Machine Learning Boosting Algorithm: A study case with Tic Tac Toe

Trabalho Prático 2

Raphael Ottoni
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
rapha@dcc.ufmg.br

KEYWORDS

Machine Learning, boosting, Adaboost

ACM Reference format:
Raphael Ottoni. 2017. Machine Learning Boosting Algorithm: A study case with Tic Tac Toe. In *Proceedings of Machine Learning Class, Belo Horizonte, Minas Gerais, Brasil, June 2017 (UFMG 2017)*, 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUÇÃO

Boosting é um meta-algoritmo de agregação de classificadores em *machine learning* cujo objetivo é criar um classificador forte baseado na combinação de classificadores fracos. Neste trabalho prático, iremos construir um classificador de partidas de jogo da velha utilizando a técnica de *boosting* conhecida como *Adaboost*. Nela, é introduzido o conceito de importância de classificação nos exemplos de treinamento. Em princípio, todos exemplos tem a mesma importância, porém a cada iteração um classificador fraco é escolhido e a importância dos exemplos é modificada: aquelas em que este classificador acertou, tem sua importância reduzida, e aquelas em que ele errou, tem sua importância aumentada. Desta maneira, na próxima iteração, a escolha do próximo classificador fraco será enviesada para escolher um que acerte aqueles em que o anterior errou. Resultando assim em uma combinação de classificadores (*ensamble*) mais robusta (figura. 1).

2 DATASET

O *dataset* utilizado neste experimento é o *tic-tac-toe*, disponível em https://archive.ics.uci.edu/ml/datasets/Tic-Tac-Toe+Endgame. Cada linha desde *dataset* representa um jogo da velha realizado e o label de qual jogador venceu aquela partida. Caso tenha sido o jogador "x" ele é positivo e se foi o jogador "o", este label é negativo. É interessante relatar, que nesta base da dados, não há exemplos de empate. Acredito que isto tenha sido feito propositalmente para que padecemos utilizar um classificador binário na classificação ao invés de alguma outra técnica, por exemplo a *one-against-all*. A tabela 1 mostra a caracterização deste *dataset*.

UFMG 2017, June 2017, Belo Horizonte, Minas Gerais, Brasil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

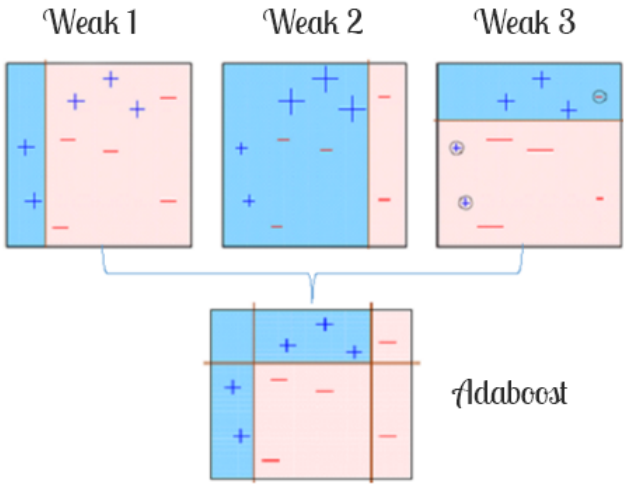
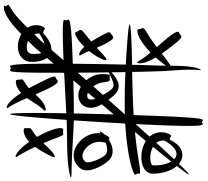


Figure 1: Adaboost com 3 classificadores fracos

Vencedor	Quantidade
x	626
o	332
total	958

Table 1: Tic-Tac-Toe Dataset

Como dito antes, cada linha dos dados representa uma instância do jogo em que o jogador 'x' ganhou ("positive") ou perder ("negative"). A figura 2 descreve com mais detalhe a representação de um jogo: um vetor de 10 posições, em que as primeiras nove são mapeadas diretamente para as 9 posições de um jogo da velha (começando do canto superior esquerdo), seguido de um label mostrando se o jogador 'x' ganhou ou perdeu. Cada uma das 9 primeiras posições contem um dos três valores possíveis: 'b' para blank, 'x' para o primeiro jogador e 'o' para o segundo.



[o,x,x,x,o,b,x,o,o,negative]

Figure 2: Representação de um jogo.

3 IMPLEMENTAÇÃO

Conforme dito na seção 1, o *Adaboost* se caracteriza por criar um classificador forte a partir de classificadores fracos. Portanto, antes de implementarmos o *Adaboost* em si, devemos introduzir os classificadores fracos que teremos disponíveis. A especificação deste trabalho prático pede para que seja utilizado como classificadores fracos *stumps* de decisão. Estes *stumps*, nada mais são do que uma árvore de decisão de apenas um separando duas folhas. Também podemos ver eles como uma classificador extremamente simples que toma sua decisão de acordo com um único par de (variavel,valor). A figura 1, ilustra bem este conceito, mostrando 3 *stumps* (classificadores fracos).

3.1 Stumps Possíveis

Traduzindo esses stumps para a nossa aplicação de jogo da velha, teremos ao todo 54 *stumps* disponíveis. Sendo a combinação de pares (variáveis,valores) disponíveis na representação de um jogo (Figura. 2), associada com uma predição de classificação "positive" ou "negative". As variáveis são as 9 posições no vetor (0-8), os valores são as três possíveis marcações $[o, x, b]$, a tabela. 2 mostra todos os *stumps* existentes.

3.2 Detalhes de implementação

O algoritmo *Adaboost* foi implementado em python3 (não funcionará em python2) seguindo o pseudocódigo 1. O código fonte foi disponibilizado no github sob licença GLP-3.0 e pode ser encontrado na url: https://github.com/raphaottoni/boosting.

Algorithm 1 Adaboost

(1) Inicializa o peso de cada observação como $w_i = 1/N$, $i = 1, 2, \ldots, N$

(2) For $m = 1$ to $nstumps$:

(a) Calcula o errro de todos *stumps* disponíveis e escolhe o stump $S_m(x)$ sendo aquele com menor $erro_j$:

$$\text{Erro}_j = \sum_{erros} w_i$$

(b) Calcula o alpha deste stump escolhido:

$$\alpha_m = \frac{1}{2}\ln(\frac{1-erro}{erro})$$

(c) Atualiza os pesos de todas as observações considerando a classificação dada pelo stump escolhido:

$$w_i = \frac{w_i}{2}\begin{cases}\frac{1}{1-erro}, & \text{Se classificado correto}\\ \frac{1}{erro}, & \text{Se classificado errado}\end{cases}, i = 1, 2, \ldots, N$$

(3) Resultado $Adaboost(x) = \text{sign}\left[\sum_{m=1}^{nstumps} \alpha_m S_m(x)\right]$.

Além disso, a implementação foi separada em 1 módulo de leitura de dados e uma classe *Boost*. Eles são usados por dois programas principais: main.py e main_step_by_step.py. A seguir será discorrido um pouco mais sobre essas separações e como são utilizadas para implementar o *Adaboost*.

3.2.1 Modulo datareader.py. Neste modulo são encontrados as funções de leitura do dataset de jogos da velha. Sua função básica é ler os dados no formato csv e clocá-los em memoria para que sejam acessados pela *boost*.

3.2.2 Classe Boost. Nesta classe é implementado todo o cerne do *Adaboost*. O objeto dessa classe representa um algoritmo de *Adaboost* completo que recebe como parâmetros para sua instanciação o número de observações no dataset e a quantidade de classificadores fracos que serão utilizados. Suas principais funções são:

- build_classifier(observações_de_treino)
- classify(sample)

Um pseudo código simples de funcionamento desta classe, pode ser visto em Algorithm. 2

Algorithm 2 Simples utilização da classe Boost

(1) Instância a classe:

adaboost = Boost(len(training_data),nstumps)

(2) Treina o classificador:

adaboost.build_classifier(training_data)

(3) Utiliza o classificador treinado para classificar:

adaboost.classify(sample)

3.3 Exemplo de execução

Conforme informado anteriormente, foram disponibilizado dois programas principais: 1) main.py e 2) main_step_by_step.py. Todos eles são executados a mesma maneira:

```
./python3 main.py <number_of_stumps>
./python3 main_step_by_step.py <number_of_stumps>
```

A única diferença entre eles, é que no main.py, já é reportado o resultado final do *Adaboost*, já tendo escolhido os 5 *stumps* (Figura. 3).

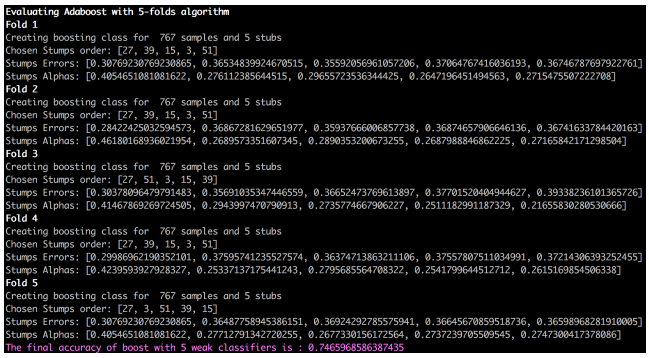


Figure 3: Resultado da escolha de 5 stumps e 5-fold analise do programa main.py

Já o main_step_by_step.py, ele apresentará os resultados parciais de cada escolha de stump, como pode ser visto na Figura. 4

Vale a pena atentar para o fato de que os resultados do *Adaboost* com 1 ou 2 stumps são exatamente os mesmos, como mostrado na Figura. 4. Este comportamento especifico é esperado pois para cada novo stump escolhido o alpha associado a ele tende a ser menor que

```
Evaluating Adaboost with 5-folds algorithm
Fold 1
Creating boosting class for  767 samples and 1 stubs
Chosen Stumps order: [27]
Stumps Errors: [0.3050874576271277]
Stumps Alphas: [0.4116001544040694]
Fold 2
Creating boosting class for  767 samples and 1 stubs
Chosen Stumps order: [27]
Stumps Errors: [0.29335071707953131]
Stumps Alphas: [0.4395827996176115]
Fold 3
Creating boosting class for  767 samples and 1 stubs
Chosen Stumps order: [27]
Stumps Errors: [0.30769230769230865]
Stumps Alphas: [0.4054651081081622]
Fold 4
Creating boosting class for  767 samples and 1 stubs
Chosen Stumps order: [27]
Stumps Errors: [0.28813559322033955]
Stumps Alphas: [0.4522281371135747]
Fold 5
Creating boosting class for  767 samples and 1 stubs
Chosen Stumps order: [27]
Stumps Errors: [0.30769230769230865]
Stumps Alphas: [0.4054651081081622]
The final accuracy of boost with 1 weak classifiers is : 0.6984293193717279
Evaluating Adaboost with 5-folds algorithm
Fold 1
Creating boosting class for  767 samples and 2 stubs
Chosen Stumps order: [27, 3]
Stumps Errors: [0.30508474576271277, 0.36647904940587994]
Stumps Alphas: [0.4116001544040694, 0.27367585733583266]
Fold 2
Creating boosting class for  767 samples and 2 stubs
Chosen Stumps order: [27, 3]
Stumps Errors: [0.29335071707953131, 0.37184091840918232]
Stumps Alphas: [0.4395827996176115, 0.26216366244993494]
Fold 3
Creating boosting class for  767 samples and 2 stubs
Chosen Stumps order: [27, 39]
Stumps Errors: [0.30769230769230865, 0.36534839924670515]
Stumps Alphas: [0.4054651081081622, 0.276112385644515]
Fold 4
Creating boosting class for  767 samples and 2 stubs
Chosen Stumps order: [27, 15]
Stumps Errors: [0.28813559322033955, 0.35811247575953359]
Stumps Alphas: [0.4522281371135747, 0.2917829886489466]
Fold 5
Creating boosting class for  767 samples and 2 stubs
Chosen Stumps order: [27, 39]
Stumps Errors: [0.30769230769230865, 0.36817325800376732]
Stumps Alphas: [0.4054651081081622, 0.27003078849315615]
The final accuracy of boost with 2 weak classifiers is : 0.6984293193717279
```

Figure 4: Resultado da escolha de 2 stumps e 5-fold analise do programa main_step_by_step.py

o anterior. Logo, no caso de uma combinação de dois classificadores fracos, a classificação sempre será a mesma da da pelo primeiro classificador fraco pois seu alpha é maior que o alpha do segundo, logo a soma do produto da classificação dos dois pelos seus alphas (Algoritimo 1, passo 3) sempre terá o sinal do primeiro classificador fraco.

3.4 Decisões de implementação

Durante a implementação do algoritmo, algumas decisões importantes foram feitas:

- Conforme a especificação original do *Adaboost*, foi permitido que stumps previamente escolhidos pudessem ser escolhidos novamente no futuro.
- O base do logaritmo utilizado no calculo dos alphas (Pseudocódigo 1) foi a natural *e*
- No calculo de novos pesos de cada observação (Pseudocódigo 1) a equação original foi expandida e separada em dois casos: quando o stump escolhido acerta e quando ele erra.

4 ANÁLISE

Todas as analises foram feitas baseadas em calculo de erro simples e utilizando o algoritmo de validação cruzada com 5 partições. Nestas análises não foram variadas o número de partições por acreditar que este trabalho prático é voltado para o algoritmo de *boosting* e não de validação cruzada. Em todos os gráficos a seguir, o erro apresentando é a dado como a media do erro encontrado em cada validação cruzada.

A primeira análise interessante a ser feita é a relação dos errors tanto de treino quanto de validação com o número de iterações do *Aadaboost*. Lembrando que o número de iterações é o número de classificadores fracos que queremos utilizar.

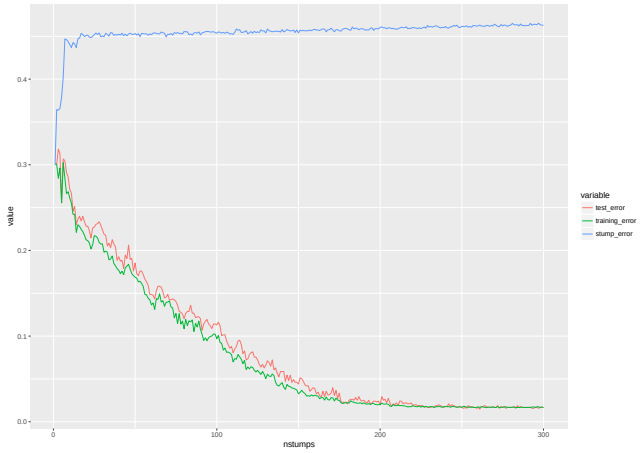


Figure 5: Adaboost erros por quantidade de *weak classifiers*

Como pode ser visto na Figura. 5, como esperado os errors de treino e de teste tendem a cair com o aumento de stumps escolhidos. Demonstrando assim a força da combinação de classificadores fracos, podemos ver que ambos convergem para um valor próximo de 0.02 tendo o erro de validação maior que o erro de treino durante a convergência. É interessante realçar que mesmo com uma grande escolha de um número grande de stumps, cerca de 6 vezes mais do que os stumps disponíveis, o *Adaboost* não sofreu de *overfit* e convergiu.

Outra padrão esperado que foi mostrado neste gráfico é o aumento do erro de cada stump escolhido (em cada uma das iterações) e sua convergência para um valor próximo de 0.5, mostrando assim, que idealmente, classificadores interessantes para o *Adaboost* são aqueles um pouco melhores que o aleatório (0.5).

Podemos ver também, que o primeiro stump escolhido já e responsável por aproximadamente 0.7 de acurácia (erro de 0.3 quando todas as observações tem a mesma importância), isso provavelmente deve ser fruto do desbalanceamento de classes, quase o dobro

de partidas no dataset foram vencidas pelo jogador "x" (Tabela 1). Se olharmos com mais detalhes as Figuras [4,**??**] veremos que o primeiro stump escolhido é o de identificação 27, que se consultarmos a Tabela 2, vemos que é o caso quando o jogador 'o' marcou na posição central do jogo da velha. O que faz sentido se pensarmos na estratégia ótima do jogo da velha. Este jogo é conhecido por ser desbalanceado de modo que quem começa se adotar a estratégia ótima não poderá perder, Se entrar em detalhes, tal estratégia envolve nunca preencher o quadrado central.

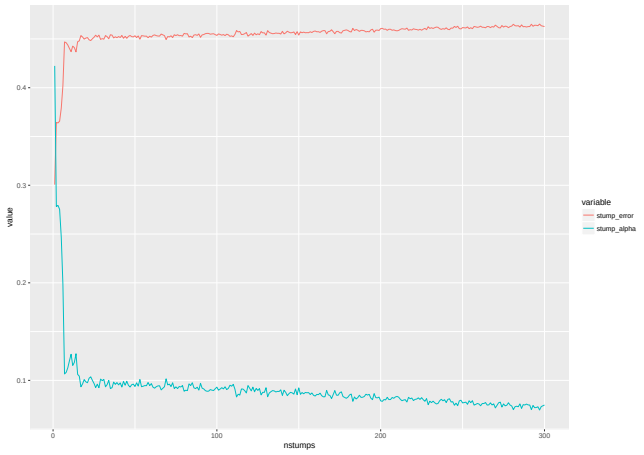


Figure 6: Adaboost *stumps* escolhidos em cada iteração com seus alphas e errors associados

Outra análise interessante é relação dos alphas e dos errors de cada stump escolhido (Pseudocódigo. 1, calculo α_m), por definição matemática o alpha e erro do stump estão inversamente relacionados: quem erra mais tem um alpha menor e consecutivamente contribui menos para a classificação final. Como podemos ver pela Figura. 6, os primeiros stumps tendem a ser escolhidos com uma importância (alpha) mais alta e subsecutivamente essas importâncias tendem a decrescer suavemente fazendo com que a importância dos últimos stumps escolhidos sejam aproximadamente iguais. Porem como o *Adaboost* atualiza os pesos de forma que o próximo stump é enviesado para escolher aquele que acerta mais os que o anterior errou, na *big-picture*, mesmo tendo importância próxima eles tendem a contribuir com pouca interseção, são especialistas em partes diferentes dos dados.

5 DISCUSSÃO E CONCLUSÃO

Este trabalho prático foi interessante pois pude ver na prática a ideia de que se pode criar um classificador forte a partir de classificadores fracos. Mais do que isso, a técnica utilizada, *Adaboost*, se provou muito resistente a *overfitting* e pode ser utilizada com qualquer tipo de classificadores. Não sendo limitada apenas a *stumps*, como foi feito neste trabalho prático.

Outra parte importante deste trabalho, foi comprovar conceitos sobre boosting aprendidos em sala de aula, como o porque é interessante apenas utilizar classificadores fracos no *ensamble* e o porque idealmente eles deveriam ser um pouco melhores do que um classificador aleatório.

Stump_ID	Variável	Valor	Predição
0	0	o	positive
1	0	x	positive
2	0	b	positive
3	0	o	negative
4	0	x	negative
5	0	b	negative
6	1	o	positive
7	1	x	positive
8	1	b	positive
9	1	o	negative
10	1	x	negative
11	1	b	negative
12	2	o	positive
13	2	x	positive
14	2	b	positive
15	2	o	negative
16	2	x	negative
17	2	b	negative
18	3	o	positive
19	3	x	positive
20	3	b	positive
21	3	o	negative
22	3	x	negative
23	3	b	negative
24	4	o	positive
25	4	x	positive
26	4	b	positive
27	4	o	negative
28	4	x	negative
29	4	b	negative
30	5	o	positive
31	5	x	positive
32	5	b	positive
33	5	o	negative
34	5	x	negative
35	5	b	negative
36	6	o	positive
37	6	x	positive
38	6	b	positive
39	6	o	negative
40	6	x	negative
41	6	b	negative
42	7	o	positive
43	7	x	positive
44	7	b	positive
45	7	o	negative
46	7	x	negative
47	7	b	negative
48	8	o	positive
49	8	x	positive
50	8	b	positive
51	8	o	negative
52	8	x	negative
53	8	b	negative

Table 2: Stumps Disponíveis