06

Testando diferentes modelos e validando o vencedor

Testando diferentes modelos e validando o vencedor

Até agora vimos como a implementação do *naive bayes* funciona, ou seja, levando em consideração as informações do passado, ele 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` :

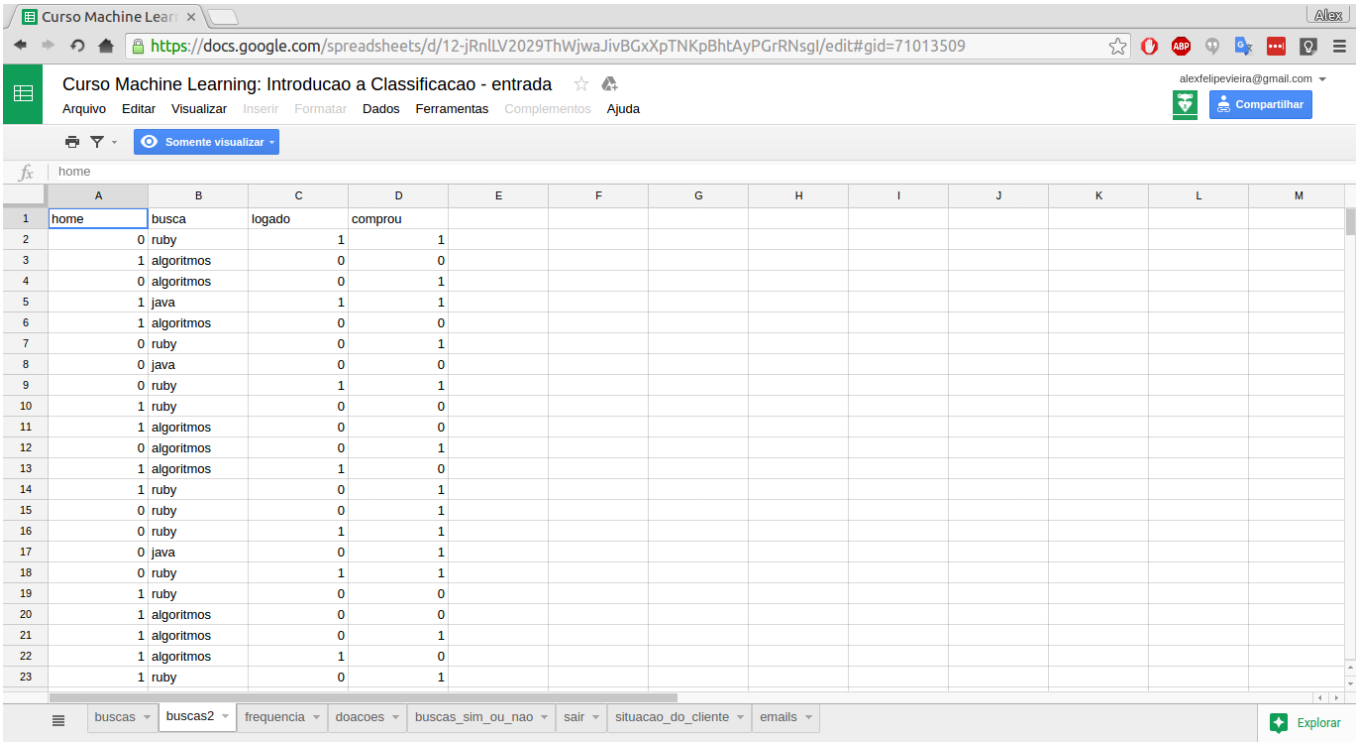
```
home,busca,logado,comprou
0,algoritmos,1,1
0,java,0,1
1,algoritmos,0,1
1,ruby,1,0
1,ruby,0,1
0,ruby,1,0
0,algoritmos,1,1
0,ruby,0,1
1,algoritmos,1,1
...
0,ruby,1,0
```

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?



Curso Machine Learning: Introducao a Classificacao - entrada

	A	B	C	D
1	home	busca	logado	comprou
2	0	ruby	1	1
3	1	algoritmos	0	0
4	0	algoritmos	0	1
5	1	java	1	1
6	1	algoritmos	0	0
7	0	ruby	0	1
8	0	java	0	0
9	0	ruby	1	1
10	1	ruby	0	0
11	1	algoritmos	0	0
12	0	algoritmos	0	1
13	1	algoritmos	1	0
14	1	ruby	0	1
15	0	ruby	0	1
16	0	ruby	1	1
17	0	java	0	1
18	0	ruby	1	1
19	1	ruby	0	0
20	1	algoritmos	0	0
21	1	algoritmos	0	1
22	1	algoritmos	1	0
23	1	ruby	0	1

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 tragetória 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_dados` e `teste_marcacoes` precisam ser enviadas para essa função, então vamos adicioná-las na lista de parâmetros da `fit_and_predict` :

```
def fit_and_predict(modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes):
# restante do código da função
```

Além disso, a impressão do total de elementos que está fora da função `fit_and_predict` :

```
# 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_marcacoe

from sklearn.ensemble import AdaBoostClassifier
modelo = AdaBoostClassifier()
fit_and_predict("AdaBoostClassifier", modelo, treino_dados, treino_marcacoes, teste_dados, teste_mar

# restante do código
```

Porém, ainda não estamos exibindo a variável `nome` , para isso, usaremos o método `format` para agrupar as nossas duas variáveis na mensagem, isto é, `nome` e `taxa_de_acerto` :

```
def fit_and_predict(modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)

    # restante do código

    print("Taxa de acerto do algoritmo {0}: {1}".format(nome, taxa_de_acerto))
```

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 `nome` e 1 para `taxa_de_acerto`. Além disso, esse texto refere-se à mensagem que estamos exibindo para o usuário, e uma boa prática é justamente isolarmos `string`s grandes como essa para uma variável, 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_marcacoe

from sklearn.ensemble import AdaBoostClassifier
modelo = AdaBoostClassifier()
fit_and_predict("AdaBoostClassifier", modelo, treino_dados, treino_marcacoes, teste_dados, teste_mar

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.

Antes tinhamos 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
```

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)
```

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, dada essas variáveis, precisamos pegar os dados dos nossos data frames. Será que precisaremos modificar os nossos calculos 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_treino` :

```
fim_de_treino = tamanho_de_treino + tamanho_de_teste

teste_dados = X[tamanho_de_treino:fim_de_treino]
teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]
```

E agora? 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_treino = tamanho_de_treino + tamanho_de_teste

# 800 até 899
teste_dados = X[tamanho_de_treino:fim_de_treino]
teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]
```

Nesse instante, precisamos nos atentar, pois os dados de validação não podem fazer parte dos dados de treino (0 a 799) e nem dos de teste (800 a 899), ou seja, precisamos fazer com que ele seja a partir do 900 a 999. E como podemos fazer isso? Da mesma forma que fizemos com os dados de teste, em outras palavras, basta pegarmos os valores a partir da variável `fim_de_treino` , pois ela é justamente o momento em que os dados de teste acabam:

```
validacao_dados = X[fim_de_treino:]
validacao_marcacoes = Y[fim_de_treino:]
```

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_treino = tamanho_de_treino + tamanho_de_teste
>>>
>>> teste_dados = X[tamanho_de_treino:fim_de_treino]
>>> teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]
>>>
>>> validacao_dados = X[fim_de_treino:]
>>> validacao_marcacoes = Y[fim_de_treino:]
```

Vamos testar primeiro os nossos dados de treino. Começaremos vendo o seu primeiro valor, isto é, a posição 0:

```
>>> treino_dados[0]
array([0, 1, 1, 0, 0])
>>>
```

Então, vamos verificar também o nosso `X` na sua primeira posição:

```
>>> X[0]
array([0, 1, 1, 0, 0])
>>>
```

Por enquanto funcionou como o esperado, então vejamos o último valor dos nossos dados de treino, nesse caso, o valor 799:

```
>>> treino_dados[799]
array([1, 1, 0, 1, 0])
>>> X[799]
array([1, 1, 0, 1, 0])
>>>
```

Como podemos ver está funcionando como o esperado. Será que os nossos dados de teste estão corretos também? Da mesma forma que fizemos com os dados de treino, pegaremos o seu primeiro valor e o último, começaremos pelo primeiro valor:

```
>>> teste_dados[0]
array([1, 1, 0, 0, 1])
>>>
```

Agora vejamos o nosso `x`, porém, há um detalhe, a qual posição o primeiro valor dos nossos dados de teste refere-se 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])
>>>
```

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, test

from sklearn.ensemble import AdaBoostClassifier
modelo = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modelo, treino_dados, treino_marcacoes, te
```

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_marc

from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost, treino_dados, treino_marca
```

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_ac
    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_treino = tamanho_de_treino + tamanho_de_teste

teste_dados = X[tamanho_de_treino:fim_de_treino]
teste_marcacoes = Y[tamanho_de_treino:fim_de_treino]

validacao_dados = X[fim_de_treino:]
validacao_marcacoes = Y[fim_de_treino:]


def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes, teste_dados, teste_marcacoes):
    modelo.fit(treino_dados, treino_marcacoes)

    resultado = modelo.predict(teste_dados)

    acertos = resultado == teste_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(teste_dados)
```

```
        taxa_de_acerto = 100.0 * total_de_acertos / total_de_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_ac
        print(msg)

    from sklearn.naive_bayes import MultinomialNB
    modeloMultinomial = MultinomialNB()
    resultadoMultinomial = fit_and_predict("MultinomialNB", modeloMultinomial, treino_dados, treino_marc

    from sklearn.ensemble import AdaBoostClassifier
    modeloAdaBoost = AdaBoostClassifier()
    resultadoAdaBoost = fit_and_predict("AdaBoostClassifier", modeloAdaBoost, treino_dados, treino_marca

    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)
```

Resumindo

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.