07

Introdução

Novos conceitos de classificação

Até o momento, vimos diversos exemplos de classificação, tais como, verificar se um e-mail é `spam` ou não, se um animal é um cachorro ou um porco, se um cliente vai comprar ou não no nosso site entre outros exemplos. Vimos que podem existir milhares de classificações no nosso dia a dia que conseguimos fazer, porém, é interessante o computador realizar esse tipo de tarefa para nós, justamente para que consigamos prevêr resultados esperados, ou seja, evitar de abrir um e-mail que é `spam`, entregar um produto que o nosso cliente espera, conseguir conversar com alunos que provavelmente irão mal na prova entre diversas ações que podemos tomar para alcançar o nosso objetivo. Todo esse conteúdo foi visto até agora, então vem a pergunta:

- Existe mais alguma coisa que precisamos saber?

Perceba que, no mundo real, poderíamos levar mais adiante, em outras palavras, ao invés de querermos saber apenas se o cliente vai comprar ou não, ou então, se vai renovar ou não, podemos também ter o interesse em uma gama maior de informações, por exemplo, saber sobre sua satisfação, isto é, se ele está feliz ou neutro ou triste ou super contente ou com raiva... Perceba que podemos ter todos esses clientes, então vem as questões:

- Com qual desses clientes temos que interagir?
- Como devemos interagir com cada um desses tipos de clientes?

Um outro exemplo para esse tipo de cenário seria, dado um produto novo que que estamos querendo lançar, surgem dúvidas como:

- Será que vai ser um produto de sucesso?
- Será que vai ser um fracasso?
- Será que vai ser um produto neutro?

Perceba que nesse instante, podemos ter mais de 2 categorias, ou seja, 3, 5, 10 ou muito mais categorias para um serviço, produto ou qualquer tipo de elemento que queremos classificar.

Além disso, atualmente, estamos testando e validando os nossos dados de uma maneira bem simples, que é testar, escolher o melhor entre os 2 algoritmos e validar. Porém, e quando tivermos, por exemplo, 4, 10 ou mais algoritmos? Também queremos ser capazes de validá-los! Em outras palavras, eleger o vencedor entre uma grande variação de algoritmos que iremos utilizar. Também queremos realizar testes diferentes aos quais vimos até agora, por exemplo, os nossos testes são realizados apenas uma única vez, e se esses dados tiverem algum vício? Ou então, e se tiver um erro que desconhecemos? Como podemos ter a certeza de que, dado os testes que realizamos para os nossos algoritmos, o resultado final que obtivemos é de fato esperado para o mundo real? Esse também será um dos assuntos que iremos abordar no decorrer dos próximos capítulos.

Até agora estávamos trabalhando apenas com números, isto é, categorias númericas, ou seja, essas categorias baseadas em apenas números ou em palavras que eram transformadas em números diretamente, por exemplo, 0, 1 ou 2. Porém, no mundo real, também trabalharemos com produção de texto, em outras palavras, precisamos também classificar textos! Ou melhor, dado um parâgrafo escrito de um livro, queremos saber se esse livro vendará muito ou não. Consegue perceber que dessa vez a classificação é referente ao conteúdo do texto? Isto é, a língua portuguesa, as próprias palavras em si. Mas como podemos processar os nossos texto para classificar as palavras? Seja por algum sentimento, intenção ou qualquer categoria dessas palavras que queremos classificar. Veremos que para esse tipo de classificação tomaremos alguns cuidados, como por

exemplo, realizar limpeza nos nossos textos para evitar qualquer lixo, ou seja, qualquer informação que não faça sentido algum com o que queremos classificar. Perceba que agora temos diversas novas situações que ainda não vimos até agora, e no decorrer dos próximos capítulos, abordaremos cada uma delas.

Classificando um elemento com 3 categorias

No mundo real, sabemos que em diversas situações podemos realizar classificações entre duas categorias, como por exemplo verificar os alunos que irão reprovar ou não, ou os alunos que irão desistir da escola durante o período letivo ou não. Mas, além de duas categorias, podemos nos interessar em classificar mais de duas categorias, porém, como fazemos para classificar um elemento que possui mais de duas categorias diferentes? Por exemplo, quando recebemos um e-mail, podemos categorizar ele como `spam` ou não, por enquanto estamos classificando apenas duas categorias, mas, além da classificação de `spam`, podemos também classificá-lo como `spam`, `promoção`, `fórum`, `update`, `importante`, `familiar` ou `normal`. Apenas com esse pequeno exemplo que fizemos, apareceram 7 categorias distintas para classificarmos um e-mail. Então observe que agora, quando um e-mail chegar, não iremos apenas tentar classificá-lo entre `spam` ou não, teremos 7 categorias possíveis para a sua classificação! No nosso primeiro exemplo utilizaremos apenas 3 categorias que demonstra, conceitualmente, a mesma situação, porém, de uma forma reduzida. Então vamos dar uma olhada no nosso primeiro cliente? Esse nosso cliente possui as seguintes características:

- **Último acesso**: Visitou ontém (1 dia atrás).
- **Frequência de acesso**: Visitou 4 dias.
- **Se inscreveu**: 4 semanas atrás.
- **Está se sentido**: Alegre.

A primeira característica informa a última visita que o cliente fez no nosso site, nesse caso, ontém, um dia atrás. Mas o que ela significa exatamente? Essa informação responde a seguinte pergunta: "O quão recente foi o último acesso desse cliente?". Identificamos isso como **recência**. Quando dizemos o último acesso, ou melhor, a recência do nosso cliente ao nosso site, não significa que precisa ser necessariamente em dias, poderia, por exemplo ser em horas, semanas, meses ou anos, ou seja, podemos medir com qualquer unidade, nesse exemplo utilizaremos dias como unidade de medida. Mas por que nos interessaria saber a recência desse cliente ao nosso site? Pois, baseado nessa informação, podemos tentar medir como ele está se sentindo em relação a um cliente que acessou há uma semana, ou há um mês ou há ano atrás.

A segunda característica descreve qual é a frequência de visitas que o cliente teve após sua inscrição. Isso não significa uma frequência seguida. Em outras palavras, estamos respondendo a seguinte pergunta: "O quão frequente foi o acesso do nosso cliente"?, nesse caso, 4 dias distintos. Da mesma forma que vimos na característica de recência, poderíamos utilizar outras unidades de medida para essa informação, como horas, meses ou anos. Então repara que agora estamos utilizando uma variável diferente, pois estamos medindo quantos **dias distintos** o cliente visitou o nosso site, isto é, a sua frequência. Esse tipo de informação pode ser relevante, pois uma pessoa que acessou 4 dias e outra que acessou nenhuma dia após a inscrição, nos indica se o meu produto é ou não interessante.

A próxima característica refere-se ao tempo que o cliente se inscreveu no nosso site. Mas porque é importante saber há quanto tempo um cliente se inscreveu no nosso site? Pois dependendo do tempo de inscrição, cada cliente poderá conter diferentes tipos de comportamentos, ou seja, um cliente que se inscreveu semana passada tem um comportamento totalmente diferente de um cliente que se inscreveu há 3 anos, podemos usar o site do Alura como exemplo, pois no início haviam entre 5 a 10 cursos, porém, atualmente, existem mais de 200 cursos. Percebe que o tempo que um cliente está dentro de uma plataforma influencia? Porém, isso não significa que quanto mais ou menos tempo o cliente estiver associado a um produto ele vai se sentir melhor ou pior, por exemplo, existem empresas que, quanto mais tempo ficamos vinculados aos seus produtos, mais contentes ficamos, mas também, existem empresas que, quanto menos tempo estamos atrelados ao seu produto, mais chateados ficamos. Então podemos concluir que não existe uma regra geral para dizer se ele ficará contente ou não dependendo do tempo que ele se inscreveu, portanto, teremos que analisar essa variável para que o nosso algoritmo consiga encontrar um padrão para esse tipo de dado. Repare que além dessas 3 variáveis (`recencia`, `frequência e tempo de`

`cadastro`), poderíamos utilizar diversas outras, por exemplo, quanto dinheiro ele gastou, quantos vídeos ele assistiu (considerando o Alura como exemplo), quantos exercícios ele fez, quantas pessoas ele ajudou, quantas dúvidas ele postou entre outras. Porém, iremos apenas utilizar essas variáveis para o nosso exemplo.

Então o que queremos classificar com apenas essas 3 variáveis? Para esse exemplo, tentaremos prevêr se o nosso cliente está ou alegre ou neutro ou chatedo com o nosso produto. Nesse primeiro exemplo, vimos as características de um cliente que está alegre. Mas o que significa cada um desses estados? Vejamos:

- **Alegre**: Significa que ele está satisfeito, portanto não precisamos nos preocupar tanto.
- **Neutro**: Significa que ele não demonstra nem satisfação ou insatisfação com o nosso produto, ou seja, de repente podemos considerar um contato para entender melhor a situação dele.
- **Chateado**: Significa que ele está insatisfeito, logo, é de extrema importância entendermos o que está acontencendo, em outras palavras, precisaremos verificar o motivo dele ficar chateado para tentar ajudá-lo de alguma maneira e recuperar a situação.

Observe que agora, ao invés de duas categorias, iremos classificar os nossos clientes entre 3 categorias, isto é, alegre, neutro e chateado. Vejamos um outro exemplo de cliente:

- **Último acesso**: Visitou anteontem (2 dia atrás).
- **Frequência de acesso**: Acessou 1 dia.
- **Se inscreveu**: 2 semanas atrás.
- **Está se sentido**: Neutro.

Como foi o comportamento desse cliente no nosso site? Podemos ver que o último acesso dele foi há 2 dias, ou seja, sua recência é 2. Esse cliente acessou em um único dia, porém, isso não significa que ele acessou apenas uma vez, pois, nesse mesmo dia, ele poderia ter acesso várias vezes, mas perceba que não estamos analisando a quantidade de vezes que ele acessou e sim quantos **dias distintos** ele acessou, por isso marcamos apenas como 1 dia. Além disso, esse cliente se inscreveu há 2 semanas, e então, sabemos que ele está neutro, pois o pessoal do comercial entrou em contato com ele e verificou que ele está neutro. Precisamos entrar em contato justamente para entender como ele está se sentido, da mesma forma como classificamos se um e-mail é ou não um *spam*, ou seja, precisamos ler o e-mail para podermos classificá-lo. Então perceba que para essa situação, temos esses conjuntos de dados que foram extraídos de uma base de dados de um site de vendas e precisamos entrar em contato para verificar qual é a sensação dele com o nosso produto. Mas teremos que fazer exatamente a mesma coisa em todos os casos? Não! Tudo depende do conjunto de dados utilizado e o que queremos classificar, por exemplo, poderíamos conter um determinado conjunto de dados sobre um produto nosso, e então, queremos prevêr se ele fará sucesso ou não antes mesmo de lançar. Perceba que para esse caso, não faz sentido perguntarmos ao produto se ele fará sucesso ou não, em outras palavras, utilizaremos apenas o nosso conjunto de dados e, baseado nesse conjunto de dados, o classificaremos. Por fim, vejamos mais um possível cliente para o nosso exemplo;

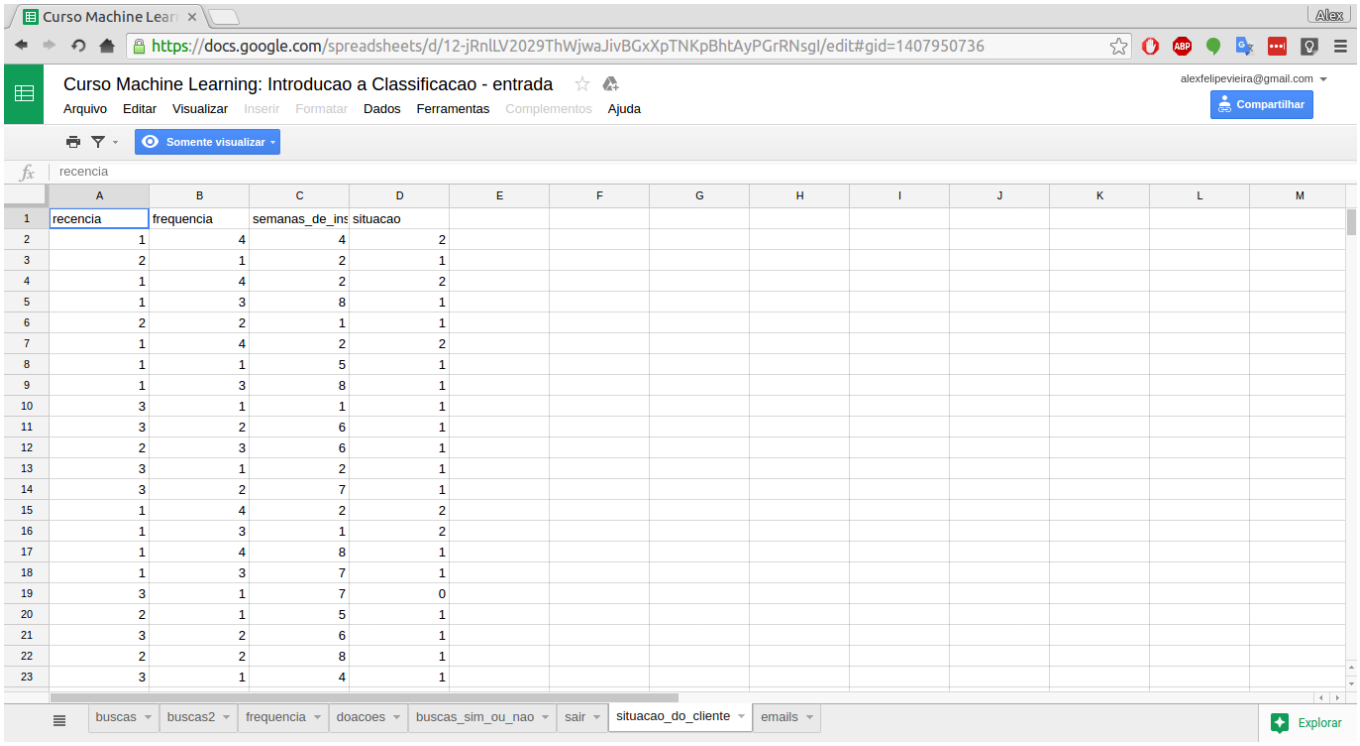
- **Último acesso**: Visitou 3 dias atrás.
- **Frequência de acesso**: Acessou 1 dia.
- **Se inscreveu**: 7 semanas atrás.
- **Está se sentido**: Chateado.

Observe que esse cliente teve uma recência de 3 dias atrás, então obteve uma frequência de apenas 1 dia, porém esse cliente se inscreveu há 7 semanas. Por fim, verificamos que ele está cheteado.

Mas e agora? O que fazemos com essas informações? Iremos criar e preencher a nossa tabela de dados da mesma forma que fizemos anteriormente. Vejamos a estrutura da nossa tabela:

Repare que a nossa tabela contém 4 colunas referente aos dados que temos de nossos clientes, ou seja, `recencia`, `frequencia`, `semanas` e `situacao` dos clientes que vimos como exemplo. Veja que no último campo preenchemos com palavras, isto é, `alegre`, `neutro` e `chateado`, porém, iremos traduzir essas palavras em números, nesse caso usaremos 2 para `alegre`, 1 para `neutro` e 0 para `chateado`:

Note que caímos em um cenário muito similar ao que já vimos até agora, então qual é a diferença desse caso para os demais que vimos? É justamente o fato de utilizar 3 categorias para classificar os nossos elementos, em outras palavras, verificar se o cliente está `alegre`, `neutro` ou `chateado`. Consirando todo esse contexto que acabamos de ver, precisamos agora implementar o nosso código. Antes mesmo de criar o nosso algoritmo, vamos primeiro pegar os novos dados na [planilha do Google Spreadsheets (http://bit.ly/1NogiPr)](http://bit.ly/1NogiPr):



Curso Machine Learning: Introducao a Classificacao - entrada

	recencia	frequencia	semanas_de_ins	situacao
2	1	4	4	2
3	2	1	2	1
4	1	4	2	2
5	1	3	8	1
6	2	2	1	1
7	1	4	2	2
8	1	1	5	1
9	1	3	8	1
10	3	1	1	1
11	3	2	6	1
12	2	3	6	1
13	3	1	2	1
14	3	2	7	1
15	1	4	2	2
16	1	3	1	2
17	1	4	8	1
18	1	3	7	1
19	3	1	7	0
20	2	1	5	1
21	3	2	6	1
22	2	2	8	1
23	3	1	4	1

Perceba que agora estamos pegando os dados da aba "situacao_do_cliente". Salve o arquivo como CSV com o nome `situacao_do_cliente.csv` dentro do diretório onde preferir, lembrando que **iremos trabalhar exatamente no local onde salvar esse arquivo**. Agora, crie um arquivo dentro do diretório onde salvou o CSV com o nome `situacao_do_cliente.py` que será o nosso arquivo python para escrevermos o nosso código. Não implementaremos todo o código desde o zero, pois reutilizaremos uma boa parte do algoritmo que implementamos no arquivo `classifica_buscas.py`, segue o código abaixo:

```
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)

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)
```

Copie o código e cole no arquivo `situacao_do_cliente.py`. A única diferença entre esse código e o que utilizamos no `classifica_buscas.py`, é que apagamos as marcações dos testes que realizamos, pois eles eram válidos para aquele outro conjunto de dados que havíamos utilizados e também o nome das variáveis:

- **De**: dados: `'home','busca','logado'` e marcação: `'comprou'`.
- **Para**: dados: `'recencia','frequencia','semanas_de_inscricao'` e marcação: `'situacao'`

Pois, utilizaremos os dados para essa nova situação que estamos, que é avaliar a situação do cliente entre: `alegre`, `neutro` ou `chateado`. Mas e esses novos dados? Qual é a diferença deles com o que tinhamos anteriormente? Vejamos:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,2
2,1,2,1
1,4,2,2
1,3,8,1
2,2,1,1
1,4,2,2
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Observe que anteriormente, não havíamos utilizados dados com marcações com mais de 2 valores, porém, dessa vez, estamos variando entre 3 valores (0, 1 ou 2).

Será que tanto o algoritmo `AdaBoost` quanto o `Multinomial` conseguem trabalhar de forma eficiente com uma variável de marcação que varia entre 3 valores? Como podemos verificar? Simples! Rodando o nosso arquivo `situacao_do_cliente.py`:

```
> python situacao_do_cliente.py
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Nesse primeiro teste que realizamos podemos concluir que o algoritmo `Multinomial` obteve um resultado de 72,72% e o `AdaBoost` 68,18%, ou seja, o algoritmo vencedor foi o `Multinomial`. Ao executar o teste do mundo real ele obteve um resultado superior ao esperado, nesse caso, 82,60%, porém, mesmo sendo um resultado superior ao dos testes, ele é equivalente ao resultado do algoritmo base, ou seja, que chuta o mesmo valor para todos. Podemos até mesmo verificar o resultado do nosso algoritmo `Multinomial` imprimindo o resultado dentro da função `teste_real`:

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

    print resultado

    # restante do código
```

Resultado:

```
> python situacao_do_cliente.py
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Repare que o nosso algoritmo que faz diversos calculos para tentar adivinhar o que cada dado é chuta o mesmo valor para todos! Portanto, tanto o `Multinomial` quanto o `AdaBoost` , não são tão bons para dados com variáveis que possuem 3 valores distintos. Isso significa que, para esse tipo de cenário que nos deparamos, não faz sentido utilizarmos algum desses algortimos. Então o que faremos? Precisamos utilizar algum outro algoritmo que seja capaz de resolver esse problema de uma maneira mais eficaz, isto é, com 3 valores diferentes para a variável de classificação.

A grande sacana para essa situação seria encontrar alguma maneira de utilizarmos os mesmos algoritmos, ou seja, alguma forma que os permitam classificar elementos com mais de duas categorias, isto é, além de zeros e uns, nesse caso, 0, 1 ou 2, ou então, e 0, 1, 2, 3 a N, portanto, classificar com qualquer quantidade acima de 2. Primeiramente precisamos investigar a fundo os resultados desses algoritmos, por exemplo, podemos começar verificando o resultado de cada um no momento em que eles realizam os testes, isto é, imprimindo a variável `resultado` da função `fit_and_predict` :

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

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

Além disso, retire a impressão que está sendo realizada na função `teste_real` . Rodando novamente o nosso algoritmo temos o seguinte resultado:

```
> python situacao_do_cliente.py
[1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1]
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Repare tanto o `Multinomial` quanto `AdaBoost` obtiveram resultados bem similares, porém, o `Multinomial` variou em apenas um dado prevendo com valor 2 ao invés de apenas 1. Quando nos deparamos com um problema grande, nesse caso, o nosso preblema atual, a grande sacana para resolvê-lo é diminuí-lo para pequenos problemas que já saibamos resolver. Então por onde podemos começar? Um bom ponto de partida seria pelos nossos dados, ou seja, o arquivo `situacao_do_cliente.csv` :

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,2
2,1,2,1
1,4,2,2
1,3,8,1
2,2,1,1
1,4,2,2
```

```
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Observe que atualmente temos os 3 valores distintos para a coluna `situacao`, então que tal transformarmos todos os valores 2 em uns? Por exemplo, todas as vezes que for 0 reconheceremos como 0, porém, quando o valor for 1 ele pode ser tanto 1 como 2:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,1
2,1,2,1
1,4,2,1
1,3,8,1
2,2,1,1
1,4,2,1
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Então o nosso problema vai para:

```
0=>0 1=>1,2
```

E, supostamente, quando rodamos o `Multinomial`, ele consegue verificar o percentual tanto de zeros quanto de uns (resto) e obtem o resultado respectivamente de (38% (zeros) e 62% (resto)):

```
0=>0 1=>1,2 multinomial 0 ou do resto (38%, resto 62%)
```

Aparentemente essa redução do problema não foi de grande ajuda, ou seja, vamos retornar aos valores que estavam anteriormente:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,2
2,1,2,1
1,4,2,2
1,3,8,1
2,2,1,1
1,4,2,2
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Que tal tentarmos reduzir novamente? Mas, ao invés de transformarmos o 2 em uns, vamos transformar o 2 em zeros:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,0
2,1,2,1
1,4,2,0
1,3,8,1
2,2,1,1
1,4,2,0
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Observe que novamente entramos no caso em que o nosso algoritmo consegue resolver, ou seja, com apenas dois valores, então os nossos dados estão sendo representados da seguinte forma:

```
0=>0,2 1=>1
```

Então, quando rodamos novamente o algoritmo `Multinomial` para esses dados, supostamente, obtemos o resultado de 44% para uns e o resto é de 56%:

```
0=>0,2 1=>1 multinomial 1 ou do resto (44%, resto 56%)
```

Como podemos ver, ainda não podemos ter a certeza, pois o resto ainda é maior que os valores uns. Novamente, vamos retornar ao valores anteriores:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,2
2,1,2,1
1,4,2,2
1,3,8,1
2,2,1,1
1,4,2,2
1,1,5,1
1,3,8,1
3,1,1,1
...
4,1,6,0
```

Então vamos fazer a última redução e verificar o resultado que obtemos, isto é, transformar o número 1 em 0:

```
recencia,frequencia,semanas_de_inscricao,situacao
1,4,4,2
2,1,2,0
1,4,2,2
1,3,8,0
2,2,1,0
1,4,2,2
1,1,5,0
1,3,8,0
```

```
3,1,1,0
...
4,1,6,0
```

Note que novamente temos apenas duas categorias, ou seja, cateria 0 e categoria 2 e então rodamos o `Multinomial` e, supostamente, ele nos devolve o seguinte resultado:

```
0=>0,1 2=>2 multinomial 2 ou do resto (20%, resto 80%)
```

Apenas reduzindo o problema das 3 possíveis maneiras, aparentemente não resolveu o nosso problema, porém, se pedirmos para o nosso algoritmo rodar baseado nas 3 informações distintas que conseguimos ao diminuir o nosso problema maior:

```
0=>0 1=>1,2 multinomial 0 ou do resto (38%, resto 62%)
0=>0,2 1=>1 multinomial 1 ou do resto (44%, resto 56%)
0=>0,1 2=>2 multinomial 2 ou do resto (20%, resto 80%)
```

Ele irá analisar cada um dos resultados e provavelmente escolhará o que tiver maior probabilidade, em outras palavras, ele vai verificar o 0 (38%), 1 (44%) e o 2 (20%), então, ele escolhará o 1, pois, dentre os 3 valores, o 1 é o que contem maior chance. Você pode estar pensando que isso foi apenas uma manipulação de dados para conseguir chegar a um resultado esperado, e de fato foi, porém, consegue perceber o que acabamos de fazer? Inicialmente, não sabíamos como resolver o problema para uma classificação de 3 categorias, então, reduzimos esse problema para um menor que fossemos capazes de resolver, ou seja, um problema de duas categorias:

- 0=>0 1=>1,2
- 0=>0,2 1=>1
- 0=>0,1 2=>2

Observe que o primeiro resultado que chegamos quando aplicamos a redução foi a categoria 0 versus o resto. O segundo foi a categoria 1 versus o resto. Por fim, a categoria 2 versus o resto. Em outras palavras, se tivermos 5, 10, 15 ou N categorias, rodaríamos o nosso algoritmo para cada uma das categorias versus o resto. Esse algoritmo é conhecido como "um versus o resto" ou "um versus todos", mas, tecnicamente, é chamado de [One-vs-the-rest (http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html)](http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html). Porém, o processo desse algoritmo é mais demorado, pois, ao invés de utilizar um único classificador e rodar, estão sendo criados 3 classificadores, rodam os 3 e comparam, ou seja, para esse caso, ficou 3 vezes mais "lerdo". Em outras palavras, esse algoritmo roda a quantidade de categorias diferentes que tivermos, por exemplo, se tivermos 5 categorias 5 vezes, 10 categorias 10 vezes, N categorias, N vezes! Mas será que teremos que implementar todos esses passos que fizemos na mão? Felizmente, o pacote **sklearn** já fornece uma implementação do One-vs-the-rest para nós, que é o `OneVsRestClassifier` , então vamos importar esse classificador no nosso código:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
# 0=>0 1=>1,2 multinomial 0 ou do resto (38%, resto 62%)
# 0=>0,2 1=>1 multinomial 1 ou do resto (44%, resto 56%)
# 0=>0,1 2=>2 multinomial 2 ou do resto (20%, resto 80%)

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

```
from sklearn.ensemble import AdaBoostClassifier
modeloAdaBoost = AdaBoostClassifier()
```

Quando utilizamos o `OneVsRest`, costumamos utilizar o algoritmo `LinearSVC`, em outras palavras, o `OneVsRestClassifier` é basicamente um algoritmo que roda o modelo que damos pra ele em diversas vezes. Portanto, quando criarmos o modelo do `OneVsRestClassifier`:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
OneVsRestClassifier
# 0=>0 1=>1,2 LinearSVC 0 ou do resto (38%, resto 62%)
# 0=>0,2 1=>1 LinearSVC 1 ou do resto (44%, resto 56%)
# 0=>0,1 2=>2 LinearSVC 2 ou do resto (20%, resto 80%)
```

Teremos que enviar o modelo que queremos que ele rode várias vezes, nesse caso, o `LinearSVC`. Para isso precisamos importar o `LinearSVC` e enviar por parâmetro no momento que criamos o modelo do `OneVsRestClassifier`:

```
# restante do código

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

Além disso, iremos enviar o parâmetro `random_state = 0`, para rodar o `LinearSVC` de uma maneira fixa ao invés de aleatória:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))
# 0=>0 1=>1,2 LinearSVC 0 ou do resto (38%, resto 62%)
# 0=>0,2 1=>1 LinearSVC 1 ou do resto (44%, resto 56%)
# 0=>0,1 2=>2 LinearSVC 2 ou do resto (20%, resto 80%)
```

E o restante do código? Faremos da mesma forma como fizemos anteriormente, ou seja, treinamos com o método `fit`:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))
modelo.fit(treino_dados, treino_marcacoes)
```

Após treinar o modelo pedimos para ele prevêr e imprimir o resultado do que ele classificou:

```
# restante do código
```

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))
modelo.fit(treino_dados, treino_marcacoes)
print(modelo.predict(teste_dados))
```

Será que ele vai chutar tudo 1 da mesma forma que o `Multinomial` e o `AdaBoost` fizeram? Será que ele vai variar entre zeros, uns e dois? Vamos testar então? Antes de executar o código, vamos retirar a impressão do resultado dos 2 algoritmos dentro da função `fit_and_predict` :

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

    resultado = modelo.predict(teste_dados)

# restante do código
```

Rodando o nosso código, obtemos o seguinte resultado:

```
> python situacao_do_cliente.py
[2 0 1 1 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 2 1]
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Repara que um resultado foi 2, 0, uma sequência de uns depois um 2 e assim por diante. Realmente ele usou valores diferentes em sua classificação. Podemos fazer uma comparação simples, apenas imprimindo a variável `teste_marcacoes` embaixo:

```
# restante do código

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
modelo = OneVsRestClassifier(LinearSVC(random_state = 0))
modelo.fit(treino_dados, treino_marcacoes)
print(modelo.predict(teste_dados))
print(teste_marcacoes)
```

Vejamos o resultado:

```
> python situacao_do_cliente.py
[2 0 1 1 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 2 1]
[2 0 1 1 1 1 1 1 2 1 2 1 1 0 1 1 1 1 0 1 2 1]
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Observe que ele errou apenas dois valores, ou melhor, ele errou menos de 10%, ou seja, ele acertou mais de 90%! Portanto, podemos concluir que ele é o algoritmo ideal para realizar esse tipo classificação, em outras palavras, classificações com mais de 2 categorias. Então repara que os algoritmos capazes de classificar em várias classes (categorias) são conhecidos como algoritmos *multiclasses*, e utilizamos o `OneVsRestClassifier` . Porém, precisamos incluir esse algoritmo dentro do nosso processo que envolve as 3 fases (treino, teste e teste real), certo? Para isso vamos renomear o modelo da mesma forma como fizemos com os outros algoritmos:

```
# restante do código

modeloOneVsRest = OneVsRestClassifier(LinearSVC(random_state = 0))
```

Então, ao invés de realizar aquele treino e teste manualmente, usaremos a nossa função `fit_and_predict` retornando o resultado do nosso novo modelo:

```
# restante do código

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

Podemos apagar todo aquele código que fizemos para testarmos manualmente o `OneVsRestClassifier` deixando apenas o seguinte código:

```
# restante do código

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

Vamos verificar se ele está funcionando conforme o esperado:

```
> python situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Note que o resultado do `OneVsRestClassifier` foi impresso, porém ainda não o incluímos no terceiro passo que é justamente eleger o vencedor e utilizá-lo para o teste real. Vejamos como estamos fazendo atualmente:

```
if resultadoMultinomial > resultadoAdaBoost:
    vencedor = modeloMultinomial
else:
    vencedor = modeloAdaBoost
```

Perceba que atualmente estamos fazendo um `if` e `else` para tentar verificar entre o `Multinomial` e o `AdaBoost` qual é o vencedor. Isso significa que teremos que adicionar mais um `if` para o `OneVsRest` também? percebe que como essa solução não parece boa? Afinal, se adicionarmos mais um algoritmo, precisaremos adicionar mais `if` s... Como podemos resolver isso? Por meio de um dicionário! Então vamos criar o nosso dicionário chamado `resultados` :

```
# restante do código

resultados = {}

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

from sklearn.naive_bayes import MultinomialNB
modeloMultinomial = MultinomialNB()
```

Então, identificamos o resultado do `OneVsRest` como `modeloOneVsRest` :

```
# restante do código

resultados = {}

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

Então faremos o mesmo para todos os outros algoritmos:

```
# restante do código

resultados = {}

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

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

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

Vamos imprimir o nosso dicionário para verificar os valores que foram inseridos:

```
# restante do código

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

print(resultados)
```

Vejamos o resultado do nosso dicionário:

```
> python situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
{MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True): 72.72727272727273, OneVsRestClassifier(
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1): 90.9090909090909, AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
          learning_rate=1.0, n_estimators=50, random_state=None): 68.18181818181819}
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Repare que é impresso uma mensagem gigante, porém, note que nessa mensagem, estão inseridos os modelos que informamos junto com os seus respectivos valores. Então agora temos o nosso dicionário que contém tanto o modelo e a porcentagem da taxa de acerto. Mas e agora? O que faremos com esse dicionário? Precisamos pegar o melhor desses modelos, em outras palavras, o modelo que contém o maior valor para a taxa de acerto. Será que podemos usar a função `max` ? Até poderíamos, mas ao invés de retornar o maior valor, a função `max` retornará a maior chave, portanto, precisamos alternar as chaves e seus valores, em outras palavras, colocaremos os resultados como chaves e os modelos como os valores do dicionário:

```
# restante do código

resultados = {}

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

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
```

Se olharmos rodarmos novamente o nosso código:

```
> python situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
{68.18181818181819: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
          learning_rate=1.0, n_estimators=50, random_state=None), 72.72727272727273: MultinomialNB(a
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)}
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 82.6086956522
Taxa de acerto base: 82.608696
Total de teste: 23
```

Veja que agora os resultados são as chaves e os modelos os valores. Portanto, podemos utilizar o `max` enviando o nosso dicionário (`resultados`) e atribuíndo para uma variável chamada `maximo` :

```
# restante do código

maximo = max(resultados)
```

E o vencedor? É justamente o valor do `maximo` , ou seja, basta apenas pedirmos para o dicionário o modelo por meio da chave que é justamente a variável `maximo` :

```
# restante do código

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

Podemos até imprimir a variável `vencedor` para verificar os valores. Porém, vamos eliminar a impressão da variável `resultados` e também o `if` e `else` que verificava quem era o vencedor entre o `Multinomial` e o `AdaBoost` :

```
# restante do código

resultados = {}

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

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
```

Vejamos o resultado:

```
> python situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
Vencerdor:
OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 86.9565217391
Taxa de acerto base: 82.608696
Total de teste: 23
```

Note que agora estamos imprimindo o algoritmo vencedor e a taxa de acerto no mundo real que é de 86,95% que é superior ao resultado do algoritmo base. Então podemos concluir que para esses dados o algoritmo `OneVsRest` é mais eficaz. Porém, será que o algoritmo base está funcionando bem para esse tipo de dado? Isto é, para dados com mais de duas categorias? Vamos verificar a resposta que ele está chutando, imprimindo o nosso `Counter` com a variável `validacao_marcacoes` que representa os nossos dados de validação:

```
# restante do código

print Counter(validacao_marcacoes)
acerto_base = max(Counter(validacao_marcacoes).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)
```

Vejamos o resultado:

```
> python situacao_do_cliente.py
Taxa de acerto do algoritmo OneVsRest: 90.9090909091
Taxa de acerto do algoritmo MultinomialNB: 72.7272727273
Taxa de acerto do algoritmo AdaBoostClassifier: 68.1818181818
```

```
Vencerdor:
OneVsRestClassifier(estimator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0),
          n_jobs=1)
Taxa de acerto do vencedor entre os dois algoritmos no mundo real: 86.9565217391
Counter({1: 19, 0: 3, 2: 1})
Taxa de acerto base: 82.608696
Total de teste: 23
```

De fato ele funciona como o esperado, então não precisamos nos preocupar com ele, portanto, podemos retirar essa impressão.

Atualmente vimos 3 modelos de algoritmos diferentes:

- MultinomialNB.
- AdaBoostClassifier.
- OneVsRestClassifier.

Dentre os 3 algoritmos, acabamos de ver como o `OneVsRest` funciona por de trás dos panos que é justamente comparar um elemento com todos os outros, considerando o conjunto {0,1,2}:

- 1º Classificação: 0 versus 1,2.
- 2º Classificação: 1 versus 0,2.
- 3º Classificação: 2 versus 0,1.

Por exemplo, o elemento 0 versus o 1 e 2, o elemento 1 versus o 0 e 2 e assim por diante. Porém, poderíamos fazer uma abordagem um pouco diferente. Em outras palavras, ao invés de ficar realizando o teste de um elemento versus todos os outros, poderíamos pegar um único elemento e testá-lo versus um outro único elemento, por exemplo, suponhamos que tenhamos 5 elementos {0,1,2,3,4} distintos:

- 1º Classificação: 0 versus 1.
- 2º Classificação: 0 versus 2.
- 3º Classificação: 0 versus 3.
- 4º Classificação: 0 versus 4.
- 5º Classificação: 1 versus 2.
- 6º Classificação: 1 versus 3.
- 7º Classificação: 1 versus 4.
- 8º Classificação: 2 versus 3.
- 9º Classificação: 2 versus 4.
- 10º Classificação: 3 versus 4.

Repara que dessa vez, ao invés de realizar um único teste para cada elemento contra todos os outros, estamos criando um teste para cada elemento, isto é ,um elemento versus apenas um outro elemento, portanto, 0 vs 1, 0 vs 2 e assim por diante. Esse tipo de algoritmo chamamos de **um versus um** ou, tecnicamente, `OneVsOne` . Antes mesmo de implementar esse algoritmo no nosso código, precisamos entender um detalhe importante em relação ao `OneVsRest` , por exemplo, vimos que, para o conjunto de dados {0,1,2,3,4} foram realizados 10 testes. Mas, para esse mesmo conjunto de dados, quantos testes seriam realizados com o `OneVsRest` ? Vejamos:

- 1º Classificação: 0 versus 1,2,3,4.
- 2º Classificação: 1 versus 0,2,3,4.
- 3º Classificação: 2 versus 0,1,3,4.
- 4º Classificação: 3 versus 0,1,2,4.
- 5º Classificação: 4 versus 0,1,2,3.

Observe que seriam apenas 5 testes! Se fizermos a analise matemática do algoritmo `OneVsOne` (http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsOneClassifier.html) (que não é o nosso foco, mas, se tiver interesse, consulte o curso de algoritmos!), veremos que ele é um algoritmo quadrático, portanto, podemos concluir que sua performance é pior em relação ao `OneVsRest` que é linear.

Para quem se interessa na fórmula matemática arredondada, o número de comparações do algoritmo `OneVsOne`, onde elementos é o número de elementos, temos algo como:

```
elementos * (elementos - 1) / 2
```

Vejamos algumas simulações da quantidade de testes:

- **3 elementos**: 3 *(3 - 1) / 2 -> 3* 2/2 = 3.
- **4 elementos**: 4 *(4 - 1) / 2 -> 4* 3/2 = 6.
- **5 elementos**: 5 *(5 - 1) / 2 -> 5* 4/2 = 10.

Perceba que esse algoritmo cresce quadraticamente de acordo com a quantidade de elementos. Portanto, podemos concluir que quanto mais elementos utilizarmos para testá-lo, cada vez mais lento ele ficará em relação ao `OneVsRest`. Agora que vimos os pontos principais entre os 2 algoritmos, vamos implementar também o `OneVsOne`. Para isso precisamos primeiro importá-lo:

```
# restante do código

resultados = {}

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

from sklearn.multiclass import OneVsOneClassifier
```

A sua implementação é praticamente identica ao do `OneVsRest`, a única diferença é que mudaremos o nome das variáveis:

```
resultados = {}

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

Note que realizamos exatamente os mesmos passos na implementação do `OneVsRest`, isto é, importamos, criamos o modelo enviando o algoritmo `LinearSVC` por parâmetro, pedimos para treinar e testar com a função `fit_and_predict` e por fim adicionar o seu resultado na variável `resultados`, ou seja, o nosso dicionário. Vamos verificar o resultado do nosso algoritmo agora?

```
> 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
```

Repare que o `OneVsOne` acertou 100% dos testes e foi escolhido como o algoritmo vencedor. Além disso, no teste de validação, a taxa de acerto no mundo real foi de 100% também! Lembre-se que caímos numa situação bem rara que é justamente acertar 100% dos dados.

Percebe o quão importante é rodarmos todos os algoritmos de uma vez para verificar qual deles se sairá melhor dado um determinado conjunto de dados? Porém, ao utilizar algoritmos como `OneVsOne`, ou seja, algoritmos quadráticos, precisamos nos atentar, pois se precisamos de performance, provavelmente esse algoritmo não poderá ser utilizado, devido ao fato do crescimento de testes para uma grande quantidade de elementos de um conjunto de dado.

Resumindo

Repara que além de dados com duas categorias, podemos também conter dados com mais de duas categorias, como por exemplo, a situação do cliente em relação ao nosso site, isto é, se ele está alegre, neutro ou chateado. Levando em consideração apenas esse exemplo, vimos que os algoritmos que utilizamos anteriormente (`AdaBoost` e `Multinomial`) não conseguiam classificar esses dados de forma eficiente, em outras palavras, obtinham o mesmo resultado que o algoritmo base, portanto, não poderíamos utilizá-los no mundo real. Porém, além do `AdaBoost` e `Multinomial`, que são algoritmos bons para dados com duas categorias, temos também os algoritmos `OneVsRest` e o `OneVsOne` que servem justamente para classificar elementos que podem ter mais de duas categorias. Além da eficiência dos algoritmos `OneVsRest` e `OneVsOne`, vimos que ambos possuem uma certa diferença quanto à performance, pois o `OneVsRest` cresce constantemente de acordo com a quantidade de elementos dado um conjunto de dados, ou seja, se tivermos 3 elementos, serão 3 testes, 4 elementos 4 testes. Mas, no `OneVsOne`, a abordagem é bem diferente, pois ele é um algoritmo quadrático, ou seja, ele cresce muito de acordo com a quantidade de elementos, em outras palavras, quantos mais elementos tivermos, serão realizados muito mais testes.

Vimos também a importância de rodar diversos algoritmos para o mesmo conjunto de dados, porém, precisamos sempre ficar atentos para não viciar o nosso algoritmo, ou seja, não podemos rodar um algoritmo, obter um resultado e então, de

acordo com o resultado rodar um outro algoritmo. O ideal é que sempre rodemos todos os algoritmos de uma vez, pois dessa forma elegemos o algoritmo vencedor e rodamos para um conjunto de dados desconhecidos.