09

O problema do sucesso e o algoritmo burro

O problema do sucesso e o algoritmo burro

Quando rodamos o nosso algoritmo `classifica_buscas.py`, ele conseguiu prever 82% dos casos e, a princípio, ficamos felizes por um número "grande", afinal, podemos considerar que 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 delicidado 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? 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
>>>
```

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, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 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, 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, 0, 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, 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, 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, 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, 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, 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, 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? Podemos pegar a quantidade de zeros por meio do tamanho do `Y` menos 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 escolhessemos 0, iriamos 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.

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 com o data frame

Repare o calculo 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 `len` e 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, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 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, 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, 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, 1, 1, 1, 1, 1, 1, 1, 0, 1, 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, 1, 0, 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, 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, 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, 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, 1, 0, 1, 1, 1, 0,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1,
       1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0])
>>>
```

Sabemos que o nosso `Y` é uma sequência de zeros e uns. E se pedirmos todos os valores de `Y` que são iguais a 1? Será que 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,  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, 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, False,
       False,  True,  True,  True,  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, 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, 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, `True` s e `False` s. Isso significa que todos os `True` s são os valores iguais a 1 e **qualquer valor diferente de 1** são os `False` s. E se quiséssemos saber quais são os zeros? Simples, bastaria pedir todos os valores de `Y` que são iguais a 0:

```
>>> Y==0
array([False, False, False,  True, False,  True, False, False, False,
       False, 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,  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 `True` s que apareceram quando pedimos todos os valores iguais a 1 se tornaram `False` , pois, nesse instante, pedimos todos os valores que são iguais a 0, ou seja, todos os `True` s 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])
>>>
```

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 `len` deles, ou seja, a quantidade de cada um:

```
>>> len(Y[Y==1])
832
>>> len(Y[Y==0])
168
```

Como podemos ver, foram retornados os 832 uns e 168 zeros. No nosso código, basta apenas 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 `sim` e `não`.

Até agora, quando analisamos os nossos dados, todas as nossas marcações, coluna `comprou` foram 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',
       '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 `sim` s e `nao` s, 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 `sim` s e `nao` s! Porém, isso é bem fácil de corrigir! Basta apenas alterarmos a comparação de uns para `sim` s:

```
# restante do código
acerto_de_um = len(Y[Y=='sim'])
```

E a comparação de zeros para `nao` s:

```
# restante do código
acerto_de_um = len(Y[Y=='sim'])
acerto_de_zero = len(Y[Y=='nao'])
```

Vamos verificar se isso funciona? Novamente no interpretador do python:

```
>>> acerto_de_um = len(Y[Y=='sim'])
>>> acerto_de_zero = len(Y[Y=='nao'])
>>> taxa_de_acerto_base = 100.0 * max(acerto_de_um, acerto_de_zero) / len(Y)
>>> print("Taxa de acerto base: %f" % taxa_de_acerto_base)
Taxa de acerto base: 83.200000
>>>
```

Perceba que está funcionando como o esperado. 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:]
```

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 `sim` s e `nao` s, 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', '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 `sim` s e `nao` s em zeros e uns.
- Comparar as duas variáveis (`resultado` e `teste_marcacoes`) e verificar quais possuem os valores iguais.

Vamos verificar como seria a segunda abordagem:

```
>>> resultado==teste_marcacoes
array([ True,  True,  True,  True,  True,  True,  True,  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 `True` s e `False` s, 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 `True` s 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 `True` s:

```
>>> 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)
```

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',
>>>
```

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 `sim` s ou `nao` s existentes dentro dela:

```
>>> list(Y).count('sim')
832
>>> list(Y).count('nao')
```

```
168
>>>
```

Então, ao invés de utilizar o a função `len` diretamente com os arrays para pegar a quantidade de `sim` s e `nao` s, 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 `sim` s e `nao` s 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>
```

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 `sim` s e `nao` s, em outras palavras, poderia ser outros textos diferentes de `sim` s e `nao` s, como por exemplo `yes` ou `not` que 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()
```

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)
```

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, independemente de qual informação esteja contida dentro do arquivo de dados.

Resumindo

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, `sim`s e `nao`s. 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 `0`s e `1`s, `sim`s e `nao`s 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. 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 `Y` utilizaremos 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.