradutor que vai traduzi-lo:

```
def vetorizar_texto(texto, tradutor):
```

Dentro dessa função, o nosso primeiro passo é calcular o tamanho do vetor que representará cada palavra contida no texto.

Podemos reutilizar o código que fizemos anteriormente:

```
def vetorizar_texto(texto, tradutor):
    vetor = [0] * totalDePalavras
```

Porém, perceba que não estamos recebendo a variável `totalDePalavras` e precisamos recebê-la como parâmetro. Mas realmente precisamos de uma variável para saber a quantidade total de palavras contidas no nosso `tradutor`? Não!

Podemos pegar o próprio tamanho do `tradutor`, que corresponde à quantidade total de palavras, isto é, a quantidade exata que o nosso `vetor` precisa ter:

```
def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
```

Agora podemos copiar e colar todo o `for` usado para iterar sobre cada palavra contida em um texto:

```
def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
    for palavra in texto:
        if palavra in tradutor:
            posicao = tradutor[palavra]
            vetor[posicao] += 1
```

Por fim, precisamos retornar o vetor obtido:

```
def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
    for palavra in texto:
        if palavra in tradutor:
            posicao = tradutor[palavra]
            vetor[posicao] += 1

    return vetor
```

Antes de rodarmos o nosso código, em vez de imprimirmos a variável `vetor`, vamos imprimir o retorno da função

```
vetorizar_texto :  
  
# restante do código  
  
def vetorizar_texto(texto, tradutor):  
    vetor = [0] * len(tradutor)  
    for palavra in texto:  
        if palavra in tradutor:  
            posicao = tradutor[palavra]  
            vetor[posicao] += 1  
  
    return vetor  
  
texto = textosQuebrados[0]  
  
print(texto)  
print(vetorizar_texto(texto, tradutor))
```

Vamos testar o nosso código e verificar o resultado:

Nossa função devolve o nosso texto vetorizado da mesma forma que antes. Vamos testar outros textos para verificar seus

vetores? Vejamos como fica:

```
print(vetorizar_texto(textosQuebrados[1], tradutor))
print(vetorizar_texto(textosQuebrados[2], tradutor))
```

Testando novamente o nosso arquivo `classificando_emails.py`, temos o seguinte resultado:

O nosso algoritmo imprime um vetor diferente para cada texto distinto que temos. Vale lembrar que, nesse instante, nosso código está equivalente a:

```
import pandas as pd

def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
    for palavra in texto:
        if palavra in tradutor:
            posicao = tradutor[palavra]
            vetor[posicao] += 1

    return vetor

classificacoes = pd.read_csv('emails.csv')
textosPuros = classificacoes['email']
textosQuebrados = textosPuros.str.lower().str.split(' ')
dicionario = set()

for lista in textosQuebrados:
    dicionario.update(lista)

totalDePalavras = len(dicionario)
tuplas = zip(dicionario, range(totalDePalavras))
tradutor = {palavra:indice for palavra, indice in tuplas}

print(vetorizar_texto(textosQuebrados[1], tradutor))
print(vetorizar_texto(textosQuebrados[2], tradutor))
```

Precisamos realizar essa tarefa para todos os textos que temos dentro da variável `textosQuebrados`. Como faremos isso? Fazemos um `for` para pegar cada texto contido na variável `textosQuebrados`:

```
# restante do código

for texto in textosQuebrados
```

Em seguida, aplicamos a função `vetorizar_texto()`:

```
# restante do código

vetorizar_texto(texto, tradutor) for texto in textosQuebrados
```

Entretanto, precisamos transformar cada resultado desse em um array, portanto, faremos isso:

```
# restante do código

[vetorizar_texto(texto, tradutor) for texto in textosQuebrados]
```

Esses serão os nossos vetores de textos, ou melhor, atribuiremos esses arrays para a variável `vetoresDeTexto`:

```
# restante do código

vetoresDeTexto = [vetorizar_texto(texto, tradutor) for texto in textosQuebrados]
```

Vamos rodar o nosso código. Porém, apague as três linhas que usamos para testar a chamada de outros textos para a nossa função e imprima apenas a variável `vetoresDeTexto`:

```
# restante do código

def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
    for palavra in texto:
        if palavra in tradutor:
            posicao = tradutor[palavra]
            vetor[posicao] += 1

    return vetor

vetoresDeTexto = [vetorizar_texto(texto, tradutor) for texto in textosQuebrados]
print(vetoresDeTexto)
```

Testando novamente o nosso arquivo `classificando_emails.py`, temos:

Ele imprime um array de arrays bem gigante. Nesse caso, é um array que contém 43 arrays, sendo que cada array contém 364 posições que representam cada um dos textos vetorizados.

Agora concluímos o nosso primeiro objetivo, que era justamente pegar todos os textos que temos e transformá-los em

arrays numéricos do mesmo tamanho para todos. Qual é o próximo passo? Utilizar o nosso algoritmo que é capaz de realizar as classificações.

Para classificar, nosso algoritmo precisa dos dados e das marcações. Os nossos dados estão contidos na variável `vetoresDeTexto`, mas e as nossas marcações? Qual coluna do nosso arquivo `emails.csv` se refere às marcações para os nossos textos? Vejamos:

```
email,classificacao
"Se eu comprar cinco anos antecipados, eu ganho algum desconto?",1
0 exercício 15 do curso de Java 1 está com a resposta errada. Pod
e conferir pf?,2
Existe algum curso para cuidar do marketing da minha empresa?,3
"Gostaria de renovar antecipadamente meu plano, como posso fazer?
",1
0 vídeo não está travando as vezes no meu navegador. Como trocar
o player?,2
Que trilha vocês recomendam para quem quer começar com programação
o?,3
...
...
```

Temos a coluna `email` e a `classificacao`, ou seja, as nossas marcações para os nossos textos são todos os valores contidos na coluna `classificacao`. Logo, precisamos pedir para o nosso datagrama `classificacoes`:

```
import pandas as pd
classificacoes = pd.read_csv('emails.csv')

# restante do código
```

E usamos os valores da coluna `classificacoes` e atribuímos para a variável `marcas`:

```
# restante do código
```

```

def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
    for palavra in texto:
        if palavra in tradutor:
            posicao = tradutor[palavra]
            vetor[posicao] += 1

    return vetor

vetoresDeTexto = [vetorizar_texto(texto, tradutor) for texto in t
extosQuebrados]
marcas = classificacoes['classificacao']

```

A partir de agora, precisamos pegar tanto o X quanto o Y. Você se lembra de que o X se refere aos nossos dados, e o Y, às marcações? Em outras palavras, atribuiremos a variável vetoresDeTexto ao X e as marcas para o Y:

```

# restante do código

vetoresDeTexto = [vetorizar_texto(texto, tradutor) for texto in t
extosQuebrados]
marcas = classificacoes['classificacao']

X = vetoresDeTexto
Y = marcas

```

No nosso algoritmo de classificação, após coletarmos o X e o Y, definíamos os percentuais de treino. Neste caso, vamos usar 80%:

```

# restante do código

X = vetoresDeTexto
Y = marcas

porcentagem_de_treino = 0.8

```

Esses 80% significam treino e teste, portanto, os 20% restantes serão usados para validação. Agora precisamos definir o tamanho do treino e teste, e também o tamanho de validação. Para o

tamanho de treino e teste, multiplicaremos a variável porcentagem\_de\_treino com o tamanho do Y :

```
# restante do código

X = vetoresDeTexto
Y = marcas

porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
```

Entretanto, o tamanho de validação será o tamanho do Y menos a variável tamanho\_de\_treino :

```
# restante do código

X = vetoresDeTexto
Y = marcas

porcentagem_de_treino = 0.8

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino
```

O nosso próximo passo é justamente separar os dados e as marcações de treino com os dados e marcações de validação. Começaremos pelos dados de treino:

```
# restante do código

treino_dados = X[0:tamanho_de_treino]
treino_marcações = Y[0:tamanho_de_treino]
```

Para os dados e marcações de teste, pegaremos desde o primeiro elemento (posição 0) até a variável tamanho\_de\_treino . E para os dados e marcações de validação? Pegaremos a partir da variável tamanho\_de\_treino em diante:

```
validacao_dados = X[tamanho_de_treino:]
```

```
validacao_marcacoes = Y[tamanho_de_treino:]
```

Definimos todos os nossos dados que vamos usar para os nossos algoritmos. Qual era o próximo passo? Realizar o `fit` e `predict`, a mesma tarefa que fazíamos anteriormente nos nossos algoritmos anteriores. Logo, vamos dar uma olhada no nosso arquivo `situacao_do_cliente_kfold.py`:

```
import pandas as pd
from collections import Counter
import numpy as np
from sklearn.model_selection import cross_val_score

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia', 'frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
# tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[:tamanho_de_treino]
treino_marcacoes = Y[:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
    cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto
```

```

print(msg)
return taxa_de_acerto

def teste_real(modelo, validacao_dados, validacao_marcacoes):
    resultado = modelo.predict(validacao_dados)

    acertos = resultado == validacao_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(validacao_marcacoes)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

    msg = "Taxa de acerto do vencedor entre os dois algoritmos no mundo real: {}".
format(taxa_de_acerto)
    print(msg)

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne,
treino_dados, treino_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMultinomial,
treino_dados, treino_marcacoes)
resultados[resultadoMultinomial] = modeloMultinomial

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost,
treino_dados, treino_marcacoes)
resultados[resultadoAdaBoost] = modeloAdaBoost

```

```

daBoost, treino_dados, treino_marcacoes)
resultados[resultadoAdaBoost] = modeloAdaBoost

print(resultados)

maximo = max(resultados)
vencedor = resultados[maximo]

print("Vencedor: ")
print(vencedor)

vencedor.fit(treino_dados, treino_marcacoes)

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)

```

Veja que realizamos todos os passos iniciais da mesma forma como foi implementado no `situacao_do_cliente_kfold.py`, portanto, vamos pegar as funções adiante. Começaremos pelo `fit_and_predict`. Copie e cole no arquivo `classificando_emails.py`. O nosso arquivo fica com o seguinte código:

```

import pandas as pd

def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
    for palavra in texto:
        if palavra in tradutor:
            posicao = tradutor[palavra]
            vetor[posicao] += 1

    return vetor

```

```

classificacoes = pd.read_csv('emails.csv')
textosPuros = classificacoes['email']
textosQuebrados = textosPuros.str.lower().str.split(' ')
dicionario = set()

for lista in textosQuebrados:
    dicionario.update(lista)

totalDePalavras = len(dicionario)
tuplas = zip(dicionario, range(totalDePalavras))
tradutor = {palavra:indice for palavra, indice in tuplas}

vetoresDeTexto = [vetorizar_texto(texto, tradutor) for texto in textosQuebrados]
marcas = classificacoes['classificacao']

X = vetoresDeTexto
Y = marcas

porcentagem_de_treino = 0.8

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[0:tamanho_de_treino]
treino_marcacoes = Y[0:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
    cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto

```

Mas ainda precisamos de um algoritmo para testar o nosso código. Utilizaremos o mesmo exemplo do OneVsRest no

arquivo `situacao_do_cliente_kfold.py` , isto é, copie e cole no arquivo `classificando_emails.py` :

```
# restante do código

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
es, cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest
```

Vamos rodar o nosso arquivo `classificando_emails.py` e verificar qual é o resultado para o algoritmo `OneVsRest` :

```
> python3 classificando_emails.py
Traceback (most recent call last):
  File "classificando_emails.py", line 61, in <module>
    resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVs
Rest, treino_dados, treino_marcacoes)
  File "classificando_emails.py", line 50, in fit_and_predict
    scores = cross_val_score(modelo, treino_dados, treino_marcaco
es, cv = k)
NameError: name 'cross_val_score' is not defined
```

Ele apresenta um erro. Perceba que é de extrema importância rodarmos o passo a passo durante o processo de implementação. Quando aderimos a essa prática, temos um maior controle em

detectar as falhas do nosso algoritmo.

Por exemplo, imagine que pegamos todo o código do arquivo `situacao_do_cliente_kfold.py` e que começou a dar erro em diversos pontos. Seria um pesadelo ficar verificando todos os pontos ao mesmo tempo. É válido lembrar que, se estivéssemos utilizando dados muito similares aos quais usávamos anteriormente, provavelmente não haveria problema algum copiar e colar todo o código.

Vamos consertar esse erro que está sendo apresentado. Observe a mensagem:

```
NameError: global name 'cross_val_score' is not defined
```

Isso significa que não importamos a função `cross_val_score`, então vamos fazer isso! Mas aproveitando que vamos importar essa função, vamos também verificar os demais imports contidos no arquivo `situacao_do_cliente_kfold.py`, pois eles provavelmente serão solicitados.

```
import pandas as pd
from collections import Counter
import numpy as np
from sklearn.model_selection import cross_val_score

# restante do código
```

Dentro do arquivo `classificando_emails.py`, usaremos os mesmos imports:

```
import pandas as pd
from collections import Counter
import numpy as np
from sklearn.model_selection import cross_val_score
```

```
# restante do código
```

Agora que realizamos todos os `import`s, vamos testar o nosso algoritmo e verificar o resultado:

```
> python3 classificando_emails.py  
Taxa de acerto do OneVsRest: 0.723333333333
```

Agora sim o nosso algoritmo funcionou! Observe que a taxa de acerto foi de 72,33% no teste, ou seja, em 72,33% das vezes ele saberá encaminhar o e-mail recebido para o lugar certo. É válido lembrar que precisamos realizar todo o processo que fazíamos antes, ou seja, comparar com o algoritmo base e também realizar o teste de validação.

Logo mais implementaremos todos esses passos, mas, vamos adicionar os demais algoritmos. O próximo que implementaremos será o `OneVsOne`. Vamos copiar e colá-lo a partir do arquivo `situacao_do_cliente_kfold.py`:

```
# restante do código
```

```
resultados = []  
  
from sklearn.multiclass import OneVsRestClassifier  
from sklearn.svm import LinearSVC  
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))  
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,  
, treino_dados, treino_marcacoes)  
resultados[resultadoOneVsRest] = modeloOneVsRest  
  
from sklearn.multiclass import OneVsOneClassifier  
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))  
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t  
reino_dados, treino_marcacoes)  
resultados[resultadoOneVsOne] = modeloOneVsOne
```

Vamos rodar e verificar o resultado:

```
> python3 classificando_emails.py
Taxa de acerto do OneVsRest: 0.723333333333
Taxa de acerto do OneVsOne: 0.656666666666
```

Agora o nosso código funcionou, resultando em 72,33% para o OneVsRest e 65,66% para o OneVsOne . Vamos para o próximo algoritmo, ou seja, o Multinomial :

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t
reino_dados, treino_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMul
tinomial, treino_dados, treino_marcacoes)
resultados[resultadoMultinomial] = modeloMultinomial
```

Vejamos o resultado:

```
> python3 classificando_emails.py
364
34
Taxa de acerto do OneVsRest: 0.723333333333
Taxa de acerto do OneVsOne: 0.656666666666
Taxa de acerto do MultinomialNB: 0.715000001
```

Rodou sem nenhum problema! Por fim, adicionaremos o

```
AdaBoost :  
  
# restante do código  
  
resultados = {}  
  
...  
  
from sklearn.ensemble import AdaBoostClassifier  
modeloAdaBoost = AdaBoostClassifier()  
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloA  
daBoost, treino_dados, treino_marcacoes)  
resultados[resultadoAdaBoost] = modeloAdaBoost
```

Vejamos o resultado:

```
> python3 classificando_emails.py  
364  
34  
Taxa de acerto do OneVsRest: 0.723333333333  
Taxa de acerto do OneVsOne: 0.656666666667  
Taxa de acerto do MultinomialNB: 0.715  
Taxa de acerto do AdaBoostClassifier: 0.43999999
```

Todos os algoritmos estão rodando. O que está faltando agora? Adicionar os demais passos, isto é, em que elegemos o algoritmo vencedor, rodamos o teste do mundo real, e então rodamos o algoritmo base e verificamos o resultado final. Vamos adicionar este código:

```
# restante do código  
  
print(resultados)  
  
maximo = max(resultados)  
vencedor = resultados[maximo]  
  
print("Vencedor: ")  
print(vencedor)  
  
vencedor.fit(treino_dados, treino_marcacoes)
```

```

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)

```

Note que estamos usando a função `teste_real`, portanto, vamos adicioná-la também:

```

def teste_real(modelo, validacao_dados, validacao_marcacoes):
    resultado = modelo.predict(validacao_dados)

    acertos = resultado == validacao_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(validacao_marcacoes)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

    msg = "Taxa de acerto do vencedor entre os dois algoritmos no mundo real: {0}".format(taxa_de_acerto)
    print(msg)

```

Testando o nosso código:

```

> python3 classificando_emails.py
Taxa de acerto do OneVsRest: 0.7233333333333334
Taxa de acerto do OneVsOne: 0.6566666666666666
Taxa de acerto do MultinomialNB: 0.7150000000000001
Taxa de acerto do AdaBoostClassifier: 0.4233333333333323
{0.7233333333333338: OneVsRestClassifier(...)}
Vencedor:
OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None,
dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real

```

: 88.88888888888889

Taxa de acerto base: 44.444444

Total de teste: 9

Ele está funcionando conforme o esperado. O vencedor foi o `OneVsRest` com 72,33%, acertando 88,88% das vezes no teste de validação. Note que o algoritmo base teve um resultado de 44,44%.

Podemos concluir que, se utilizássemos o algoritmo base, em 56% das vezes teríamos um resultado errado. Porém, se usássemos o `OneVsRest`, em apenas 11% das vezes teríamos um erro. Além disso, repare no valor que o `AdaBoost` nos apresentou dessa vez:

Taxa de acerto do `AdaBoostClassifier`: 0.423333333333

Anteriormente, ele havia nos apresentado 47,33% de taxa de acerto, mas desta vez ele nos apresentou 43,99%. Por que será que isso aconteceu? O algoritmo `AdaBoostClassifier` tenta adaptar o conjunto de dados que ele recebe para encontrar o melhor resultado e, nesse processo, ele utiliza cópias dos dados para realizar essas tarefas.

Porém, esses processos são baseados em parâmetros aleatórios. A aleatoriedade significa que, a cada nova execução, podemos ter um resultado diferente.

Como vimos anteriormente, quando estamos utilizando algoritmos classificadores, precisamos que os resultados sejam fixos, caso contrário, teremos olhado diversas vezes os mesmos dados, viciando nossa decisão. Não poderemos tomar alguma decisão concreta sabendo que os resultados podem acontecer por "sorte" dessa vez ou "azar da próxima".

Precisamos também que a execução possa ser repetida da

mesma maneira que antes, se desejamos atualizar o nosso programa com novas partes. Como podemos resolver isso?

Em algoritmos que se baseiam em valores aleatórios, existe um parâmetro, geralmente chamado *seed* ([https://en.wikipedia.org/wiki/Random\\_seed](https://en.wikipedia.org/wiki/Random_seed)), que é um número de inicialização para a geração dos números aleatórios. Ou seja, por meio desse número, os demais números "aleatórios" são gerados.

Por exemplo, se o valor do *seed* for fixo, significa que os números gerados a partir dele serão sempre os mesmos; caso o contrário, se o *seed* for sempre um valor diferente a cada vez que rodarmos, haverá a possibilidade de termos números diferentes aos anteriores.

No caso do AdaBoost, quando não informamos o valor do *seed*, a cada instância ele utiliza um valor diferente. É justamente por esse motivo que tivemos resultados diferentes. Para resolver esse problema, basta apenas informarmos um valor fixo para o *seed* por meio do parâmetro *random\_state* no momento em que criamos o *AdaBoostClassifier*. Utilizaremos o valor 0:

```
# restante do código  
  
from sklearn.ensemble import AdaBoostClassifier  
modeloAdaBoost = AdaBoostClassifier(random_state=0)  
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost, treino_dados, treino_marcacoes)  
resultados[resultadoAdaBoost] = modeloAdaBoost
```

Vamos rodar mais uma vez o nosso algoritmo e verificar o resultado:

```
> python3 classificando_emails.py  
...
```

```
Taxa de acerto do AdaBoostClassifier: 0.4233333333333323
```

```
...
```

Independentemente da quantidade de vezes que rodarmos o nosso algoritmo, sempre teremos o mesmo resultado para o algoritmo `AdaBoostClassifier`, portanto, o seu resultado é de 42,33%. O nosso código final fica da seguinte forma:

```
import pandas as pd
from collections import Counter
import numpy as np
from sklearn.model_selection import cross_val_score

def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
    for palavra in texto:
        if palavra in tradutor:
            posicao = tradutor[palavra]
            vetor[posicao] += 1

    return vetor

classificacoes = pd.read_csv('emails.csv')
textosPuros = classificacoes['email']
textosQuebrados = textosPuros.str.lower().str.split(' ')
dicionario = set()

for lista in textosQuebrados:
    dicionario.update(lista)

totalDePalavras = len(dicionario)
tuplas = zip(dicionario, range(totalDePalavras))
tradutor = {palavra:indice for palavra, indice in tuplas}

vetoresDeTexto = [vetorizar_texto(texto, tradutor) for texto in textosQuebrados]
marcas = classificacoes['classificacao']

X = vetoresDeTexto
Y = marcas

porcentagem_de_treino = 0.8
```

```

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[0:tamanho_de_treino]
treino_marcacoes = Y[0:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes,
    cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto

def teste_real(modelo, validacao_dados, validacao_marcacoes):

    resultado = modelo.predict(validacao_dados)

    acertos = resultado == validacao_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(validacao_marcacoes)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

    msg = "Taxa de acerto do vencedor entre os dois algoritmos no
    mundo real: {0}".format(taxa_de_acerto)
    print(msg)

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest,
, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

```

```

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t
reino_dados, treino_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMul
tinomial, treino_dados, treino_marcacoes)
resultados[resultadoMultinomial] = modeloMultinomial

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier(random_state=0)
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloA
daBoost, treino_dados, treino_marcacoes)
resultados[resultadoAdaBoost] = modeloAdaBoost

print(resultados)

maximo = max(resultados)
vencedor = resultados[maximo]

print("Vencedor: ")
print(vencedor)

vencedor.fit(treino_dados, treino_marcacoes)

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcaco
es)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)

```

Lembre-se de que todo código pode ser encontrado no GitHub mencionado na introdução do livro, em <https://github.com/guilhermesilveira/machine-learning>.

## 10.1 RESUMINDO

Neste capítulo, aprendemos como podemos classificar textos utilizando os nossos algoritmos de classificação. Nosso primeiro passo foi armazenar todas as palavras distintas dentro de uma lista ou estrutura de dados. Para garantir que não existirão valores iguais, aprendemos que precisamos usar o mesmo conceito da teoria dos conjuntos na matemática. No nosso código, utilizamos o `set`.

Além disso, vimos que precisamos transformar todas as palavras em números, isto é, para cada palavra contida no nosso texto, precisamos associá-la a um número. Por exemplo, a palavra "se" será o número 0, a palavra "come" será 1 e assim por diante. Para esse problema, usamos a função `zip`, que nos devolve um array com tuplas que são justamente pares ordenados que associam um valor ao outro, por exemplo: `{se, 1}, {como, 2}` etc.

Com as tuplas, criamos uma estrutura que, a partir de uma palavra, permitia encontrarmos o seu número. Chamamos essa estrutura de `tradutor`. Em seguida, criamos uma função que, dado um texto qualquer, "vetorizava-o", ou seja, criava um array com a quantidade de posições igual à quantidade de palavras contidas no `tradutor`.

Então, consultamos o número da palavra pelo `tradutor` e, a partir do número obtido, somamos 1 no array para representar a quantidade de vezes que aquela palavra se repetia no texto. Por fim, pegamos todos os arrays obtidos dessa função, extraímos todos os nossos dados e marcações e, a partir deles, testamos os nossos algoritmos.



## CAPÍTULO 11

# QUEBRANDO NA PONTUAÇÃO ADEQUADA

Conseguimos criar um algoritmo capaz de classificar os nossos e-mails em três categorias distintas. Esses e-mails são dados reais que extraí do meu dia a dia, e as categorias são:

- **Categoria 1:** comercial.
- **Categoria 2:** técnico.
- **Categoria 3:** carreira.

O nosso próximo passo é justamente entender como o nosso algoritmo funcionou de acordo com o nosso dicionário, isto é, entender o processo por de trás dos panos, como verificar quais foram as palavras que ele utilizou e a quantidade.

```
# restante do código

for lista in textosQuebrados:
    dicionario.update(lista)

totalDePalavras = len(dicionario)
print(totalDePalavras)
print(dicionario)

Rodando      novamente      o      nosso      arquivo
classificando_emails.py :
```

```
> python3 classificando_emails.py
364
{'', 'seguinte', 'é', 'desconto?', 'moderador?', 'linguagens', '',
programação.', 'cadastrei', 'conhecimento', 'devem', 'travando',
'os', 'javascrip', 'bancária?', 'que', 'parte', 'ferramenta', 'py
qt', 'sobre', 'ele', 'minutos.', 'bom', 'email,', '0', 'curso', '',
renovar', 'maior', 'além', 'tenho', 'transferencia', 'quais', 'al
gum', 'outras', 'pela', 'filho', 'exercícios', 'novo', 'mudar',
'email', 'noite,', 'correção', 'almejando', 'sou', 'passos', 'dia
nte', 'início', 'a', 'melhor', 'indicam', 'cobol,', 'pagamento,', ''
gostaria', 'plus,', 'independência',
...
'com', 'antigo', 'cartão', 'python', 'fórmula', 'conheço', 'cuid
ar', '3', 'usada',
...
'prática', 'possar', 'muito', 'filho,', 'onde', 'web?', 'e', 'e
sses', 'qualquer', 'ótimo', 'dúvida', 'serei', 'faculdade', 'play
er?', 'desde', 'pagar', 'pesquisas', 'ver', 'já', 'deve', 'premiu
m', 'moderador', 'mas', 'comprei', 'em', 'capaz', 'nenhum', 'sent
i'}
# restante da impressão
```

Observe que, dentre essas palavras, existem diversas que fazem todo o sentido para analisarmos o conteúdo de um e-mail, como: "recomendam", "carreira", "preço", "certificado", "trocar", "ferramenta", entre outras que nos permitem compreender a que se referem. Entretanto, ao mesmo tempo em que temos palavras com um grande significado para o nosso negócio, temos também as que não fazem sentido: "com", "o", "uma", "isto", entre outras que utilizamos para construir os textos, mas que não servem como parâmetro para que possamos analisar o conteúdo dos nossos e-mails.

Essas palavras que não precisamos utilizar para avaliarmos o nosso texto são chamadas de palavras de parada, ou tecnicamente, *stop words* ([https://en.wikipedia.org/wiki/Stop\\_words](https://en.wikipedia.org/wiki/Stop_words)).

Não faz sentido termos essas *stop words* dentro de nosso dicionário, pois se analisarmos esse tipo de palavra, provavelmente estaremos enganando o nosso algoritmo. Por motivos de coincidência, ele pode tomar uma decisão que na verdade não faz muito sentido, porque se baseou nas *stop words*. Ou seja, quanto mais focarmos em palavras que fazem sentido para o nosso negócio, que mostrem algum sentimento ou intenção, provavelmente o nosso algoritmo obterá melhores resultados no processo de classificação.

Um outro detalhe muito importante é a questão do desempenho do nosso algoritmo, pois quanto mais dados usarmos, mais lento o algoritmo ficará. Por exemplo, suponhamos que sejam 1 milhão de palavras para analisarmos; quanto tempo será que o nosso algoritmo levaria para processar? Provavelmente bem mais do que com 100 palavras ou menos. Portanto, ao realizarmos um filtro de *stop words*, além de melhorar a precisão do nosso algoritmo, também melhoramos o seu desempenho.

Mas como faríamos isso no nosso código? Leríamos todos os nossos e-mails, anotaríamos todas as palavras que consideramos como *stop words*, e então pediríamos para o nosso dicionário remover uma a uma? Parece um tanto trabalhoso.

Felizmente, alguém já fez isso para nós. Usaremos uma biblioteca que fará esse filtro. Vamos começar criando um novo arquivo chamado `classificando_emails_limpos.py`, dentro do diretório onde estão os nossos arquivos Python. Em seguida, copie todo o código contido no arquivo `classificando_emails.py` e cole nesse arquivo que acabamos de criar.

Faremos essa abordagem para analisar os nossos dois algoritmos, ou seja, verificar as alterações que fizemos no nosso código original após a limpeza. Tendo conhecimento do que será realizado daqui para a frente, podemos iniciar a nossa análise do código. Vejamos um trecho antes de imprimir o nosso dicionário:

```
dicionario = set()  
  
for lista in textosQuebrados:  
    dicionario.update(lista)  
  
print(dicionario)
```

Nesse instante, estamos adicionando palavra por palavra dentro da variável `dicionario`, porém, não impomos nenhuma condição para que a palavra seja adicionada. Então o que devemos fazer? Precisamos adicionar uma condição que diz:

- Adicione essa palavra **apenas** se não fizer parte do conjunto de palavras de paradas, ou seja, stop words.

Para realizarmos essa condição, primeiramente precisamos do conjunto de stop words. Entretanto, atualmente não temos esse conjunto, logo, faremos uso de uma biblioteca de processamento de linguagem do Python, o *National Language Toolkit* (<http://www.nltk.org/>).

Agora, precisamos instalá-lo. Podemos utilizar o `pip` para isso:

```
> pip3 install nltk
```

Lembre-se de que, se você estiver utilizando Linux, precisará da permissão de superusuário, ou seja, o `sudo`.

Com o `nltk` instalado, podemos testá-lo dentro do

interpretador do Python:

```
> python3  
>>> import nltk  
>>>
```

Qual é o nosso próximo passo? Precisamos pedir ao `nltk`, a partir da sua biblioteca, o corpo de textos `nltk.corpus` e as palavras de parada `nltk.corpus.stopwords`. Também queremos que elas sejam da língua portuguesa, logo, `nltk.corpus.stopwords.words("portuguese")`. Vejamos o resultado que ele nos apresenta:

```
>>> nltk.corpus.stopwords.words("portuguese")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    except LookupError: raise e
LookupError:
*****
Resource u'corpora/stopwords' not found. Please use the NLTK
Downloader to obtain the resource: >>> nltk.download()
Searched in:
- '/home/alex-felipe/nltk_data'
- '/usr/share/nltk_data'
- '/usr/local/share/nltk_data'
- '/usr/lib/nltk_data'
- '/usr/local/lib/nltk_data'
*****
>>>
```

Repare que ele nos apresenta esse erro. Por que será que isso aconteceu? Será que não foi instalado corretamente?

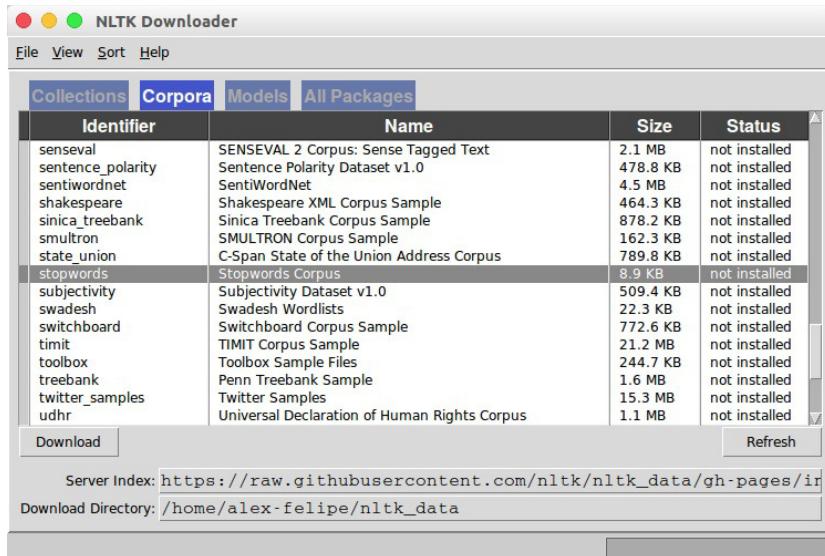
Instalamos normalmente a biblioteca do `nltk`, porém, essa biblioteca dá suporte a diversas linguagens, como inglês, japonês, italiano, espanhol, entre outras. Lembra que a instalação do `nltk` foi bem rápida? Será que, no momento em que foi realizada a

instalação, foram instalados também todos os pacotes que dão suporte para todas as linguagens?

Nesse caso, não! A instalação foi apenas da biblioteca. As demais bibliotecas em que temos interesse, como a de suporte ao português, precisarão ser instaladas à parte. Podemos ver que a própria mensagem nos diz isso, informando que o 'corpora/stopwords' não foi encontrado: '*corpora/stopwords not found*'. Em seguida, ele pede para baixá-lo utilizando a instrução `nltk.download()`. Então faremos isso:

```
>>> nltk.download()
```

Após executar essa instrução, a seguinte janela abrirá:



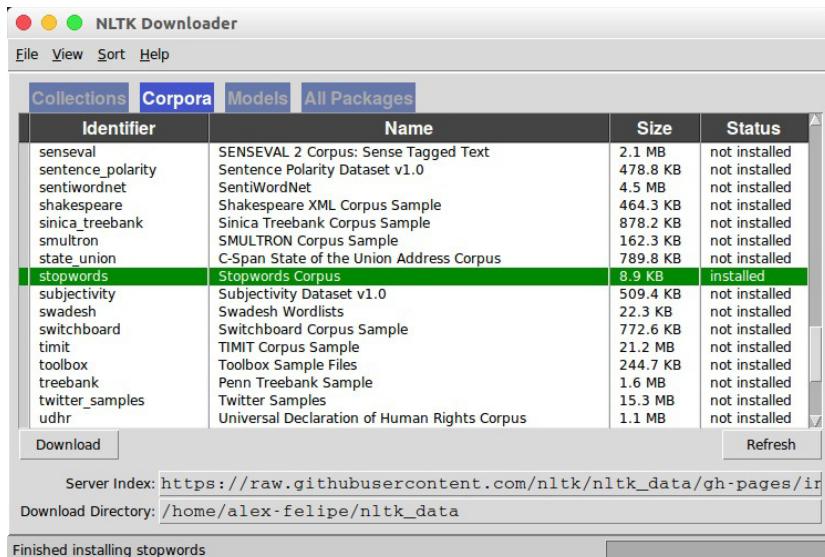
Caso você esteja usando um Mac OSX, é capaz de receber uma mensagem de erro em relação aos certificados SSL de segurança

usados pelo NLTK. Nesse caso, você pode fechar o programa e baixar os certificados com (atualize para a versão do seu Python):

```
/Applications/Python\ 3.6/Install\ Certificates.command
```

Nesse caso de uso de Mac OSX, feche o Python3, abra novamente e entre no `nltk.download()`.

Dentro dessa janela, vá à aba Corpora e procure o pacote stopwords conforme a figura anterior. Após selecionar o pacote, clique em Download. Ao finalizar o download, será apresentada a mensagem installed, indicando que foi instalado:



Agora podemos fechar a janela para download de pacotes e voltarmos ao terminal.

```
>>> nltk.download()
showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-
pages/index.xml
```

True

>>>

Vamos tentar novamente executar o comando que pega todas as stop words da língua portuguesa:

```
>>> nltk.corpus.stopwords.words("portuguese")
['de', 'a', 'o', 'que', 'e', 'do', 'da', 'em', 'um', 'para', 'com',
 'não', 'uma', 'os', 'no', 'se', 'na', 'por', 'mais', 'as', 'dos',
 'como', 'mas', 'ao', 'ele', 'das', 'à', 'seu', 'sua', 'ou', 'quando',
 'muito', 'nos', 'já', 'eu', 'também', 'só', 'pelo', 'pela',
 'até', 'isso', 'ela', 'entre', 'depois', 'sem', 'mesmo', 'aos',
 'seus', 'quem', 'nas', 'me', 'esse', 'eles', 'você', 'essa', 'num',
 'nem', 'suas', 'meu', 'ás', 'minha', 'numa', 'pelos', 'elas',
 'qual', 'nós', 'lhe', 'deles', 'essas', 'esses', 'pelas', 'este',
 'dele', 'tu', 'te', 'vocês', 'vos', 'lhes', 'meus', 'minhas', 'teu',
 'tua', 'teus', 'tuas', 'noso', 'nossa', 'nossos', 'nossas',
 'dela', 'delas', 'esta', 'estes', 'estas', 'aquele', 'aqueila', 'a
 queles', 'aqueelas', 'isto', 'aquilo', 'estou', 'está', 'estamos',
 'estão', 'estive', 'esteve', 'estivemos', 'estiveram', 'estava',
 'estávamos', 'estavam', 'estivera', 'estivéramos', 'esteja', 'est
 ejamos', 'estejam', 'estivesse', 'estivéssemos', 'estivessem', 'e
 stiver', 'estivermos', 'estiverem', 'hei', 'há', 'havemos', 'hão',
 'houve', 'houvemos', 'houveram', 'houvera', 'houvéramos', 'haja',
 'hajamos', 'hajam', 'houvesse', 'houvéssemos', 'houvessem', 'hou
 uver', 'houvermos', 'houverem', 'houverei', 'houverá', 'houveremo
 s', 'houverão', 'houveria', 'houveríamos', 'houveriam', 'sou', 's
 omos', 'são', 'era', 'éramos', 'eram', 'fui', 'foi', 'fomos',
 'foram', 'fora', 'fôramos', 'seja', 'sejamos', 'sejam', 'fosse',
 'fôssemos', 'fossem', 'for', 'formos', 'forem', 'serei', 'será',
 'seremos', 'serão', 'seria', 'seríamos', 'seriam', 'tenho', 'tem',
 'temos', 'tém', 'tinha', 'tínhamos', 'tinham', 'tive', 'teve',
 'tivemos', 'tiveram', 'tivera', 'tivéramos', 'tenha', 'tenhamos',
 't
 enham', 'tivesse', 'tivéssemos', 'tivessem', 'tiver', 'tivermos',
 'tiverem', 'terei', 'terá', 'teremos', 'terão', 'teria', 'teríamo
 s', 'teriam']
```

**NOTA:** as palavras 'houverei', 'houverá', 'houveremos', 'houverão', 'houveria', 'houveríamos', 'houveriam' e "tém" não existem na língua portuguesa, mas estão nesse *corpus*.

O nltk nos apresenta todas as palavras que não possuem intenções ou sentimento para a nossa regra de negócio. Consegue observar o que essas palavras têm em comum? Todas elas aparecem com muita frequência nos textos da língua portuguesa e é justamente por isso que queremos tirá-las durante a análise dos nossos textos.

Basicamente, é essa a definição das stop words, por isso não adicionaremos essas palavras dentro do nosso dicionário. Como faremos isso no nosso código? Vá até o arquivo `classificando_emails_limpos.py` e, antes de criarmos o nosso dicionário, importaremos o nltk e suas stop words da língua portuguesa:

```
# restante do código

import nltk
stopwords = nltk.corpus.stopwords.words('portuguese')

dicionario = set()

for lista in textosQuebrados:
    dicionario.update(lista)

print(dicionario)
```

Qual é o nosso próximo passo? Em vez de adicionarmos toda a lista dentro do dicionário, queremos colocar apenas as palavras

válidas, mas quais são elas? Inicialmente, são todas as contidas na variável `lista` :

```
# restante do código

import nltk
stopwords = nltk.corpus.stopwords.words('portuguese')

dicionario = set()

for lista in textosQuebrados:
    validas = lista
    dicionario.update(validas)

print(dicionario)
```

Por enquanto, não teve nenhuma diferença. Vamos pegar cada palavra contida na variável `lista` :

```
# restante do código

import nltk
stopwords = nltk.corpus.stopwords.words('portuguese')

dicionario = set()

for lista in textosQuebrados:
    validas = [palavra for palavra in lista]
    dicionario.update(validas)

print(dicionario)
```

Repare que estamos pegando cada palavra da lista, porém, ainda não estamos adicionando nenhuma condição para filtrá-las, portanto, precisamos adicioná-la. Podemos adicionar uma palavra como válida, apenas se ela não fazer parte do conjunto das stop words:

```
# restante do código

import nltk
```

```

stopwords = nltk.corpus.stopwords.words('portuguese')

dicionario = set()

for lista in textosQuebrados:
    validas = [palavra for palavra in lista if palavra not in stopwrods]
    dicionario.update(validas)

print(dicionario)

```

Agora estamos adicionando apenas as palavras que não fazem parte das stop words do `nltk`.

Anteriormente, nosso dicionário continha 365 palavras. Vamos verificar quantas palavras ele conterá sem as stop words. Primeiro, saia do interpretador do Python e execute o arquivo `classificando_emails_limpos.py`:

```

> python3 classificando_emails_limpos.py
315
{'', 'prática', 'fórmula', 'cadastrar', 'vídeo', 'paypal', 'quer', 'distanciada', 'base', 'ajuda', 'certificados?', 'pro', 'travando', 'python', 'duvida', 'certificado', 'criarmos', ... 'diante', 'pagamento', 'curso!', 'pagar', 'senti', 'premium', 'deparei', 'designer', 'sobre', 'vontade', 'bancária?', 'javascrip', 'sistemas', 'letra', 'seguinte', 'ainda', 'quero', 'completa.', 'lógica?', 'alguém', 'quanto', 'novamente?', 'acesso', 'aprender', 'certificado?', 'devem', 'terminei', 'paypal', 'mudar', 'interface', 'forma', 'parabéns!', 'ux', 'pf?', 'pouco', 'alura!', 'ótimo', 'vezes', 'quais', 'comprar', 'usada', '3', 'valido', 'vai', 'necessidade', 'trabalhando', 'deseja', 'ganho', 'web?', 'sei', 'plano', 'dependente', 'player?', 'além', 'desconto?', 'navegador.', 'realizados', 'programa', 'trocar', 'computador', 'escolher', 'filho', 'maiúscula', 'trabalhar', 'digitação', 'faço?', 'obs:', 'anos?', 'conheço', 'faculdade', 'porém', 'site', 'virtualização', 'explicação?', 'gráfica?', 'convidado', 'gosta', 'tentar', 'equipe', 'alura?', 'fica', 'depoimento', 'alura', 'existe', 'acessar', 'necessário', 'já', 'optar', 'indicam', 'css', 'possar', 'todo', 'informação', 'mim', 'cobol,'}

```

Ele imprimiu 315 palavras, ou seja, 49 a menos que o nosso dicionário anterior continha. Vale lembrar que todas essas 49 palavras são as stop words, isto é, palavras que em geral não dizem sobre o que realmente o texto se trata, como intenções ou sentimentos.

Conseguimos limpar o nosso dicionário, e agora ele possui diversas palavras valiosas para o nosso negócio. Entretanto, ainda existe um detalhe nas nossas palavras, pois, atualmente, temos as seguintes palavras "distintas": "deve" e "devem". Elas são realmente tão distintas assim? Além destas, temos outros exemplos como "curso" e "cursos". Realmente faz tanta diferença assim?

Não faz muito sentido tratarmos palavras derivadas de outra também. Ou seja, palavras conjugadas no plural, no passado ou qualquer tipo de variação de uma palavra raiz. Tanto a palavra "deve" quanto "devem", "deverão", "deveram" ou "deveria" possuem o mesmo significado!

Um outro exemplo seria a diferença de gêneros, por exemplo, a palavra "aluno" e "aluna" são originadas da mesma palavra, então, não faz sentido tratá-las como distintas. Mas será que, para esse caso, teremos de resolver na mão?

Felizmente, o `nltk` também nos fornece uma biblioteca, uma ferramenta, que extrai a raiz de uma palavra. Temos o *stemmer*, responsável em retirar a palavra raiz de acordo com suas variantes. O nome da biblioteca para o stemmer é `RSLPStemmer` e podemos criá-lo a partir da instrução `nltk.stem.RSLPStemmer()`. Porém, quando tentamos executá-lo pela primeira vez:

```
> python3  
>>> import nltk
```

```
>>> stemmer = nltk.stem.RSLPStemmer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
LookupError:
*****
Resource u'stmmers/rslp/step0.pt' not found. Please use the
NLTK Downloader to obtain the resource: >>> nltk.download()
Searched in:
  - '/home/alex-felipe/nltk_data'
  - '/usr/share/nltk_data'
  - '/usr/local/share/nltk_data'
  - '/usr/lib/nltk_data'
  - '/usr/local/lib/nltk_data'
  - u''
*****
*****
>>>
```

Novamente aparece aquela mensagem informando que não temos o pacote para essa biblioteca, ou seja, precisamos baixá-la. Da mesma forma como fizemos anteriormente, usaremos a instrução `nltk.download()`, porém, dessa vez, com o parâmetro `rslp`, que já baixa o pacote desejado sem a necessidade de procurar no assistente de download:

```
>>> nltk.download('rslp')
[nltk_data] Downloading package rslp to /home/alex-felipe/nltk_da
ta...
[nltk_data]  Unzipping stemmers/rslp.zip.
True
>>>
```

Agora podemos criar o nosso stemmer:

```
>>> stemmer = nltk.stem.RSLPStemmer()
>>>
```

Faremos o nosso primeiro teste. Vamos pedir a raiz da palavra "amigos":

```
>>> stemmer = nltk.stem.RSLPStemmer()
>>> stemmer.stem("amigos")
u'amig'
>>>
```

amig ? Vejamos para a palavra "amigas":

```
>>> stemmer = nltk.stem.RSLPStemmer()
>>> stemmer.stem("amigos")
u'amig'
>>> stemmer.stem("amigas")
u'amig'
>>>
```

Observe que a tanto a palavra amigo(a) como amigos(as) são compostas por "amig", por isso ele nos retornar isso. Vamos fazer mais um teste, agora com a palavra "carreira":

```
>>> stemmer.stem("carreira")
u'carr'
>>>
```

Vejamos agora no plural:

```
>>> stemmer.stem("carreira")
u'carr'
>>> stemmer.stem("carreiras")
u'carr'
>>>
```

Funcionando conforme o esperado, o stemmer retorna uma raiz baseada na palavra que informamos. Entretanto, existe um detalhe importante: em qual momento informamos que queremos que ele extraia a raiz de uma palavra da língua portuguesa? Em nenhum, certo? Então será que o suporte desse *stemmer* será para a língua portuguesa, para a inglesa ou qualquer outra língua por padrão?

Diferentemente do conjunto de stop words que contém

diversas palavras para diferentes línguas, o `RSLPStemmer` é um extrator de raiz das palavras baseado no *removedor de sufixo da Língua Portuguesa*, portanto, essa biblioteca foi desenvolvida para ela. É exatamente por esse motivo que não precisamos informar qual língua estamos usando.

Vamos adicioná-lo ao nosso código, mas em qual parte? Vejamos o momento em que extraímos a palavra da lista de palavras:

```
validas = [palavra for palavra in lista if palavra not in stopwords]
```

Nesse instante, estamos pegando uma palavra da lista de palavras que não estão contidas nas stop words. O que está faltando agora? No momento em que devolvemos a palavra, precisamos dizer que devolveremos o `stem` dela, ou seja, a sua raiz:

```
validas = [stemmer.stem(palavra) for palavra in lista if palavra not in stopwords]
```

Porém, antes de testarmos, precisamos declarar o `stemmer` no nosso código:

```
import nltk
stopwords = nltk.corpus.stopwords.words('portuguese')

stemmer = nltk.stem.RSLPStemmer()

dicionario = set()

for lista in textosQuebrados:
    validas = [stemmer.stem(palavra) for palavra in lista if palavra not in stopwords]
    dicionario.update(validas)

print(dicionario)
```

Vamos organizar o nosso código. Veja que temos o `import` do `nltk` no meio dele. Vamos adicioná-lo na parte superior junto aos demais `imports`:

```
import pandas as pd
from collections import Counter
import numpy as np
from sklearn.model_selection import cross_val_score
import nltk

# restante do código
```

Agora vamos testar o nosso código e verificar o resultado:

```
> python3 classificando_emails_limpos.py
Traceback (most recent call last):
  File "classificando_emails_limpos.py", line 27, in <module>
    validas = [stemmer.stem(palavra) for palavra in lista if pala
vra not in stopwords]
  File "classificando_emails_limpos.py", line 27, in <listcomp>
    validas = [stemmer.stem(palavra) for palavra in lista if pala
vra not in stopwords]
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/pyt
hon3.6/site-packages/nltk/stem/rslp.py", line 103, in stem
    if word[-1] == "s":
IndexError: string index out of range
```

Repare que ele apresentou um erro na linha 27, informando o seguinte: "*IndexError: string index out of range*". Vejamos o código nessa linha:

```
validas = [stemmer.stem(palavra) for palavra in lista if palavra
not in stopwords]
```

O erro se refere a uma posição do array que não foi possível acessar. Que posição é essa? Veja que ele nos mostra o seguinte trecho: "if word[-1] == "s":". Neste ponto, ele tentar pegar a posição -1, isso significa que ele tentou pegar a última posição de uma palavra, porém não deu certo. Mas todas as palavras possuem

a última posição, certo? Se pegarmos o nosso dicionário:

```
('', 'letra', 'email', ..., 'termos'])
```

Todas as palavras possuem a última posição. Entretanto, a primeira palavra é uma palavra vazia! No momento em que ele pega a palavra vazia e tenta pegar a última posição, isto é, seu último caractere, será apresentado o erro. Como podemos resolver esse detalhe? No momento em que extraímos as palavras da lista:

```
validas = [stemmer.stem(palavra) for palavra in lista if palavra  
not in stopwords]
```

Podemos também dizer que queremos apenas as palavras com tamanho maior que 0:

```
validas = [stemmer.stem(palavra) for palavra in lista if palavra  
not in stopwords and len(palavra) > 0]
```

```
Ao      rodarmos      novamente      o      nosso      arquivo  
classificando_emails_limpos.py :
```

```
> python3 classificando_emails_limpos.py  
285  
{'cresc', 'é', 'dep',  
...  
'fórum?', 'dess', 'tod', 'python', 'plan', 'pouc', 'cadastrar,'  
, 'di', 'já,', 'antig', 'filh', 'seguint', 'conheç', 'nom', 'algu  
ém', 'além', '0', 'exist', 'pai', 'conteúdo?', 'trav', 'coloq', '  
mudar', 'algum'}  
Taxa de acerto do OneVsRest: 0.4083333333333333  
Taxa de acerto do OneVsOne: 0.375  
Taxa de acerto do MultinomialNB: 0.4  
Taxa de acerto do AdaBoostClassifier: 0.3733333333333335  
{0.4083333333333333: OneVsRestClassifier(estimator=LinearSVC(C=1  
.0, class_weight=None, dual=True, fit_intercept=True,  
intercept_scaling=1, loss='squared_hinge', max_iter=1000,  
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,  
verbose=0),  
n_jobs=1}, 0.375: OneVsOneClassifier(estimator=LinearSV  
C(C=1.0, class_weight=None, dual=True, fit_intercept=True,
```

```

        intercept_scaling=1, loss='squared_hinge', max_iter=1000,
        multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
        verbose=0),
        n_jobs=1), 0.4000000000000002: MultinomialNB(alpha=1.0
, class_prior=None, fit_prior=True), 0.3733333333333335: AdaBoos
tClassifier(algorithm='SAMME.R', base_estimator=None,
            learning_rate=1.0, n_estimators=50, random_state=0)}
Vencedor:
OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None,
dual=True, fit_intercept=True,
intercept_scaling=1, loss='squared_hinge', max_iter=1000,
multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
verbose=0),
n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real
: 66.66666666666667
Taxa de acerto base: 44.444444
Total de teste: 9

```

O nosso algoritmo funcionou normalmente. Além disso, repare que, em vez de 364 palavras, o nosso dicionário contém 285, ou melhor, 285 raízes de palavras distintas. Também não temos mais a palavra vazia e nem as stop words.

Será que o nosso algoritmo já está 100%, ou ainda existe um detalhe que precisamos verificar? Vamos verificar quantas palavras estão sendo computadas no primeiro vetor de palavras:

```

# restante do código

vetoresDeTexto = [vetorizar_texto(texto, tradutor) for texto in t
extosQuebrados]
marcas = classificacoes['classificacao']

print(vetoresDeTexto[0])

```

Lembra qual é o primeiro texto? Vamos verificar:

Se eu comprar cinco anos antecipados, eu ganho algum desconto?

Rodando o nosso algoritmo, temos o seguinte vetor para esse

texto:

Apenas três palavras foram computadas. Será que temos apenas três dentro dessa frase? Consultando no nosso dicionário, temos os seguintes resultados para cada uma das palavras:

- **Comprar:** 'compr'
  - **Cinco:** 'cinc'
  - **Anos?:** 'anos?'
  - **Antecipados,:** 'antecipados, '
  - **Ganho:** 'ganh'
  - **Algum:** 'algum'
  - **Desconto:** 'desconto?'

Temos pelo menos sete palavras contidas no nosso dicionário, porém, apenas três delas estão sendo computadas no momento em que a vetorizamos, isto é, pegamos palavra por palavra de um texto e convertemos para um array numérico que computa a quantidade de repetições. O que será que aconteceu?

Provavelmente esse problema deve-se ao fato de estarmos adicionando a raiz de uma palavra ao nosso dicionário. Mas o

quão impactante isso pode ser? Se analisarmos os resultados de cada algoritmo nesse último teste que fizemos, a taxa de acerto de ambos foi reduzida, portanto, o simples fato de vetorializar um texto sem computar todas as suas palavras que fazem sentido piora, e muito, o nosso algoritmo.

Nesse instante, precisamos verificar em qual ponto do código precisamos realizar uma análise para entender o motivo pelo qual aconteceu esse problema. Começaremos analisando a partir do trecho do código em que criamos o dicionário:

```
# restante do código

dicionario = set()

for lista in textosQuebrados:
    validas = [stemmer.stem(palavra) for palavra in lista if palavra not in stopwords and len(palavra) > 0]
    dicionario.update(validas)

print(dicionario)
```

Até esse ponto do código, realizamos os ajustes necessários e verificamos que eles não têm problema algum. Vamos para o próximo:

```
totalDePalavras = len(dicionario)
tuplas = zip(dicionario, range(totalDePalavras))
tradutor = {palavra:indice for palavra, indice in tuplas}
print(totalDePalavras)
```

Nesse trecho também não há problema, pois é simplesmente pegar todas as palavras e transformar em um tradutor, ou seja, associar um número distinto para cada uma delas. Vejamos o próximo:

```
def vetorizar_texto(texto, tradutor):
    vetor = [0] * len(tradutor)
```

```
for palavra in texto:  
    if palavra in tradutor:  
        posicao = tradutor[palavra]  
        vetor[posicao] += 1  
  
return vetor
```

Nessa função, realizamos de fato o processo de vetorializar um texto. Provavelmente existe algum detalhe dentro dessa função que precisamos ajustar. Vamos analisar passo a passo o que está acontecendo:

```
vetor = [0] * len(tradutor)
```

Nesse trecho, não há problema algum, já que só estamos pegando o total de palavras que o tradutor contém para que possamos criar o vetor. Vejamos a próxima linha:

```
for palavra in texto:  
    if palavra in tradutor:
```

Observe que estamos criando um laço para iterar sobre todas as palavras contidas no texto que enviamos. Por enquanto, não temos problemas. Entretanto, logo abaixo, verificamos se a palavra contida no texto está contida também no tradutor, mas, atualmente, o nosso tradutor contém apenas a raiz da palavra.

Faz sentido verificar se existe uma determinada palavra em um tradutor que contém apenas as raízes de todas as palavras que conhecemos? Por exemplo, no texto, temos a palavra "cinco", porém, no nosso tradutor, para a palavra "cinco", temos a raiz "cinc". Agora vem a questão: "cinco" é a mesma coisa que "cinc"?

De fato, são distintas! Quando realizamos uma comparação entre duas strings, ambas precisam ser **estritamente** iguais! Nesse caso, temos um tradutor com as raízes das palavras, portanto,

quando tivermos de verificar se uma palavra existe no tradutor, primeiro precisamos transformá-la em sua raiz para depois compará-la.

Vamos realizar essas modificações no código. Antes de adicionarmos esses detalhes que vimos, precisamos de um stemmer na função `vetorizar_texto`, portanto, adicione o stemmer como parâmetro:

```
def vetorizar_texto(texto, tradutor, stemmer):
```

Qual é o nosso próximo passo? É justamente pegar o stem da palavra para verificar se existe no tradutor:

```
def vetorizar_texto(texto, tradutor, stemmer):  
    vetor = [0] * len(tradutor)  
    for palavra in texto:  
        raiz = stemmer.stem(palavra)  
        if raiz in tradutor:
```

Será que não precisamos mexer no restante do código?  
Vejamos:

```
def vetorizar_texto(texto, tradutor, stemmer):  
    vetor = [0] * len(tradutor)  
    for palavra in texto:  
        raiz = stemmer.stem(palavra)  
        if raiz in tradutor:  
            posicao = tradutor.index(palavra)  
            vetor[posicao] += 1  
  
    return vetor
```

Observe que ainda estamos pedindo a palavra inteira para o tradutor, sendo que, agora, precisamos pedir as suas raízes. Em outras palavras, em vez de pedir a posição a partir de uma palavra completa, precisamos pedir para sua raiz:

```
def vetorizar_texto(texto, tradutor, stemmer):
```

```

vetor = [0] * len(tradutor)
for palavra in texto:
    raiz = stemmer.stem(palavra)
    if raiz in tradutor:
        posicao = tradutor[raiz]
        vetor[posicao] += 1

return vetor

```

Lembre-se de invocar o `vetorizar_texto` passando o `stemmer`:

```

vetoresDeTexto = [vetorizar_texto(texto, tradutor, stemmer) for t
exto in textosQuebrados]

```

Aparentemente, fizemos todos os ajustes necessários. Vamos testar novamente o nosso arquivo `classificando_emails_limpos.py`:

```

> python3 classificando_emails_limpos.py
...
Traceback (most recent call last):
  File "classificando_emails_limpos.py", line 36, in <module>
    vetoresDeTexto = [vetorizar_texto(texto, tradutor, stemmer) f
or texto in textosQuebrados]
  File "classificando_emails_limpos.py", line 36, in <listcomp>
    vetoresDeTexto = [vetorizar_texto(texto, tradutor, stemmer) f
or texto in textosQuebrados]
  File "classificando_emails_limpos.py", line 29, in vetorizar_te
xto
    raiz = stemmer.stem(palavra)
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/pyt
hon3.6/site-packages/nltk/stem/rslp.py", line 103, in stem
    if word[-1] == "s":
IndexError: string index out of range

```

Novamente aquele problema de acessar o último caractere. O que será que aconteceu? Note que, quando pegamos uma palavra:

```

def vetorizar_texto(texto, tradutor, stemmer):
    vetor = [0] * len(tradutor)
    for palavra in texto:

```

```
raiz = stemmer.stem(palavra)  
#restante do código
```

Não filtramos se ela está vazia ou não, portanto, precisamos adicionar mais uma condição que é justamente fazer todos os passos de pegar a raiz da palavra, sua posição e somar **somente** se a palavra contida no texto tiver tamanho maior do que 0:

```
def vetorizar_texto(texto, tradutor, stemmer):
    vetor = [0] * len(tradutor)
    for palavra in texto:
        if len(palavra) > 0:
            raiz = stemmer.stem(palavra)
            if raiz in tradutor:
                posicao = tradutor[raiz]
                vetor[posicao] += 1

    return vetor
```

Testando novamente o nosso código, temos o seguinte resultado:

Agora foram contabilizadas as sete palavras da frase que havíamos analisado. Agora estamos realizando todos os passos da

forma adequada, isto é, dado um conjunto de palavras, extraímos suas raízes para diminuir uma quantidade considerável de palavras a serem analisadas. Isso melhora tanto a performance do nosso algoritmo como também a sua precisão, pois ele terá menos variáveis para serem analisadas como, nesse caso, stop words ou palavras compostas que não fazem tanta diferença para a tomada de decisão do nosso algoritmo.

Há mais um detalhe que precisamos analisar em nosso dicionário, pois ainda existem palavras como:

- **eu,**
- **minutos.**
- **empresa?**

Note que cada uma dessas palavras está concatenada com algum tipo de pontuação. Ou seja, a palavra "eu" tem uma vírgula, a "minutos" tem um ponto final e a "empresa", um ponto de interrogação. O stemmer não é capaz de retirar essas pontuações nos nossos textos, portanto, se existem palavras como "eu," ou "eu.", serão consideradas como distintas.

Em nenhum momento trabalhamos com pontuação. O que precisamos fazer para retirar esses pontos durante a separação das palavras?

Nesse caso, precisamos verificar como estamos quebrando as palavras, ou seja, qual é o critério que estamos utilizando para separar palavra por palavra dentro de um texto. Vejamos como estamos fazendo atualmente:

```
classificacoes = pd.read_csv('emails.csv')
textosPuros = classificacoes['email']
textosQuebrados = textosPuros.str.lower().str.split(' ')
```

```
# restante do código
```

Estamos separando cada palavra no momento em que encontramos um espaço. Ou seja, em vez de apenas quebrar com espaços em branco, precisamos quebrar também com a pontuação. Da mesma forma como fizemos com as stop words e as raízes, usaremos uma biblioteca que é capaz de separar palavras de um texto, tanto pelos espaços em branco quanto pela pontuação.

Essa biblioteca se chama `tokenize` e, da mesma forma como as outras, ela faz parte do conjunto de bibliotecas do `nltk` e precisa ser baixada. Usaremos a instrução `nltk.download("punkt")`, que é justamente o módulo para pontuação:

```
>>> import nltk
>>> nltk.download("punkt")
[nltk_data] Downloading package punkt to /home/alex-
[nltk_data]     felipe/nltk_data...
[nltk_data]     Unzipping tokenizers/punkt.zip.
True
>>>
```

Antes de adicionarmos ao nosso código, vamos realizar um teste. A partir dessa biblioteca, pediremos para separar palavra por palavra da frase:

- "Voce vai viajar? Este ano, eu penso que sim! Claro!?Mas e voce? Sim!".

Essa frase é apenas um teste, portanto, estamos utilizando espaços em branco, pontuação com espaço em branco e pontuação junto com palavras em ambos os lados. Para testar, vamos utilizar a seguinte instrução:

```
>>> nltk.tokenize.word_tokenize("Voce vai viajar? Este ano, eu pe  
nso que sim! Claro!Mas e voce? Sim!")  
['Voce', 'vai', 'viajar', '?', 'Este', 'ano', ',', 'eu', 'penso',  
'que', 'sim', '!', 'Claro', '!', '?', 'Mas', 'e', 'voce', '?', 'S  
im', '!']  
>>>
```

Observe que a função `word_tokenize` foi capaz de quebrar todas as palavras por meio da pontuação. Porém, diferentemente do stemmer, o `tokenize` não é exclusivo da língua portuguesa, portanto, ele pode variar de acordo com a linguagem.

Agora, vamos adicionar o `tokenize` no nosso código. Mas, em qual momento precisamos adicioná-lo? O nosso primeiro passo é justamente separar frase por frase, logo, vamos extrair cada frase para uma variável chamada `frases` antes mesmo de realizarmos a separação de palavras:

```
classificacoes = pd.read_csv('emails.csv')  
textosPuros = classificacoes['email']  
frases = textosPuros.str.lower()  
  
# restante do código
```

Em seguida, para cada frase contida na variável `frases` :

```
for frase in frases
```

Precisamos devolver a `frase` processada pelo `tokenize` :

```
nltk.tokenize.word_tokenize(frase) for frase in frases
```

Transformaremos esse resultado em um array e devolvemos para a variável `textosQuebrados` :

```
classificacoes = pd.read_csv('emails.csv')  
textosPuros = classificacoes['email']  
frases = textosPuros.str.lower()  
textosQuebrados = [nltk.tokenize.word_tokenize(frase) for frase in  
frases]
```

```
# restante do código
```

Vamos testar o nosso código e verificar o resultado:

```
> python3 classificando_emails_limpos.py  
244  
...
```

Agora o nosso dicionário contém 244 palavras. Entretanto, quando observarmos algumas, temos os seguintes resultados:

- '.'
- '::'

Faz sentido tratarmos esses pontos como uma palavra? Com certeza, não! Portanto, vamos descartar também essas pontuações. Como fazemos isso?

Durante todo esse processo de separar cada palavra para adicionar ao nosso dicionário, é muito comum desconsiderarmos palavras com dois ou menos caracteres. Se uma palavra tiver apenas duas letras ou menos, podemos descartá-la.

Dessa forma, além de eliminar palavras que não fazem tanta diferença durante a classificação, resolvemos o problema das pontuações. Vamos alterar o nosso código no momento em que estamos inserindo as palavras no nosso dicionário:

```
# restante do código  
  
dicionario = set()  
  
for lista in textosQuebrados:  
    validas = [stemmer.stem(palavra) for palavra in lista if pala  
    vra not in stopwords and len(palavra) > 0]  
    dicionario.update(validas)
```

```
print(dicionario)
```

Em vez de adicionar palavras com a quantidade de caracteres maior que 0, pediremos para adicionar palavras com pelo menos três caracteres, ou seja, maior do que dois:

```
dicionario = set()  
  
for lista in textosQuebrados:  
    validas = [stemmer.stem(palavra) for palavra in lista if pala  
vra not in stopwords and len(palavra) > 2]  
    dicionario.update(validas)  
  
print(dicionario)
```

Testando novamente o nosso arquivo  
classificando\_emails\_limpos.py :

```
> python3 classificando_emails_limpos.py  
230  
{'mud', 'almej', 'inter', 'prát', 'seman', 'depend', 'program', 'comput', 'identifiq', 'agradeç', 'aul', 'design', 'letr', 'por', 'djang', 'parabém', 'nom', 'curs', 'cuid', 'trav', 'qual', 'sent', 'aind', 'tent', 'vou', 'cust', 'pouc', 'porém', 'vai', 'vers', 'nenhum', 'ond', 'pra', 'expir', 'inform', 'possu', 'indic', 'io', 'trilh', 'err', 'mim', 'minut', 'ajud', 'proxim', 'pal', 'quant', 'ver', 'imprim', 'maiúscul', 'distanc', 'real', 'favor', 'dess', 'seguint', 'part', 'aprend', 'recomend', 'duvid', 'ont', 'fórmu l', 'cs', 'poss', 'inic', 'desej', 'via', 'renov', 'qualqu', 'us', 'ótim', 'framework', 'control', 'faz', 'vontad', 'sistem', 'grá f', 'aplic', 'dúv', 'transferenc', 'filh', 'olá', 'respost', 'noi t', 'plataform', 'html', 'dificuldad', 'víde', 'boa', 'play', 'cri', 'pro', 'alguém', 'ganh', 'quer', 'sab', 'fórum', 'necess', 'parec', 'comunidad', 'carr', 'web', 'envi', 'compr', 'termin', 'py thon', 'algum', 'avis', 'pass', 'sobr', 'dep', 'dev', 'realiz', 'escolh', 'window', 'ambi', 'interfac', 'bas', 'moder', 'antecip', 'pesquis', 'excel', 'vez', 'torn', 'cont', 'pert', 'além', 'ach', 'aqu', 'ano', 'encontr', 'bancár', 'entend', 'gost', 'pod', 'cada str', 'pai', 'começ', 'segund', 'tod', 'laravel', 'naveg', 'absor v', 'interess', 'premiun', 'opt', 'payp', 'jav', 'dia', 'descont', 'pret', 'plan', 'post', 'nov', 'pyqt', 'virtu', 'cancel', 'depo', 'desejo-t', 'trabalh', 'plu', 'hav', 'outr', 'faç', 'desd', 'digit', 'ob', 'qu', 'conheç', 'sit', 'lid', 'pag', 'própri', 'premi
```

```
um', 'capaz', 'exercíci', 'coloq', 'brasil', 'ferrament', 'email'
, 'mai', 'poi', 'sum', 'di', 'acess', 'lingu', 'cobol', 'javascri
p', 'estud', 'muit', 'conteúd', 'faculdad', 'googl', 'convid', 'v
ist', 'antig', 'use', 'empr', 'cresc', 'emit', 'efetu', 'cinc', '
alur', 'explic', 'caminh', 'melhor', 'confer', 'troc', 'sei', 'fi
c', 'bom', 'exist', 'val', 'correç', 'complet', 'mac', 'colun', '
lógic', 'independ', 'antecipad', 'certific', 'form', 'equip', 'te
rm', 'ser', 'moip', 'inici', 'cart', 'desenvolv', 'marketing', 'á
re', 'conhec'}
```

Agora não temos mais pontuações ou palavras com menos de três caracteres no nosso dicionário. Além disso, agora ele contém 230 palavras distintas! O nosso código final fica da seguinte forma:

```
import pandas as pd
from collections import Counter
import numpy as np
from sklearn.model_selection import cross_val_score
import nltk

classificacoes = pd.read_csv('emails.csv')
textosPuros = classificacoes['email']
frases = textosPuros.str.lower()
textosQuebrados = [nltk.tokenize.word_tokenize(frase) for frase in
frases]

stopwords = nltk.corpus.stopwords.words('portuguese')
stemmer = nltk.stem.RSLPStemmer()

dicionario = set()

for lista in textosQuebrados:
    validas = [stemmer.stem(palavra) for palavra in lista if palavra
not in stopwords and len(palavra) > 2]
    dicionario.update(validas)

totalDePalavras = len(dicionario)
print(totalDePalavras)
print(dicionario)
tuplas = zip(dicionario, range(totalDePalavras))
tradutor = {palavra:indice for palavra, indice in tuplas}

def vetorizar_texto(texto, tradutor, stemmer):
```

```

vetor = [0] * len(tradutor)
for palavra in texto:
    if len(palavra) > 0:
        raiz = stemmer.stem(palavra)
        if raiz in tradutor:
            posicao = tradutor[raiz]
            vetor[posicao] += 1

return vetor

vetoresDeTexto = [vetorizar_texto(texto, tradutor, stemmer) for t
exto in textosQuebrados]
marcas = classificacoes['classificacao']

print(vetoresDeTexto[0])

X = vetoresDeTexto
Y = marcas

porcentagem_de_treino = 0.8

tamanho_de_treino = int(porcentagem_de_treino * len(Y))
tamanho_de_validacao = len(Y) - tamanho_de_treino

treino_dados = X[0:tamanho_de_treino]
treino_marcacoes = Y[0:tamanho_de_treino]

validacao_dados = X[tamanho_de_treino:]
validacao_marcacoes = Y[tamanho_de_treino:]

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):

    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes
    , cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print(msg)
    return taxa_de_acerto

def teste_real(modelo, validacao_dados, validacao_marcacoes):
    resultado = modelo.predict(validacao_dados)

    acertos = resultado == validacao_marcacoes

```

```

total_de_acertos = sum(acertos)
total_de_elementos = len(validacao_marcacoes)

taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

msg = "Taxa de acerto do vencedor entre os dois algoritmos no m  
undo real: {}".format(taxa_de_acerto)
print(msg)

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest
, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, t
reino_dados, treino_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMul
tinomial, treino_dados, treino_marcacoes)
resultados[resultadoMultinomial] = modeloMultinomial

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier(random_state=0)
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloA
daBoost, treino_dados, treino_marcacoes)
resultados[resultadoAdaBoost] = modeloAdaBoost

print(resultados)

maximo = max(resultados)
vencedor = resultados[maximo]

print("Vencedor: ")
print(vencedor)

```

```
vencedor.fit(treino_dados, treino_marcacoes)

teste_real(vencedor, validacao_dados, validacao_marcacoes)

acerto_base = max(Counter(validacao_marcacoes).values())
taxa_de_acerto_base = 100.0 * acerto_base / len(validacao_marcacoes)
print("Taxa de acerto base: %f" % taxa_de_acerto_base)

total_de_elementos = len(validacao_dados)
print("Total de teste: %d" % total_de_elementos)
```

## RESUMINDO

Neste capítulo, o nosso foco foi justamente na limpeza dos dados que estávamos usando para o nosso algoritmo. Inicialmente, vimos que muitas palavras (como "com", "o", "uma", "isto" etc.) são consideradas para a construção de um texto da língua portuguesa, porém, não faz sentido considerá-las para classificação.

Aprendemos que essas palavras são consideradas como stop words e, justamente por não fazerem diferença durante a nossa classificação de texto, as desconsideramos. Para isso, utilizamos o pacote de bibliotecas do `nltk`, nesse caso, o módulo `corpus` para a língua portuguesa, a partir da seguinte instrução:

```
nltk.corpus.stopwords.words("portuguese")
```

Mesmo retirando as palavras de paradas (stop words), notamos que não era o suficiente, pois ainda existiam palavras no nosso dicionário que estavam sendo consideradas "distintas", mas que continham o mesmo significado, por exemplo:

- "trocar" ou "troca".
- "curso" ou "cursos".

Vimos que cada uma dessas palavras contém uma raiz, isto é, uma palavra da qual foram originadas. E para o nosso algoritmo, o importante seria lidar apenas com as raízes das palavras, justamente para reduzir a quantidade das que ele precisaria lidar durante a vetorização de textos, melhorando consideravelmente sua performance e resultado.

Da mesma forma que fizemos com as stop words, usamos o stemmer, uma biblioteca do `nltk` que nos permitiu adicionar apenas a raiz da palavra dentro do dicionário. Assim, foi reduzido consideravelmente a quantidade de palavras que tínhamos no nosso dicionário.

Por fim, vimos que, mesmo realizando o filtro de stop words e a raiz das palavras, ainda contínhamos palavras como:

- 'eu,'
- **minutos.**
- **empresa?**

São palavras concatenadas com pontuações. Para essa situação, verificamos que o problema estava justamente no momento em que separamos cada palavra do texto, ou seja, utilizamos apenas o critério de espaços vazios. Para resolvermos esse problema de separar as palavras tanto por espaços em brancos quanto para pontuações, utilizamos mais um módulo do `nltk`, o `tokenize`.

Além disso, consideramos apenas palavras com mais de dois caracteres, pois é bem comum desconsiderarmos essas palavras que não darão tanto significado durante a classificação. Dessa forma, também resolvemos o problema que adicionava as pontuações no nosso dicionário.

Todo esse processo de limpeza de nossos textos e dicionários é fundamental quando queremos analisar textos. Por isso mesmo que a biblioteca `nltk` é muito usada quando temos esse tipo de trabalho.

# CONCLUSÃO

## 12.1 O CAMINHO

Começamos questionando como funciona o pensamento humano na hora de distinguir categorias de animais diferentes. Abstraímos esse nosso pensamento como sendo a detecção da existência ou não de determinadas características. Como só existiam dois valores possíveis para cada característica, chamamos de variáveis binárias, com valor 0 ou 1.

Diversos matemáticos e cientistas desenvolveram algoritmos para, dado um array  $\mathbf{x}$  de valores binários, calcular  $\mathbf{y}$  binário de forma a moldar essa função aos valores que damos ao algoritmo. Com isso, tivemos nosso primeiro contato com o aprendizado matemático: um algoritmo que aprende qual a função  $f$ , de acordo com os dados que são fornecidos a ela.

Passamos para a utilização de diversos desses algoritmos. Vimos a importância de treiná-los e testá-los em dados distintos. Falamos sobre tipos diferentes de testes, incluindo o k-fold. Vimos o quanto importante é ficarmos cientes de quantos testes diferentes estamos fazendo, evitando viciar o resultado e a nossa decisão final.

Mais adiante, vimos como trabalhar com textos, limpando e

padronizando-os em dicionários de raízes de palavras simples. Estes poderiam ser transformados em arrays binários, novamente permitindo a aplicação de nossos algoritmos.

Vimos como trabalhar com uma classificação de múltiplas categorias, com o nosso resultado não sendo simplesmente binário.

### ARQUIVOS

Relembrando de que você pode acessar todo o código e arquivos deste livro no nosso GitHub:

<https://github.com/guilhermesilveira/machine-learning>

## 12.2 COMO CONTINUAR OS ESTUDOS

Existem diversas maneiras de aprofundar ou continuar nossos estudos. Independente do caminho que escolha, sugiro praticar. Bastante!

### Teoria e prática: matemática

Se você gostou da introdução a Bayes, vai adorar ler mais sobre como os algoritmos de machine learning funcionam por trás. Os livros que focam nessa parte costumam mostrar uma abordagem pesada em abstração matemática, por isso não recomendo para quem não tem uma vivência mais profunda com abstração e formalismo. Por outro lado, a própria Wikipédia possui explicações razoáveis e, por vezes, simplificadas de algoritmos

como o Naive Bayes.

## Outras linguagens de programação

Existem implementações de algoritmos de machine learning em diversas linguagens. As mais populares são as de R, Octave e Python. Dependendo da área de estudo, existem bibliotecas muito boas em outras linguagens também, como Java.

Aqui não existe segredo: temos de procurar por bibliotecas na linguagem que pretendemos usar. Por outro lado, em linguagens que não foram mencionadas anteriormente, não é raro que bibliotecas sejam abandonadas com o passar do tempo, então evite-as. Foque nas linguagens que o mercado usa e que possuem maior suporte e comunidade ativa.

Mas como todo algoritmo de machine learning costuma ser intensivo no uso de matemática, a otimização desses cálculos acaba sendo vital, e as três linguagens citadas anteriormente são famosas por tal trabalho.

Não existe segredo para a busca de bibliotecas nessas linguagens. Se você procurar no google por "R naive bayes", vai encontrar implementações de classificação que o usam. Se procurar por "R classification algorithm", também encontrará. Melhor do que um ou outro link pontual é entender que basta usar uma das linguagens *mainstream* de machine learning junto com o tipo de classificação ou algoritmo que quer trabalhar.

## Teoria e prática: algoritmos

Utilize os dados que você tem acesso em sua empresa, escola ou que estão disponíveis na internet para trabalhar com os

algoritmos que vimos até aqui. Faça perguntas aos seus dados e tente responder com os algoritmos. Lembre-se: quanto mais testes fizer, maior a chance de encontrar alguma conclusão por sorte ou azar. Anote os testes com carinho, inclusive os que falharam na sua busca por respostas.

O livro *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*, dos autores Aoife D'Arcy, John D. Kelleher e Brian Mac Namee, é um bom próximo passo, mostrando casos de uso dos mais diversos algoritmos.

## Teoria e prática: negócios

Converse com os clientes e os usuários de seu software e produto, e tente entender os problemas que eles passam no dia a dia. Quais perguntas poderiam ser respondidas com acesso a dados e que facilitaria o trabalho deles? A análise de dados pode ser feita de diversas maneiras, e machine learning é uma delas.

Finalizo com a recomendação de um livro que mostra aplicações nas mais diversas áreas, o *The elements of statistical learning: Data Mining, Inference, and Prediction*, de Trevor Hastie, Robert Tibshirani e Jerome H. Friedman. Se você gosta de teoria é sempre interessante manter um olho em como ela pode ser aplicada. Se você gosta de aplicação, é importante ver exemplos dela. Este livro serve de próximo passo nos dois casos.