06

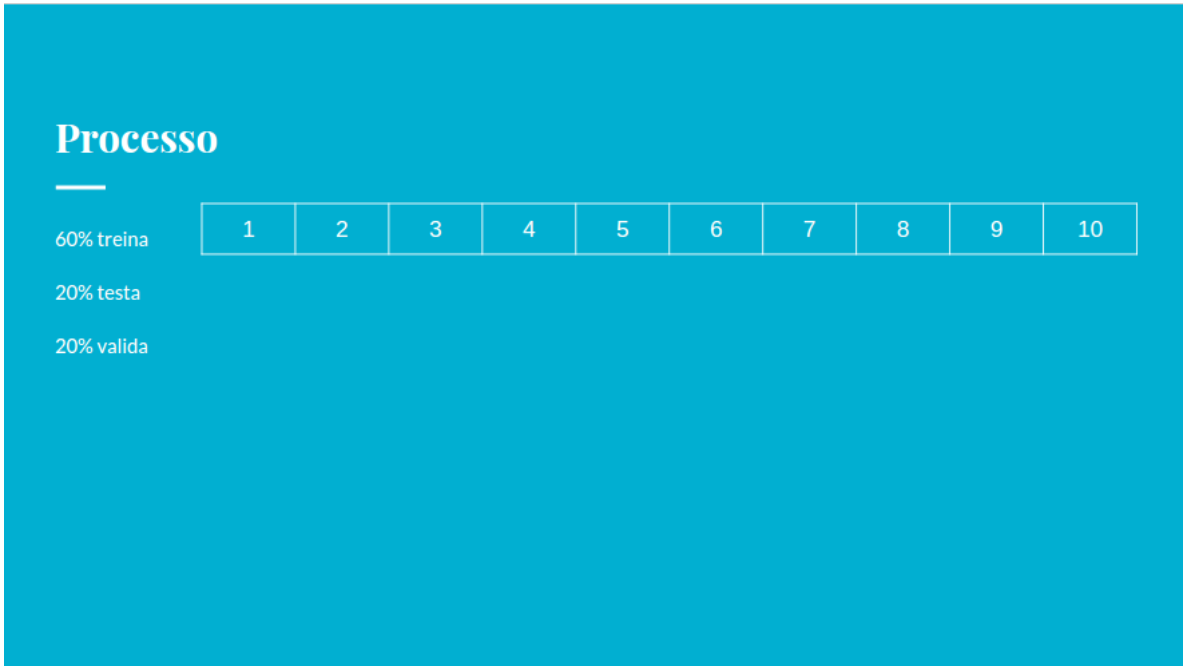
Utilizando o kfold

Utilizando o kfold

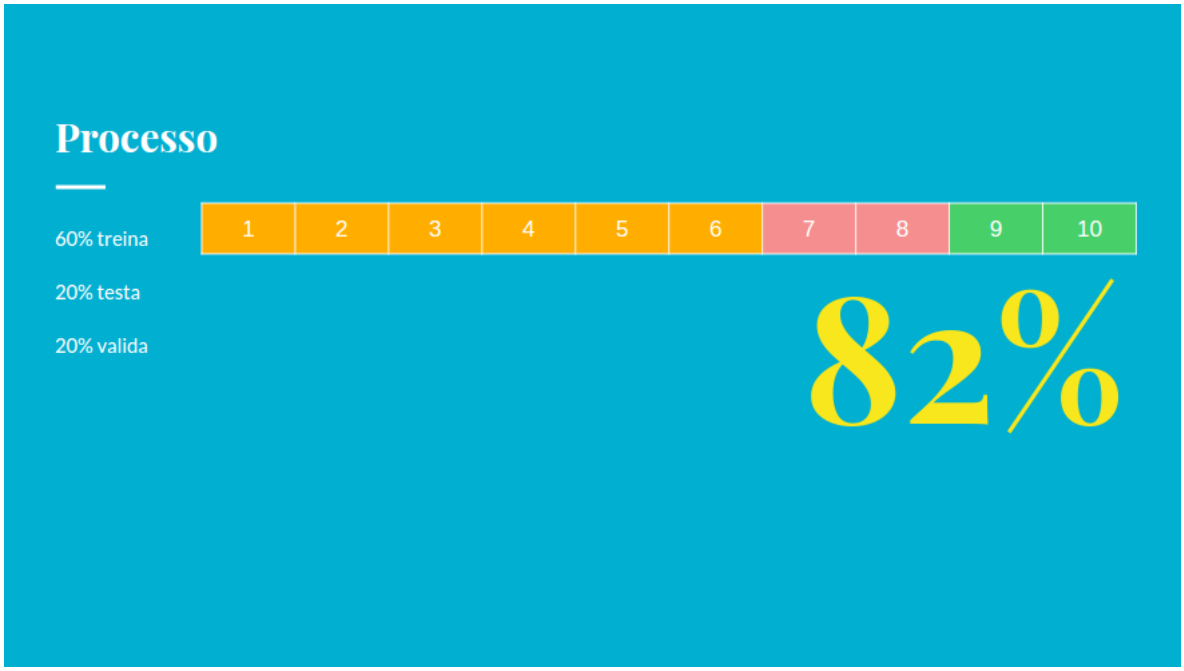
Todos os processos que realizamos até agora seguiram 3 passos, ou seja, primeiro pegamos mais de um algoritmo e pedimos para eles treinarem com determinados dados, e então, realizamos o mesmo teste para cada um deles. Por fim, verificamos qual dentre eles obteve o melhor resultado e o escolhemos como o algoritmo vencedor, isto é, o algoritmo que realiza a validação (teste do mundo real) e nos diz o resultado final que desejamos. Porém, vamos analisar melhor esses passos que realizamos para verificar se realmente é eficaz. Começaremos pelo treino, suponhamos que temos um conjunto de dados e que dividimos ele em:

- **Treino**: 60%.
- **Teste**: 20%.
- **Validação**: 20%.

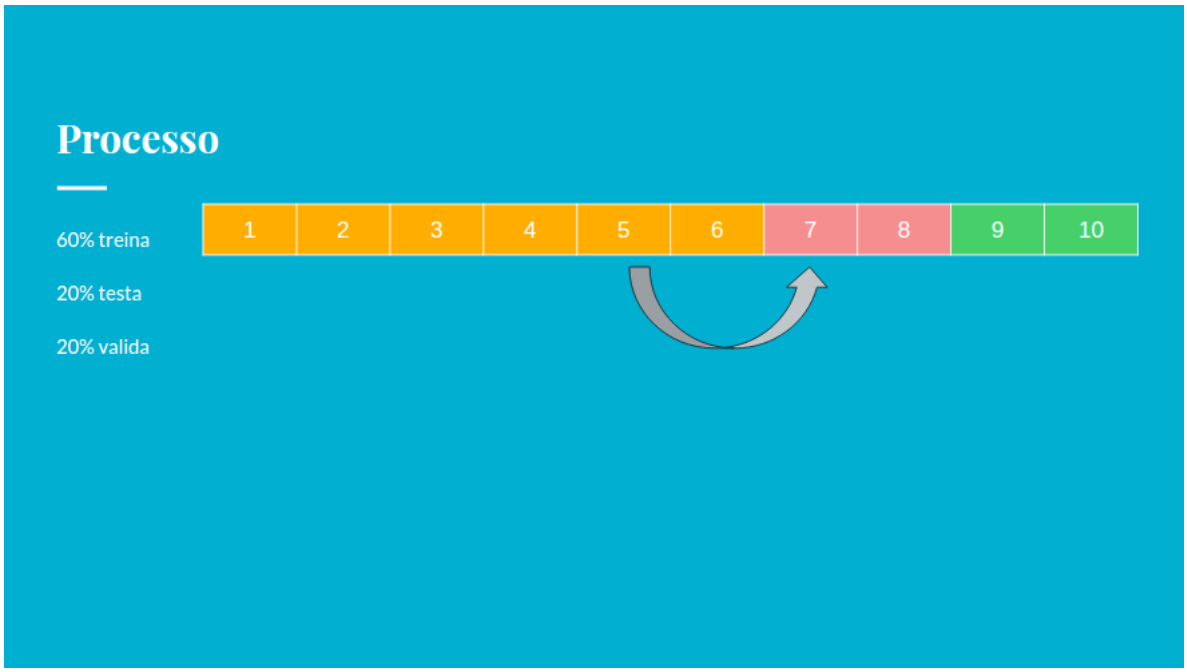
Então vamos utilizar um conjunto de dados de 10 elementos para uma demonstração simplicaficada do processo:



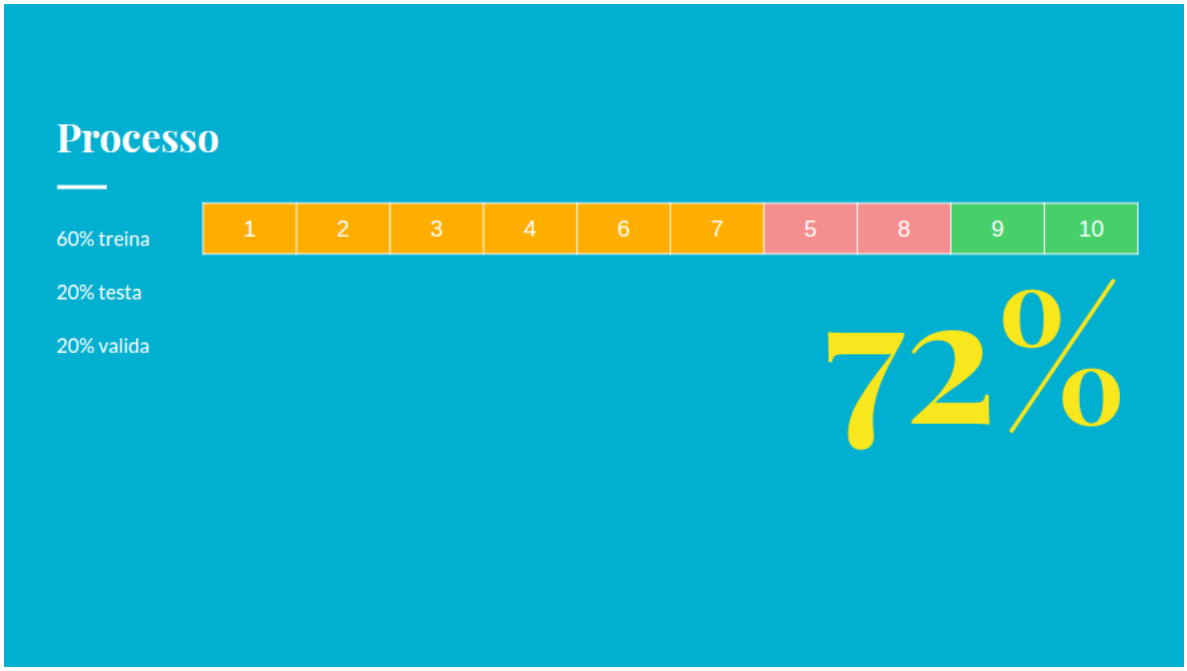
Considerando esse conjunto de dados, faremos a divisão dos mesmos da seguinte maneira:



Observe que foram utilizados 60% {1, 2, 3, 4, 5, 6} dos dados para treino, 20% {7, 8} para teste e mais 20% {9, 10} para validação. Então, quando rodamos esse algoritmo escolhemos o algoritmo vencedor e o seu resultado é de 82%. Aparentemente não temos nenhuma novidade do que já havíamos visto, porém, o que aconteceria se o nosso suposto cliente 5 não tivesse acessado naquele exato momento? Em outras palavras, suponhamos que esse cliente 5 ficasse doente e então aparecesse no lugar do cliente 7, o que aconteceria?

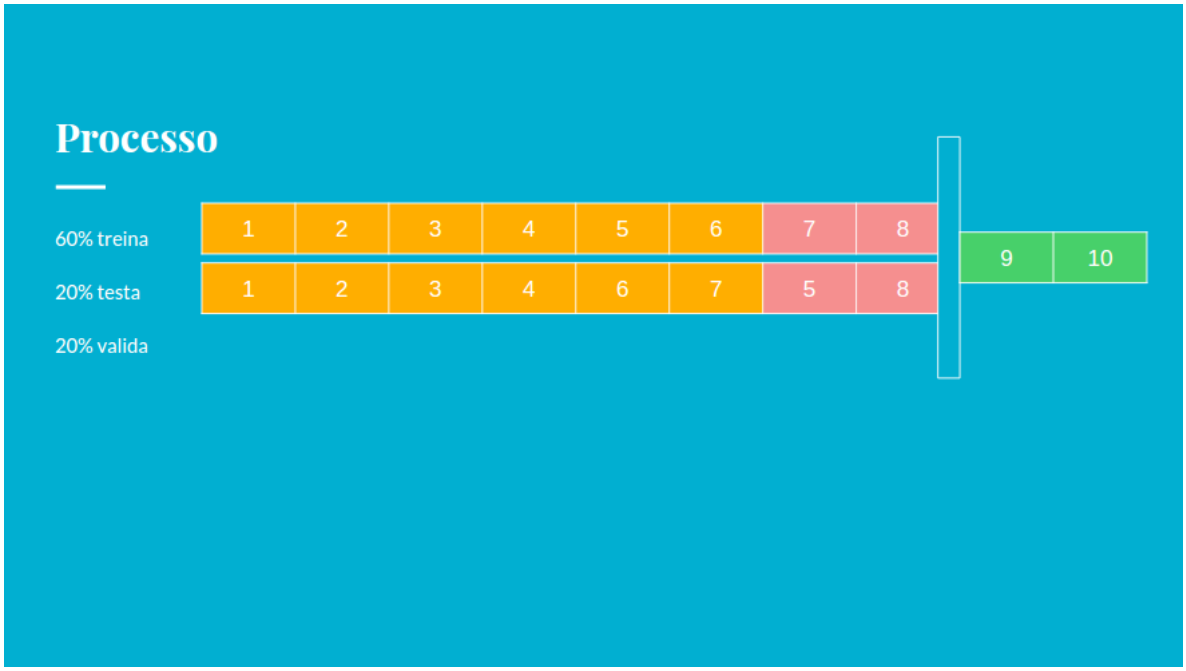


Repare que com essa simples alteração a nossa analise será totalmente diferente, mas por que será totalmente diferente? Pois a função `fit` não vai mais treinar com os dados {1, 2, 3, 4, 5 e 6} e sim com {1, 2, 3, 4, 6, 7}. Mas qual é o impacto que temos nesse cenário? Observe que, se o nosso treino for diferente, provavelmente o resultado será diferente, logo, poderemos utilizar um algoritmo diferente, e então, o resultado da nossa avaliação pode ser diferente, conforme o exemplo abaixo:

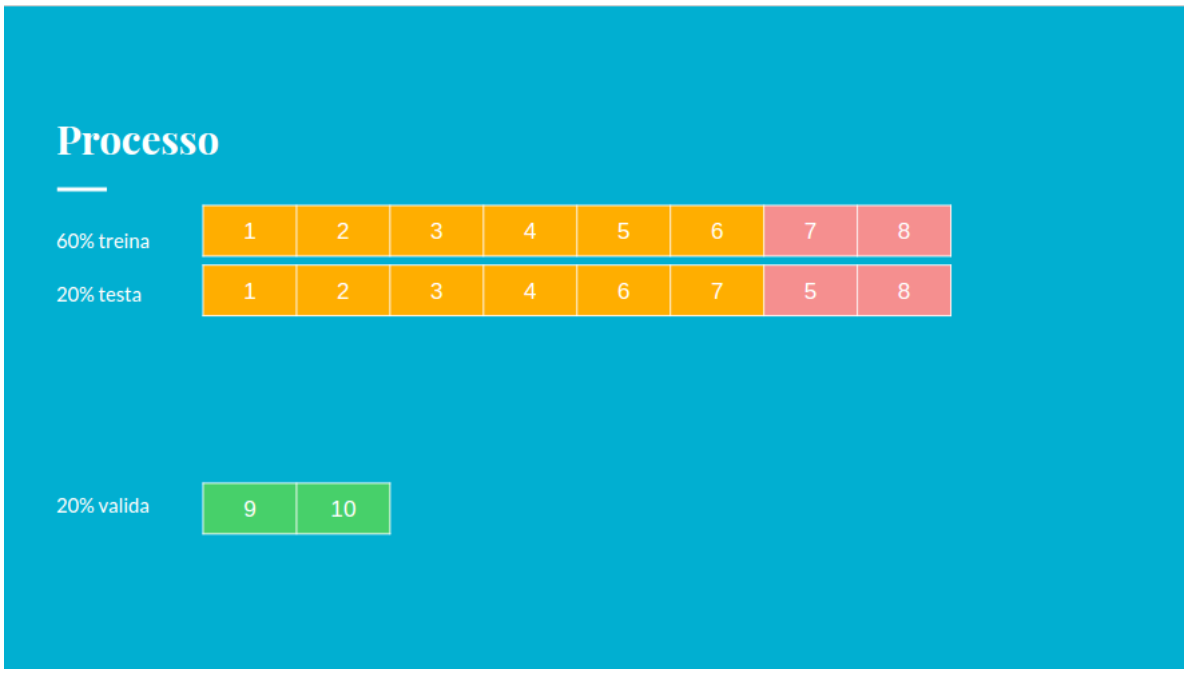


Essa foi apenas uma simulação para verificarmos o quão suscetíveis estamos para qualquer alteração dos nossos dados. Pois, com apenas uma simples alteração, temos resultados totalmente diferentes, como por exemplo, enquantos os nossos dados estavam seguindo a ordem de {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} obtemos 82%, mas, colocando o dado 5 no lugar do dado 7 {1, 2, 3, 4, 6, 7, 5, 8, 9, 10}, conseguimos 72%. Perceba que além de diferente, obtivemos um resultado bem inferior ao anterior, mas, e se ao invés de 72% fosse 92%? Será que podemos confiar nos resultados obtidos da forma que estamos realizando os testes? Pois, se qualquer tipo de variável que os nossos dados contenha for alterada, por exemplo, um cliente que ia comprar num dia, mas comprou em outro dia, ou então, se alguém ficou doente, ou, se ao invés de ir hoje decidiu ir amanhã, ou, qualquer tipo de evento que **altere a ordem dos dados** que recebemos, fará com o que o nosso algoritmo resulte em determinado valor que ele não deveria obter. Em outras palavras, tanto 72%, quanto 82% ou qualquer outro resultado obtido alterando a ordem dos nossos dados são válidos, portanto, os nossos resultados estão muito dependentes da ordem dos nossos dados.

Note que, ao invés de treinarmos e testarmos apenas com uma sequência dos nossos dados, como por exemplo: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ou {1, 2, 3, 4, 6, 7, 5, 8, 9, 10}. Precisamos calcular diversas combinações rodando o `Multinomial` , a partir de todas essas combinações, tiramos a média. Quando tiramos a média de vários resultados do nosso algoritmo, estamos eliminando esse caso de temporalidade, isto é, considerar um resultado que é válido para uma sequência específica de um conjunto de dados. Observe que agora, ao invés de realizar apenas um teste, iremos fazer vários! Repara que tudo que abordamos até agora, ao invés de realizar apenas uma vez, fazemos várias! Em outras palavras, estamos sempre modificando o nosso algoritmo para que ele não seja dependente de um determinado algoritmo, resultado ou sequência de dados, pois queremos que ele funcione para diferentes cenários. Então vejamos como faremos para treinar e testar o nosso algoritmo variando a sequência de dados. Uma abordagem que podemos fazer é:

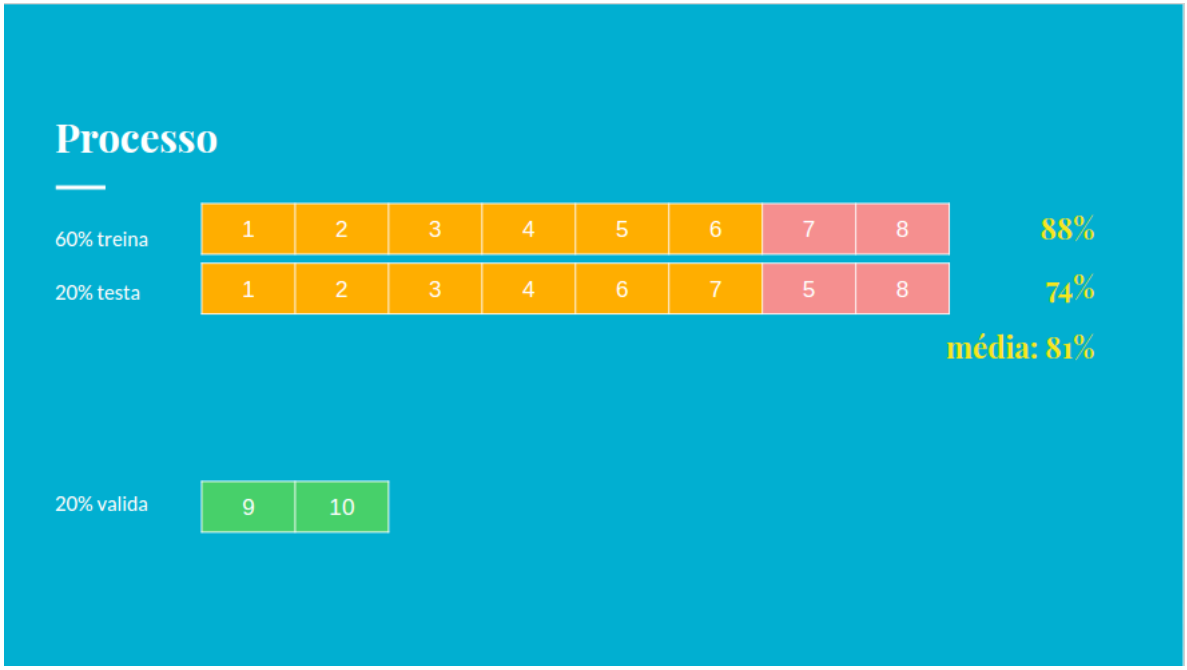


Dado as duas sequências distintas, separamos os mesmos dados de validação para ambas, nesse caso, 9 e 10. Lembre-se que os dados de validação são todos os dados que o nosso algoritmo nunca viu, portanto, todas as vezes que iniciar esse tipo de teste, o primeiro passo é pegar os dados de validação que será utilizado para todas as sequências de dados que escolhemos. Mas qual é o problema de usar os dados de validação durante os testes? É o fato de viciarmos nos resultados obtidos durante esses testes. Portanto, iremos separar 20% dos dados para validação desse teste:

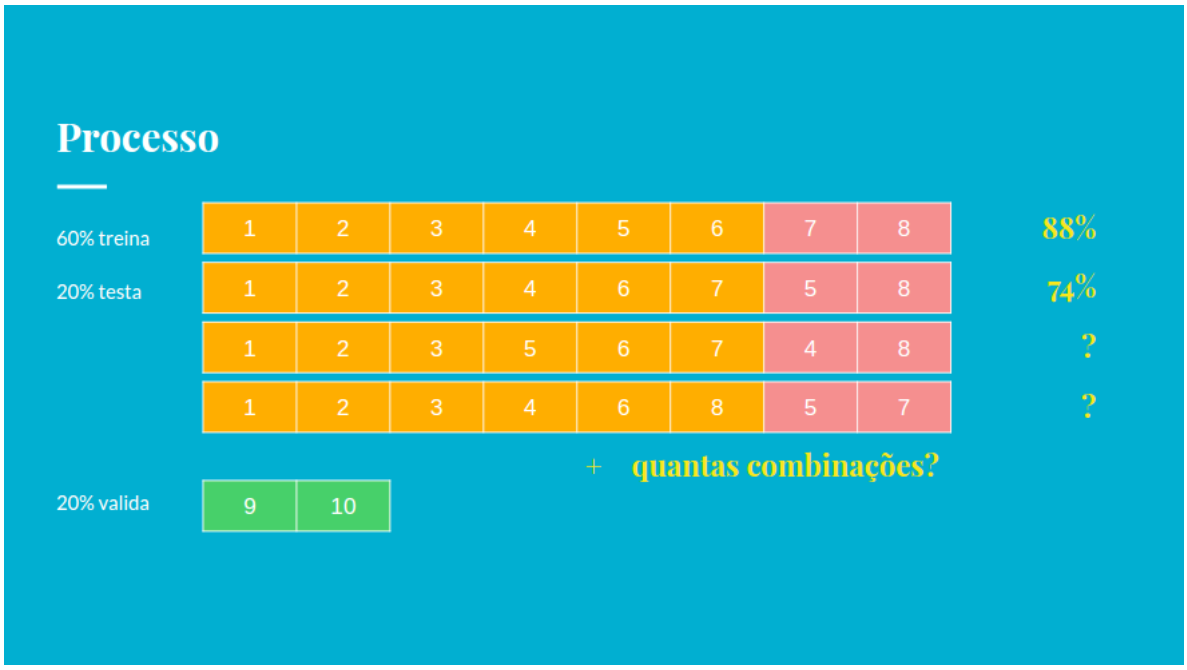


Então rodamos o `Multinomial` tanto com a primeira sequência, quanto a segunda sequência e, por exemplo, obtemos os resultados 88% e 74% respectivamente. Em seguida, tiramos a média entre 88% e 74%:

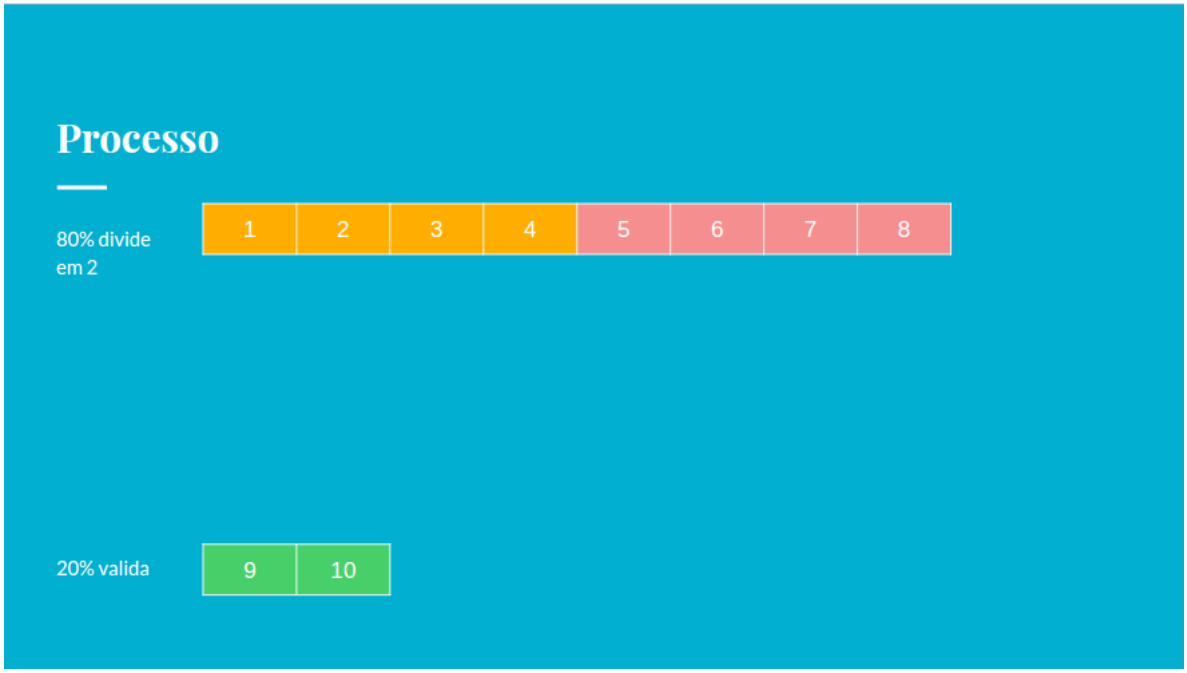
- (88 + 74) / 2 = 81%.



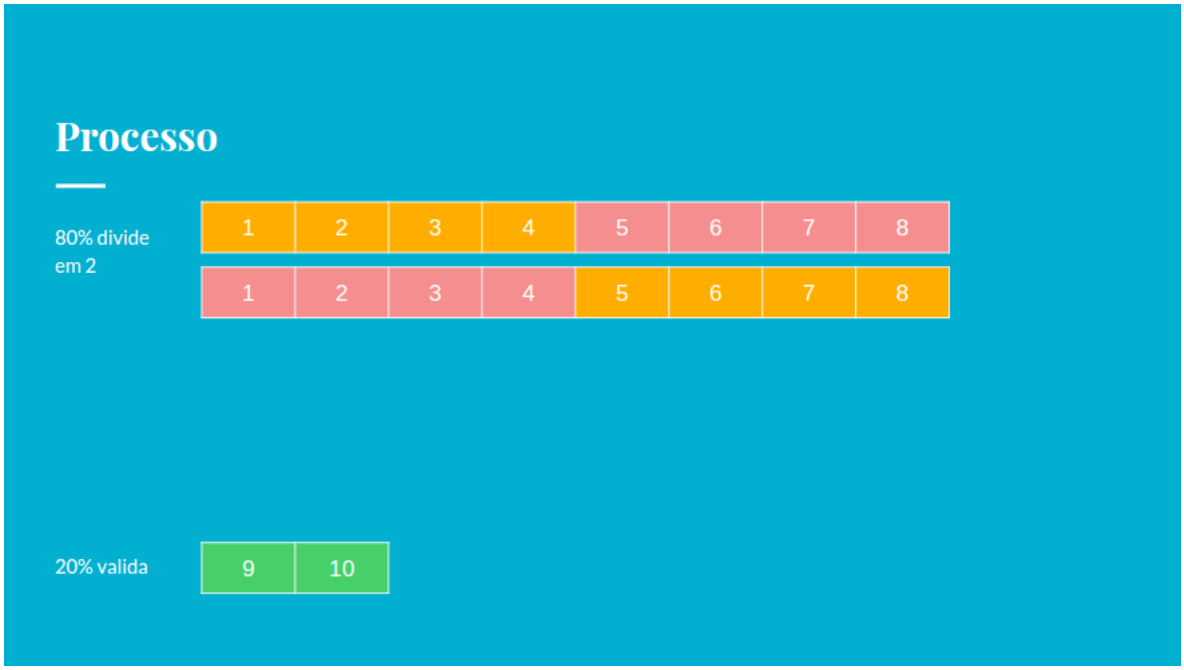
Agora podemos afirmar que, para esse conjunto de dados, independemente da sua sequência, o `Multinomial` obtem o resultado de 81%, pois ambos os resultados testados (81% e 74%) são válidos, por isso precisamos tirar a média de todos os testes realizados e, a partir dela, informar qual é o resultado do algoritmo. Porém, calculamos a média para apenas essas duas sequências, então como ficaria o resultado para essas outras duas sequências?



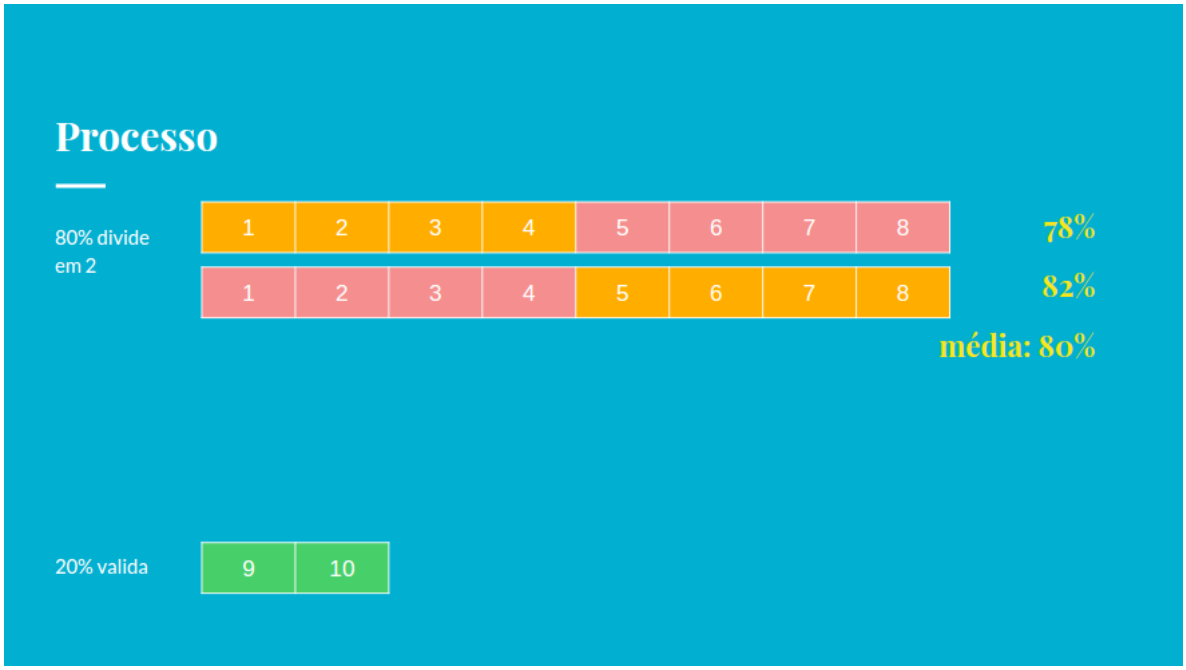
Perceba que mesmo adicionando mais duas sequências, não estamos cobrindo todas as outras possíveis sequências para esse conjunto de dados {1, 2, 3, 4, 5, 6, 7, 8}. Existem diversas outras sequências para esse conjunto de dados, além de ser trabalhoso realizar manualmente, é custoso para a CPU, ou seja, precisamos achar alguma forma mais eficaz para realizar esses testes. Uma abordagem que podemos fazer é quebrar esse conjunto de dados de uma forma que possibilite rodarmos diversas vezes sem que entremos no cenário de temporalidade. Vamos então iniciar quebrando o nosso conjunto de dados em 2:



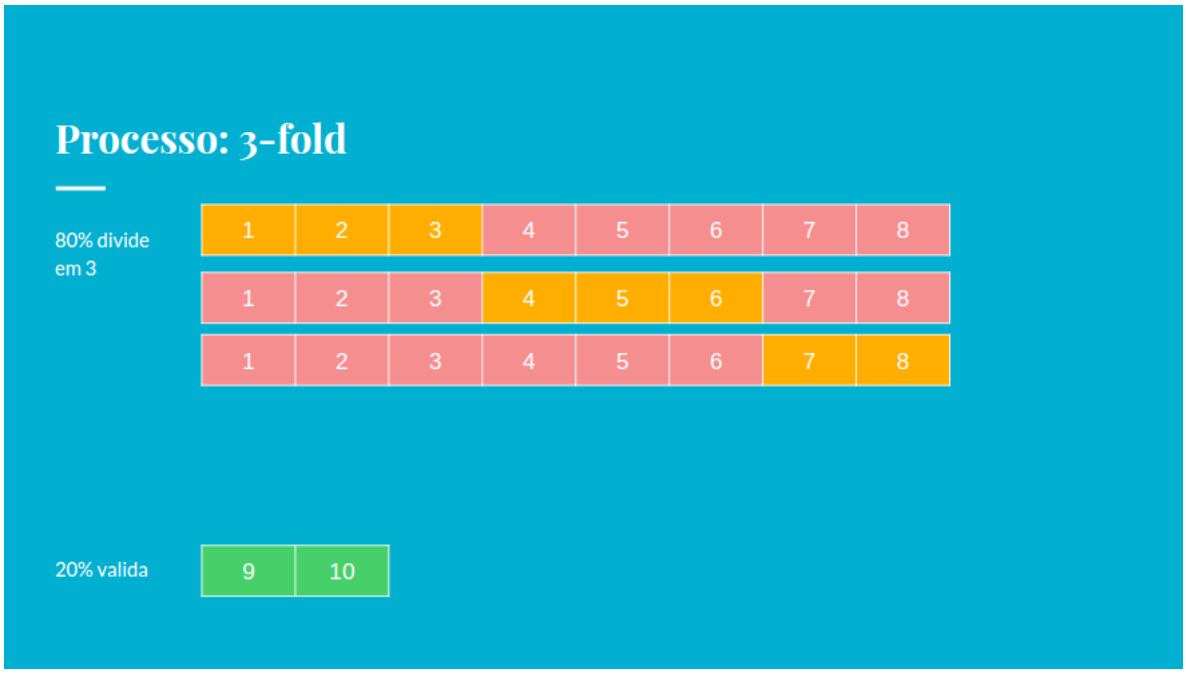
Então, utilizamos a primeira metade para treinar e a segunda para testar. Agora quebramos novamente em dois:



Observe que agora temos duas metades distintas. Então vamos tirar a média:



Note que agora o nosso processo de teste mudou. Ao invés de fazer o teste com diversas sequências, estamos dividindo os dados em duas metades distintas e inversas, ou mellhor, a primeira vez utilizamos os dados {1, 2, 3, 4} para treinar e {5, 6, 7 , 8} para testar, porém, na segunda vez, os dados {1, 2, 3, 4} para testar e {5, 6, 7, 8} para treinar. Quando quebramos um processo de treino e teste em dois pedaços, estamos *foldando* em dois pedaços, nesse caso, *2-fold*. Mas e se quiséssemos quebrar em 3 pedaços? Como seria essa quebra? Vejamos:

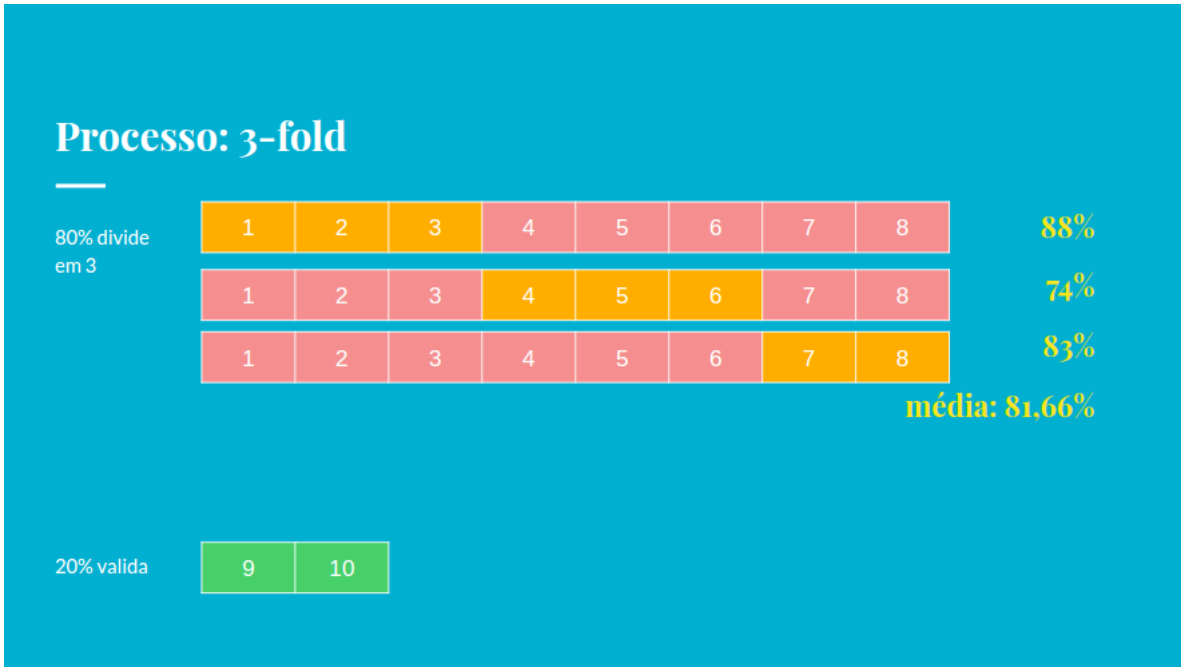


Veja que divimos em 3 trechos proporcionais, ou seja, 3 {1, 2, 3}, 3 {4, 5, 6} e 2 {7, 8}. Então agora, para 3-fold, os nossos treinos e testes ficam da seguinte maneira:

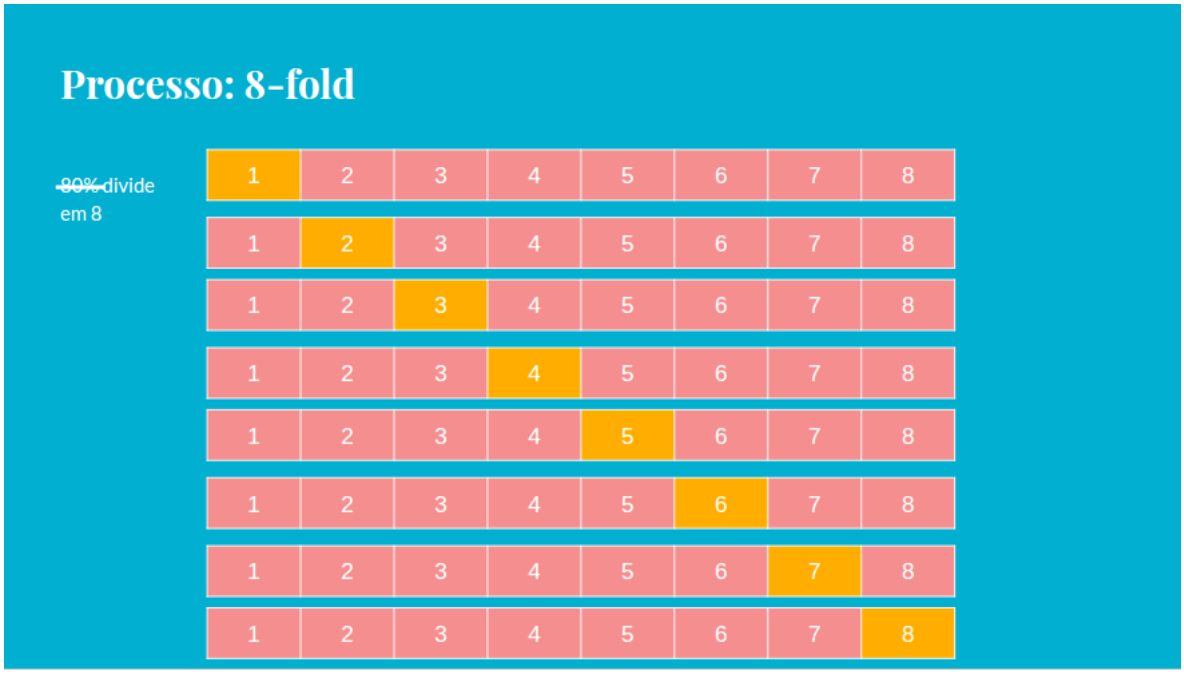
- **treino {4, 5, 6, 7, 8}**: teste {1, 2, 3}
- **treino {1, 2, 3, 7, 8}**: teste {4, 5, 6}
- **treino {1, 2, 3, 4, 5, 6}**: teste {7, 8}

Após separar os dados de treino e teste, o que precisamos fazer? Calcular e tirar a média da mesma forma como fizemos com o 2-fold:

- (88 + 74 + 83) / 3 = 81,66%.



Dessa vez o nosso `Multinomial` está acertando em 81,66% das vezes. E quebrássemos 4? E se fosse N pedaços? Como faríamos? Exatamente como fizemos com os exemplos anteriores, ou seja, quebraríamos pela quantidade de vezes desejadas, e então, usaríamos o primeiro pedaço para testar e o resto para treinar, depois, o segundo para testar e o resto para treinar... Em outras palavras, cada vez que quebramos, precisamos treiná-lo com o resto e testá-lo com o pedaço que separamos. Percebe que agora o nosso processo está ficando cada vez mais lento? Antes testávamos apenas uma única vez, porém, com essa abordagem, estamos rodando o nosso algoritmo de acordo com a quantidade de pedaços, isto é, se quebramos 2 vezes, rodamos duas vezes, se forem 3 pedaços 3 vezes, N pedaços serão N vezes. Porém, precisamos sempre nos atentar que a cada quebra que realizarmos a média pode variar. Perceba que, quebrar os nossos dados em N vezes, não significa que podemos quebrar por qualquer quantidade, pois a quantidade de quebras só pode variar entre 2 e a quantidade total de elementos, ou seja, se temos 8 elementos significa que podemos quebrar os nossos dados de 2 a 8 pedaços. Considerando esse exemplo, como ficaria um 8-fold? Vejamos:



Observe que cada elemento representa uma quebra dos dados, ou seja, cada elemento do conjunto de dados será utilizado para teste e o restante para treino, por exemplo, primeiro utilizaremos o dado 1 para teste e o restante para treino, então, o

dado 2 para teste e o restante para treino e assim sucessivamente. Além disso, temos o caso extremo que é justamente fazer o contrário, em outras palavras, utilizar cada pedaço como treino e o resto como teste. Esse algoritmo que quebra, corta, particiona pedaços dos nossos dados, para realizarmos o nosso treino e teste e depois tirar a média, e então, obtermos o resultado final, é chamado de *k-fold* (https://pt.wikipedia.org/wiki/Valida%C3%A7%C3%A3o_cruzada).

Implementando o k-fold

Vamos então implementar o k-fold no nosso algoritmo! Porém, dessa vez, vamos criar um novo arquivo python com o nome `situacao_do_cliente_kfold.py` , justamente para que não percamos o algoritmo que realizamos no nosso arquivo anterior. Entretanto, iremos copiar o algoritmo do arquivo `situacao_do_cliente.py` para esse novo arquivo. Então o arquivo `situacao_do_cliente_kfold.py` fica da seguinte forma:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia','frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8
porcentagem_de_teste = 0.1

tamanho_de_treino = 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)

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest, treino_dados, treino_marcacoes, t
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, treino_dados, treino_marcacoes, test
resultados[resultadoOneVsOne] = modeloOneVsOne

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

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

maximo = max(resultados)
vencedor = resultados[maximo]

print "Vencerdor: "
print vencedor

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)
```

Então qual será a diferença entre esse novo arquivo e o `situacao_do_cliente.py` ? Será justamente na forma que iremos trabalhar com os dados. Então vejamos esse trecho de código:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia','frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8
porcentagem_de_teste = 0.1
```

Veja que anteriormente estávamos lendo os dados por meio do pandas, e então, pegávamos os dummies e, logo em seguida, separávamos a parte de treino e de teste. Continuaremos com a mesma aborgem, porém, dessa vez, ao invés de separar a parte de treino e de teste, iremos utilizar o mesmo pedaço para ambos os objetivos. Portanto, iremos descartar a variável `porcentagem_de_teste` . Vejamos o restante do código:

```
porcentagem_de_treino = 0.8

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
```

O tamanho de treino ainda permanece, porém, não utilizaremos mais o tamanho de teste, pois, dessa vez, só precisaremos do tamanho de treino que será compreendido como os dados de treino e teste ao mesmo tempo. Então apagaremos a variável `tamanho_de_teste` . Porém, veja que o tamanho de validação não utiliza mais a variável `tamanho_de_teste` , ou seja, iremos fazer apenas a diferença com a variável `tamanho_de_treino` :

```
porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
tamanho_de_validacao = len(Y) - tamanho_de_treino
```

Vamos continuar verificando o nosso código:

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

fim_de_treino = tamanho_de_treino + tamanho_de_teste

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

Tanto os dados e marcações de treino iremos manter. Porém, os dados de testes removeremos, portanto, as variáveis `fim_de_treino` , `teste_dados` e `teste_marcacoes` serão apagadas. E ficamos apenas com os dados e marcações de treino e validação:

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

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

Mas repara que ainda estamos usando a variável `fim_de_treino` para calcular os dados e marcações de validação. Como faremos agora? Ao invés de pegar a partir do `fim_de_treino` utilizaremos a variável `tamanho_de_treino` :

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

Então como estão sendo calculados os nossos dados e marcações nesse instante? Estamos pegando para treino de 0 a 80% e para validação de 80% em diante. Entretanto, a variável `tamanho_de_validacao` não está sendo utilizada, então, por enquanto, podemos deixá-la comentada. Vejamos como ficou o nosso código agora que separamos os dados necessários para esse algoritmo:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia','frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
# tamanho_de_validacao = len(Y) - tamanho_de_treino

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

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

# restante do código
```

Com os dados de treino e validação em mãos, qual é o nosso próximo passo? É de fato realizar um teste, correto? Porém, dessa vez, precisamos utilizar o algoritmo k-fold. Para esse primeiro exemplo, utilizaremos o algoritmo `OneVsRest` . Portanto implementaremos esse algoritmo logo abaixo:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))
```

E agora? Função `fit` ? Não! Lembre-se que iremos utilizar o k-fold, portanto, faremos uma outra abordagem. Nosso primeiro passo será definir o valor do k, isto é, a quantidade de pedaços que iremos quebrar, nesse exemplo, utilizaremos o valor 3:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))

k = 3
```

Qual é o próximo passo? Quebrar os dados em 3 pedaços, e então, rodar o algoritmo com o primeiro pedaço e o resto, depois o segundo pedaço e o resto, em seguida, o terceiro pedaço e o resto. Então, pegaremos os resultados de cada um desses grupos. Para isso, utilizaremos a função `cross_val_score` do `sklearn.cross_validation` , porém, precisamos importá-la:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
from sklearn.cross_validation import cross_val_score
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))

k = 3
```

Agora precisamos chamar a função `cross_val_score` enviando o modelo, os dados de treino e de marcação e por fim, a quantidade de pedaços que iremos quebrar:

```
cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
```

Então retornamos o valor dessa função para uma variável chamada `scores` :

```
scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
```

Essa variável representará todos os resultados obtidos de acordo com a quantidade de pedaços que enviamos. Então vamos imprimí-la também:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
from sklearn.cross_validation import cross_val_score
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))

k = 3

scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
print scores
```

O restante do código abaixo pode ser comentado ou removido, pois nesse instante não utilizaremos, nesse caso eu irei removê-lo para uma melhor visibilidade. Então o nosso código fica da seguinte forma:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia','frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
# tamanho_de_validacao = len(Y) - tamanho_de_treino

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

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


from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
from sklearn.cross_validation import cross_val_score
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))

k = 3

scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
print scores
```

Ao rodar o nosso arquivo `situacao_do_cliente_kfold.py` , obtemos o seguinte resultado:

```
> python situacao_do_cliente_kfold.py
[ 0.91803279  0.93333333  0.91525424]
```

Precisamos agora realizar o próximo passo que é justamente tirar a média a partir dos resultados obtidos. Para isso, utilizaremos o `numpy` do python para calcular a média:

```
# restante do código

scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
print scores
import numpy as np
```

Para extraírmos a média, precisamos utilizar a função `mean`, em seguida, retornamos para a variável `media` e então imprimos essa variável:

```
# restante do código

scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
print scores
import numpy as np
media = np.mean(scores)
print media
```

Rodando novamente o nosso código, obtemos o seguinte resultado:

```
> python situacao_do_cliente_kfold.py
[ 0.91803279  0.93333333  0.91525424]
0.922206785836
```

Como podemos ver a nossa média foi de 92,22%. Mas e se o `k` fosse, por exemplo, 4? Vejamos o resultado:

```
> python situacao_do_cliente_kfold.py
[ 0.93478261  0.91111111  0.93333333  0.93181818]
0.92776130874
```

Veja que agora o resultado foi de 92,77%. E se fosse 10? Vejamos:

```
> python situacao_do_cliente_kfold.py
[ 0.94736842  0.89473684  0.84210526  0.89473684  0.94736842  0.94444444
  0.94444444  0.94117647  0.9375      0.9375    ]
0.923138114895
```

Para `k` igual a 10 temos o resultado de 92,31%. Repara que cada vez que alteramos o valor de `k`, estamos viciando a nossa decisão, pois tentaremos sempre nos basear no valor de `k` que obtivemos o melhor resultado para aquele conjunto de dados, ou seja, todas as vezes que fizermos uso do algoritmo k-fold, precisamos decidir primeiro o valor de `k` para não ficarmos alterando. Portanto, utilizaremos o valor 10.

Implementando o novo fit_and_predict

Repara que mudamos a forma de treinar e testar o nosso algoritmo. Em outras palavras, esses passos que realizamos é justamente o `fit_and_predict`. Portanto, vamos implementá-lo:

```
def fit_and_predict():
```

O que enviaremos como parâmetro? O nome do algoritmo, modelo e os dados e marcações de treino. Lembrando que para esse algoritmo, não teremos os dados ou marcações de teste:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
```

Então vamos definir o valor padrão de `k`, ou seja, vamos atribuir o valor 10:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
```

Agora precisamos calcular o scores, portanto, vamos utilizar a função `cross_val_score`:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
```

Com os scores em mãos, qual é o próximo passo? Calcular a taxa de acerto, certo? Como fazemos isso mesmo? Tirando a média a partir dos `scores`. Para tirarmos a média, utilizaremos novamente o `numpy`:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
    taxa_de_acerto = np.mean(scores)
```

Agora construiremos a mensagem que será exibida com o nome do algoritmo e a taxa de acerto. Então, vamos imprimí-la também:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print msg
```

Por fim, retornamos a taxa de acerto:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print msg
    return taxa_de_acerto
```

Repara que agora o nosso `fit_and_predict` não utiliza a função `fit` ou `predict` como vimos na outra implementação que fizemos. Para o algoritmo k-fold, dentro do `fit_and_predict`, defimos primeiro a quantidade de pedaços que serão quebrados os nossos dados por meio da variável `k`, então pedimos para ele calcular os scores que é justamente chamar a função `cross_val_score` que, internamente, faz todo faz todo o processo para particinar e calcular cada um dos resultados. Por fim, ele calcula a média com o `numpy`, imprime a mensagem com o nome do algoritmo e taxa de acerto, por fim, retorna o resultado da taxa de acerto. Portanto, podemos agora retirar o trecho de código que realizava todos esses passos e, a partir de agora, utilizaremos esse `fit_and_predict`:

```
import pandas as pd
from collections import Counter

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia','frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
# tamanho_de_validacao = len(Y) - tamanho_de_treino

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

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

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print msg
    return taxa_de_acerto
```

Antes de utilizá-lo, vamos primeiro importar o `numpy` e o `cross_val_score` logo no começo do arquivo:

```
import pandas as pd
from collections import Counter
import numpy as np
from sklearn.cross_validation import cross_val_score

# restante do código
```

Para utilizar o `fit_and_predict`, faremos da mesma forma como fizemos no arquivo `situacao_do_cliente`, ou seja, criamos o nosso dicionário:

```
resultados = {}
```

Agora importamos o algoritmo que iremos utilizar, e criamos o seu modelo:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
```

Com o modelo em mãos, chamamos a função `fit_and_predict` enviando o nome do algoritmo, modelo, dados e marcações de treino. Em seguida, atribuímos para a variável que representará o resultado, nesse caso, `resultadoOneVsRest` :

```
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest, treino_dados, treino_marcacoes)
```

Por fim, adicionamos o nosso modelo no dicionário `resultados` enviando o resultado obtido pelo `fit_and_predict` dentro de chaves, nesse caso, o `resultadoOneVsRest` e imprimimos o dicionário para verificar o resultado que o nosso algoritmo nos devolve:

```
resultados[resultadoOneVsRest] = modeloOneVsRest
print resultados
```

O nosso código fica da seguinte maneira:

```
import pandas as pd
from collections import Counter
import numpy as np
from sklearn.cross_validation import cross_val_score

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia','frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df

X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
# tamanho_de_validacao = len(Y) - tamanho_de_treino

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

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


def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print msg
    return taxa_de_acerto

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
```

```
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

print resultados
```

Rodando o nosso algoritmo, obtemos o seguinte resultado:

```
> python situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
{0.92313811489508102: OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None, dual=True, 1
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)}
```

Observe que a média do nosso algoritmo, nesse caso o `OneVsRest`, foi de 92,31% para esses dados. Lembra quanto era o resultado do `OneVsRest` sem utilizar o k-fold? Vejamos:

```
> python situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo OneVsOne: 100.0
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Vencerdor:
OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 100.0
Taxa de acerto base: 82.608696
Total de teste: 23
```

Como vimos, esse 90,90% é um resultado que foi obtido por sorte ou azar, pois quando testamos o nosso algoritmo com apenas uma sequência, ficamos suscetíveis a qualquer evento que possa ocorrer.

Atualmente estamos utilizando o k-fold apenas para o `OneVsRest`, ou seja, precisamos também adicionar os demais algoritimos. Como podemos fazer isso? Simples! Basta copiarmos da mesma forma que está escrito no arquivo `situacao_do_cliente.py`. Entretanto, vamos copiar apenas um algoritmo para verificar se tudo ocorre como o esperado, nesse caso, copiaremos o `OneVsOne`:

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest
```

```
from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, treino_dados, treino_marcacoes, test
resultados[resultadoOneVsOne] = modeloOneVsOne

print resultados
```

O único detalhe sobre copiar o código do arquivo `situacao_do_cliente.py` , é que não utilizamos mais os dados e marcações de teste, portanto, vamos excluí-los:

```
# restante do código

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, treino_dados, treino_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

print resultados
```

Vamos testar o arquivo `situacao_do_cliente_kfold.py` para verificar se tudo ocorre como o esperado:

```
> python situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
Taxa de acerto do OneVsOne: 0.994444444444
{0.92313811489508102: OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None, dual=True, 
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1), 0.99444444444444446: OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)}
```

Funcionou sem nenhum problema, porém, observe o resultado obtido pelo algoritmo `OneVsOne` foi de 99,44%. Lembra que na implementação que fizemos no arquivo `situacao_do_cliente.py` ele apresentou um resultado de 100%? Em outras palavras, quando utilizamos o método do k-fold, alcançamos um resultado mais realista ao qual tinhamos anteriormente. Agora precisamos adicionar os demais algoritmo, o nosso código fica da seguinte maneira:

```
import pandas as pd
from collections import Counter
import numpy as np
```

```
from sklearn.cross_validation import cross_val_score

df = pd.read_csv('situacao_do_cliente.csv')
X_df = df[['recencia','frequencia', 'semanas_de_inscricao']]
Y_df = df['situacao']

Xdummies_df = pd.get_dummies(X_df).astype(int)
Ydummies_df = Y_df


X = Xdummies_df.values
Y = Ydummies_df.values

porcentagem_de_treino = 0.8

tamanho_de_treino = porcentagem_de_treino * len(Y)
# tamanho_de_validacao = len(Y) - tamanho_de_treino

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

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

def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)
    print msg
    return taxa_de_acerto

resultados = {}

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
resultadoOneVsRest = fit_and_predict("OneVsRest", modeloOneVsRest, treino_dados, treino_marcacoes)
resultados[resultadoOneVsRest] = modeloOneVsRest

from sklearn.multiclass import OneVsOneClassifier
modeloOneVsOne = OneVsOneClassifier(LinearSVC(random_state = 0))
resultadoOneVsOne = fit_and_predict("OneVsOne", modeloOneVsOne, treino_dados, treino_marcacoes)
resultados[resultadoOneVsOne] = modeloOneVsOne

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

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

print resultados
```

Vejamos o resultado obtido por todos os algoritmos utilizando o k-fold:

```
> python situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
Taxa de acerto do OneVsOne: 0.994444444444
Taxa de acerto do MultinomialNB: 0.829977210182
Taxa de acerto do AdaBoostClassifier: 0.762947196422
{0.8299772101823184: MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True), 0.7629471964224285
          learning_rate=1.0, n_estimators=50, random_state=None), 0.92313811489508102: OneVsRestClas
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1), 0.99444444444444446: OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)}
```

Observe que ele calculou normalmente a média de todos os algoritmos, e obtemos os seguintes resultados:

- **OneVsRest**: 92,31%.
- **OneVsOne**: 99,44%.
- **Multinomial**: 82,99%.
- **AdaBoost**: 76,29%.

Mesmo com uma abordagem diferente, o `OneVsOne` continua sendo o vencedor. Entretanto, precisamos verificar qual é o algoritmo vencedor. Como podemos implementar isso no nosso algoritmo? Exatamente da mesma forma como fizemos no `situacao_do_cliente.py`. Em outras palavras, podemos copiar e colar o código novamente:

```
# restanto do código

print resultados

print "Vencerdor: "
print vencedor

maximo = max(resultados)
vencedor = resultados[maximo]

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)
```

Perceba que adicionamos a função `teste_real`, portanto, precisamos adicioná-la no nosso arquivo `situacao_do_cliente_kfold.py` também:

```
def fit_and_predict(nome, modelo, treino_dados, treino_marcacoes):
    k = 10
    scores = cross_val_score(modelo, treino_dados, treino_marcacoes, cv = k)
    taxa_de_acerto = np.mean(scores)
    msg = "Taxa de acerto do {0}: {1}".format(nome, taxa_de_acerto)

    print msg
    return taxa_de_acerto

def teste_real(modelo, validacao_dados, validacao_marcacoes):
    resultado = modelo.predict(validacao_dados)

    acertos = resultado == validacao_marcacoes

    total_de_acertos = sum(acertos)
    total_de_elementos = len(validacao_marcacoes)

    taxa_de_acerto = 100.0 * total_de_acertos / total_de_elementos

    msg = "Taxa de acerto do vencedor entre os dois algoritmos no mundo real: {0}".format(taxa_de_ac
    print(msg)

# restante do código
```

Vamos testar novamente o nosso arquivo `situacao_do_cliente_kfold.py` :

```
> python situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
Taxa de acerto do OneVsOne: 0.994444444444
Taxa de acerto do MultinomialNB: 0.829977210182
Taxa de acerto do AdaBoostClassifier: 0.762947196422
{0.8299772101823184: MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True), 0.7629471964224285
          learning_rate=1.0, n_estimators=50, random_state=None), 0.92313811489508102: OneVsRestClas
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1), 0.99444444444444446: OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)}
Vencerdor:
OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)
Traceback (most recent call last):
  File "situacao_do_cliente_kfold.py", line 79, in <module>
    teste_real(vencedor, validacao_dados, validacao_marcacoes)
  File "situacao_do_cliente_kfold.py", line 36, in teste_real
    resultado = modelo.predict(validacao_dados)
  File "/usr/local/lib/python2.7/dist-packages/sklearn/multiclass.py", line 538, in predict
    Y = self.decision_function(X)
  File "/usr/local/lib/python2.7/dist-packages/sklearn/multiclass.py", line 557, in decision_funct
    check_is_fitted(self, 'estimators_')
```

```
  File "/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py", line 678, in check_is_f
    raise NotFittedError(msg % {'name': type(estimator).__name__})
sklearn.utils.validation.NotFittedError: This OneVsOneClassifier instance is not fitted yet. Call 'f
```

Observe que o algoritmo apresentou um erro na linha 79 que é justamente a chamada à função `teste_real` . Mas porque isso aconteceu? Veja que o erro nos informa que o `OneVsOneClassifier` (algoritmo vencedor) não foi "fitted", ou melhor, não chamamos a função `fit` , portanto, ele não consegue prever esses dados. Para resolver esse detalhe, basta apenas adicionarmos o `fit` antes de fazer a chamada à função `teste_real` :

```
# restante do código

print "Vencerdor: "
print vencedor

vencedor.fit(treino_dados, treino_marcacoes)

teste_real(vencedor, validacao_dados, validacao_marcacoes)
```

Testando novamente o nosso algoritmo:

```
> python situacao_do_cliente_kfold.py
Taxa de acerto do OneVsRest: 0.923138114895
Taxa de acerto do OneVsOne: 0.994444444444
Taxa de acerto do MultinomialNB: 0.829977210182
Taxa de acerto do AdaBoostClassifier: 0.762947196422
{0.8299772101823184: MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True), 0.7629471964224285
          learning_rate=1.0, n_estimators=50, random_state=None), 0.92313811489508102: OneVsRestClas
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1), 0.99444444444444446: OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)}
Vencerdor:
OneVsOneClassifier(estimator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 100.0
Taxa de acerto base: 75.555556
Total de teste: 45
```

Agora o nosso algoritmo funcionou como o esperado. Note que no teste do mundo real o `OneVsOne` acertou 100% das vezes. Além disso, observe que agora, ao invés de utilizar 23 elementos para teste, foram utilizados 45. Mas porque aconteceu isso? Lembra que para o k-fold estamos utilizar 80% para treinar e testar? Então, sobraram 20% para validação, por isso que foram 45 testes ao invés de 23. E quanto usavamos no arquivo `situacao_do_cliente.py` para atingir 23? Vejamos:

```
porcentagem_de_treino = 0.8
porcentagem_de_teste = 0.1
```

Observe que no nosso outro algoritmo utilizamos 80% para treino e 10% para teste, ou seja, sobravam apenas 10% para validação, é exatamente por esse motivo que o teste do mundo real era realizado com 23 elementos. Repara que não podemos também ficar alterando o número de elementos que utilizaremos tanto para os dados quanto para os testes, pois dessa forma estaremos viciando a nossa decisão humana. Por exemplo, vimos o resultado com 80%, mas, quanto será que ele acertaria se fosse 90%? E 60%? Que tal dessa vez X%? Perceba que dessa forma escolheremos o melhor resultado para esse caso, em outras palavras, estaremos viciando a nossa decisão. Esse é um grande problema da ciência, pois existem diversas práticas que precisamos seguir para ter a certeza de que tudo que estamos fazendo, de fato é válido. Entretanto, se não anotarmos todo o processo que foi realizado para chegarmos a um resultado, como por exemplo, uma mudança de valores para um determinado teste, todos irão pensar que o nosso algoritmo acerta sempre e pra tudo, pois não apresentamos todos os passos que foram realizados para chegar ao resultado adquirido. Repare que é bem comum as pessoas não publicarem exatamente todos os passos realizados, em outras palavras, todas as falhas e sucessos de um algoritmo, pois existe um "vício humano" que exibe apenas tudo que obteve sucesso, e então, acreditamos que a amostra que nos foi apresentada, funciona muito bem para tudo sendo que, na verdade, existem diversos problemas que não foram apresentados. Portanto, sempre que tivermos que apresentar os resultados adquiridos, precisaremos apresentar também todo o processo que foi realizado, justamente para comprovar se existe ou não um vício que possa invalidar os nossos resultados.