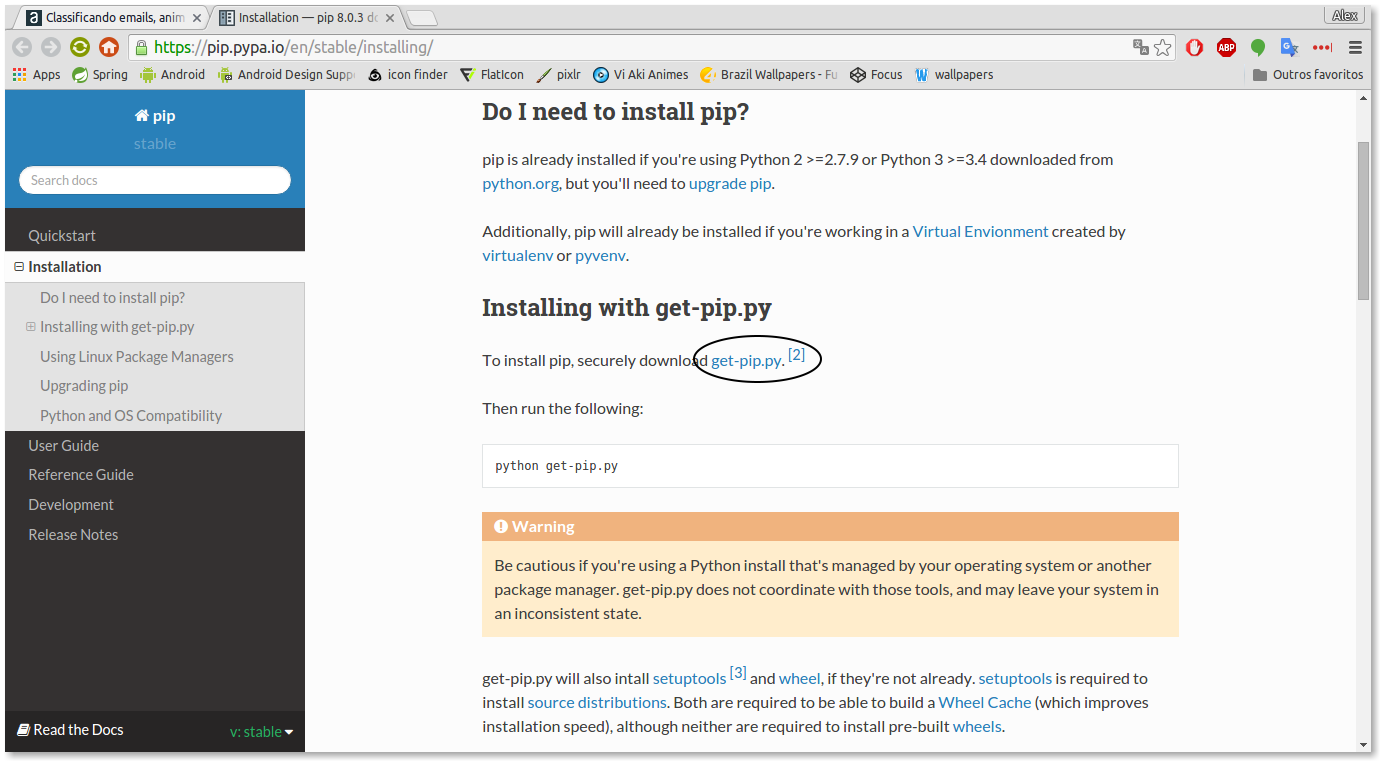
***1 - Classificando e-mails, animais e muito mais...***

Introdução ao código.

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, se é porco ou se é cachorro, eu quero classificá-las. Pra isso usaremos *Python* e instalaremos nele o *pip*, que é o instalador simples de pacotes. Vá até a “https://pip.pypa.io/en/stable/installing/” do pip e clique em get-pip.py:



Ele vai abrir o arquivo na página atual, salve em um diretório de fácil acesso via terminal. A seguir, no terminal, entre no diretório que você salvou o arquivo e digite o comando:

> python get-pip.py

Caso você esteja no linux será necessário utilizar o sudo para permissão de super usuário.

Pronto! O pip foi instalado. Na própria página de instalação existem outras alternativas de instalação, como por exemplo, no Ubuntu, podemos utilizar um package manager como o apt-get. Utilizamos o método demonstrado, pois funciona em qualquer sistema operacional!

Agora que nós temos o pip, podemos instalar as bibliotecas do python de nosso interesse para trabalhar com as informações científicas que queremos, como por exemplo, o machine learning e a classificação. A biblioteca que usaremos é o scikit-learn. Então vamos pedir para o pip instalar para nós por meio do comando:

> pip install scikit-learn

Da mesma forma que aconteceu na instalação do *pip*, caso esteja no Linux, será necessário utilizar o sudo.

Se você estiver utilizando Python3 pode ser necessário instalar outras duas bibliotecas. A NumPy e a SciPy

Para instalá-las, podemos utilizar o pip, só que como estamos utilizando o Python3, temos que utilizar a versão 3 do pip (pip3)

pip3 install numpy scipy

Bacana, instalei a biblioteca do python, o scikit-learn. Além da biblioteca, nós temos acesso à [documentação](http://scikit-learn.org/stable/documentation.html) para consultarmos tudo o que ela disponibiliza para nós. O que queremos fazer, a princípio, é a classificação entre diversos porcos e cachorros, depois iremos fazer do nosso mundo web. Então, abriremos um editor de texto para escrevermos o nosso código python. Utilize um de sua preferência.

Primeiro vamos criar um arquivo chamado classificao.py e vamos salvá-lo, lembre-se de salvar o arquivo dentro de um diretório de fácil acesso ao terminal. E nesse arquivo iremos 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 tem uma característica. Mas, e quais são as 3 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, ele é gordinho:

porco1 = [1]

Ele tem perninha curta? Sim, ele tem perninha curta:

porco1 = [1, 1]

Esse porquinho faz *auau*? Não, ele não faz *auau*:

porco1 = [1, 1, 0]

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

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

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

porco1 = [1, 1, 0]

porco2 = [1, 1, 0]

Por fim eu 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 3 porquinhos, agora eu preciso cadastrar 3 cachorros que eu já conheço. O primeiro cachorro ele é gordinho, ele tem perninha curta e ele faz auau:

porco1 = [1, 1, 0]

porco2 = [1, 1, 0]

porco3 = [1, 1, 0]

cachorro1 = [1, 1, 1]

Os outros 2 cachorros não são gordinhos, tem 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 6 elementos que eu, Guilherme, conheço. Então vamos comentar o 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 6 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 o que eles são, ou seja, dizer se eles são 1 ou 0, se são 1 ou -1, se é cachorro ou porco... Eu posso escolher o que eu quiser. 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 são, usaremos então uma variável chamada marcacoespara indicar o que cada um representa:

marcacoes

Sabemos que do primeiro ao terceiro, são porcos e eu vou marcar com 1, tudo bem?

marcacoes = [1, 1, 1]

E para marca como cachorro eu vou querer marcar com um outro número, eu posso marcar com 0, -1, 10000... Eu posso marcar com o que eu quiser. Para dar ênfase na distinção dos 2 elementos, ou seja, dizer que um é oposto do outro, eu irei usar 1 positivo(1) para porcos 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 os nossos elementos, porém, nós temos agora um elemento misterioso, quem é esse misterioso? Bom, eu tenho um colega 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 esse elemento misterioso é um cachorro ou é um porco?". Então nós queremos treinar um algoritmo de classificação baseado nesses dados para que, baseado nesses dados, ele me diga se é um cachorro ou é um porco. Vamos lá?

O que precisamos fazer agora é treinar um algoritmo baseado nesses dados. Então nós precisamos importar da biblioteca do sklearnuma parte dela que faz o treinamento baseado no algoritmo que se chama *bayesiano*, chamado naive\_bayes. A partir dessa biblioteca iremos 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 nosso 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:

> python classificacao.py

>

Não aconteceu nada? Claro que não aconteceu nada! Apenas criamos o modelo! Nem mandamos ele treinar... Como nós, como ser humanos, treinamos? Olhamos para um elemento e classificamos como porquinho, olhamos para outro elemento e classificamos como cachorrinho, isto é, baseado nos dados e 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 é 1 os dados do cachorro4 ao cachorro6 é -1. É dessa maneira que precisamos **adequar** o nosso modelo, e para isso usaremos o método fit:

modelo.fit(dados, marcacoes)

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

> python classificacao.py

>

Não aconteceu nada... Nesse momento 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:

> python classificacao.py

>/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:386: DeprecationWarning: Passing 1d arrays as data is deprecated 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 veremos como resolver esse *warning*. E o que significa o -1 pra 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 único caso? Não! Nós queremos prever muitos outros casos! Então precisamos de outros elementos misteriosos para que o programa classifique para nós, como por exemplo o misterioso2 que é gordinho, não tem perninha curta e não faz *auau*:

misterioso1 = [1, 1, 1]

misterioso2 = [1, 0, 0]

Bom, ele não é gordinho e não tem perna curta, porém ele não faz *auau*, eu imagino que seja um porco, pois todos que não fazem *auau* são porquinhos. Então agora, em vez de prever o misterioso1, pediremos para que ele preveja o misterioso2:

print(modelo.predict(misterioso2))

Se rodarmos novamente:

> python classificacao.py

> /usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:386: DeprecationWarning: Passing 1d arrays as data is deprecated 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]

Na interpretação dele, a maior chance que ele dá é que esse misterioso2 seja um porquinho, bom, se observarmos o misterioso2 vemos que ele é gordinho, não tem perna longa e não faz *auau*... Eu chutaria que é um porquinho também! 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))

Repara que o parâmetro do método predictrecebe 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*, pode ser que em versões futuras ele nem funcione! Então, mesmo que for um *array* de uma única posição, precisamos passar um array! Vamos testar novamente?

> python 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 Multinominal para classificação?

* Treinamos o nosso modelo pedindo para se adequar aos dados e marcações utilizando o método fit.
* Preveja pra mim o que eles são por meio do método predict.

E então ele vai prevendo se esses elementos são porcos ou cachorros. Mas agora eu estou pensando, será que esse código que criamos funciona apenas para porco ou cachorro? E se, ao invés 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 traduzimos isso pra código, extremamente simples! E quais são os passos mesmo?

* Para cada um dos elementos a gente define 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 diferenciar um do outro, 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 eu usei -1 e 1. Eu optei por essa escolha, pois dessa 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:

> python classificacao.py

> [0 1]

Mas vamos deixar -1 e 1, justamente pra dar essa ênfase de um ser o oposto do outro no momento em que eu tenho duas categorias!

Sabendo a taxa de acerto

Vimos lá no começo da aula que nós tínhamos 3 porcos e 3 cachorros, cada um com suas variáveis, ou seja, suas características que definiam se eles eram gordinhos (1) ou não (0), se tinham perninhas curtas (1) ou não (0) e, por fim, verificamos se eles faziam *auau* (1) ou não (0) e 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 nós testamos 2 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 nós rodarmos o nosso algoritmo e não fazer 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 se 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, se iremos funcionar conforme o esperado para verificar o quão bons estamos sendo com esse algoritmo. E como podemos testar? Podemos criar um cenário de teste utilizando os 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 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 são os nossos resultados, para uma variável chamada resultado:

resultado = modelo.predict(teste)

print(resultado)

Agora repare o 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:

> python 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, ele conseguiu prever! Mas é importante lembrar que no mundo real, ou seja, na prática, esse resultado não costuma ser 100% e o próximo exemplo que veremos logo mais, de compras na web, não iremos conseguir acertar sempre. Porém, observe que estamos testando isso manualmente, ou seja, estamos verificando se o teste passou a olho nu, para o nosso caso atual, não tem problema, mas no mundo real, iremos realizar testes gigantes, como por exemplo, 1000 elementos! E não faz sentido nós ficarmos olhando 1 a 1 para verificar o quanto ele acertou ou o quanto ele errou... O que precisamos saber é: quantos elementos valores 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 2 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 e marcação: -1, resultará em: 1 - (-1) -> 1 + 1 = 2.
* Resultado: -1 e marcação: -1, resultará em: -1 + 1 = 0.
* Resultado: -1 e marcação: 1, resultará em: -1 - 1 = -2

Resumindo 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?

> python classificacao.py

> [0 0 0]

O resultado foi tudo 0, então acertou 100% novamente! Mas o ideial seria que o nosso algoritmo mostrasse o percentual ao invés de só a diferença entre o resultado e a marcacoes\_teste. Então vamos fazer com que a diferença dessas duas variáveis sejam retornadas para uma variável chamada diferencas:

diferencas = resultado - marcacoes\_teste

Mas como iremos fazer para verificar os acertos? Sabemos que qualquer valor igual a 0 é um acerto, ou seja, qualquer valor da variável diferencas que seja 0. Isso significa que, para cada diferença dentro da variável diferencas:

for d in diferencas

Se o valor for igual a 0:

for d in diferencas if d == 0

retorne:

acertos = for d in diferencas if d == 0

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

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

Se imprimirmos os acertos:

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

print(acertos)

Resultado:

> python classificacao.py

> [0, 0, 0]

Observe que foi impresso foi um array com todos os 0 contidos no array diferencas, ou seja, todos os valores, porém, não queremos os valores, precisamos saber a quantidade de elementos que existem nesse array, e como fazemos para retornar a quantidade de elementos existentes 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:

> python 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)

diferencas = resultado - marcacoes\_teste

print(diferencas)

acertos = [d for d in diferencas 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 extraírmos 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:

> python classificacao.py

> [0 0 0]

> 3

> 3

> 1

Vamos melhorar a nossa impressão, vamos retirar todas as impressões. Então imprimimos primeiro o nosso resultado: print(resultado), depois a diferença entre o resultado e as marcações de teste: print(diferenca), por fim, faremos a impressão da taxa de acerto: print(taxa\_de\_acerto), porém vamos multiplicar 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)

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

print(resultado)

print(diferencas)

print(taxa\_de\_acerto)

Rodando o nosso algoritmo, obtemos o seguinte resultado:

> python 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:

> python classificacao.py

> [-1 1 -1]

> [ 0 0 -2]

> 66.6666666667

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

Repara que a taxa de acerto é fundamental para que o nosso algoritmo, no momento em que recebe novos dados, nesse caso, animais novos com características novas, seja possível verificar o quão bom ele foi em uma situação do mundo real, como por exemplo, nesse último teste, ele foi capaz de acertar 66%.

Daqui a pouco a nossa situação será diferente, iremos utilizar um exemplo real em uma aplicação web onde iremos 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%.

Calculando a taxa de acerto

Imprimir apenas a classificação dos animais não é a forma ideal para mostrarmos o quão bom foi o nosso algoritmo, principalmente se temos diversos dados! Quando estamos classificando, é bem comum mostrarmos um percentual que indica a taxa de acerto que o nosso algoritmo obteve para uma melhor analise.

Crie uma marcação de teste para os elementos misteriosos utilizando o array marcacoes\_testeindicando o que é cada elemento misterioso.

Calcule a diferença das variáveis resultado e as marcacoes\_teste e atribua a uma variável chamada diferencas.

Percorra o array diferencas e devolva todos os seus elementos que forem iguais a 0 e atribua a uma variável chamada acertos.

Calcule o total de acertos pegando o tamanho do array acertos com o método len() e atribua à variável total\_de\_acertos.

Calcule o total de elementos pegando o tamanho do array teste com o método len() e atribua à variável total\_de\_elementos.

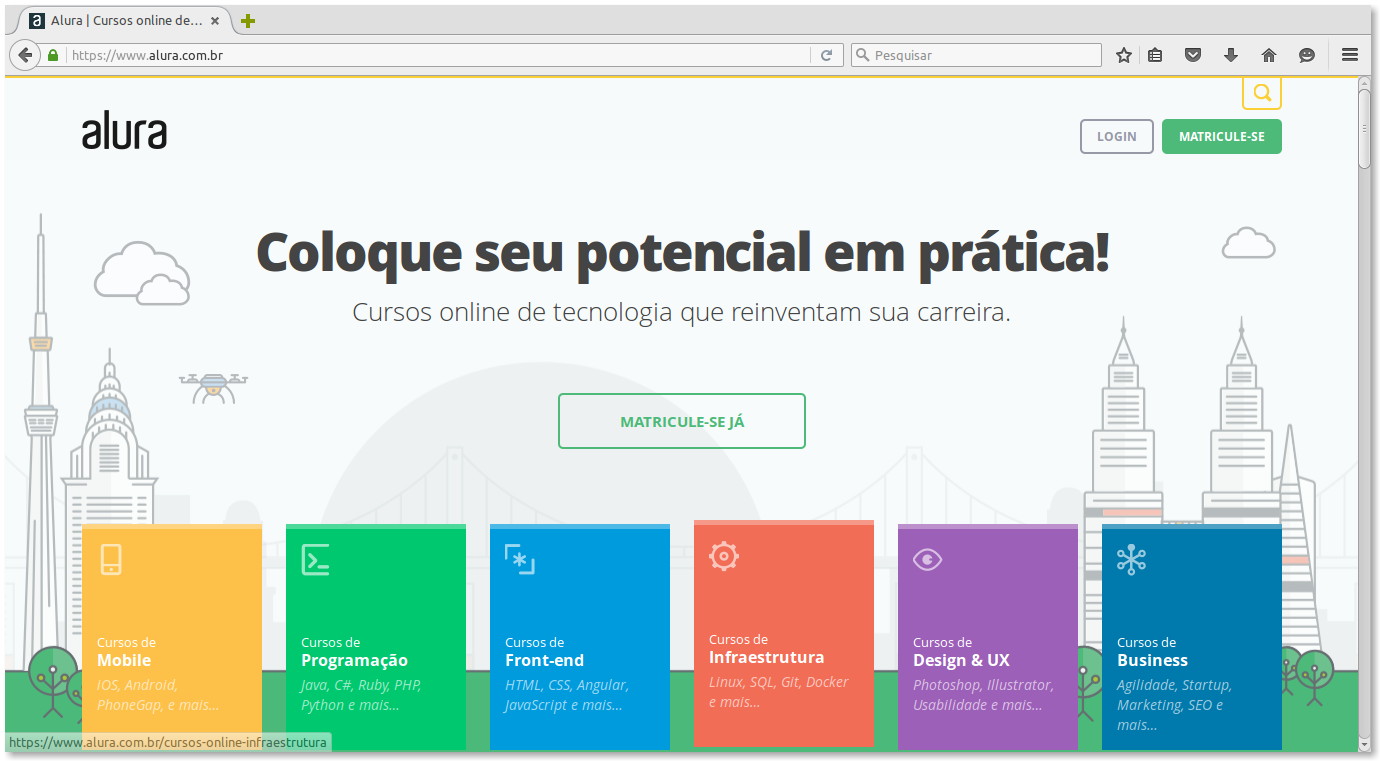
Calcule a taxa de acerto multiplicando 100.0 pela divisão das variáveis total\_de\_acertos e total\_de\_elementos e atribua à variável taxa\_de\_acerto.

Por fim, imprima as variáveis resultado, diferencas e taxa\_de\_acerto.

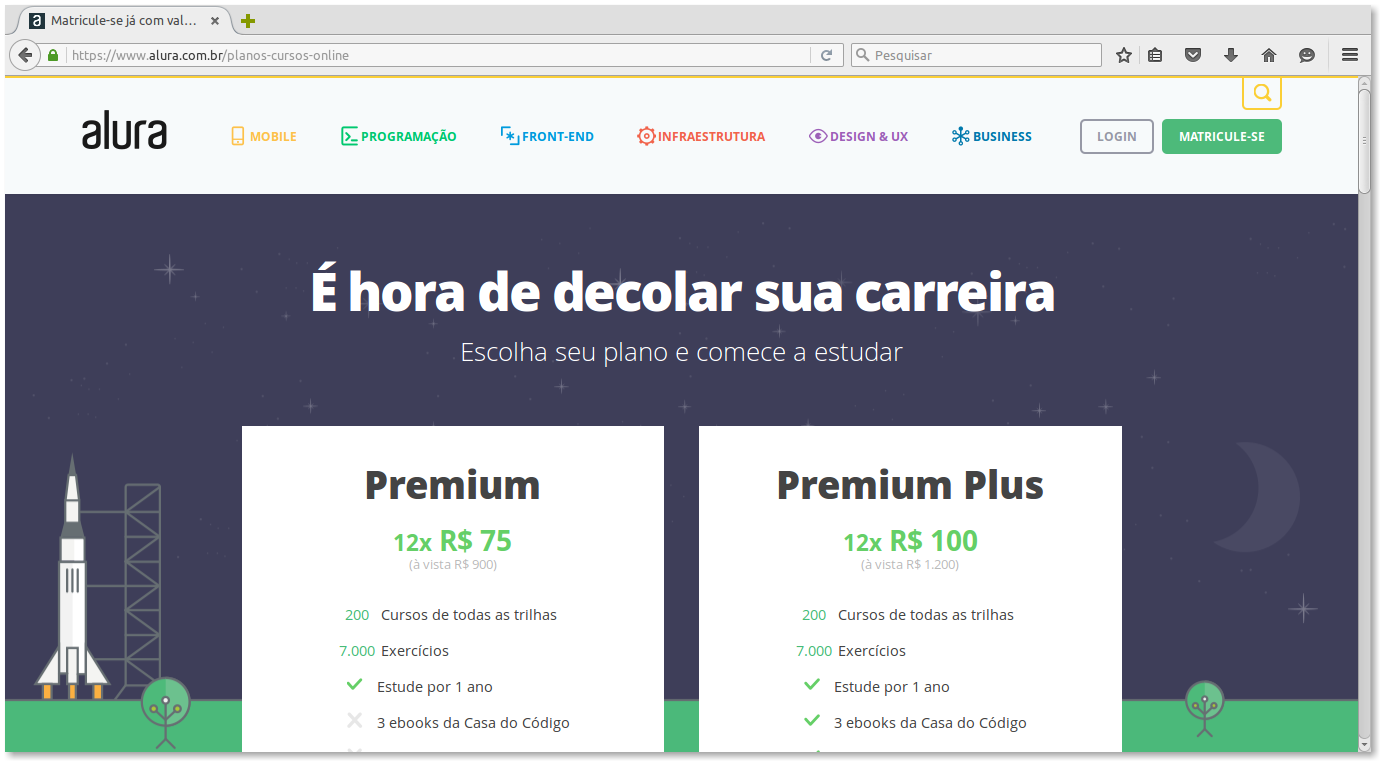
Exemplo no mundo web

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

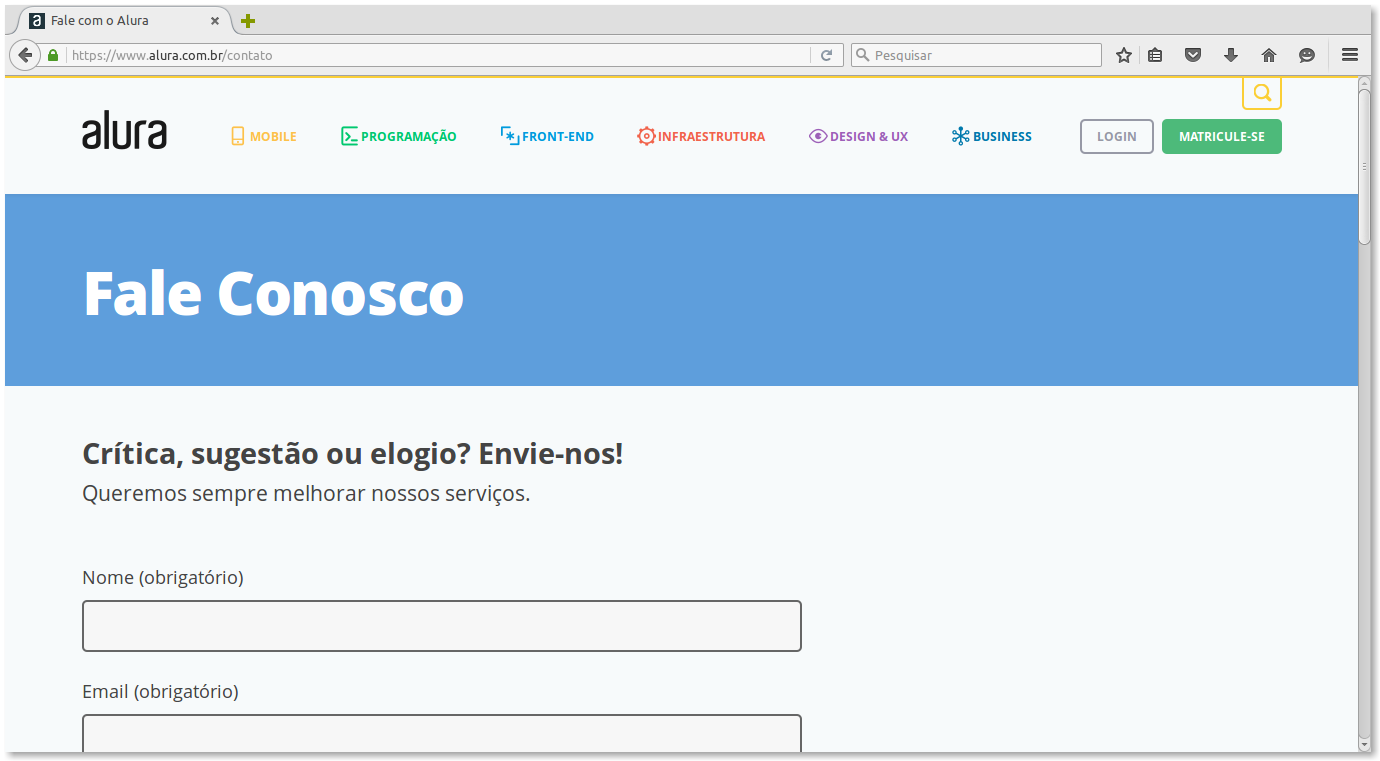
Um dos exemplos seria um usuário que entra num site de cursos.



Então ele entra numa [segunda página](https://www.alura.com.br/cursos-online-programacao):



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



Visto apenas essas 3 páginas, surgem 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, um deles seria:

Eu 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, 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 usado no mundo a fora, por exemplo, essa situação para detectar se um funcionário vai ou não continuar. A [Accenture](https://www.accenture.com/) é uma das empresas muito famosa por usar esse tipo de classficação. Um outro exemplo, poderia ser esse que fizemos do usuário que acessou as página do site, quais serão as chances desse usuário comprar ou não? Se ele não comprar, não seria melhor eu ajudar ele 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 ele está com dúvida sobre o produto e precisa de ajuda.

Veja o quão interessante é utilizar a classificação em diferentes contextos, ou seja, não utilizamos a classficação para classificar apenas cachorros ou porcos, também não utilizamos apenas para as coisas mais loucas do [Google](https://www.google.com.br/), e também, não é só para fazer com que um carro dirija sozinho... Queremos vender um produto! Queremos ajudar um cliente, queremos 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.

Repara que, 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 analise ele visitou as 3 páginas [1, 1, 1], ele vai comprar ou não? Podemos fazer diversas variações, como por exemplo: Ele visitou apenas a página 1 e paǵina 3 [1, 0, 1], será que ele vai comprar? Agora ele visitou a página 2 e a terceira apenas [0, 1, 1], será que ele vai comprar?

Percebeu que estamos caindo no mesmo problema de sempre? Isso é, dada uma matriz de características de cada usuário do passado e seus comportamentos, ou seja, 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 que ele vai comprar ou não?". Então repara que sempre caímos nessa situação de classificação, baseado em características!

Classificando um usuário.

Nessa primeira aula, 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:

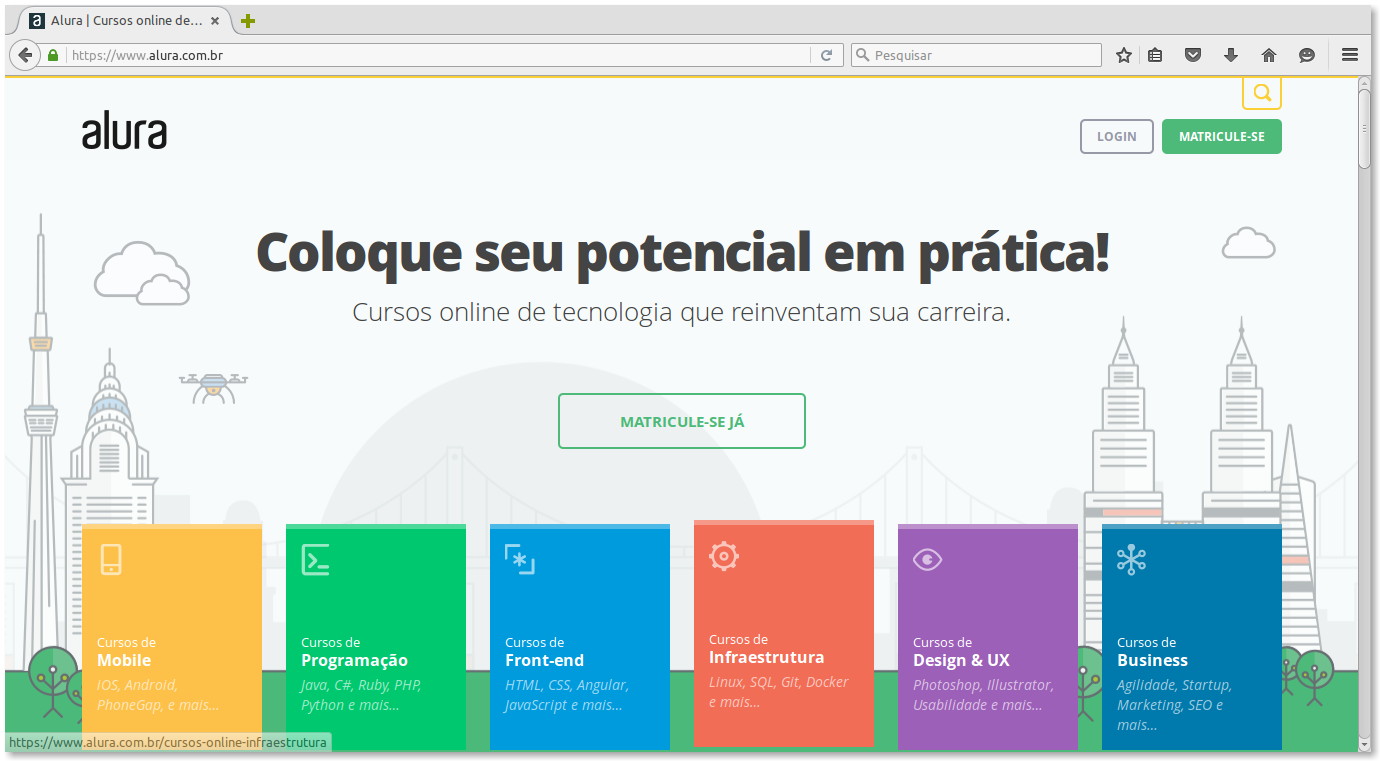
teste = [misterioso1, misterioso2, misterioso3]

E as nossas marcações de teste:

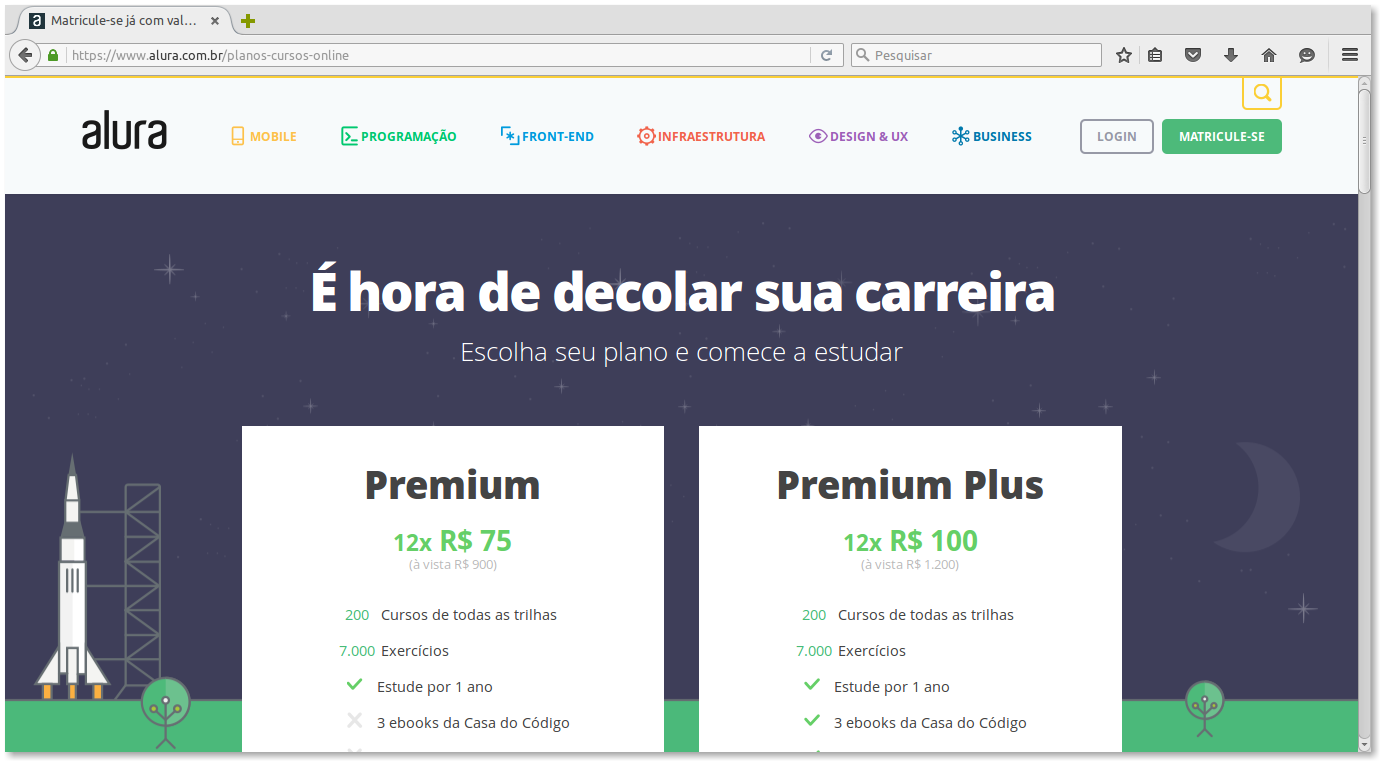
marcacoes\_teste = [-1, 1, 1]

E tudo isso representava o que? 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?

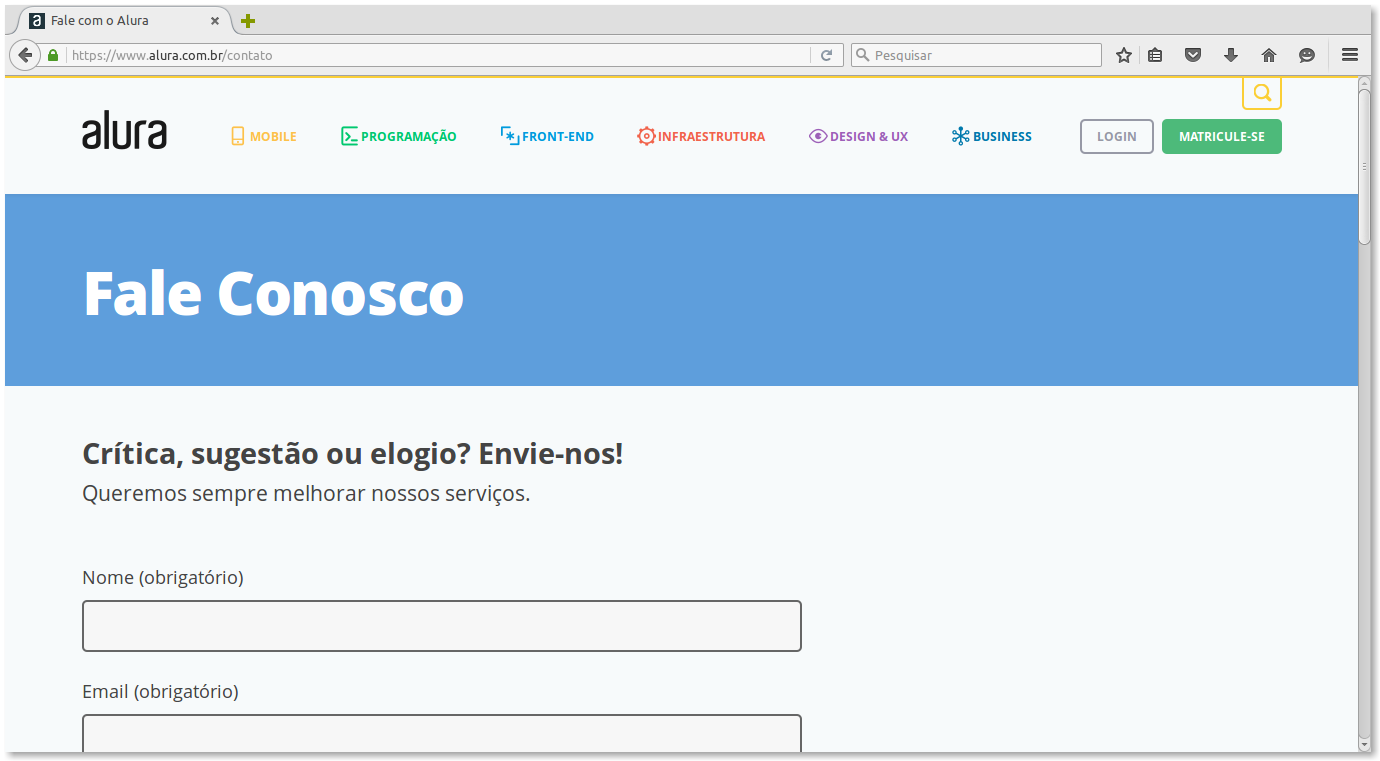
No mundo web, temos diversos usuários acessando meu sistema, então teremos um log de acesso, vamos utilizar a página do [Alura](https://www.alura.com.br/) como exemplo. A primeira é página principal:



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



Por fim, chegamos a uma página de um curso:

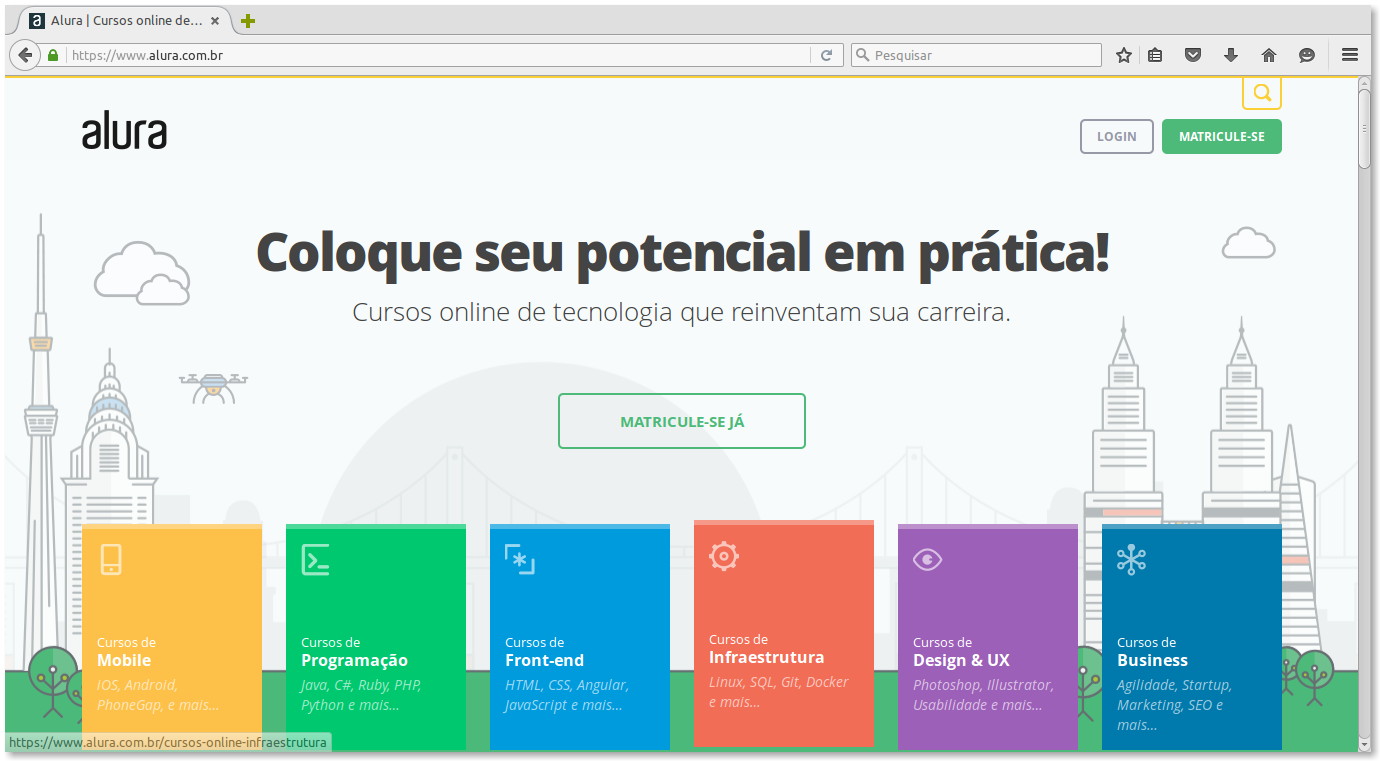


E, de alguma forma, sabemos quais foram as páginas que o usuário acessou, ou seja, se acessou as 3 páginas, se acessou a página 1 e a 3 se acessou a 2 e 3 e assim por diante. É claro que num cenário real, a analise é feita mais 100 páginas ou mais, porém, inicialmente, faremos com 3 páginas, 3 características! Assim como fizemos com os animais (cachorros e porcos). Tudo bem?

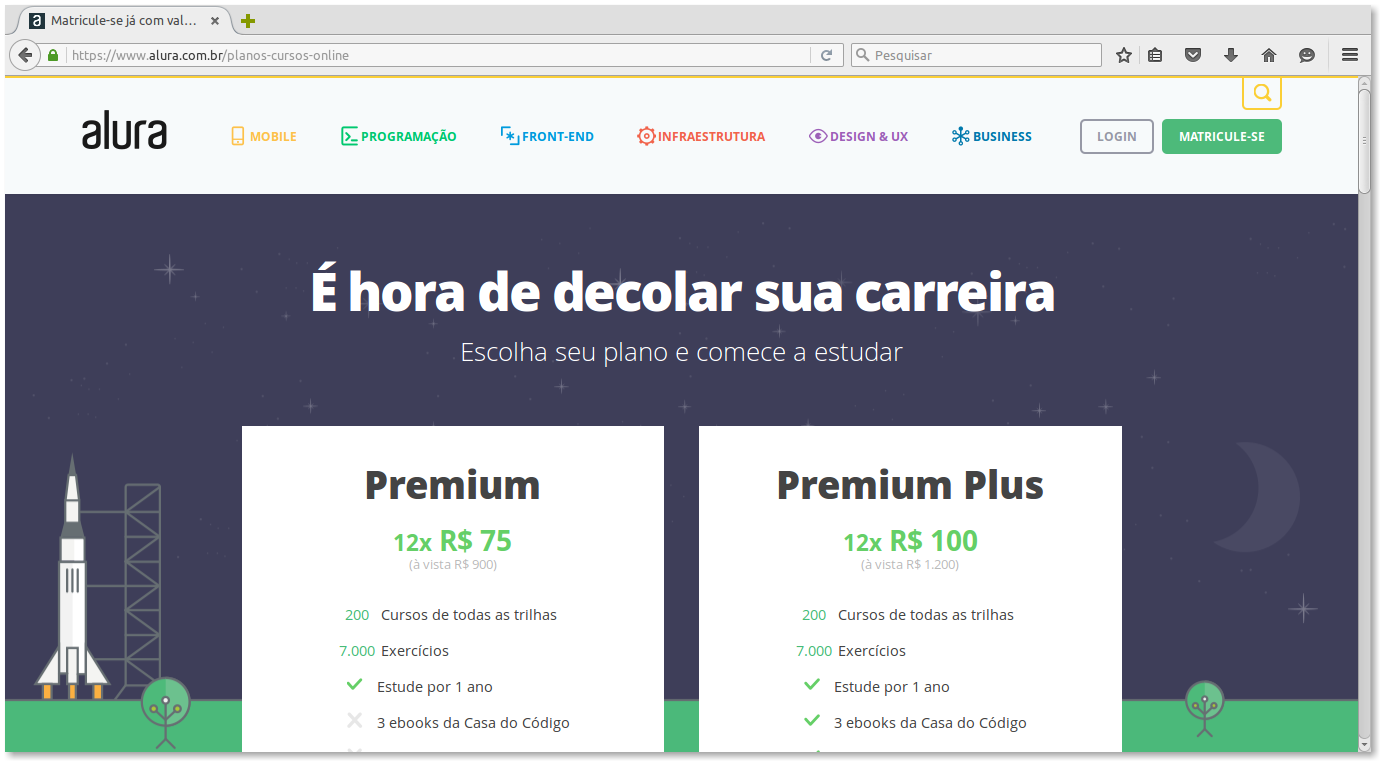
Sabemos quais páginas o usuário acessou, então o que precisamos saber? Queremos saber se esse mesmo usuário vai comprar ou não, se vai assinar o meu produto ou não vai, se ele vai virar um cliente ou não, se ele vai entrar em contato ou não, tudo isso se resume entre 0 e 1, ou seja, é um tipo de classificação! Estamos classificando um usuário que está acessando o meu site de acordo com quais características? 3 características:

* Se ele visitou a primeira página.
* Se ele visitou a segunda página
* Se ele visitou a terceira página

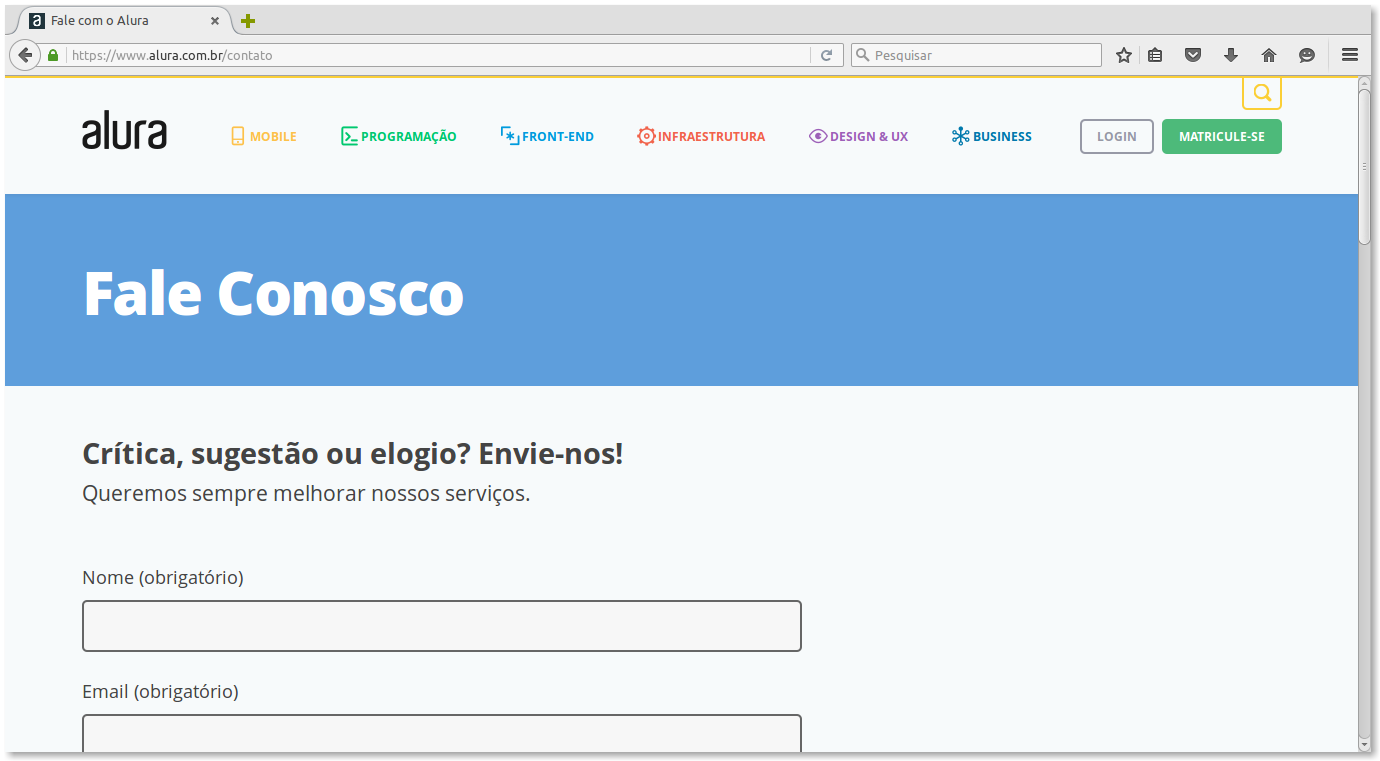
Quais páginas poderia ser essas? Podemos utilizar a página [principal](https://www.alura.com.br/):



Será que o usuário entrou na página de [como funciona](https://www.alura.com.br/planos-cursos-online)?



Por fim, será que o usuário entrou na [página de contato](https://www.alura.com.br/contato)?



Será que ele entrou nessas 3 páginas? Vamos utilizar 0 para indicar que não entrou e 1 para indicar que entrou. Então vamos criar um novo arquivo. Agora vamos preencher as características do usuário.

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

1,0,1,1

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):

1,0,1,1

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).

1,0,1,1

0,1,0,0

1,1,0,1

As 3 primeiras variáveis representa 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

1,0,1,1

0,1,0,0

1,1,0,1

Observe que estamos analisando passo-a-passo as ações do usuário 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 não o agradou do que estamos fornecendo ou então o que ele procura ainda não está disponível e, então, iremos correr atrás para disponibilizar o conteúdo, por exemplo. Ou então, é só uma dúvida que ele tem e não conseguiu tirar com ninguém e nós mesmo iremos tirar a dúvida para que ele adquira o produto... Então eu estou demonstrado um exemplo de como podemos classificar no mundo da web. Independentemente se você natura ou se é um e-commerce que vende de tudo ou é um blog e você precisa saber se ele acessou determinadas páginas e se ele vai clicar em um propaganda ou não, se ele vai voltar daqui a x dias ou não. Ou seja, não importa a qual necessidade estamos fazendo a classificação, o importante é, dadas as características do usuário eu quero saber se ele comprou ou não. Observe que na primeira linha, as 3 primeiras colunas: acessou\_home, acessou\_como\_funciona, acessou\_contato do no nosso novo arquivo, são as características que desejamos analisar dos nossos usuários e 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 cada uma das linhas são separadas por vírgula, ou seja, de acordo com o cabeçalho:

acessou\_home, acessou\_como\_funciona,

acessou\_contato, comprou

Esse tipo de arquivo chamamos de **comma separated value** (valor separado por vírgula) mais conhecido por csv, esse arquivo separa todos os valores por vírgula. Esse tipo de arquivo, nós conseguimos ler pelo python facilmente por meio de uma função que carrega esse arquivo e então tratamos os dados de uma maneira bem fácil e é exatamente isso que faremos no próximo capítulo. Iremos utilizar esse padrão para a leitura dos dados, pois é um padrão bem simples de manter.

***2 – importando, classificando e validando um modelo***

Lendo um arquivo .CSV

No arquivo **acesso.csv** estarão todos os dados referente ao acesso de páginas do usuário no nosso site, ou seja, se ele acessou a página home, como funciona e contato, por fim, informamos se ele comprou ou não. Nesse arquivo temos diversos dados do nosso usuário no passado, ou seja, suas características (acessos às páginas) e a marcação (se comprou), no total são 100 linhas.

Agora que temos o arquivo 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 leitura desse arquivo e colocar os dados em um array que tenha as informações dos nossos elementos, porém nós 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 divido da seguinte forma:

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

1,1,0,0

Os valores referentes a acessou\_home, acessou\_como\_funciona, 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 dados que iremos prever.

**Importando os dados**

Agora que sabemos a forma que iremos representar os dados do nosso arquivo csv, precisamos importar esse arquivo. Dentro do arquivo dados.py, escreveremos o código para importar o csv:

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 irá associar os valores para o array de dados e os valores para o array de marcações:

import csv

def carregar\_acessos():

dados = []

marcacoes = []

Agora que definimos os nossos arrays, precisamos abrir o arquivo utilizando a função open() e enviando o nome do arquivo como parâmetro:

import csv

def carregar\_acessos():

dados = []

marcacoes = []

arquivo = open('acesso.csv', 'rb')

Observe que utilizamos também o parâmetro rbindicando que queremos ler o arquivo.

Se estiver utilizando Python3, no segundo parâmetro da função open() pode passar apenas a string 'r'

import csv

def carregar\_acessos():

dados = []

marcacoes = []

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

Agora precisamo ler esse arquivo csv, para isso iremos utilizar a função reader() da biblioteca que importamos:

import csv

def carregar\_acessos():

dados = []

marcacoes = []

arquivo = open('acesso.csv', 'rb')

csv.reader(arquivo)

A função reader devolve um leitor:

import csv

def carregar\_acessos():

dados = []

marcacoes = []

arquivo = open('acesso.csv', 'rb')

leitor = csv.reader(arquivo)

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

import csv

def carregar\_acessos():

dados = []

marcacoes = []

arquivo = open('acesso.csv', 'rb')

leitor = csv.reader(arquivo)

for linha in leitor:

Porém, perceba que 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). Mas e agora? O que precisamos fazer? Precisamos adicionar ao nosso array dados as colunas: acessou\_home, acessou\_como\_funciona, acessou\_contato utilizando a função append():

import csv

def carregar\_acessos():

dados = []

marcacoes = []

arquivo = open('acesso.csv', 'rb')

leitor = csv.reader(arquivo)

for acessou\_home, acessou\_como\_funciona,

acessou\_contato, comprou in leitor:

dados.append([acessou\_home,

acessou\_como\_funciona,

acessou\_contato])

Por fim, adicionamos as marcações:

import csv

def carregar\_acessos():

dados = []

marcacoes = []

arquivo = open('acesso.csv', 'rb')

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)

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

import csv

def carregar\_acessos():

dados = []

marcacoes = []

arquivo = open('acesso.csv', 'rb')

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

Vamos testar o nosso código? Vá no 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 importa a função carregar\_acessos do arquivo dados.py:

>>> from dados import carregar\_acessos

>>>

Agora podemos utilizar a nossa função! Vamos retorna o valor dessa função:

>>> dados, marcacoes = carregar\_acessos()

>>>

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

>>> marcacoes

['comprou', '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', '0', '1', '0', '0', '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', '0', '1', '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:

1,1,0,0

1,1,0,0

1,1,0,0

1,1,0,0

1,1,0,0

1,0,1,1

...

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

>>> dados

[['acessou\_home', 'acessou\_como\_funciona', 'acessou\_contato'], ['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'], ['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', '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'], ['1', '0', '1'], ['0', '1', '0'], ['0', '0', '1'], ['0', '1', '0'], ['1', '0', '1'], ['0', '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'], ['0', '0', '1'], ['1', '1', '0'], ['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'], ['1', '1', '0'], ['0', '0', '1'], ['0', '0', '1'], ['1', '1', '1'], ['0', '0', '1'], ['0', '1', '0'], ['0', '0', '0']]

>>>

Os dados batem, conforme o esperado! Observe que agora nós 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 de X, ou seja, os valores que conhecemos do nosso usuário como por exemplo, se ele acessou uma página, se ele tem perninha curta ou qualquer informação que nós já sabemos!

def carregar\_acessos():

X = []

marcacoes = []

# restante do código

Mas e as nossas marcações? Como é mesmo que chamávamos? Lembre-se que as marcações são os valores que não sabemos, ou seja, os valores que queremos calcular, prever! Tradicionalmente, esses valores, essas marcações que queremos prever, 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', 'rb')

leitor = csv.reader(arquivo)

for acessou\_home,acessou\_como\_funciona,acessou\_contato, comprou in leitor:

X.append([acessou\_home,acessou\_como\_funciona,acessou\_contato])

Y.append(comprou)

return X, Y

**Analisando os valores adicionados**

Vamos dar uma olhada novamente no resultado das nossas marcações:

>>> marcacoes

['comprou', '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', '0', '1', '0', '0', '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', '0', '1', '0', '0', '0']

>>>

Repare que veio também o array com a informação do cabeçalho, ou seja, a palavra 'comprou', porém nós não queremos isso no nosso array, lembra como era 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 next(leitor) que pulará para a linha a seguir:

import csv

def carregar\_acessos():

X = []

Y = []

arquivo = open('acesso.csv', 'rb')

leitor = csv.reader(arquivo)

next(leitor)

for acessou\_home,acessou\_como\_funciona,acessou\_contato, comprou in leitor:

X.append([acessou\_home,acessou\_como\_funciona,acessou\_contato])

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, ao invés 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', '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', '0', '1', '0', '0', '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', '0', '1', '0', '0', '0']

Aparentemente tudo está funcionando como o esperado! Porém, '0'? '1'? Isso é uma *string*! Não queremos *string*s, nós queremos números! Afinal estamos lendo números... E como resolver isso? Podemos converter as *string*s para números, mas que tipo de números? Nesse caso estamos trabalhando com números inteiros, então converteremos para números inteiros! Para converter uma string para inteiro, 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', 'rb')

leitor = csv.reader(arquivo)

next(leitor)

for acessou\_home,acessou\_como\_funciona,acessou\_contato, comprou in leitor:

X.append([int(acessou\_home),int(acessou\_como\_funciona)

,int(acessou\_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 Y:

>>> from dados import carregar\_acessos

>>> X, Y = carregar\_acessos()

>>> Y

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

E o X?

>>> X

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

Agora sim nós carregamos os nossos números!

**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', 'rb')

leitor = csv.reader(arquivo)

next(leitor)

for acessou\_home,acessou\_como\_funciona,acessou\_contato, comprou in leitor:

X.append([int(acessou\_home),int(acessou\_como\_funciona)

,int(acessou\_contato)])

Y.append(int(comprou))

return X, Y

Veja que utilizamos os nomes acessou\_home, acessou\_contato, acessou\_alguma\_coisa! Vamos retirar esse acessou, pois já sabemos que nossos dados refere-se a acesso de página do usuário, então não precisamos repetir a mesma palavra em todos os nossos dados! Veja o resultado do nosso código:

import csv

def carregar\_acessos():

X = []

Y = []

arquivo = open('acesso.csv', 'rb')

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, mas perceba que ainda existe alguns detalhes, vejamos esse trecho de código:

X.append([int(home),int(como\_funciona)

,int(contato)])

Observe que enviamos um monte de informação para o X que, a primeira vista, não da pra saber a 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', 'rb')

leitor = csv.reader(arquivo)

next(leitor)

for home,como\_funciona,contato, comprou in leitor:

dado = [int(home),int(como\_funciona)

,int(contato)]

X.append(dado)

Y.append(int(comprou))

return X, Y

Bem mais limpo e de fácil compreensão! Vamos verificar se o nosso código ainda funciona? Porém, dessa vez, ao invés de usar o interpretador python, vamos criar um novo arquivo chamado classifica\_acessos.py e 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:

> python 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, 1, 0], [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, 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], [1, 0, 1], [0, 1, 0], [0, 0, 1], [0, 1, 0], [1, 0, 1], [0, 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], [0, 0, 1], [1, 1, 0], [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], [1, 1, 0], [0, 0, 1], [0, 0, 1], [1, 1, 1], [0, 0, 1], [0, 1, 0], [0, 0, 0]]

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

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

from dados import carregar\_acessos

X, Y = carregar\_acessos()

Importando os dados do arquivo CSV.

Crie um arquivo chamado **dados.py**.

Dentro do **dados.py**, importe a biblioteca que consegue ler arquivos csv utilizando o comando import csv.

Agora defina uma função chamada carregar\_acessos(). Dentro da função, defina 2 arrays dados e marcacoes, para representar os nossos dados e marcações que estão no arquivo.

Utilize a função open() passando como primeiro parâmetro o nome do arquivo e, como segundo parâmetro, "rb" para indicar que está realizando uma leitura e retorne para uma variável chamada arquivo.

Se estiver utilizando o Python3, no segundo parâmetro da função open() passe a string "r" para indicar que está realizando uma leitura.

Agora utilize a função reader da biblioteca csvenviando como parâmetro o arquivo e retorne um leitor. Então faça um for para cada uma das colunas: for acessou\_home, acessou\_como\_funciona, acessou\_contato, comprou in leitor: e atribua o valor das colunas: acessou\_home, acessou\_como\_funciona, acessou\_contato para o array dados e a coluna comprou para o array marcacoes utilizando a função append. Por fim, retorne os arrays dados e marcacoes.

Verificando os dados do arquivo CSV

Abra o interpretador do python no terminal e importe a função carregar\_acessos utilizando o comando from dados import carregar\_acessos.

Teste os valores dos dados e marcações que foram retornados.

Melhorando a leitura dos dados e código

No nosso primeiro exemplo, os nossos dados não continham cabeçalho, porém, os nossos novos dados estão vindo com esse cabeçalho:

[['acessou\_home', 'acessou\_como\_funciona', 'acessou\_contato'], ['1', '1', '0']...

Para corrigir esse detalhe, basta pular uma linha antes de percorrer o arquivo utilizando o comando next(leitor).

Além disso, perceba que os nossos dados ainda aparecem como string:

[['1', '1', '0'], ['1', '1', '0'], ['1', '1', '0'], ['1', '1', '0'],...]

Antes de atribuir os valores das colunas para os arrays dados e marcacoes converta-os para inteiro utilizando a função int()

Vimos também que os nossos dados e marcações possuem um nome específico para algoritmos de classificação que é X e Y. Substitua o array dados por X e o array marcacoes por Y.

Abra novamente o interpretador do python e teste o código com as novas alterações.

Utilizando o algoritmo Multinomial

Temos o X(dados) e o Y(marcações), mas e agora? O que faremos com eles? Lembra que no exemplo anterior, nós tinhamos 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, lembra que o predict espera um array de arrays para não mostrar aquele warning? Então 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:

> python classifica\_acessos.py

> [1]

Ele imprimiu 1 então isso significa que esse usuário vai comprar! 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 vai comprar? 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:

> python classifica\_acessos.py

> [1 0]

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

Além de testar apenas 2 casos, podemos testar muito mais, 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:

> python 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 página 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:

> python classifica\_acessos.py

> [1 0 0 0]

Nenhum desses usuários compraram... 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:

> python 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 prevenção 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 cachorros! Faremos isso no próximo vídeo!

Obtendo a taxa de acerto.

Lembra que 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

Perceba que nós temos os dados que representa a variável teste, ou seja, o nosso X, então 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 irá prever se cada um desses usuários vai ou não comprar. Se imprimirmos essa variável:

print(resultado)

E rodarmos o nosso algoritmo:

> python classifica\_acessos.py

> [0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 1 1 1

0 0 1 0 0 0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 1 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0]

Além dos dados, temos também as marcações(Y), ou seja, por meio dessas marcações que sabemos se os nossos elementos, nesse caso os usuários, que acessaram as páginas do nosso web site, compraram ou não! Se imprirmos também o nosso Y:

print(resultado)

print(Y)

O resultado é:

> python classifica\_acessos.py

> [0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 1 1 1

0 0 1 0 0 0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 1 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0]

> [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, 0, 1, 0, 0, 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, 0, 1, 0, 0, 0]

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

diferencas = resultado - Y

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

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

O total de acerto 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, ou seja, 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 multiplicado 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)

diferencas = resultado - Y

acertos = [d for d in diferencas 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 testar o nosso código? Vejamos o resultado:

> python classifica\_acessos.py

> 93.9393939394

> 99

Acertou 93,93%? Já podemos ficar felizes e contente e declarar vitória! Porém, lembra que eu havia falado que é muito difícil chegarmos em taxas de acerto absurdamente altas? Como por exemplo, 100%!

Se isso é verdade, então porque o nosso algoritmo está acertando tanto? Se verificarmos quem utilizamos 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 denovo! Será que estamos testando da maneira correta? Vamos analisar uma situação similar:

Suponhamos que eu dê para você 10 porquinhos e 10 cachorrinhos e lhe digo quais são os porcos e quais são os cachorros.

Você aprendeu quais são os porcos e quais são os cachorros. Em seguida, eu te dou os **mesmos 10 porquinhos** e os **mesmos 10 cachorros** e 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 mesmo porquinhos e os mesmo cachorros que eu te ensinei em poucos instantes atrás.

Um outro exemplo similar seria, eu te apresentar a minha mão direita e, logo em seguida, te apresento a minha mão esquerda e então eu mostro a minha mão direita e pergunto: "Que mão é essa?", e depois mostro a minha mão esquerda e pergunto: "Que mão é essa?", com certeza você vai acertar, pois eu acabei de te falar qual é cada uma das mãos que estou te perguntado...

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

No mundo real, quando treinamos um algoritmo de classificação nós o treinamos 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 elementos desconhecidos 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 com que o nosso algoritmo seja treinado e testados por todos esses 99 registros sem que aconteça o mesmo caso problemático que vimos anteriormente?

Treinando e testando o algoritmo com metade dos dados

Podemos dividir em duas partes os nossos registros! Por exemplo, podemos deixar os primeiros 50 registros para treinar o nosso algoritmo e os 49 demais para testá-lo, ou então, podemos deixar 80 registros para treiná-lo e os 19 restantes 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 iremos utilizar 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 Y:

X, Y = carregar\_acessos()

Perceba que 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 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 retornando para o X e 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], [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, 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], [1, 0, 1], [0, 1, 0], [0, 0, 1], [0, 1, 0], [1, 0, 1], [0, 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], [0, 0, 1], [1, 1, 0], [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]]

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 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)

Ao invés de utilizar os nossos dados reais para treinar (X e Y), utilizaremos os nossos dados de treino, ou seja, treino\_dados e treino\_marcacoes:

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 está atualmente o nosso código:

resultado = modelo.predict(X)

Mas não podemos mais pedir para ele prever com dados que ele já conheça, ou seja, ao invés 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:

diferencas = resultado - Y

Observe que estamos fazendo a diferença com as marcações reais (Y) que refere-se aos dados reais! Precisamos então, mudar para as nossas marcações que representam os nossos dados de teste (teste\_marcacoes):

diferencas = resultado - teste\_marcacoes

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

acertos = [d for d in diferencas 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 diferencas que forem iguais a 0, a taxa\_de\_acertos será também o tamanho do array acertos, porém, o total\_de\_elementosé 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 diferencas 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:

> python classifica\_acessos.py

88.8888888889

9

Veja que o nosso algoritmo treinou com 90 elementos e 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 cachorros o nosso cenário anterior se resume em:

* Eu te apresento 90 animais entre porcos e cachorros e eu te ensino quais são os porcos e quais são os cachorros.
* Então eu te mostro os mesmos 90 animais entre porcos e cachorros que eu havia te apresentado e peço para que você os classifique mim quais são os porcos e quais são os cachorros.

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

* Eu te apresento 90 animais entre porcos e cachorros e eu te ensino quais são os porcos e quais são os cachorros.
* Então eu te mostro 9 novos animais que também são porcos e cachorros, mas você nunca os viu e então eu peço para que você classifique para mim quais são os porcos e quais são os cachorros

Se você tiver que 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, o nosso algoritmo está lidando com elementos que ele nunca viu na vida!

Treinando e testando o algoritmo

Crie o arquivo classifica\_acesso.py e importe o arquivo dados.py utilizando o comando from dados import carregar\_acessos.

Chame a função carregar\_acessos() e atribua para a variável X e Y.

Agora precisamos dos dados de treino (treino\_dados e treino\_marcacoes) e de teste (teste\_dados e teste\_marcacoes), para os dados de treino, utilizaremos as primeiras 90 linhas de X e Y e para teste as 9 últimas, ou seja, 90% dos nossos elementos para treino e 10% para teste.

Com os dados de treino em mãos, peça para o modelo treinar com os nossos dados de treino utilizando a função fit() passando os arrays treino\_dados e treino\_marcacoes.

Por fim, peça para ele prever (predict) o array teste\_dados e atribuir para a variável resultado e imprima a variável resultado.

Calculando a taxa de acerto

Em algoritmos de classificação, imprimir apenas o resultado, a primeira vista, não significa se o nosso algoritmo foi bem ou não. Por isso, em geral, precisamos calcular a taxa de acerto que demonstra o quão bom o algoritmo foi em percentual. Começaremos calculando a diferença entre o resultado e as marcações de teste.

Calcule a diferença entre a variável resultado e teste\_marcacoes e atribua para um variável chamada diferencas.

Dentro de um array, faça um for para o array diferencas e verifique cada valor desse array se é igual a 0. Por fim, atribua a uma variável chamada acertos.

Calcule o total de acerto pegando o tamanho da variável acertos por meio do método len() e atribua a variável total\_de\_acertos

Agora Calcule o total de elementos pegando o tamanho da variável teste\_dados e atribua para a variável total\_de\_elementos

Com todos os dados necessários, calcule a taxa de acerto realizando a divisão entre a variável total\_de\_acertos e total\_de\_elementos e, multiplique por 100.0 para aparecer o ponto flutuante.

Por fim, imprima o a taxa de acerto e o total de elementos.

Resumo

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 as características de um e-mail? Poderia ser o tamanho do e-mail, número de palavras repetidas, se usa letra maiúscula, se o remetente é conhecido, se já foi marcado como *SPAM* entre diversas características! E se forem animais? Outras características. E se forem alunos que irão repetir de ano? Outras características.

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

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

Além disso, precisamos sempre lembrar que existe uma taxa de erro, pois nem sempre nós iremos acertar, ou seja, precisamos sempre verificar a forma que o algoritmo foi treinado, analisando a quantidade de dados e características, para determinar um retorno aceitável do nosso objetivo, por exemplo, se o nosso objetivo é conversar com mais pessoas que provavelmente irão pedir para serem demitidas ou com menos pessoas, ou então, se queremos conversar com os alunos que estão com uma chance maior para reprovarem, talvez seja mais interessante conversar com mais alunos do que menos alunos, porém, se fosse entre um cachorro e porco, para qual dos 2 eu gostaria de errar mais? Depende do caso! Isso significa que cada caso, precisaremos verificar o quão interessante é, ou seja, se aceitamos uma taxa de erro maior ou menor.

Temos que nos atentar a todos esses detalhes, pois é dessa forma que o processo de classificação, baseado nas características, funciona.

Revisando o código

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 eletronica e converter para o formato csv. E vimos que podemos ler um arquivo csv por meio de uma função de leitura de arquivo do próprio python:

arquivo = open('acesso.csv', 'rb')

leitor = csv.reader(arquivo)

Nesse exemplo nós só trabalhamos com dados inteiros, mas, mais pra frente, veremos como trabalhar com float, string e outras coisas. Além disso, vimos que, de acordo com os dados que lemos:

X,Y = carregar\_acessos()

Aprendemos a importância de treinar um modelo e testá-lo, porém existe um grande desafio nessa abordagem, pois se treinarmos o nosso modelo com diversos dados:

modelo.fit(X,Y)

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

resultado = modelo.predict(X)

A chance dele 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 utilizamos os mesmos dados, que foram usados para treino, para testar o nosso modelo.

Perceba que utilizamos uma estratégia para separar os dados de treino com os 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% pra teste e 10% pra 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.

***3 – classificação de varias categoricas***

Entendo o problema de busca

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

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

Perceba que todas as nossas características tiveram apenas 2 tipos de valores, ou seja, 0 ou 1. Além disso, nossas marcações também tiveram apenas 2 valores, 0 ou 1, ou então, -1 ou 1... Observe que, por enquanto, todas as nossas características e marcações tiveram apenas 2 tipos de valores para distinguir os nossos elementos, porém, nem sempre iremos nos deparar com esses tipos de valores para os nossos dados. Iremos verificar um exemplo um pouco diferente ao qual vimos até agora, vamos analisá-lo?

No Alura, um site de cursos online, nós temos um cliente que visitou a home.

De acordo com o que vimos até agora, como que classificaríamos 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? Sem segredo também! Marcamos com 1 para indicar que ele estava logado. Vejamos a próxima ação dele:

E então, esse cliente comprou um curso

Da mesma forma que fizemos anteriormente, ou seja, marcamos com 1 para indicar que ele comprou... Porém, observe essa 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?)

Perceba que, 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 características com apenas 2 valores distintos, ou seja, 0 ou 1, acessou ou não, está logado ou não, comprou ou não, porém, para o valor de busca, podemos ter diversos valores, por exemplo, buscou algoritmos ou buscou Java ou buscou Ruby. Vamos verificar algumas possibilidades para esse caso na tabela abaixo:



Cada coluna refere-se a:

* **home**: se o cliente visitou a página home (0 ou 1).
* **busca**: o que o cliente buscou (algortimos ou Java ou Ruby ou ...).
* **estava logado?**: se o cliente estava logado (0 ou 1).
* **comprou??**: se o cliente comprou ou não (0 ou 1).

Analisando cada coluna, qual é a novidade dentre esses conjuntos de dados? É a coluna busca, pois é um tipo de dado que 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 comportamente 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 programação. E se agora você quiser algum livro de medicina? Novamente 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? Por exemplo, diversos valores, ou seja, pode ser "algoritmos" ou "Java" ou "Ruby" e assim por diante.

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? O que podemos fazer?

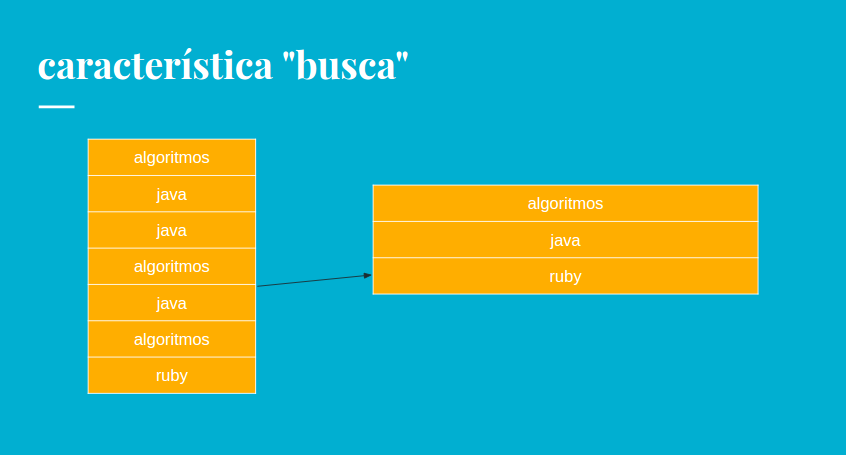
Precisamos, de alguma forma, transformar essas informações da coluna busca em variáveis númericas com valores entre 0 e 1, pois se conseguirmos fazer isso, ou seja, verificar se o cliente só fez busca de "algoritmos" ou "Java" ou "Ruby"... Consegue imaginar como podemos transformar diversos valores diferentes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

Mas por que pra 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?

Variável categórica

Vamos começar isolando essa coluna:



Observe que identificamos 3 tipos de características diferentes para essa coluna **busca**:

* Algoritmos
* Java
* Ruby

Cada um dos nossos clientes, fez apenas uma dessas buscas, ou seja, ou ele pesquisou "algoritmos" ou "Java" ou "Ruby". 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?

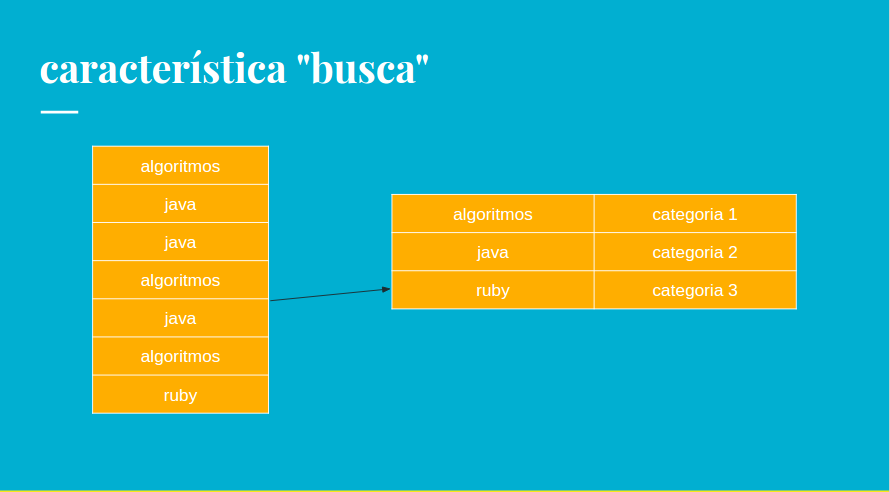
Em 3 perguntas:

* O cliente buscou algoritmos?
* O cliente buscou Java?
* O cliente buscou Ruby?

Vamos considerar agora, um cliente que buscou por "Java" e fazer novamente as 3 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 3 perguntas para classificar o que um outro cliente buscou! Então o que fizemos exatamente? Demos 3 categorias para a coluna busca:



Observe que cada característica tem a sua própria categoria:

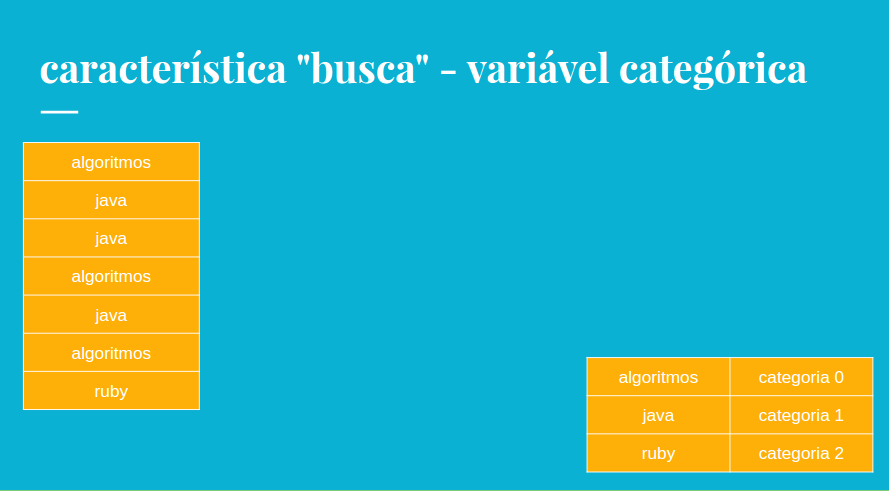
* Algoritmos - categoria 1
* Java - categoria 2
* Ruby - categoria 3

Isso significa que a nossa variável busca não é simplesmente uma variável que atribuímos apenas um valor (0 ou 1), ou seja, podemos atribuir 3 valores para essa mesma variável! Chamamos esse tipo de variável de ***variável categórica***, pois ela possui diversas categorias!

Um outro exemplo que poderíamos utilizar esse tipo de variável seria na seguinte pergunta:

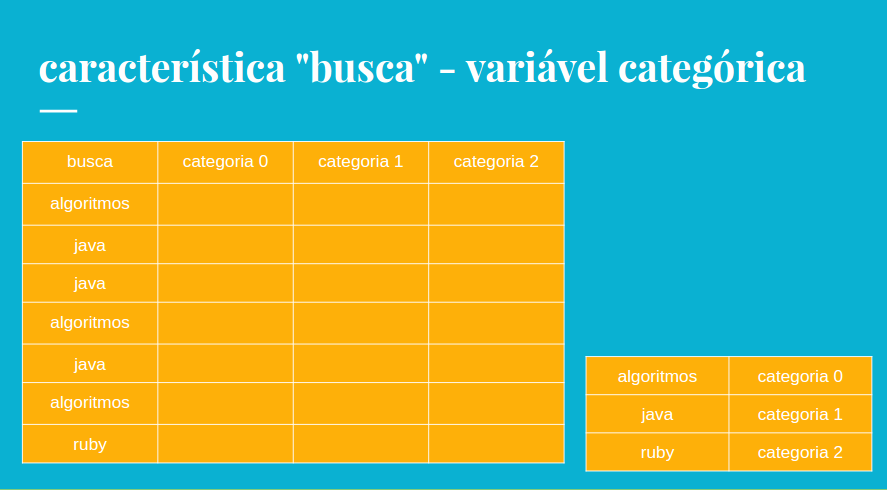
* Em qual estado você está presente nesse instante?

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



Note que agora adicionamos os valores de 0 a 2 para as categorias, pois utilizaremos arrays para classificar cada uma dessas características.

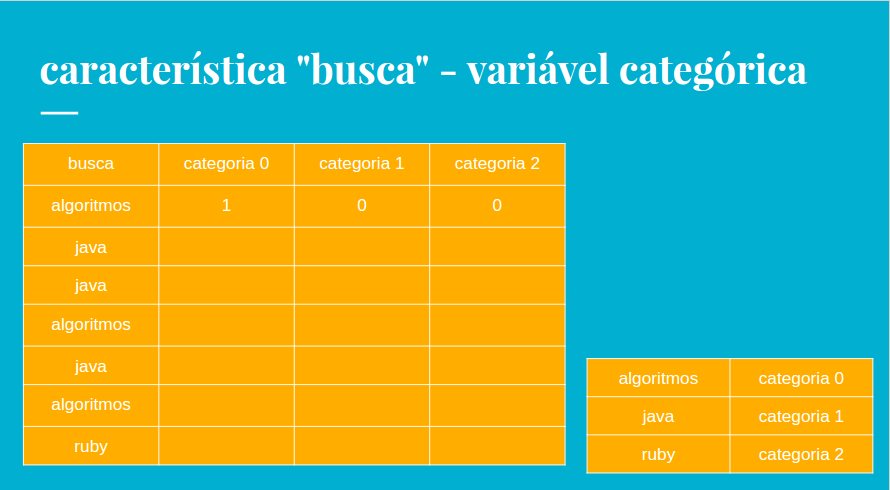
Mas e na prática? Como poderíamos montar a nossa tabela para cada uma dessas categorias? Vejamos um modelo:



Já tem uma ideia de como podemos preencher essa tabela?

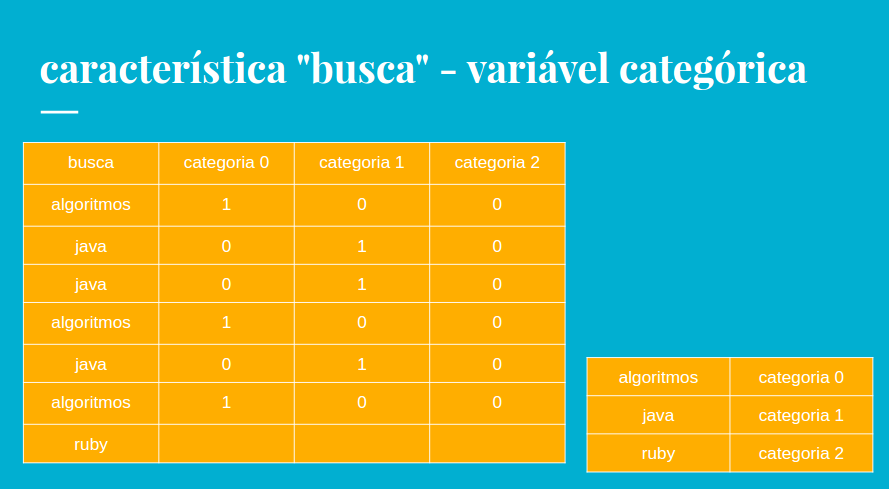
Classificando variáveis categóricas

Vamos pegar o primeiro exemplo de um cliente que buscou por "algoritmos":

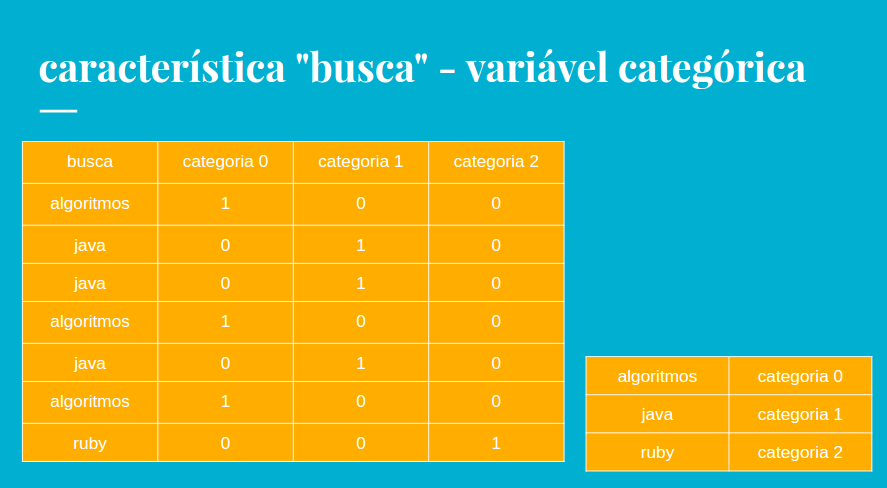


Já que ele buscou por algoritmos, significa que ele pesquisou pela "categoria 0", ou seja, marcamos a coluna "categoria 0" com 1 e as demais como 0. E para o cara que pesquisou sobre Java? Como marcamos?

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). Então vamos preencher todos as buscas de "algoritmos" e "Java":



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



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 3 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:



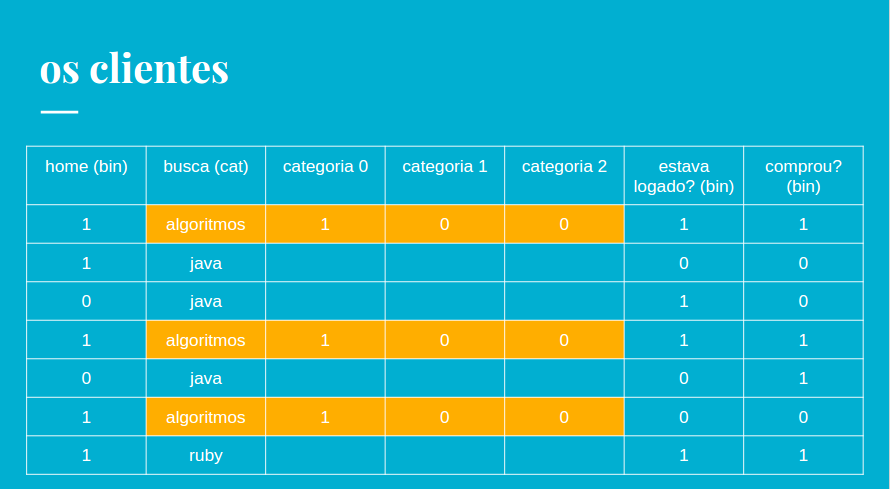
Observe que todas as categorias, exceto a busca, são do tipo bin que significa binário(0 ou 1) e a coluna busca é do tipo cat que significa uma variável categórica, ou seja, que possui categorias!

Esses dados que nós temos atualmente, já estão prontos para passarmos para um algoritmo de machine learning? Ou existe alguma dessas colunas que precisamos traduzir? Nesse caso, a variável categórica precisa ser trabalhada! O que devemos fazer então? Primeiro precisamos converter essa única coluna para as 3 possíveis categorias:

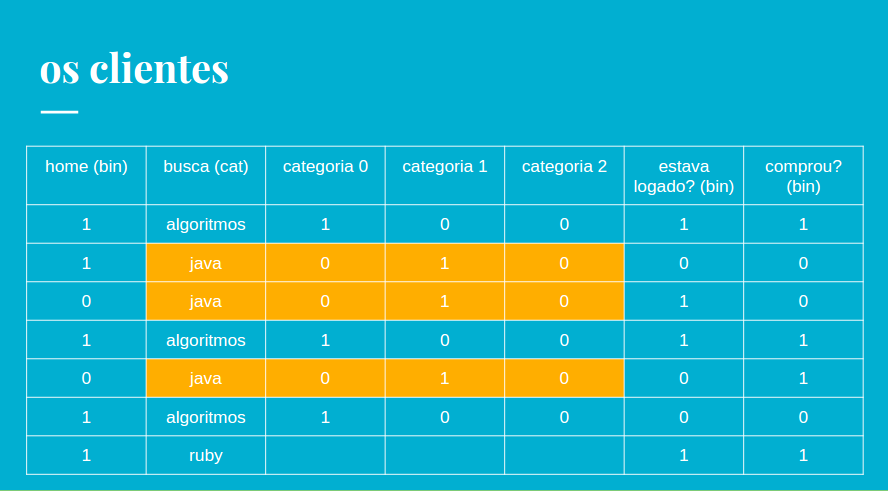


Então encaixamos essas novas colunas dentro da nossa tabela original:

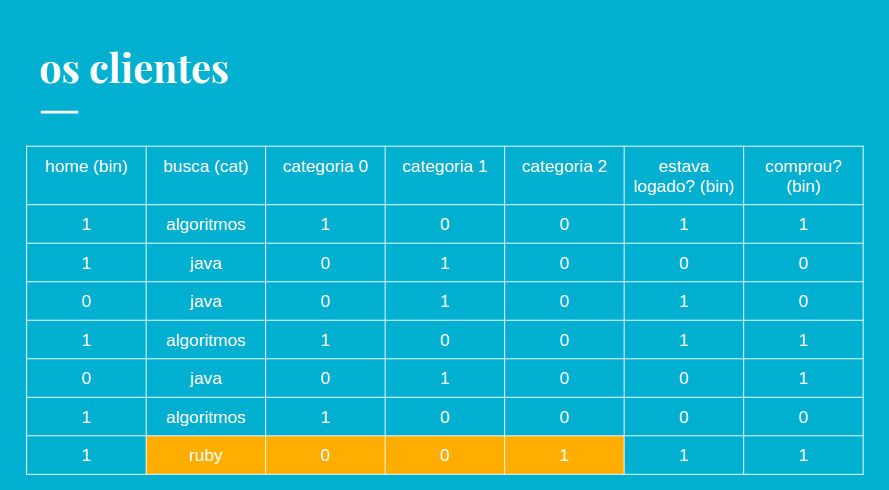
Agora nós precisamos preencher cada uma dessas categorias. Lembra que para busca de algoritmos adicionávamos na categoria 0? Então fica 1, 0, 0:



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



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



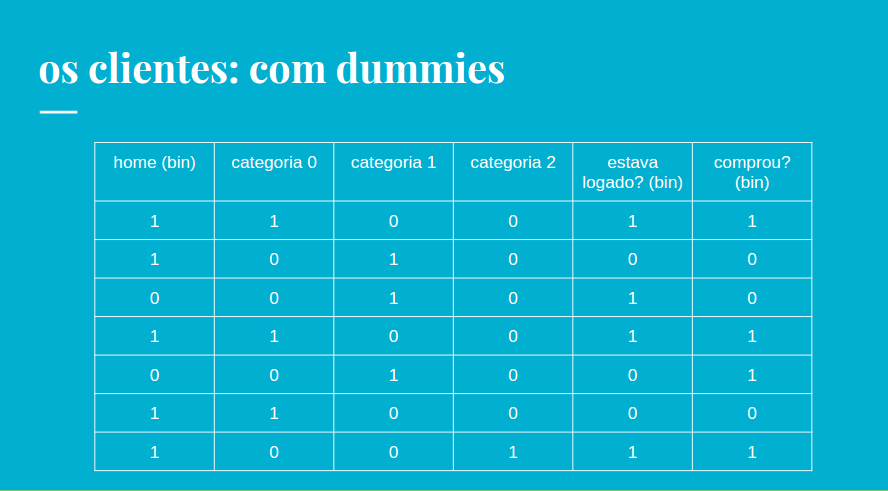
Agora que conseguimos transformar todas as colunas em tipos binários, não precisamos mais da coluna busca, ou seja, a variável categórica. Então jogamos fora essa coluna e a nossa tabela atual mantém os seguintes dados:



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! Aprendendo a partir desses dados e tentando prevê-los para o futuro. Note também que a partir das 5 variáveis iniciais, iremos prever a sexta variável 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 elas estavam presentes de uma maneira diferente!

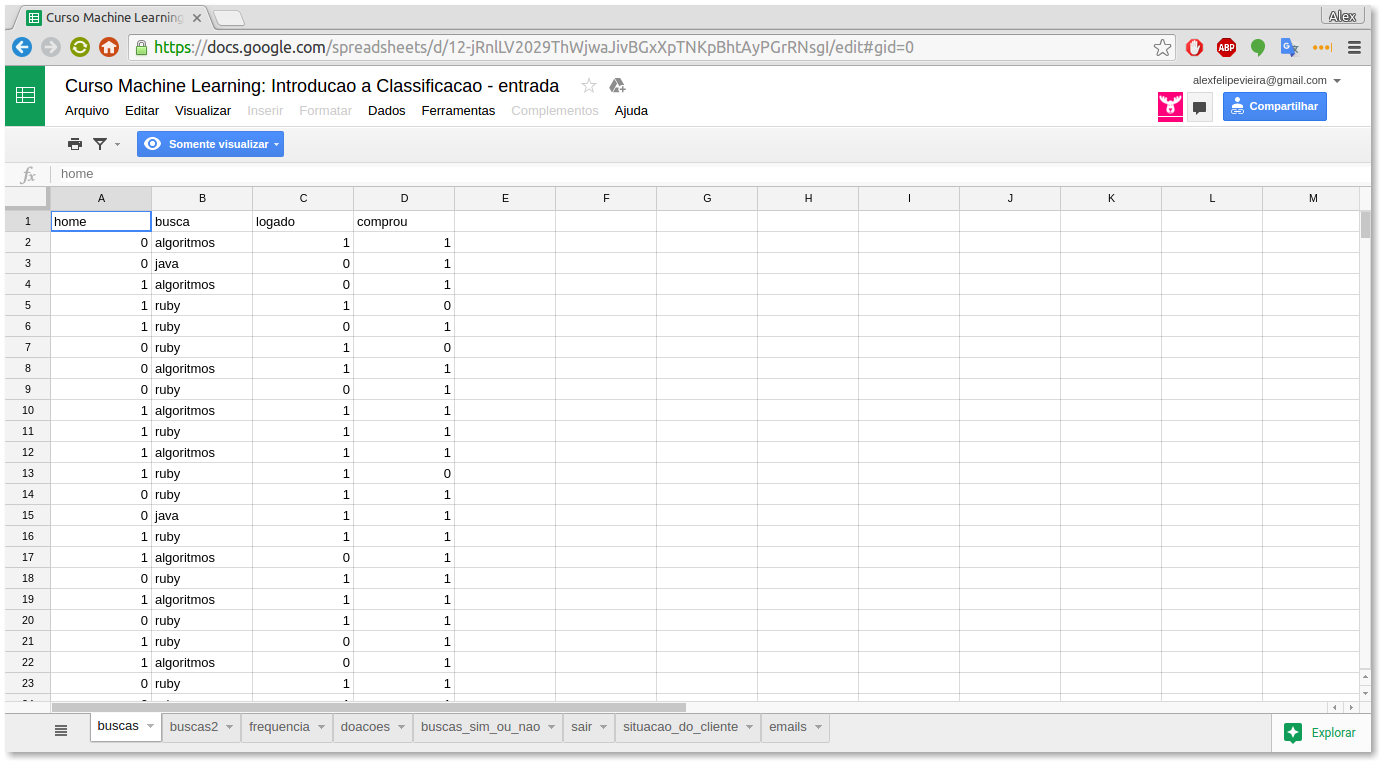
Isso significa que elas foram abordadas e questionadas de uma maneira, porém preenchemos de outra! Chamamos esse tipo de variáveis de *dummies*.



O objeto desse tipo de variável é justamente preencher um espaço de uma outra pergunta que fizemos, ou seja, as variáveis de categorias são os nossos *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.

Exportando um arquivo .CSV

Temos aqui uma planilha chamada “Curso Machine Learning\_ Introducao a Classificacao - entrada - buscas” que contém diversos dados:



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 ou 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, de alguma forma, transportar esses dados para um formato que o nosso algoritmo saiba trabalhar.

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

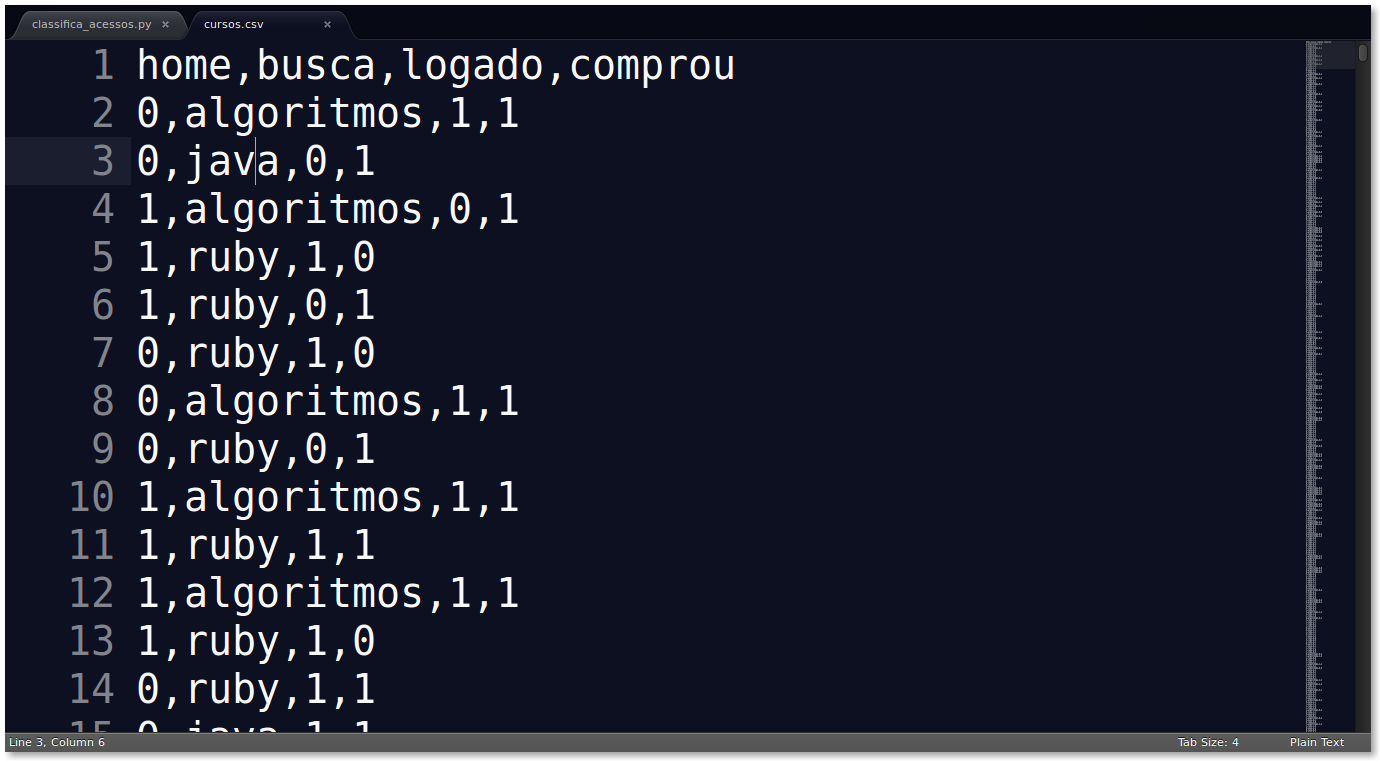
Se entrarmos no menu "Arquivo > Fazer download como" do Google Spreadsheets, veremos que ele nos fornece diversas formas para exportarmos esses arquivos, como por exemplo, no formato .xlsx para trabalharmos com o Excel, porém nós não estamos trabalhando com o Excel! Estamos trabalhando com o python.

Então qual era o tipo de arquivo que trabalhamos no python? Lembra que era o arquivo CSV? Aquele que separava cada valor por vírgula. Então, iremos importar 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 irá fazer isso para a página atual, mas o que isso significa exatamente? Se você der uma olhada na planilha:

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

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



Veja que o arquivo foi importado sem nenhum problema.

Lendo um arquivo .CSV e classificando as buscas

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 utilizar 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, ou seja, se está lendo cursos.csv, ele se chamará classifica\_cursos.py.

Porém, é valido pensar se o nome dos nossos arquivos são coesos, pois o arquivo cursos.csvrepresenta 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.

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 4 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', 'rb')

leitor = csv.reader(arquivo)

leitor.next()

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-lo, pois os procedimentos serão identicos 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, possuem também palavras! Ou seja, não podemos reutilizar essa mesma função. Então iremos criar uma nova função chamada carregar\_buscas:

def carregar\_buscas():

Essa função será muito similar ao carregar\_acessos, porém ela será utilizada para ler e tratar os dados do arquivo buscas.csv da forma correta:

def carregar\_buscas():

X = []

Y = []

arquivo = open('buscas.csv', 'rb')

leitor = csv.reader(arquivo)

leitor.next()

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

def carregar\_buscas():

X = []

Y = []

arquivo = open('buscas.csv', 'rb')

leitor = csv.reader(arquivo)

leitor.next()

for home,busca,logado,comprou in leitor:

Agora 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)

Repare que 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 Y e a nossa função carregar\_buscas() fica da seguinte maneira:

def carregar\_buscas():

X = []

Y = []

arquivo = open('buscas.csv', 'rb')

leitor = csv.reader(arquivo)

leitor.next()

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 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:

> python 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:

> python classifica\_buscas.py

[1, 1, 1, 0, ...]

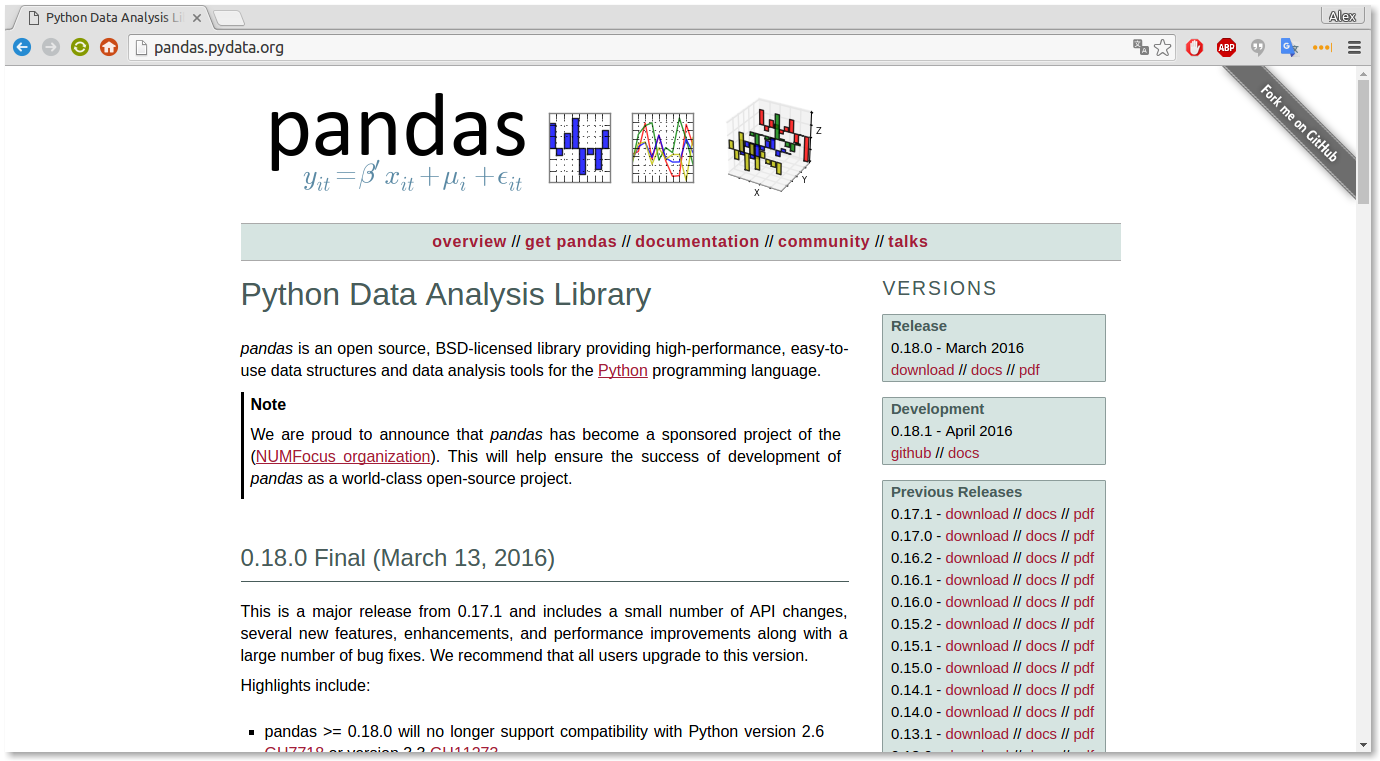
Os valores do Y também estão corretos! Porém, todos os nossos dados estão corretos? Ou seja, valores binários (0 ou 1). Observe que ainda estamos lidando com uma variável do tipo *string*e o nosso algoritmo não sabe lidar com esse tipo de variável!

Lembra que a coluna busca é uma variável categórica? Isso significa que precisamos convertê-la para 3 colunas, ou seja, as 3 categorias possíveis para essa coluna.

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

Traduzindo as categorias com pandas

Acessando a página do [pandas](http://pandas.pydata.org/) vc encontra toda documentação necessária para fazer um primeiro contato com a mesma:



Para instalar o pandas, poderíamos entrar no link "get pandas", e seguir as instruções, porém, já que temos o pip, iremos utilizá-lo para instalar o pandas por meio do comando:

> sudo pip install pandas

Caso você já tenha o pandas, atualize a biblioteca utilizando o comando:

> sudo pip install pandas --upgrade

É importante manter a versão mais atualizada, pois, dependendo da API que utilizaremos, pode existir variações que podem causar incompatibilidade ou simplesmente não funcionar conforme o esperado por ser depreciado. **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. Se observamos as nossas funções que realizam a leitura dos nossos dados:

import csv

def carregar\_acessos():

X = []

Y = []

arquivo = open('acesso.csv', 'rb')

leitor = csv.reader(arquivo)

leitor.next()

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', 'rb')

leitor = csv.reader(arquivo)

leitor.next()

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 analise de dados. Porém, para analisarmos os dados, a primeira coisa que precisamos fazer é justamente a leitura dos dados.

Temos diversas formas para ler dados, atualmente estavamos utilizando a biblioteca padrão csv ou então, podemos utilizar uma biblioteca específica para analise de dados, ou seja, podemos 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 o pandas? Podemos utilizar 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, 1000 linhas.

Na realidade, ao invés de devolver apenas um array puro e 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, é chamado de data frame, ou seja, ao invés 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 utilizarmos bibliotecas específicas que já resolvem a mesma situação, porém, 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 de trás dos panos, porém, agora que já aprendemos todo o processo para a leitura desses tipos de dados, podemos delegar toda essa tarefa para alguém que já faça tudo isso por nós e de uma maneira bem mais inteligente, nesse caso o pandas!

Lemos o arquivo utilizando o pandas, porém ainda não resolvemos o grande desafio que é solucionar a coluna busca. Lembra que, quando nós fazíamos a leitura dos nossos dados, separávamos todos os dados que representavam o X e o Y. Mas e agora? Como faremos isso utilizando 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:

> python classifica\_buscas.py

Traceback (most recent call last):

File "classifica\_buscas.py", line 3, in <module>

print(df[0])

File "/usr/local/lib/python2.7/dist-packages/pandas/core/frame.py", line 1992, in \_\_getitem\_\_

return self.\_getitem\_column(key)

File "/usr/local/lib/python2.7/dist-packages/pandas/core/frame.py", line 1999, in \_getitem\_column

return self.\_get\_item\_cache(key)

File "/usr/local/lib/python2.7/dist-packages/pandas/core/generic.py", line 1345, in \_get\_item\_cache

values = self.\_data.get(item)

File "/usr/local/lib/python2.7/dist-packages/pandas/core/internals.py", line 3225, in get

loc = self.items.get\_loc(item)

File "/usr/local/lib/python2.7/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 uma caquinha aí! Que tal pedirmos para ele utilizando 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:

> python 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 qual é o nome desses dados e o tipo dele, nesse 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:

> python 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/python2.7/dist-packages/pandas/core/frame.py", line 1992, in \_\_getitem\_\_

return self.\_getitem\_column(key)

File "/usr/local/lib/python2.7/dist-packages/pandas/core/frame.py", line 1999, in \_getitem\_column

return self.\_get\_item\_cache(key)

File "/usr/local/lib/python2.7/dist-packages/pandas/core/generic.py", line 1345, in \_get\_item\_cache

values = self.\_data.get(item)

File "/usr/local/lib/python2.7/dist-packages/pandas/core/internals.py", line 3225, in get

loc = self.items.get\_loc(item)

File "/usr/local/lib/python2.7/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: ('home', 'busca', 'logado')

Eita, mais uma caca... 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 utilizando o pandas, é que devemos colocar todas as colunas 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:

> python 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, porém, ainda existe um pequeno detalhe... Analisando a coluna busca, vemos que os valores ainda são as variáveis categóricas! Lembra que precisamos utilizar as categorias da coluna busca? Ou seja, precisamos pegar os dummies dessa coluna. 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:

> python classifica\_buscas.py

home logado busca\_algoritmos busca\_java busca\_ruby

0 0 1 1.0 0.0 0.0

1 0 0 0.0 1.0 0.0

2 1 0 1.0 0.0 0.0

3 1 1 0.0 0.0 1.0

4 1 0 0.0 0.0 1.0

5 0 1 0.0 0.0 1.0

6 0 1 1.0 0.0 0.0

...

999 0 1 0.0 0.0 1.0

[1000 rows x 5 columns]

Conseguimos os dummies do X, mas perceba que os valores vieram como ponto flutuante, não queremos números flutuantes e sim inteiros! Para convertemos os valores do data frame para inteiro, basta utilizar o método astype()passando int por parâmetro:

import pandas as pd

df = pd.read\_csv('buscas.csv')

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

Xdummies = pd.get\_dummies(X).astype(int)

print(Xdummies)

Rodando novamente o nosso código:

python 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]

Mas e o Y? O que fazemos com ele? Precisamos pegar os dummies do Y sendo que a coluna comprou é uma coluna com valores binários? A resposta é não, pois o próprio Y já possui os valores que precisamos para o nosso algoritmo, ou seja, ele por si só já é um dummie:

import pandas as pd

df = pd.read\_csv('buscas.csv')

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

Y = df['comprou']

Xdummies = pd.get\_dummies(X).astype(int)

Ydummies = Y

print(Xdummies)

Vejamos o resultado:

> python classifica\_buscas.py

0 1

1 1

2 1

3 0

4 1

5 0

6 1

...

999 0

Name: comprou, dtype: int64

Ele é impresso conforme o esperado! Porém, vamos dar uma olhada, novamente, como o algoritmo naive bayes MultinomialNB funciona:

treino\_dados = X[:90]

treino\_marcacoes = Y[:90]

from sklearn.naive\_bayes import MultinomialNB

modelo = MultinomialNB()

modelo.fit(treino\_dados, treino\_marcacoes)

# restante do código

Observe que o algoritmo MultinomialNB recebe 2 arrays do python, ou seja, ele não recebe data frames e sim lista de dados! Então teremos que, de alguma forma, transforma 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).astype(int)

Ydummies\_df = Y

Note que X, Xdummies e Y, Ydummies são nomes ruins, pois não deixa claro se estamos trabalhando com data frames ou arrays! É sempre importante deixarmos bem claro o significado das nossas variáveis, por isso, alteraremos o 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).astype(int)

Ydummies\_df = Y

Agora nós precisamos pegar os 2 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 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).astype(int)

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 imprimeros o X:

# restante do código

print(X)

Resultado:

> python 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 o Y:

# restante do código

print(Y)

Resultado:

python classifica\_buscas.py

[1 1 1 0 1 0 1 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 0 0 1 0 1

1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 1 1

1 0 1 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 1 0 1 1 1 0 1 1 0 0 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 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0

1 0 0 1 0 1 1 1 1 0 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1 0 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 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0

0 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 0 1 0 1 1 0 0 1 0 1 1 0 1 1 1 1

1 0 0 1 1 1 1 0 0 1 1 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0

1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1

1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 1 1 1 1 1

0 1 1 1 1 1 1 1 1 1 1 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

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 0 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 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1

0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 1

1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 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 0 0 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 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 0 0 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 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1

1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 1 0 1

1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 1 1 1 1 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 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 1 0 0

0 1 1 1 1 0 1 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 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 0 1 1 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 0 1 1 1 0 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 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]

Conseguimos pegar os valores do nosso X e Y, ou seja, podemos utilizá-los para treinar e testar o nosso algoritmo.

Treinando e testando o 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 dados de teste, nesse exemplo utilizamos 90 primeiros dados para treino e 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).astype(int)

>>> Ydummies\_df = Y\_df

>>>

>>> X = Xdummies\_df.values

>>> Y = Ydummies\_df.values

>>>

>>> len(Y)

1000

Espera um pouco! Não temos apenas 99 linhas e sim 1000 linhas... 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).astype(int)

>>> 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. Precisamos apenas pegar os dados e marcações de teste, como pegamos os dados de teste? 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 Ye 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 refere-se a nossa porcentagem:

porcentagem\_treino = 0.9

Dessa forma nós 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).astype(int)

>>> 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. Como fazemos isso? Lembra 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:]

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).astype(int)

>>> 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:]

>>>

>>> treino\_dados

array([[0, 1, 1, 0, 0],

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

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

...,

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

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

[1, 1, 0, 1, 0]])

>>> treino\_marcacoes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1, 1, 1])

>>>

Agora 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, 0, 1, 0, 0],

[0, 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, 0, 0, 1],

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

[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, 0, 1, 0],

[1, 1, 1, 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, 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],

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[0, 1, 0, 0, 1]])

>>>

>>> teste\_marcacoes array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 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, 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])

>>>

Conseguimos pegar tanto os dados de treino quantos os de teste, qual que é o próximo passo? De acordo com os nossos dados de treino e de teste, queremos criar o nosso modelo! Lembra 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 reutilizá-lo também! Por fim, vamos verificar como estão sendo feitos os calculos para a taxa de acerto:

diferencas = resultado - teste\_marcacoes

acertos = [d for d in diferencas 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** utilizado no arquivo classifica\_acessos.py para treinar e testar o modelo, calcular o resultado e a taxa de acerto não mudam! Ou seja, poderemos copiar todo o código e adicioná-lo no 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).astype(int)

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)

diferencas = resultado - teste\_marcacoes

acertos = [d for d in diferencas 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)

revisão

Vamos testar e verificar se está funcionando:

> python 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 3 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** que possa comparar com o nosso código. Se o resultado fosse 100%, com certeza não restaria dúvida que o algoritmo é perfeito, porém, o que precisamos saber por agora é quão longe o nosso algoritmo consegue chegar com dados reais.

Temos que levar em consideração que em cada área, haverão dados diferentes, ou seja, os resultados, provavelmente, serão diferentes, mas, o que precisamos focar é justamente em saber o quão bom é o nosso algoritmo para os determinados números que estamos trabalhando atualmente.

# Resumindo

Nesse capítulo nós realizamos um teste para verificar quais clientes iriam ou não comprar no meu site. Vimos que o nosso algoritmo acertou 82% utilizando dados reais, a partir desses dados podemos entrar em contato com os clientes que não irão comprar para tentar entender a necessidade deles ou propor alguma coisa que seja de seu interesse. Então podemos concluir que 82% das vezes acertamos se o cliente iria ou não comprar.

Além disso, nesse mesmo capítulo, fizemos alguns ajustes no nosso algoritmo, dessa vez utilizamos uma biblioteca nova, o pandas, para realizarmos a leitura dos nossos arquivos CSV que continham os nossos dados de uma forma bem simplificada e, além de realizar a leitura, ele nos devolveu um conjunto de informações bem mais completo e inteligente 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 por exemplo, pegar uma coluna ou diversas colunas, por meio de um(ns) cabeçalho(s), facilitando, e muito, a nossa vida.

Um outro ponto importante visto nesse 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 utilizamos no arquivo classifica\_acessos.py, ou seja, reutilizamos o mesmo código! Por fim, nós chegamos a mais uma questão muito importante que é verificar se o resultado do nosso algoritmo, para esse cenário, ou seja, esses dados, foi bom ou ruim. Veremos com mais detalhes essa questão no próximo capítulo.

***4 – o problema do algoritmo de sucesso e o algoritmo burro***

Como comparar algoritmo

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 82% 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 rodar ele 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. Porém, 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 formas que poderíamos abordar 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 2 tipos de características dentro do nosso sistema, que são:

* Usuário que compra.
* Usuário que não compra.

Pra esse caso, qual seria o algoritmo mais simples de todos para classificar esses elementos? Podemos utilizar um outro exemplo bem similar. Temos diversos animais entre porcos e cachorros e precisamos definir quais desses animais são porcos ou cachorros. Consegue imaginar no código, algoritmo, solução mais simples possível?

O algoritmo burro

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 esse 1º animal? Ele é um porco.
* E esse 2º animal? Ele é um porco.
* E agora esse 3º animal? Ele é um porco.
* E esse 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 pra classificar entre spam e não spam? Simples! Responderíamos apenas spam. Perceba que nem temos o trabalho de verificar as características, ou seja, para implementar um classificador mais simples e burro do mundo, basta responder **sempre** a mesma coisa.

Vimos o quão simples esse algoritmo é, porém, a questão é: "O quão eficiente esse algoritmo 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, ou seja, todos que forem 1 ele vai acertar e todos que forem 0 ele vai errar. Porém, o quão 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).astype(int)

>>> Ydummies\_df = Y\_df

>>>

>>> X = Xdummies\_df.values

>>> Y = Ydummies\_df.values

>>>

### No Python3

>>> 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 Y? Vejamos novamente:

>>> 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, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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, 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 irá chutar 1 para todos os elementos do Y. Quanto será que ele irá acerta? Podemos verificar a quantidade de uns 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 como verificamos quantos dados nós temos nos total? É simples, len do Y:

>>> len(Y)

1000

>>>

1000 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 pra todos os dados, consegue acertar 83,20% dos casos. E o nosso algoritmo anterior? Quanto ele acerta?

> python 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 acerta 82% sendo que bastava chutarmos "sim" para todos os casos que eu acertaria 83,20% das vezes? Aparentemente, a maneira que estamos rodando esse algoritmo não está tão boa com esses dados... Esses 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 de 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 que o nosso algoritmo de classificação que implementamos (o algoritmo complexo) tiver um resultado inferior ao algoritmo mais burro possível para classificação (que chuta o mesmo valor para todos os dados), o nosso algortimo é ruim, pois ele obteve um resultado inferior ao algoritmo que nem ao menos faz uma analise de dados. Isso significa que se utilizarmos 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 algoritmo que analisa os dados e tenta se adaptar de acordo com os novos dados que ele recebe. Então repare que no mundo real, podemos cair em duas situações:

* Quando o algoritmo que chuta o mesmo valor para todos os dados obter 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 algoritmo que chuta a mesma coisa pra todos significa que ele é pior, ou seja, utilize o algoritmo que chuta o mesmo valor pra tudo. A outra situação seria:

* Quando o algoritmo que implementamos, obtem um resultado melhor que o algoritmo que chuta a mesma coisa pra todos.

Nesse caso, podemos utilizar esse algoritmo, pois ele obteve um resultado melhor.

Observe que, para afirmarmos que o nosso algoritmo que implementamos é bom ou não, é imprescindível realizar uma comparação com um outro algoritmo. No nosso exemplo, inicialmente, utilizamos o algoritmo mais simples de todos, pois se o nosso algoritmo apresentar um resultado inferior ao algoritmo que não utiliza nenhum tipo de lógica e apenas chuta o mesmo valor, significa que o nosso algoritmo é realmente muito ruim. É válido lembrar que isso depende também dos dados que estão sendo utilizados, 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?

Escolhendo o algoritmo base

E se os nossos chutes fossem apenas 0? Qual resultado teríamos? Podemos pegar a quantidade de zeros por meio do tamanho do Ymenos a soma do Y:

>>> len(Y) - sum(Y)

168

168 é a quantidade total de zeros no nosso arquivo de dados, ou seja, 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, ao invés de 1 escolhêssemos 0, iríamos comparar 82% e 16,80%, isto é, sabendo que 82% é maior que 16,80% iríamos concluir que o nosso algoritmo é realmente é muito bom. Esse é um pequeno detalhe que precisamos tomar cuidado, pois, quando utilizamos esse algoritmo mais básico de todos, precisamos saber ambos os resultados, tanto escolhendo apenas o valor 1 quanto escolhendo apenas 0, em outras palavras, dado 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.

Mas como podemos implementar isso no nosso código?

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).astype(int)

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 iremos utilizar? A maior delas! Pois o nosso algoritmo base precisa utilizar o maior resultado para o mesmo chute. Porém, como podemos pegar a maior variável entre elas? Simples, no python, podemos utilizar a função max() enviando duas variáveis como parâmetro, então, 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 imprimir esse valor, lembra qual era o nome da variável que utilizamos para imprirmir 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 refere-se a 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:

> python 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, ou seja, nesse caso, deveria nos apresentar 83,20%. Como podemos fazer isso? É simples, basta dividirmos o valor do max() pelo total 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:

> python classifica\_buscas.py

Taxa de acerto base: 83

82.0

100

Observe que o resultado foi de 83, porém, não foi impresso o ponto fluante! 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:

> python 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

Calculando a quantidade de zeros e uns

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 lindando com o lene uma operação de subtração. Além dessa forma que fizemos, podemos também utilizar os próprios data frames para realizar esses calculos 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).astype(int)

>>> 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, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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, 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])

>>>

Sabemos que o nosso Y é uma sequência de zeros e uns. E se pedirmos todos os valores de Yque são iguais a 1? Será que da 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, True, False, True, False,

False, True, True, True, True, False, False, True, False,

True, True, True, True, True, True, True, False, True,

True, True, True, True, True, False, True, True, True,

True, True, True, True, True, True, False, True, True,

True, False, True, False, True, False, True, True, True,

True, True, True, False, True, True, True, True, True,

True, False, True, False, False, True, True, True, False,

True, True, True, True, False, True, True, True, True,

True, True, True, True, False, True, True, True, False,

False, True, True, True, True, True, True, True, True,

True, True, False, True, True, True, False, True, True,

False, False, True, True, True, True, True, True, True,

True, True, True, True, True, False, False, False, False,

True, True, True, True, False, False, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, False, True, False, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, False, True, False, True, False, False, True,

False, True, True, True, True, False, True, True, True,

True, False, True, False, False, True, True, True, True,

True, True, True, True, False, True, True, True, True,

True, True, True, True, False, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, False, True, True, True, True,

True, True, True, True, True, True, False, False, True,

True, False, True, True, True, False, True, True, True,

True, True, True, True, True, False, True, False, True,

True, False, True, False, True, True, False, False, True,

False, True, True, False, True, True, True, True, True,

False, False, True, True, True, True, False, False, True,

True, False, False, True, True, True, False, True, True,

True, True, True, True, True, True, True, True, True,

False, True, True, True, True, True, True, False, False,

True, True, True, True, True, True, False, True, False,

True, False, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, False, False, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, False, True, True, True, False, True,

True, True, True, False, True, True, False, True, True,

True, True, True, True, False, False, True, True, True,

True, True, False, True, True, True, True, True, True,

True, True, True, True, False, True, True, True, True,

True, True, True, True, True, True, True, True, False,

True, False, True, True, True, True, True, 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, True, True, True, True,

True, True, True, False, True, True, True, True, True,

False, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, False, True,

True, True, True, True, False, True, True, True, True,

True, True, False, True, True, False, True, True, True,

True, True, True, True, True, False, True, True, True,

True, True, True, False, True, True, True, False, True,

True, False, True, True, True, True, False, True, True,

False, True, True, True, False, True, True, True, False,

True, True, True, True, True, True, True, True, True,

True, True, False, False, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

False, True, True, True, True, True, True, True, True,

True, True, True, True, True, False, False, True, True,

True, True, True, True, False, True, False, True, True,

True, True, True, True, True, False, True, True, True,

True, True, True, True, True, True, False, True, True,

True, True, True, True, True, True, True, True, False,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, False, False, True, False, False, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, False, True, True, True, False,

False, True, True, False, True, True, True, True, True,

True, True, False, True, True, True, True, True, True,

False, True, True, True, True, True, True, False, True,

True, True, True, True, False, False, True, True, True,

True, True, True, True, True, True, True, False, True,

False, True, True, True, True, True, True, True, False,

True, True, True, True, True, True, False, False, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, False, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, False, True, False, True, True,

True, True, True, True, True, True, True, True, False,

True, True, True, True, True, True, True, False, False,

True, True, False, False, False, True, True, True, True,

False, True, False, False, True, True, True, False, True,

True, True, True, True, True, True, True, True, True,

True, False, True, True, True, True, True, True, False,

False, True, True, True, True, True, True, True, False,

False, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, False,

True, True, True, True, False, True, True, True, False,

True, True, True, True, True, True, True, True, True,

True, 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, True, True, False, False, False, True, True, False,

True, True, True, False, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, False, True, True, False, False, False, True, False,

False, True, True, True, False, True, True, True, True,

True, True, False, True, False, False, True, True, True, False], dtype=bool)

>>>

Observe que agora ele nos devolveu valores booleanos, ou seja, Trues e Falses. Isso significa que todos os Trues são os valores iguais a 1 e **qualquer valor diferente de 1** são os Falses. 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, False, True, False, False, False, False, False, False,

False, False, False, False, False, False, True, False, True,

True, False, False, False, False, True, True, False, True,

False, False, False, False, False, False, False, True, False,

False, False, False, False, False, True, False, False, False,

False, False, False, False, False, False, True, False, False,

False, True, False, True, False, True, False, False, False,

False, False, False, True, False, False, False, False, False,

False, True, False, True, True, False, False, False, True,

False, False, False, False, True, False, False, False, False,

False, False, False, False, True, False, False, False, True,

True, False, False, False, False, False, False, False, False,

False, False, True, False, False, False, True, False, False,

True, True, False, False, False, False, False, False, False,

False, False, False, False, False, True, True, True, True,

False, False, False, False, True, True, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, True, False, True, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, True, False, True, False, True, True, False,

True, False, False, False, False, True, False, False, False,

False, True, False, True, True, False, False, False, False,

False, False, False, False, True, False, False, False, False,

False, False, False, False, True, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, True, False, False, False, False,

False, False, False, False, False, False, True, True, False,

False, True, False, False, False, True, False, False, False,

False, False, False, False, False, True, False, True, False,

False, True, False, True, False, False, True, True, False,

True, False, False, True, False, False, False, False, False,

True, True, False, False, False, False, True, True, False,

False, True, True, False, False, False, True, False, False,

False, False, False, False, False, False, False, False, False,

True, False, False, False, False, False, False, True, True,

False, False, False, False, False, False, True, False, True,

False, True, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, True, True, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, True, False, False, False, True, False,

False, False, False, True, False, False, True, False, False,

False, False, False, False, True, True, False, False, False,

False, False, True, False, False, False, False, False, False,

False, False, False, False, True, False, False, False, False,

False, False, False, False, False, False, False, False, True,

False, True, False, False, False, False, False, False, False,

False, False, False, True, False, False, False, False, False,

False, False, False, False, False, False, True, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, True, False, False, False, False, False,

True, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, True, False,

False, False, False, False, True, False, False, False, False,

False, False, True, False, False, True, False, False, False,

False, False, False, False, False, True, False, False, False,

False, False, False, True, False, False, False, True, False,

False, True, False, False, False, False, True, False, False,

True, False, False, False, True, False, False, False, True,

False, False, False, False, False, False, False, False, False,

False, False, True, True, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

True, False, False, False, False, False, False, False, False,

False, False, False, False, False, True, True, False, False,

False, False, False, False, True, False, True, False, False,

False, False, False, False, False, True, False, False, False,

False, False, False, False, False, False, True, False, False,

False, False, False, False, False, False, False, False, True,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, True, True, False, True, True, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, True, False, False, False, True,

True, False, False, True, False, False, False, False, False,

False, False, True, False, False, False, False, False, False,

True, False, False, False, False, False, False, True, False,

False, False, False, False, True, True, False, False, False,

False, False, False, False, False, False, False, True, False,

True, False, False, False, False, False, False, False, True,

False, False, False, False, False, False, True, True, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, True, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, True, False, True, False, False,

False, False, False, False, False, False, False, False, True,

False, False, False, False, False, False, False, True, True,

False, False, True, True, True, False, False, False, False,

True, False, True, True, False, False, False, True, False,

False, False, False, False, False, False, False, False, False,

False, True, False, False, False, False, False, False, True,

True, False, False, False, False, False, False, False, True,

True, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, True,

False, False, False, False, True, False, False, False, True,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, True, False, False,

False, True, False, False, False, False, False, False, False,

False, False, False, True, True, True, False, False, True,

False, False, False, True, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, False, False, False, False, False, False, False, False,

False, True, False, False, True, True, True, False, True,

True, False, False, False, True, False, False, False, False,

False, False, True, False, True, True, False, False, False, True], dtype=bool)

>>>

Observe que agora todos aqueles Trues 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, ou seja, todos os Trues acima 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 ao data frame:

>>> 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 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 lendeles, 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 substituírmos a nossa implementação, que calcula a quantidade de zeros e uns, por essa que é uma forma mais comum para esse tipo de calculo:

# restante do código

acerto\_de\_um = len(Y[Y==1])

acerto\_de\_zero = len(Y[Y==0])

É comum utilizarmos essa forma, pois, a 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 todos 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, 1, 0, 0, 1],

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[0, 1, 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 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. Então repara que o pandas é muito mais além do que o que vimos até agora, ele nos da diversas possibilidades para trabalharmos com dados, porém, veremos todas as suas funcionalidades de acordo com a necessidade que tivermos.

Lidando com valores sim e não

Até agora, quando analisamos os nossos dados, todas as nossas marcações, coluna comprouforam 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

Porém, nem sempre recebemos as informações com binários, e sim, com valores como: **sim e não** ou **true e false**, pois 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**. Então 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](https://raw.githubusercontent.com/alura-cursos/machine-learning-introducao-a-classificacao/master/busca.csv). Ao baixar o arquivo, salve-o no mesmo diretório onde os arquivo python estão. Lembre-se, caso o arquivo tiver o nome diferente de buscas.csv, altere o código que lê o arquivo CSV para ler esse novo arquivo baixado.

Já sabemos o que fizemos 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 3: algoritmos, Java e Ruby. Mas o que vai acontecer com o Y se temos apenas sim e nao? Precisamos testar, certo? Então 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).astype(int)

>>> 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',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'nao',

'nao', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim', 'nao',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim',

'sim', 'nao', 'sim', 'nao', 'sim', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'nao', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'nao',

'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim',

'nao', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'nao', 'nao',

'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'sim', 'nao', 'sim', 'nao', 'nao', 'sim',

'nao', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim',

'sim', 'nao', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim',

'sim', 'nao', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'nao', 'sim',

'sim', 'nao', 'sim', 'nao', 'sim', 'sim', 'nao', 'nao', 'sim',

'nao', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim',

'nao', 'nao', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim',

'sim', 'nao', 'nao', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'nao',

'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'nao', 'sim',

'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao',

'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim',

'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'nao', 'sim',

'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim',

'nao', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'nao', 'sim', 'nao', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'nao',

'nao', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim',

'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'nao', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'nao',

'sim', 'sim', 'nao', 'nao', 'nao', 'sim', 'sim', 'sim', 'sim',

'nao', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim', 'nao', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao',

'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao',

'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim',

'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'nao', 'nao', 'nao', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'nao', 'sim', 'sim', 'nao', 'nao', 'nao', 'sim', 'nao',

'nao', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim', 'nao'], dtype=object)

>>>

Os valores do nosso Y são sims e naos, por enquanto não tem nenhum problema. Porém e esse 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\_zero) / 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 existe mais valores entre zeros e uns, ou seja, agora os valores são sims e naos! Porém, isso é bem fácil de corrigir! Basta apenas alterarmos a comparação de uns para sims:

# restante do código

acerto\_de\_um = len(Y[Y=='sim'])

E a comparação de zeros para naos:

# 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. E essa parte 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:]

### No Python3...

No Python3, precisamos converter o resultado da multiplicação para um número inteiro, do contrário não conseguimos pegar os valores no nosso array.

# 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:]

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. Mas, e no momento que utilizamos o nosso algoritmo MultinomialNB?

from sklearn.naive\_bayes import MultinomialNB

modelo = MultinomialNB()

modelo.fit(treino\_dados, treino\_marcacoes)

Em específico no momento em que pedidos para ele treinar, ou seja, o método fit, 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>

diferencas = resultado - teste\_marcacoes

TypeError: unsupported operand type(s) for -: 'str' and 'str'

Perceba que ele conseguiu treinar com as informações de sims e naos, porém deu um erro justamente na linha:

# restante do código

diferencas = resultado - teste\_marcacoes

Mas 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', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', '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', 'nao', 'sim', 'sim',

'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'nao', 'nao', 'nao', 'sim', 'sim', 'nao',

'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim', 'sim',

'sim', 'nao', 'sim', 'sim', 'nao', 'nao', 'nao', 'sim', 'nao',

'nao', 'sim', 'sim', 'sim', 'nao', 'sim', 'sim', 'sim', 'sim',

'sim', 'sim', 'nao', 'sim', 'nao', 'nao', 'sim', 'sim', 'sim', 'nao'], dtype=object)

Observe que ambas as variáveis, nesse instante, possuem valores do tipo string, ou seja, 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. Então como poderíamos resolver esse problema? Existem duas abordagens que podemos realizar:

* Transformar todos os sims e naos em zeros e uns.
* Comparar as duas variáveis (resultadoe 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, True, True,

True, True, True, True, True, True, False, True, True,

True, False, True, True, True, True, True, True, True,

True, True, True, False, False, False, True, True, False,

True, True, True, False, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, False, True, True, False, False, False, True, False,

False, True, True, True, False, True, True, True, True,

True, True, False, True, False, False, True, True, True, False], dtype=bool)

>>>

Veja que o python nos devolveu um array de Trues e Falses, isto é, quais são os valores sim e quais não são, nesse caso, valores iguais a nao. E como mudaríamos 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:

>>> diferencas = resultado == teste\_marcacoes

>>> diferencas

array([ True, 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, True, True, False, False, False, True, True, False,

True, True, True, False, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, True, True, True, True, True, True, True, True,

True, False, True, True, False, False, False, True, False,

False, True, True, True, False, True, True, True, True,

True, True, False, True, False, False, True, True, True, 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 diferencas if d == 0]

total\_de\_acertos = len(acertos)

total\_de\_elementos = len(teste\_dados)

A variável total\_de\_acertos é a quantidade de Trues existentes na variável diferencas. 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 diferencas teremos a quantidade total de Trues:

>>> sum(diferencas)

82

Por meio dessa abordagem, nem precisamos mais da variável acertos ou fazer qualquer tipo de iteração, basta apenas, no lugar de len(acertos) adicionarmos a soma da variável diferencas:

# 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, ela 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 calculo 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).astype(int)

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 = 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)

### No Python3

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 do algoritmo: %f" % taxa\_de\_acerto)

print(total\_de\_elementos)

Vamos exercutar o arquivo e verificar qual é o resultado:

> python 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, porém, dessa vez, com valores do tipo string entre sim e nao nas marcacoes, neste caso, na coluna comprou.

## Utilizando collections do python

Atualmente, se recebermos um arquivo que não esteja com valores binários e sim com apenas dois valores distintos que sejam texto (strings), como por exemplo, sim e nao, podemos também utilizar o nosso algoritmo que suporta esse tipo de dado. Porém, um detalhe **muito importante**, para que o nosso algoritmo consiga suportar valores do tipo string, precisaremos **sempre nos atentar** no 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 à variáveis do tipo string, ainda sim estamos vinculados com os valores delas, ou melhor, sim e nao, caso os valores sejam distintos de sim e nao, será necessários modificar esse trecho de código.

Além de utilizar apenas o array do python, como por exemplo o nosso X e Y, podemos também utilizar 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).astype(int)

>>> 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 por exemplo, a função count que nos 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 sims ou naos existentes dentro dela:

>>> list(Y).count('sim')

832

>>> list(Y).count('nao')

168

>>>

Então, ao invés de utilizar o a função lendiretamente com os arrays para pegar a quantidade de sims e naos, poderíamos utilizar 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 fosse capaz de contar todos os elementos do nosso array, e também, nos informe quais são esses elementos! 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 sims e naos como um dicionário, ou seja, para cada valor distinto que ele encontra, ele nos informa a quantidade que encontrou! Mas e agora? Como podemos utilizar o Counter no nosso código? Primeiro precisamos iterar sobre todos os valores do Counter, para isso utilizamos a função itervalues():

>>> Counter(Y).itervalues()

<dictionary-valueiterator object at 0x7fb26bdccd08>

No Python3, utilizamos a função values():

>>> Counter(Y).values()

<dictionary-valueiterator object at 0x7fb26bdccd08>

Veja que ele retorna um objeto iterador. Agora, basta apenas pedirmos o maior valor por meio da função max:

>>> Counter(Y).itervalues()

<dictionary-valueiterator object at 0x7fb26bdccd08>

>>> max(Counter(Y).itervalues())

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 sims e naos, em outras palavras, poderia ser outros textos diferentes de sims e naos, como por exemplo yes ou notque funcionaria da mesma forma! Porém, ainda falta modificarmos o nosso código. Começaremos importando o Counter para o nosso código:

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).astype(int)

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, precisamos pegar os valores do Counter, pedido os itervalues();

# restante do código

Counter(Y).itervalues()

No Python3, usamos a função 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).itervalues())

Vejamos agora o restante do código:

acerto\_base = max(Counter(Y).itervalues())

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).itervalues())) e estamos retornando para a variável acerto\_base, ou seja, podemos simplesmente excluir as variaveis acerto\_de\_um e acerto\_de\_zero e no calculo de maior entre elas adicionamos a variável acerto\_base. Então o calculo da taxa de acerto base fica da seguinte forma:

# restante do código

acerto\_base = max(Counter(Y).itervalues())

taxa\_de\_acerto\_base = 100.0 \* acerto\_base / len(Y)

O nosso arquivo 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).astype(int)

Ydummies\_df = Y\_df

X = Xdummies\_df.values

Y = Ydummies\_df.values

acerto\_base = max(Counter(Y).itervalues())

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)

E no Python3:

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 = 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 do algoritmo: %f" % taxa\_de\_acerto)

print(total\_de\_elementos)

Vamos testar novamente o nosso arquivo classifica\_buscas.py:

> python 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, em outras palavras, ele consegue contar a quantidade de dados e nos retorna o maior, independentemente de qual informação esteja contida dentro do arquivo de dados.

Resumo

Nesse capítulo iniciamos a questão do sucesso do algoritmo, isto é, dado o seu resultado concluir se foi bom ou ruim. Vimos que para sabermos se o nosso resultado foi sucesso ou fracasso de verdade, precisamos comparar o nosso algoritmo. Uma das possíveis formas é compararmos com um algoritmo base, ou seja, aquele algoritmo que chuta o mesmo valor para cada elemento independentemente de suas características, como por exemplo, se os valores forem entre uns e zeros, ele pode chutar apenas 1, ou então, apenas 0.

Se realizarmos o teste, e o nosso algoritmo for pior do que o algoritmo base, podemos concluir que o nosso algoritmo não é tão bom quanto imaginamos. Podemos utilizar esse método como o primeiro critério de avaliação para identificar se o nosso 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 ao algoritmo que chuta apenas sim ou apenas não. Observe 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? Então repara 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. Se o nosso algoritmo for melhor do que o algoritmo base, a princípio podemos concluir que ele é bom, porém, pode ser que ele também preveja muitos alunos que não terão dificuldades, em outras palavras, provavelmente gastaremos mais tempo dos professores com alunos que não tenham 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 perceba que você também precisa avaliar o resultado e saber dizer se 51% ou 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, porém, dessa vez, vimos que podemos também trabalhar com palavras, isto é, sims e naos. Mas, tivemos que realizar uma refatoração no nosso código para que ele fosse o mais genérico possível, em outras palavras, aceitasse quaisquer tipos de valores, independemente se fossem 0s e 1s, sims e naos ou qualquer outro valor. Para isso utilizamos APIs mais genéricas do python, como no caso o Counter.

Por fim, vimos que, quando queremos comparar 2 algoritmos, é de extrema importância **utilizarmos os mesmos dados**, ou seja, se foram utilizados os dados **X** para treinar e testar o nosso algoritmo, teremos que utilizar esses mesmos dados **X** para o nosso algoritmo base. Lembre-se também que o algoritmo base não exige nenhum tipo de treino, pois ele simplesmente chuta o mesmo valor para todos os elementos.

Utilizando os mesmos dados no algoritmos

Será que no nosso algoritmo utilizamos 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).itervalues())

taxa\_de\_acerto\_base = 100.0 \* acerto\_base / len(Y)

* Nosso 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 algortimo base estamos utilizando o Y, ou seja, 100% dos dados, enquanto que no algoritmo que implementamos, estámos utilizando uma parte do Y, nesse caso, 90%. Será que esse teste está sendo justo? Com certeza não! Lembre-se que, quando precisamos comparar 2 algoritmos, **precisamos sempre utilizar os mesmos dados**, em outra palavras, se utilizarmos 90% para o nosso algoritmo, precisamos utilizar os mesmos 90% para o algoritmo base, e se forem 90% aleatórios? Os mesmo 90% aleatórios precisará ser utilizado 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 esse trecho de código:

acerto\_base = max(Counter(Y).itervalues())

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).itervalues())

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 Yutilizaremos a variável teste\_marcacoes que refere-se aos 90% do Y utilizados no nosso algoritmo. Então 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).astype(int)

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).itervalues())

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:

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 82.000000

100

Taxa de acerto base: 82.000000

Repare que, utilizando exatamente os mesmo 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 comparação entre 2 algoritmos.

5 – naive bayes e maxium a posteriori por tras dos panos

O algoritmo multinomial naive bayes

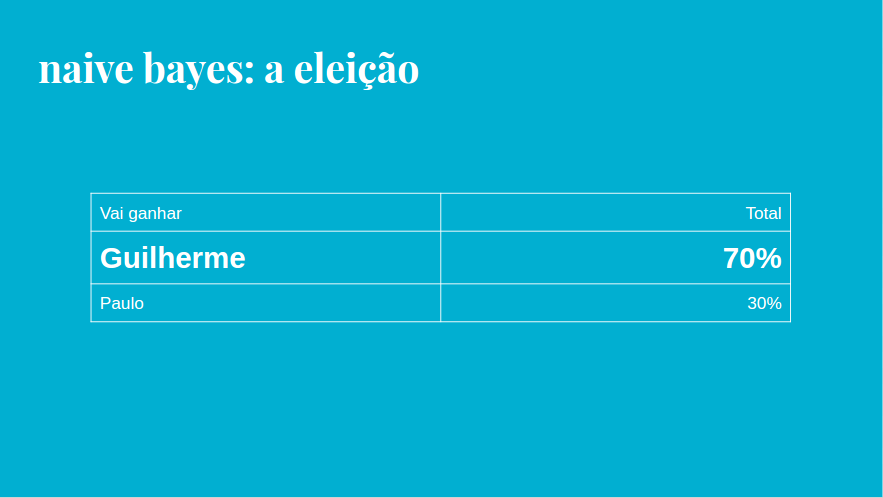
Até agora utilizamos apenas o algoritmo MultinomialNB (Naive Bayes). Como será que esse algoritmo funciona por de trás dos panos? Em outras palavras, dado um conjunto de dados que separamos entre treino e teste, como será que ele faz para treinar a partir desses dados? Ou então, como o nosso cérebro faz pra dizer se é um porco ou cachorro? Ou se o e-mail é *spam* ou não *spam*. Mas porque 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 envolve 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, entre o Guilherme e o Paulo, que iriam ganhar, os resultados foram:



Observe que 70 pessoas acreditam que o Guilherme vai ganhar e 30 pessoas acham que é o Paulo. Nesse exemplo, temos 70 pessoas de 100, que dizem que o Guilherme vai ganhar. Levando essa informação em consideração, se perguntarmos a uma pessoa quem vai ganhar, o que será que ela responderá? A probabilidade para acharem que seja o Guilherme será de 70% e o Paulo 30%. Perceba que a pergunta não tem o objetivo de saber quem vai ganhar e sim, explorar a probabilidade para as duas opções.

Se escolhermos uma pessoa aleatória e perguntarmos quem vai ganhar, utilizando 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 (treinado com esses dados)? E então? Ela responde Guilherme ou Paulo? Qual foi a regra de decisão que você utilizou para responder entre o Guilherme e o Paulo?

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 porque o de 70%?



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, pois a probabilidade é maior. 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, devido a chance de erro ser 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, que escolhe entre a maior ou a menor probababilidade, 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 utilizando apenas uma informação para decidir quem será ou não votado, ou melhor, quando consideramos apenas o maior dentre as porcentagens, não conseguimos distinguir **70% e 30%** de **99% e 1%** ou **51% e 49%**, ou seja, sempre escolheremos o maior independemente se existe alguma chance ou não para o de menor probabilidade. Repare que restringir a nossa decisão não parece uma boa ideia, ou melhor, podemos melhorar a forma que decidimos qual será o voto escolhido entre o Guilherme e o Paulo.

Uma das possibilidades que podemos 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, ou seja, o voto seria para o Guilherme, caso o contrário, entre 71 a 100, o voto seria para o Paulo que corresponde aos 30%:

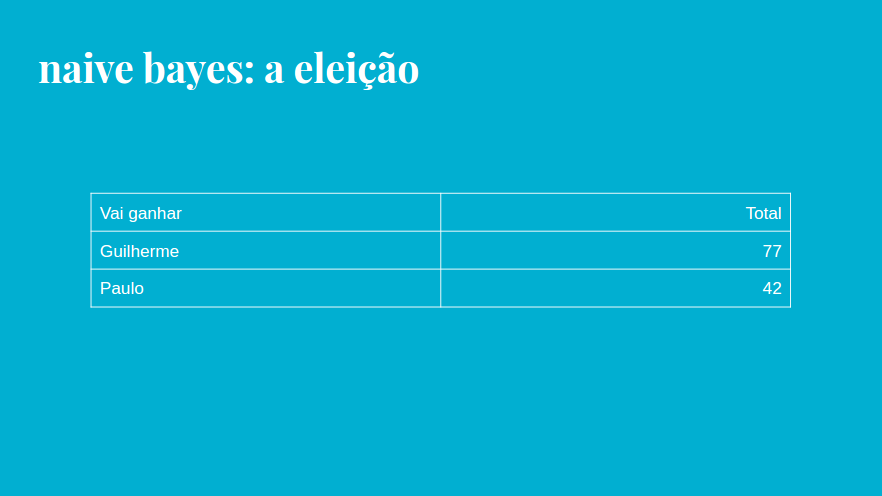


Mas o que estamos fazendo nesse momento? Estamos criando uma regra que não decide apenas utilizando a maior ou menor probabilidade, em outras palavras, nesse instante, nós temos 70 e 30 números de 100 para decidir se é o Guilherme (70 números) ou o Paulo (30 números). Levando essa nova regra em consideração, se perguntarmos para uma pessoa qualquer em quem ela votará, será que ela vai votar no Paulo ou no Guilherme? Dessa vez, ao invés de escolher o maior para tentar descobrir em quem essa pessoa votará, iremos sortear um número de 1 a 100, pois o número 100 é o número total da probabilidade. Então, se esse número for entre 1 a 70 iremos dizer que essa peossa votou no Guilherme e se for entre 71 a 100 iremos dizer que ela votou no Paulo.

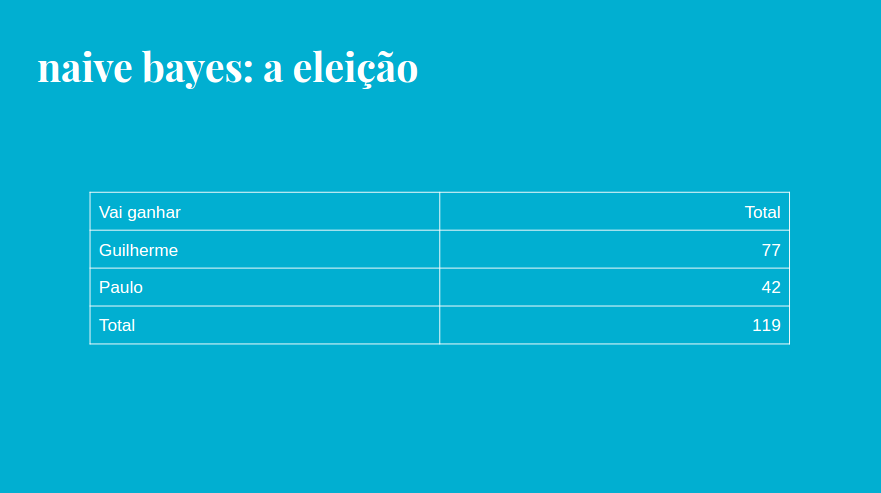
Perceba que agora, não estamos lidando apenas com uma possibilidade como estávamos fazendo antes, dessa vez, existe a possibilidade tanto para o Guilherme ou para o Paulo, independemente se o Guilherme tem mais chance do que o Paulo ou vice-versa. Observe que essa regra funciona de acordo com as probabilidades, pois existe a chance de uma pessoa qualquer escolher ou Guilherme ou Paulo. De acordo com o que vimos até agora, conseguimos definir 3 tipos de regras de decisão:

* **Preferência ao maior**: É válido 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.

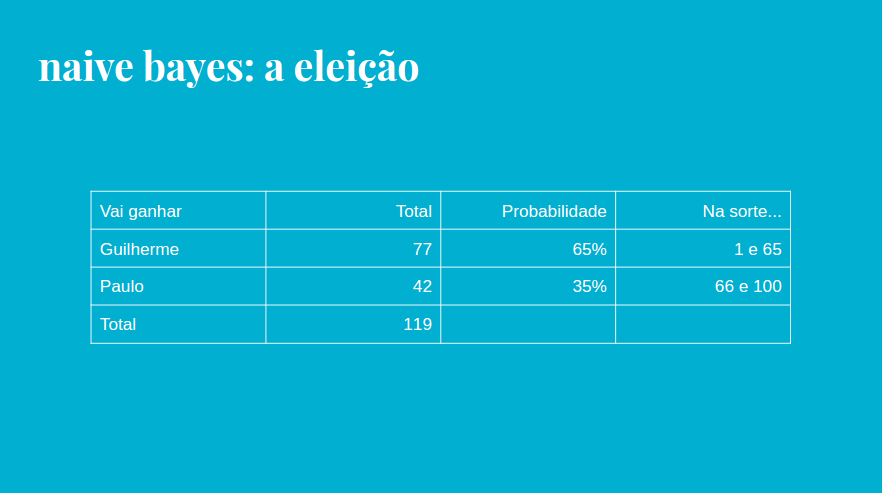
O nosso treino pode ser baseado em algumas dessas regras de decisões, porém é sempre importante compreender que para cada um dos casos uma dessas regras pode ser mais interessante do que a outra. Um outro exemplo que podemos analisar seria conforme a tabela abaixo:



Repare que dessa vez, ao invés porcentagens, temos a quantidade de votos do Guilherme e do Paulo. O que podemos fazer quando já temos a quantidade de votos ao invés do percentual? Simples, calculamos o percentual. Para calcularmos o percentual, precisamos saber a quantidade de votos totais, nesse caso, basta somar os votos do Guilherme e do Paulo:



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, quando arredondamos obtemos 65%. E para o Paulo? A mesma coisa, porém, agora com 42 votos, então, 42/119 = 0,352941176, ao arredondar esse número obtemos 35%. Então a nossa tabela fica da seguinte maneira:



Se sortearmos um número de 1 a 100 e cair entre 1 e 65, fará parte dos 65%, isto é, o Guilherme será votado, caso o número seja entre 66 a 100, estaremos pegando os 35% do Paulo, logo, o voto vai para o Paulo. Observe que por meio dessa tabela podemos utilizar uma das 3 regras que vimos até agora. Por exemplo, se utilizarmos a regra do **maior**, iremos apenas dizer que o Guilherme será votado, caso optamos pela regra do valor **menor**, falaremos que apenas o Paulo será votado, por fim, podemos escolher a regra da **probabilidade**, em outras palavras, sorteamos um número entre 1 a 100 e então, dependendo do número que for sorteado, ele cairá ou nos 65% do Guilherme (1 a 65) ou nos 35% do Paulo (66 a 100).

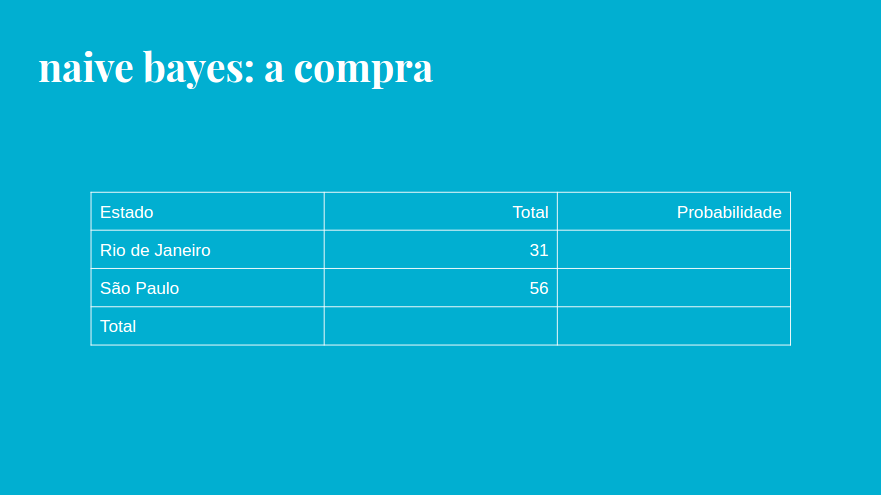
Perceba que o comportamento do algoritmo que estamos analisando, é justamente pegar uma quantidade de dados brutos, calcular as probabilidades e utilizar uma regra de decisão. Esses passos são bem similares ao que fazemos no nosso dia-a-dia, por exemplo, vemos diversos animais, e esses animais possuem pernas curtas, são rozinhas, não fazem *auau* entre demais características, então aprendemos que dada as características citadas, a maioria desses animais são porquinhos e a minoria são cachorrinhos, ou seja, calculamos essa probabilidade na nossa cabeça e a partir dela conseguimos concluir se é um porquinho ou um cachorrinho.

Na classificação de e-mails entre *spam* e não *spam* também fazemos algo bem similar, em outras palavras, também utilizamos um dos 3 critérios que vimos até agora, por exemplo, quando vemos os nossos e-mails e nos deparamos com um texto do *subject* que na maioria das vezes era um *spam*, imediatamente, sem utilizar nenhum outro critério, concluímos que se trata 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, nesse instante estamos entrando na situação 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 vai descer, então acontece um evento comum 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 utilizamos o critério da probabilidade, sorteamos um número na nossa cabeça e, a partir número, 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 que utilizaremos nos exemplos posteriores, chama-se *maximum a posteriori* que é justamente a regra que, dada as probabilidades, escolhemos o maior valor dentre as possibilidades.

Probabilidade condicional

Então vamos para o próximo exemplo referente a um site de vendas de imóveis. No meu site, tenho diversas informações dos clientes que compraram, e dentre elas eu peguei a informação de qual estado que eles são:



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? A princípio, não temos muita utilidade pra ela... Então o que realmente queremos saber? Na verdade, o que queremos saber é se, dada essas informações, a pessoa vai comprar ou não no meu site. Mesmo a informação que indica qual região os clientes compram mais, ou seja, de São Paulo ou do Rio de Janeiro, tenha alguma relevância, a informação de que o cliente que compra ou não também é de grande importância. Então observe que nesse instante teremos que lidar com duas informações conforme a tabela a abaixo:



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. Então agora 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, então dividimos os que compraram pelo total e obtemos a probabilidade dos clientes que compraram, em seguida, fazemos o mesmo para os que não compraram. O resultado da nossa tabela fica da seguinte forma:



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 temos que lidar com duas variáveis que são: **localidade** e **compra**:

* **localidade**: estado que o cliente mora.
* **compra**: se comprou ou não.

Nesse instante precisamos analisar duas informações para poder prevêr se o cliente comprará ou não. Por exemplo, utilizando o método *maximum a posteriori*, um cliente do Rio de Janeiro, compra ou não compra? Como podemos ver 68,5% compram e 31,5% não, em outras palavras, os clientes do Rio de Janeiro irão comprar. Mas e os clientes de São Paulo? 72,5% não compram e 27,5% sim, logo, não irão comprar.

Perceba que essas probabilidades possuem condições, pois de acordo com o estado do cliente, temos uma determinada possibilidade se ele vai comprar ou não, em outras palavras, para cada estado, teremos uma probabilidade de compra dele, por exemplo, se o cliente for do Rio de Janeiro ele tem mais chance de comprar, porém, se for de São Paulo ele tem menos chance de comprar. Esse é o tipo de probabilidade quem 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 o estado dela, então, se fosse do Rio de Janeiro, responderíamos que ela compraria, porém, se fosse de São Paulo, falaríamos que ele não iria comprar, 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 ao invés de escolher o maior, iríamos sortear um número de 1 a 100 e então verificaríamos em qual probabilidade ele se encaixa, por exemplo, se fosse o Rio de Janeiro, de 1 a 68,5 responderíamos que compraria, porém, de 68,6 a 100 diríamos que não compraria.

Observe que para esses tipos de dados (com probabilidades condicionais), podemos analisar diversas perguntas, como por exemplo:

* Dado que ele é do Rio de Janeiro, ele vai comprar?

Qual é a nossa atitude dessa vez? Primeiro verificamos em qual tabela ele pertence, nesse caso, Rio de Janeiro, logo, ele tem 68,5% de chance de comprar, então podemos responder que sim. A próxima pergunta seria:

* Dado que ele é do Rio de Janeiro, qual a probabilidade de ele comprar?

E para essa situação? Fazemos a mesma coisa, porém respondemos a chance, ou seja, probabilidade desse cliente comprar que é 68,5% . Perceba que na matemática podemos representar essa mesma pergunta da seguinte forma:

P(Comprar | Rio de Janeiro)

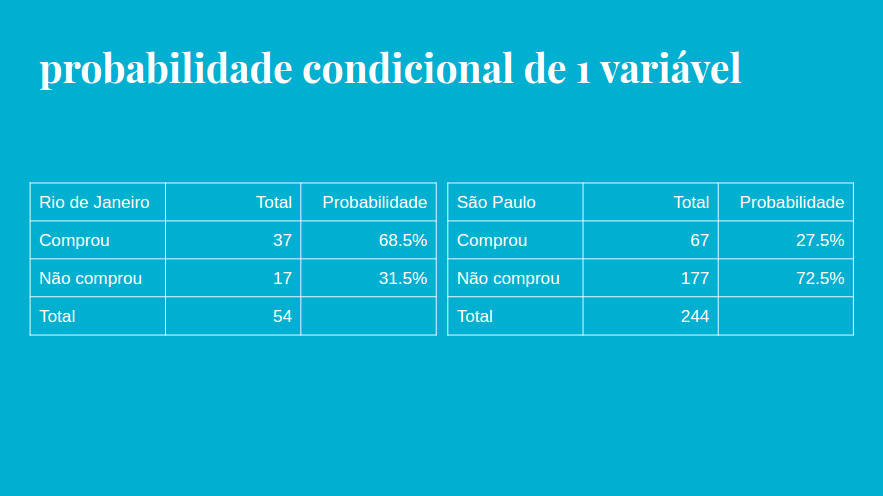
Repare que a matemática só traduz, ou seja, P significa probabilidade e (comprar | Rio de Janeiro) nos diz que, probabilidade de comprar, dado que ele é do Rio de Janeiro. Então podemos concluir que essa representação matemática é a tradução da nossa pergunta. E se fosse de São Paulo? Como ficaria? Vejamos 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 da seguinte fórmula:

P(Comprar | São Paulo)

Até o momento, tivemos apenas 1 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 abaixo:



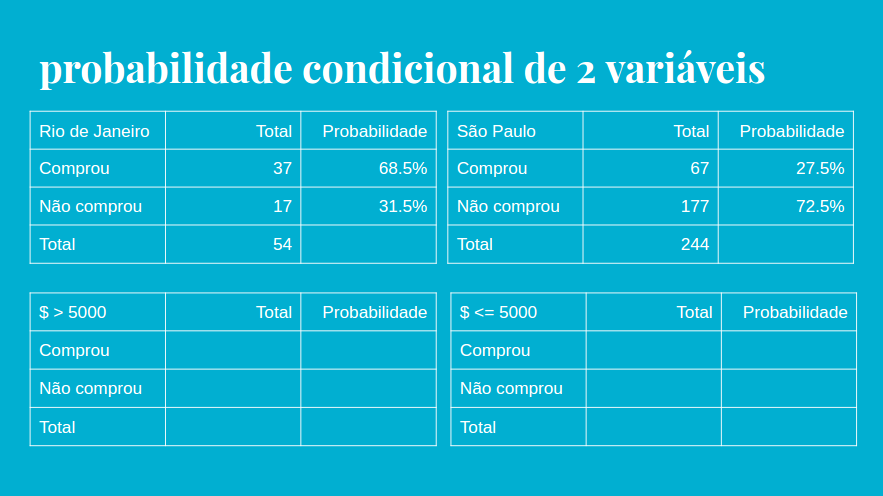
Com apenas uma variável condicional utilizamos essas duas tabelas para nos auxiliar. Porém, e se forem duas variáveis condicionais? Por exemplo, sabemos qual é o estado do cliente e, além do estado, também temos a informação sobre sua renda, em outras palavras, sabemos também se a renda do cliente é maior ou menor do que 5000. 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 entre outros estudos sobre o mercado.

Probabilidade condicional de duas variáveis

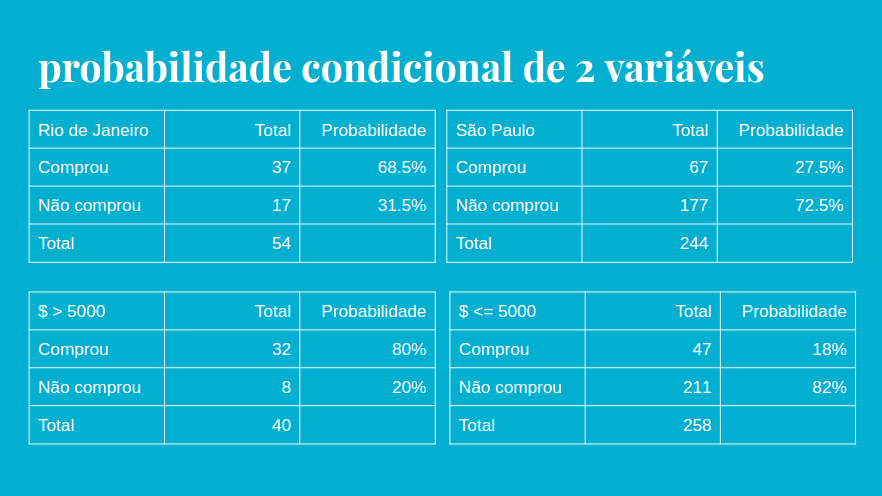
Tendo o conhecimento das informações apresentadas anteriormente, nos deparamos com a seguinte situação:

* Temos duas variáveis condicionais, o estado e a faixa salarial do cliente, como podemos fazer a nossa analise?

Observe que agora temos duas variáveis condicionais, ou seja, precisamos analisar duas variáveis distintas, mas como podemos resolver isso? Da mesma forma que fizemos para a variável que representa o estado, iremos criar uma tabela para a variável que representa a faixa de salário:



Então repara que agora temos 4 tabelas, dentre elas, duas representam os estados em que os clientes moram e as outras duas, representam a faixa salarial dos nossos clientes. O que precisamos fazer agora? Preencher a tabela da faixa salarial:



Veja que, 80% dos clientes com salário acima de 5000 compram e 20% não, porém, os clientes com salários abaixo de 5000 18% compram e 82% não. Por meio dessa tabela, podemos concluir que o perfil de uma pessoa que ganha mais que 5000 é totalmente diferente de uma pessoa que ganha menos de 5000. 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 deversos pontos para explicar o motivo para alguns clientes, com um determinado perfil, comprar 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 5000, possuem maiores chances de comprar do que os clientes com faixa abaixo de 5000, como também sabemos dizer que, dentre os clientes do Rio de Janeiro ou de São Paulo, os clientes 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 que treinávamos o nosso algoritmo? Então agora precisamos treinar 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 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, em outras palavras, iremos calcular a chance e utilizar uma regra de decisão, porém, a única diferença é que pra cada variável condicional que tivermos, precisaremos criar duas tabelas, isto é, quando tinhamos 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, seis tabelas e assim por diante. Começaremos com o seguinte exemplo:

* Dado um cliente que é de São Paulo com salário menor do que 5000. Qual é a probabilidade de ele comprar?

Lembra que podemos representar essa pergunta com uma fórmula matemática? Porém tinhamos feito apenas para uma variável condicional. Como ficaria a mesma fórmula para duas variáveis, ou seja, as variáveis estado e faixa de slário? Vejamos:

P(Comprar | São Paulo, <= 5000)

Perceba que por meio da fórmula, temos os valores de ambas as variáveis, nesse caso, estado e salário, e, além disso, já sabemos quais são os valores, ou seja, São Paulo e menor do que 5000 respectivamente. Mas como calcularemos essa probabilidade que envolve tanto um estado e 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 depender uma da outra, podemos quebrar a fórmula matemática mais complexa (P(Comprar | São Paulo, <= 5000)), em duas fórmulas mais simples:

P(Comprar | São Paulo, <= 5000) = P(Comprar | São Paulo) \* P(Comprar | <= 5000)

Observe que criamos duas fórmulas distintas, em outras palavras, uma não depende da outra. A partir de agora, estamos calculando a probabilidade separadamente! Então começaremos pela fórmula P(Comprar | São Paulo):

* P(Comprar | 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(Comprar | <= 5000):

* P(Comprar | <= 5000): Qual é a probabildade de compra para os clientes que ganham menos de 5000?
  + 18% compram.
  + 82% não compram.

Por fim, multiplicamos ambas as probabilidades 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 5000 comprar?
  + 18%.

Temos as porcentagens 27,5% e 18%, então fazemos:

27,5% \* 18%

Então obtemos o seguinte resultado:

0,275 \* 0,18 = 0,0495 -> 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 5000 foi de 4,95%, porém, arredondamos para 5% por ser um valor muito pŕoximo, ou seja, o nosso teste resultou em 5%. 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 irá comprar, pois existe 95% de chance dele não comprar. E se tentarmos a regra de probabilidade? Teremos que sortear um número de 1 a 100, se o resultado for de 1 a 5 o cliente vai comprar, caso o resultado for 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(Comprar | Rio de Janeiro, >5000)

Um cliente que é do Rio de Janeiro e tem um salário acima de 5000. Dado que esse cliente é do Rio de Janeiro e ganha mais de 5000, qual é a probabilidade dele ganhar? Faremos da mesma forma como anteriormente, ou seja, calcularemos cada probabilidade de acordo com sua variável:

* Qual a probabilidade dele comprar dado que ele é do Rio de Janeiro?
  + 68,5%
* Qual a probabilidade dele comprar dado que ele ganha mais de 5000?
  + 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 5000 contém 54,8% de chance de comprar. Um resultado bem diferente comparado ao cliente que era de São Paulo e ganhava menos de 5000 que continha apenas 5% de chance de comprar.

## Resumindo

Perceba que podemos calcular a probabilidade de um evento, como por exemplo, um cliente que é de um estado X e ganha um salário ou maior ou menor que Y. Fizemos isso multiplicando todas as variáveis existentes para obter o resultado final, pois assumidos que ambas as variáveis são independentes. Esse é o processo de treino que precisamos realizar, ou seja, calcular as probabilidades condicionais independentemente se é uma ou duas ou três variáveis condicionais. Para cada variável criamos duas tabelas, isto é, se for apenas uma variável, então duas tabelas, se forem duas variáveis, então quatro tabelas, se forem Xvariáveis, então X \* 2 tabelas.

Então utilizamos um critério de avaliação, podemos utilizar tanto o maximum a posterioriquanto a probabilidade, porém, precisamos sempre nos atentar a qual critério escolher, pois, por exemplo, se utilizarmos o maximum a posteriori e as probabilidades de compra forem sempre maiores, ele sempre vai dizer que o cliente vai comprar, como também, caso o contrário, ou seja, todas a probabilidades de compra forem menor, ele sempre irá dizer que o cliente não vai comprar. Percebe o quão 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, 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, em outras palavras, precisa somar, 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), portanto, podemos concluir que ele é um algoritmo eficiente, pois ele faz operações de acordo com a quantidade de elementos, por exemplo, se são 100 elementos ele faz apenas as 100 operações, se forem 1000 elementos, 1000 operações, ou seja, 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 utilizado para classificar texto, como foi o caso do spam, por exemplo, verificar as palavras mais frequentes e, com base nessas informações, ele tenta distinguir um e-mail normal de um spam. Perceba o naive bayes não é uma implementação de um outro mundo, pois ele está relacionado com determinados passos e rotinas que temos uma pequena noção, em outras palavras, temos a capacidade de analisar situações, como por exemplo, dado que 70% das pessoas vão recomendar o produto e 30% não, e então, perguntarmos 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, como por exemplo, se, com base na informação: "A maior parte dos brasileiros gostam de futebol". Levando em consideração essa afirmação, se perguntarmos para um brasileiro qualquer se ele gosta de futebol, o que você responderia? Provavelmente responderia sim, pois é a maior parte das pessoas, é bem provável que também terão pessoas que não gostam, mas não importa, pois estamos dando preferência a quantidade maior, ou seja, utilizando a regra do maior, ou então, poderíamos utilizar uma outra regra de decisão.

Perceba que o senso de eventos mais comuns, em outras palavras, com maior frequência de ocorrência, é uma opção válida para uma regra de decisão e é muito parecido com as regras que estamos utilizando, ou seja, o que tiver maior frequência provavelmente acontecerá denovo. Esse é justamente o comportamento do naive bayes, de acordo com a probabilidade que aconteceu no passado, eu irei conseguir classificar o que acontecerá agora.

6 – testando diferentes modelos e validando o vencedor

Registrando os testes

Até agora vimos como a implementação do naive bayes funciona, ou seja, levando em consideração as informações do passado, ele da maiores 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, um mesmo cliente com as mesma características já comprou, provavelmente ele respondará que esse novo cliente comprará também. Porém, quando utilizamos 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:

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 82.000000

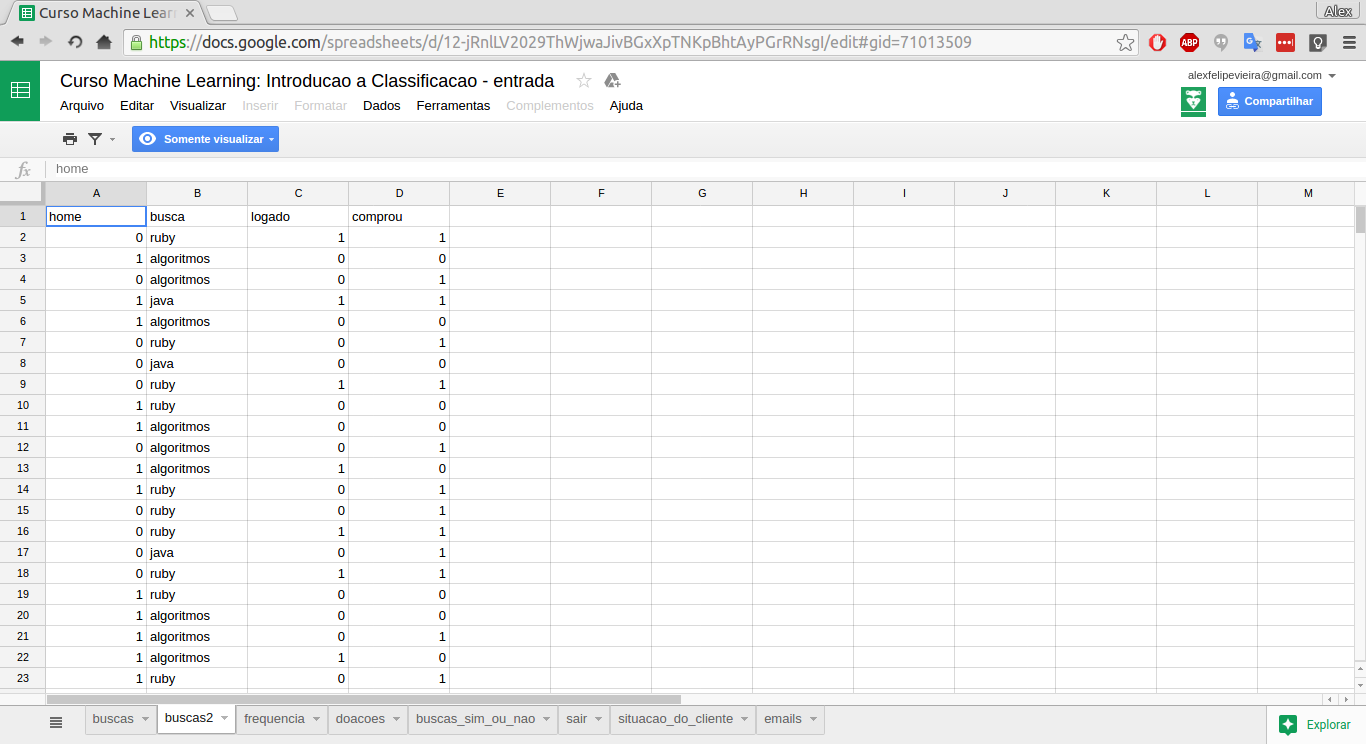
100

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, em outras palavras, se chutássemos tudo 1 ou se implementássemos um algoritmo bem complexo, cheio de código, 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 pra tudo? Repare que em alguns momentos, principalmente para classificação de texto, utilizaremos o MultinomialNB, porém, em outras situações pode ser que esse algoritmo não seja tão interessante. Porém, precisamos levar em consideração que, nessa situação, foram utilizados dados fakes, ou seja, dados que não são reais, ou melhor, informações que foram manipuladas para demonstrar uma comparação entre o nosso algoritmo e o algoritmo base. Mas e se fossem dados reais? Como por exemplo 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, ou seja, dependerá muito do que estamos tentando classificar.

Dessa vez vamos mudar um pouco os nossos dados, lembra da [planilha do Google Spreadsheets](http://bit.ly/1T7qjMS) que contém os nossos dados?



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), comprou (se ele comprou ou não), porém, nessa aba temos apenas 75 registros. Repara que agora iremos lidar com muito menos elementos do que anteriormente e, mesmo com menos dados, provavelmente teremos um resultado bem diferente que anteriormente, portanto, sempre que tiveremos 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 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:

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 85.714286

7

Taxa de acerto base: 71.428571

Observe que para esse conjunto de dados foram utilizados 7 registros para teste, em outras palavras, foram utilizados 10% desses dados. Além disso, dessa vez, repara que o nosso algoritmo acertou 85,71% enquanto o algoritmo base acertou 71,42%. Isso comprova que o resultado dos nossos testes variam de acordo com o conjunto de dados, isto é, 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, por exemplo, pode ter melhores resultados com quantidade menores do que com maiores e vice versa. E não podemos esquecer que as características são pontos cruciais para o resultado do nosso algoritmo, nesse caso, temos 3 variáveis (home, busca, logado) para esses dados sendo que, dentre elas, uma é categórica com 3 valores distintos, ou seja, no total temos 5 variáveis. Então podemos nos perguntar:

* Será que essas 5 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? Então 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:

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 85.714286

7

Taxa de acerto base: 71.428571

Ele obteve a mesma taxa de acerto, em outras palavras, 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:

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 57.142857

7

Taxa de acerto base: 71.428571

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 o nosso algoritmo 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:

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 85.714286

7

Taxa de acerto base: 71.428571

Repare que dentre todas as variáveis, apenas a busca apresentou algum impacto de verdade. Então 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! Então 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 do nosso código novamente:

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 85.714286

7

Taxa de acerto base: 71.428571

Como podemos ver, apenas com a variável busca, fomos capazes de alcançar o mesmo resultado que estávamos obtendo com todas as variáveis. Então perceba que temos muitas coisas em jogo, isto é, para o nosso algoritmo obter um resultado bom ou ruim, depende da quantidade de elementos que estamos utilizando para treinar e testar. Também depende das variáveis, ou melhor, das características que estamos utilizando, como vimos no nosso exemplo, a variável busca possui uma grande importância para que o nosso algoritmo atingir um resultado bom, permitindo até o uso exclusivo dela, ou seja, 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 contém grande impacto e então usar apenas elas? Na prática, como vimos, parece fazer sentido.

Note que 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 obtem um resultado de, por exemplo, 70%, então se tirarmos uma dessas variáveis, o resultado será melhor que 70%? Ou então, pior? Será que, se adicionarmos uma nova variável fará diferença? Como por exemplo, idade, sexo, localização entre diversas informações que temos. Então perceba que é bem comum o processo de adicionar e remover variáveis de acordo com as características que os nossos elementos contém, porém, para cada teste realizado, é um teste a mais, por exemplo, testaremos 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, então, isso vai se repetindo até o momento em que alcançamos um resultado tão bom que parece perfeito. Consegue perceber o quão problemático essa atitude é? Perceba que existe uma grande chance do 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, e então, descobrimos que dentre todos esses testes existe uma informação em comum que é:

* Todos os anos que a bolsa de valores caiu, são os mesmos anos em que o Nicolas Cage filmou um filme de ação.

Então concluímos que cada vez que o Nicolas Cage filmar um filme de ação, saberemos que a bolsa de valores cairá... Faz sentido esse tipo de conclusão? Consegue perceber que nessa atitude de tentar diversas informações em excesso, provavelmente teremos alguma coincidência entre essas informações? Em outras palavras, estaremos lidando com sorte ou azar. Então podemos chegar a conclusão de que testar com muitas variáveis, possibilita chegarmos a um resultado muito bom, porém, esse resultado será por sorte. Mas um resultado bom por sorte não é bom? Quando utilizamos um resultado por sorte, somos induzidos às coincidências, ou seja, podemos chegar a conclusões do tipo:

* Hoje está chovendo, então os meus clientes vão comprar, pois todas as vezes que choveram os clientes compraram.
* Hoje é o dia 7 do mês, então os meus clientes vão compra, pois em todos os dias 7 os meus clientes compraram.
* Amanhã é o meu aniversário, então todos os meus clientes irão comprar, pois em todo meu aniversário eles compraram.

Consegue perceber que tudo isso não faz sentido algum? Ou seja, iremos levar 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, em outras palavras, quando treinamos tanto o nosso algoritimo com muitas informações, ou chegamos a um resultado tão perfeito que foi simplesmente por pura sorte, ou então, é tão perfeito e específico que não funciona para mais nada, logo, não poderemos utilizar para o mundo real. É justamente por esse motivo que durante todo o percurso do desenvolvimento do nosso algoritmo, não variamos suas características.

Temos sempre que lembrar que, todas as vezes que alterarmos as variáveis que estamos utilizando 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 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 eu fiz uma anotação informando que testamos com 90% dos dados para treino e 10% para teste. Porque 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 percebemos que todas elas dão errado, e então, em um determinado teste dentre esse conjunto de testes, conseguimos um resultado satisfatório, tem uma grande desse resultado ter acontecido simplesmente por sorte, e como vimos, se caírmos nesse cenário nos daremos mal. Então repara que, 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 para evitar de cair no cenário que simplesmente chegamos 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, eu irei considerar 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. Então 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

Então para apresentarmos o resultado desse algoritmo, precisamos adicionar todos os passos realizados e por fim o resultado final, que foram, 7 testes com o resultado de 85,71%.

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[['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, porém, e se quisermos variar entre os algoritmos? Será que é possível?

Algoritmo adaBoost

Até agora utilizamos apenas o MultinomialNB, porém existem diversas outras variações para algoritmos de classificação, uma das variações que iremos utilizar se chama [AdaBoost](https://pt.wikipedia.org/wiki/AdaBoost). Esse algoritmo, basicamente, ele tenta melhorar um algoritmo, ou seja, ele vai refinando o algoritmo para tentar encontrar a melhor possibilidade, então, quando ele encontra, ele devolve o resultado. Mas como podemos utilizar o AdaBoost no nosso algoritmo? Precisamos apenas alterar o algoritmo que estamos utilizando no nosso arquivo classifica\_buscas.py, ou seja, no instante que estamos importando e instanciando o MultinomialNB:

from sklearn.naive\_bayes import MultinomialNB

modelo = MultinomialNB()

Comentamos o import e 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 utilizado para o MultinomialNB, será que funcionará para o AdaBoost também? Antes de testarmos, agora que estamos utilizando um algoritmo diferente, precisamos utilizar todas as variáveis que foram utilizadas 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.

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 85.714286

7

Taxa de acerto base: 71.428571

Como podemos ver, os métodos de treino e 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:

> python 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 dependedo do conjunto de dados o nosso teste pode variar, nesse caso, o AdaBoost obteve um resultado melhor que o MultinomialNB para esse conjunto de dados, pode ser que para um outro conjunto de dados, o MultinomialNB tenha um resultado melhor.

Repare que além de variar as variáveis, agora estamos variando o algoritmo também, ou seja, além de só utilizar o MultinomialNB, estamos utilizando o AdaBoost. Vimos que, quando variamos o algoritmo um obtem resultados melhores do que o outro dependendo do conjunto de dados, considerando essa afirmação, que tal utilizarmos os dois algoritmos (MultinomialNB e AdaBoost) ao mesmo tempo? Mas porque ao mesmo tempo? Justamente por apresentarem resultados diferentes, em outras palavras, 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, então 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\_elementos

print("Taxa de acerto do algoritmo: %f" % taxa\_de\_acerto)

Porém, observe que as variáveis, modelo, treino\_dados, treino\_marcacoes, teste\_dadose 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:

# restante do código

print(total\_de\_elementos)

acerto\_base = max(Counter(teste\_marcacoes).itervalues())

taxa\_de\_acerto\_base = 100.0 \* acerto\_base / len(teste\_marcacoes)

print("Taxa de acerto base: %f" % taxa\_de\_acerto\_base)

Precisa da variável total\_de\_elementos, porém 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, em outras palavras, deixar uma mensagem explícita sobre o que ele é:

acerto\_base = max(Counter(teste\_marcacoes).itervalues())

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, isto é, 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:

> python 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 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:

> python classifica\_buscas.py

Taxa de acerto do algoritmo: 85.714286

Taxa de acerto do algoritmo: 85.714286

Taxa de acerto base: 71.428571

Total de teste: 7

Também está funcionando, repare que dessa vez o total de elementos de teste são 7 e ambos os algoritmos são de 85,71%. Porém, repare que, por meio dessa impressão, não conseguimos distinguir quais são os algoritmos que estão sendo utilizados e mais, não sabemos quais são os seus respectivos resultados, ou seja, como saberemos qual deles obteve o melhor resultado? Isso significa que precisamos adicionar o nome de ambos na impressão também, podemos fazer isso enviando por parâmetro a variável nome que representa o nome do algoritmo que está sendo utilizado:

def fit\_and\_predict(nome, modelo, treino\_dados, treino\_marcacoes, teste\_dados, teste\_marcacoes):

# restante do código

Então, 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 formatpara 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))

Repare que enviamos dois argumentos no nosso texto, que é o {0} e o {1} que significa o primeiro e segundo elemento que foram adicionados no format, nesse caso, 0 para nomee 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 strings grandes como essa para uma variável, pois dessa forma identificamos com mais facilidade com o que estamos lidando, então 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, taxa\_de\_acerto)

print(msg)

Então 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).astype(int)

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:]

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).itervalues())

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)

### No Python3

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).itervalues())

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:

> python classifica\_buscas.py

Taxa de acerto do algoritmo MultinomialNB: 85.7142857143

Taxa de acerto do algoritmo AdaBoostClassifier: 85.7142857143

Taxa de acerto base: 71.428571

Total de teste: 7

Perceba que agora está claro o resultado de cada algoritmo, vamos modificar novamente para o arquivo buscas.csv e vejamos o resultado:

> python classifica\_buscas.py

Taxa de acerto do algoritmo MultinomialNB: 82.0

Taxa de acerto do algoritmo AdaBoostClassifier: 85.0

Taxa de acerto base: 82.000000

Total de teste: 100

Observe que agora é bem fácil de saber qual dos algoritmos obteve o melhor resultado. Porém, agora que implementamos os dois algoritmos (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 os algoritmos, qual a taxa de acerto que esperamos acertar no mundo real? Para esse algoritmo, esperamos que ele acerte, teoricamente, 85%, mas porque 85%? É justamente porque entre os 2 algoritmos um obteve 82% e o outro 85%, logo, pegamos o de maior valor. Isso significa que 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, escolhemos ele, pois ele apresentou um resultado melhor com esse conjunto de dados, em outras palavras, 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 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 de dados, pois utilizamos essa porcentagem para decidir qual dos algoritmos utilizaremos para um conjunto de dados desconhecido, isto é, um conjunto que não conhecemos e que queremos que ele classifique para nós.

Então repara que o nosso algoritmo terá 3 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 podemos afirmar que no mundo real ele obtem um resultado bom ou ruim. Mas e na prática? Como fazemos? Atualmente simplesmente utilizamos 90% pra treino e 10% pra teste, e os dados do mundo real? Como podemos resolver isso? Existem diversas formas para resolvermos esse cenário, uma das formas que utilizaremos será conforme os dados abaixo:

* 80% para treino.
* 10% para teste.
* 10% para teste no mundo real.

Validando o algoritmo com dados no mundo real

O nosso algoritmo terá 3 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 podemos afirmar que no mundo real ele obtém um resultado bom ou ruim. Mas e na prática? Como fazemos? Atualmente simplesmente utilizamos 90% pra treino e 10% pra teste, e os dados do mundo real? Como podemos resolver isso? Existem diversas formas para resolvermos esse cenário, uma das formas que utilizaremos será conforme os dados abaixo:

* 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, porém, considerando que o nosso algoritmo precisa ser validado com dados do mundo real, precisaremos de uma terceira variável que representará esses dados para validação. Então agora, 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% para treino:

# restante do código

porcentagem\_de\_treino = 0.8

E agora? Quantos porcento utilizaremos para teste? 10% certo? Então 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 = porcentagem\_de\_treino \* len(Y)

tamanho\_de\_teste = len(Y) - tamanho\_de\_treino

* No Python 3, convertemos a multiplicação para inteiro:

tamanho\_de\_treino = int(porcentagem\_de\_treino \* len(Y))

Veja 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, em outras palavras, está pegando os 20% restante da quantidade total dos dados, 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 o 10% que precisamos:

# restante do código

porcentagem\_de\_treino = 0.8

porcentagem\_de\_teste = 0.1

tamanho\_de\_treino = porcentagem\_de\_treino \* len(Y)

tamanho\_de\_teste = porcentagem\_de\_teste \* len(Y)

* No Python3...

# 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 tamanho para teste, mas e os dados de validação? Precisamos calculá-lo também, porém, perceba que não temos nenhuma variável que representa seu percentual. E então? Criamos uma variável para armazenar o seu percetual? Poderíamos fazer isso também, mas perceba 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%, então subtraímos esses 20% pela quantidade total de teste (10%), logo, 20% - 10% = 10%. Perceba que, 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? Como fazemos? Vejamos:

# restante do código

porcentagem\_de\_treino = 0.8

porcentagem\_de\_teste = 0.1

tamanho\_de\_treino = porcentagem\_de\_treino \* len(Y)

tamanho\_de\_teste = porcentagem\_de\_teste \* len(Y)

tamanho\_de\_validacao = len(Y) - tamanho\_de\_treino - tamanho\_de\_teste

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 = porcentagem\_de\_treino \* len(Y)

>>> tamanho\_de\_teste = porcentagem\_de\_teste \* len(Y)

>>> tamanho\_de\_validacao = len(Y) - tamanho\_de\_treino - tamanho\_de\_teste

Então vamos verificar de 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 primeiro 80%, portanto, não precisamos mexer nele. 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:]

Pois 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]

Observe que a soma do tamanho de treino e teste se tornaram instruções bem grandes, portanto, podemos extrair essa operação para uma outra variável. Vamos criar a variável fim\_de\_teste:

fim\_de\_teste = tamanho\_de\_treino + tamanho\_de\_teste

teste\_dados = X[tamanho\_de\_treino:fim\_de\_teste]

teste\_marcacoes = Y[tamanho\_de\_treino:fim\_de\_teste]

E agora? Quem 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\_teste = tamanho\_de\_treino + tamanho\_de\_teste

# 800 até 899

teste\_dados = X[tamanho\_de\_treino:fim\_de\_teste]

teste\_marcacoes = Y[tamanho\_de\_treino:fim\_de\_teste]

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\_teste, pois ela é justamente o momento em que os dados de teste acabam:

validacao\_dados = X[fim\_de\_teste:]

validacao\_marcacoes = Y[fim\_de\_teste:]

Porém, e o final? Como podemos fazer? Lembra que quando não passamos o argumento no segundo parâmetro, ele pega o restante dos dados? Então, nesse exato momento ele está pegando a partir do 900 até o final, nesse caso o 999. Será mesmo que as nossas variáveis estão pegando os valores corretos? Vamos testar! Podemos fazer esse teste pelo interpretador do python. Dentro do interpretador, 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).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 = porcentagem\_de\_treino \* len(Y)

>>> tamanho\_de\_teste = 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\_teste = tamanho\_de\_treino + tamanho\_de\_teste

>>>

>>> teste\_dados = X[tamanho\_de\_treino:fim\_de\_teste]

>>> teste\_marcacoes = Y[tamanho\_de\_treino:fim\_de\_teste]

>>>

>>> validacao\_dados = X[fim\_de\_teste:]

>>> validacao\_marcacoes = Y[fim\_de\_teste:]

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 ao X? Lembra que ele vai do 800 a 899? Ou seja, a primeira posição dos nossos dados de teste no X é justamente a posição 800. Vejamos o resultado:

>>> teste\_dados[0]

array([1, 1, 0, 0, 1])

>>> X[800]

array([1, 1, 0, 0, 1])

Então 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 vai do 900 a 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])

Então vamos pegar 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])

verificando o melhor algoritmo

Calculamos as nossas variáveis que representarão tanto os nossos dados de treino, teste e validação, porém, ainda temos que isolar o resultado de cada um dos nossos algoritmos para que possámos comparar qual foi o melhor, em outras palavras, 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 atribuir 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, treino\_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 pra isso? Fazendo um único if que verifica qual deles é o maior:

if resultadoMultinomial > resultadoAdaBoost:

Então retornamos o maior para um variável chamada vencedor. Porém, observe que ambos os modelos, estão com os mesmo 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 atribuimos à variável vencedor o modelo de acordo com o teste, isto é, atribuiremos o modeloMultinomial caso o resultado do Multinomial for 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 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, então 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)

Então realizamos o mesmo calculo de acertos conforme fizemos antes, porém, a diferença é que, ao invés 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\_elementos

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\_elementos

msg = "Taxa de acerto do vencedor entre os dois algoritmos no mundo real: {0}".format(taxa\_de\_acerto)

print(msg)

Com a função implementada, podemos chamá-la após acharmos os vencedor:

# restante do código

if resultadoMultinomial > resultadoAdaBoost:

vencedor = modeloMultinomial

else:

vencedor = modeloAdaBoost

teste\_real(vencedor, validacao\_dados, validacao\_marcacoes)

Ao testar o nosso código temos o seguinte resultado:

> python 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

Observe que o nosso algoritmo vencedor foi o AdaBoost com 84%, porém, quando utilizamos 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. Porém ainda existe um detalhe, a taxa de acerto base ainda está baseada com os dados de teste:

# restante do código

acerto\_base = max(Counter(teste\_marcacoes).itervalues())

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).itervalues())

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).itervalues())

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:

> python 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 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:

> python 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 for 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 e 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)

df = pd.read\_csv('buscas.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 = porcentagem\_de\_treino \* len(Y)

tamanho\_de\_teste = 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\_teste = tamanho\_de\_treino + tamanho\_de\_teste

teste\_dados = X[tamanho\_de\_treino:fim\_de\_teste]

teste\_marcacoes = Y[tamanho\_de\_treino:fim\_de\_teste]

validacao\_dados = X[fim\_de\_teste:]

validacao\_marcacoes = Y[fim\_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)

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)

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)

if resultadoMultinomial > resultadoAdaBoost:

vencedor = modeloMultinomial

else:

vencedor = modeloAdaBoost

teste\_real(vencedor, validacao\_dados, validacao\_marcacoes)

acerto\_base = max(Counter(validacao\_marcacoes).itervalues())

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)

**No Python3...**

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\_teste = tamanho\_de\_treino + tamanho\_de\_teste

teste\_dados = X[tamanho\_de\_treino:fim\_de\_teste]

teste\_marcacoes = Y[tamanho\_de\_treino:fim\_de\_teste]

validacao\_dados = X[fim\_de\_teste:]

validacao\_marcacoes = Y[fim\_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)

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)

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)

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

conclusão

Nesse capítulo começamos com o algoritmo Multinomial, sabemos como esse algoritmo funciona por de trás, em outras palavras, sabemos que ele utiliza um modelo baseado em probabilidades dos eventos acontecerem, como por exemplo, se em uma determinada região chove com mais frequência do que não, ele vai prevêr que nos próximos dias irão chover. É uma decisão bem boba, porém tem a capacidade de prevêr diversas coisas, como aquele exemplo que vimos, no instante que olhamos um e-mail, como é que distinguimos de um *spam* ou não? Analisamos o texto do e-mail, então, se na maioria das vezes que aquele texto apareceu foi um *spam*, consequentemente iremos achar que é um *spam*. Também vimos que as variáveis que utilizamos 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 te digo apenas que um animal tem 4 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, existe também cachorros rosinhas. 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.

Então perceba que, 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, pois dependendo da quantidade de variáveis que ele tiver que lidar, ao invés de ajudar, vai confundí-lo, pois 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 iremos utilizá-lo para dados do mundo real, acabamos percebendo que ele não era tão bom quanto imaginávamos, ou melhor, 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 obtivémos, pois é uma forma de provar que aquele resultado que chegamos não foi por acaso, em outras palavras, com esse histórico de cada teste realizado, temos a capacidade de provar todos os passos que realizamos para chegar no 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 obtem um resultado diferente. Também implementamos os 2 algoritmos no mesmo código para testá-los ao mesmo tempo, e então, aquele que retornava o melhor resultado, isto é, o maior resultado, escolhíamos como o algoritmo vencedor. Tendo o vencedor em mãos, fazíamos mais um teste que é justamente um teste do mundo real, em outras palavras, pegávamos novos elementos que ele nunca tinha visto antes, ou seja, nem no seu treino e nem no teste, então pedíamos para ele prevêr para nós, pois dessa forma podemos ter a certeza como será o seu comportamento, resultado dada uma situação do mundo real. E é justamente esse valor final que irá validar o quão bom o nosso algoritmo é.

Com isso concluímos que tivémos 3 fases para o nosso algoritmo, que são:

* Treinar os algoritmos.
* Testar os algoritmos.
* 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 quão bom o nosso algoritmo é no mundo real.