AdaBoost

João Francisco B. S. Martins

27 de Junho de 2017

1 Introdução

A ideia básica em torno de boosting surgiu em 1989, quando Michael Kearns e Leslie Valiant fizeram a seguinte pergunta em uma publicação: Podemos criar um modelo com bom aprendizado a partir de um conjuntos de modelos com aprendizado ruim?

Em 1990, outro cientista teórico da computação, Robert Schapire, respondeu a essa pergunta afirmando que sim, e, 5 anos depois, em 1995, Schapire e seu colega, Yuav Freund, publicaram um paper intitulado "A Decision-Theoretic Generalization of On-line Learning and an Application to Boosting", o qual descrevia um algoritmo nomeado Adaptive Boosting, sendo abreviado como AdaBoost. Esse algoritmo inovador fazia exatamente o que foi primeiro questionado por Kearns e Valiant, ou seja, ele utilizava diversos modelos de aprendizado fraco em conjunção para criar um modelo de aprendizado forte.

Esse método rendeu diversos prêmios aos seus criadores e foi o estado-da-arte por muitos anos, pois, além de muito eficiente, ele prevenia overfitting e seu modelo de aprendizado fraco não era pré-definido, podendo ser um perceptron ou um decision stump(árvores de decisão com apenas uma camada), por exemplo. A figura 1 mostra o trecho do artigo original onde o AdaBoost é descrito pela primeira vez.

Algorithm AdaBoost
Input: sequence of N labeled examples $\langle (x_1, y_1), ..., (x_N, y_N) \rangle$
distribution D over the N examples
weak learning algorithm **WeakLearn**
integer T specifying number of iterations
Initialize the weight vector: $w_i^1 = D(i)$ for $i = 1, ..., N$.
Do for $t = 1, 2, ..., T$

1. Set

$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^{N} w_i^t}$$

2. Call **WeakLearn**, providing it with the distribution $\mathbf{p}^t$; get back a hypothesis $h_t: X \to [0, 1]$.
3. Calculate the error of h_t: $\varepsilon_t = \sum_{i=1}^{N} p_i^t \, |h_t(x_i) - y_i|$.
4. Set $\beta_t = \varepsilon_t / (1 - \varepsilon_t)$.
5. Set the new weights vector to be

$$w_i^{t+1} = w_i^t \beta_t^{1 - |h_t(x_i) - y_i|}$$

Output the hypothesis

$$h_f(x) = \begin{cases} 1 & \text{if } \sum_{t=1}^{T} (\log 1/\beta_t) \, h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \log 1/\beta_t \\ 0 & \text{otherwise.} \end{cases}$$

Figura 1: Algoritmo de Adaptive Boosting(AdaBoost) como descrito no artigo original.

O objetivo deste trabalho é portanto implementar o algoritmo AdaBoost. Os dados utilizados serão os da base de dados Tic-Tac-Toe, disponibilizada no repositório de aprendizado de máquina da UC Irvine. A avaliação dos resultados será feita utilizando um método de validação cruzada com 5 partições(5-fold cross-validation).

2 Modelagem e Implementação

O projeto foi dividido em três módulos:

- **adaboost.py**: Contém a implementação da classe AdaBoost a qual provém todos os métodos e estruturas de dados utilizados no processo de boosting.
- **data_handler.py**: Contém métodos que carregam, manipulam e formatam a base de dados, assim como métodos de manipulação e armazenamento dos resultados.
- **main.py**: Contém a **main** do trabalho. Utiliza os módulos descritos acima para realizar o boosting com uma validação cruzada de 5 partições. O processo do cross-validation e o cálculo de sua média são feitos aqui.

Para informações mais específicas a respeito de certos pontos da implementação, é possível olhar diretamente no código, o qual está totalmente comentado, extremamente limpo e formatado.

2.1 Inicialização

Ao inicializarmos um objeto AdaBoost, passamos à ele o conjunto de treinamento e o de testes. A essa altura as saídas da base de dados já foram tratadas, traduzindo a classe de saída "positive" para o valor +1 e a classe "negative" para -1. Os pesos das instâncias são inicializados com:

$$w_i = \frac{1}{n}$$

Onde w_i é o valor do peso para instância i e n é o número de instâncias de treinamento. Sendo assim, ao menos inicialmente, $\sum_{i=1}^{n} w_i = 1$. Veremos mais para frente que isso manterá como verdade ao longo de toda a execução.

Como dito anteriormente, ao fazermos boosting estamos transformando um conjunto de mals preditores em um único bom preditor. O modelo de mal preditor escolhido nessa implementação do AdaBoost foi o decision stump por sua simplicidade e comprovada eficiência em conjunção com o método.

2.2 Decision Stumps

Decision stumps nada mais são do que árvores de decisão com apenas um nível, e é justamente daí é que vem sua nomenclatura, pois stump em inglês significa toco. Sendo assim, eles só podem fazer referência a um atributo da base de dados por vez, com cada galho saindo da raiz representando um possível valor para aquele atributo e as folhas representando a classe de saída predita para aquele valor.

No código nós representamos nossos stumps como um dicionário que guarda o número da coluna relativa ao atributo ao qual ele está relacionado, e uma entrada do dicionário para cada possível valor deste atributo. A base de dados utilizada só tem 3 possíveis valores para entrada **{x, o, b}**.

A escolha do melhor stump a cada iteração do boosting foi baseada no algoritmo de seleção de stumps **one rule** e funciona da seguinte maneira:

1. Para cada atributo(coluna) de entrada da base de dados, iteramos ao longo de todas as instâncias(linhas), acrescentando a um acumulador, para cada classe de entrada, o peso da instância corrente de acordo com sua classe de saída.

2. Escolhemos o maior somatório de pesos para cada valor como sendo a classe de saída da predição do mesmo.

3. Quando temos as predições para todos os possíveis valores daquele atributo, a função **stump_error** é chamada e calcula o erro de acordo com a seguinte fórmula:

$$error = \sum_{i=1}^{n} w_i \cdot perror_i$$

   Onde *error* é o erro empírico total daquele modelo, w_i é o peso da instância i e $perror_i$ é o erro da predição para aquela instância, sendo 0 se o modelo acertou e 1 se ele errou. Nesta equação deveríamos dividir tudo pela soma dos pesos, porém, como dito anteriormente, essa soma sempre será 1. Isso será melhor explicado na seção 2.4.

4. Escolha o modelo com o menor erro empírico.

Como vai ser possível perceber pelos gráficos, a categoria de classificadores fracos gera resultados que são, na média, pouco melhores do que as probabilidades envolvidas no lançamento de uma moeda(50%).

2.3 Peso dos classificadores(α)

Para cada classificador(stump) escolhido, ou seja, a cada rodada, precisamos calcular o peso que aquele classificador terá na conjunto final. Esse peso é:

$$\alpha_t = \frac{1}{2} ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

Onde α é o peso do classificador, *epsilon* é o erro empírico daquela classificador e os subíndices t representam a iteração atual.

Esse peso é atribuído única e exclusivamente ao classificador que forneceu o valor do erro empírico. O valor de α é importante não só para o classificador final mas também para o cálculo dos pesos das instâncias, sendo esse último explicado com detalhes na próxima seção.

2.4 Peso das instâncias

Quando falamos de boosting, o peso das instâncias é de suma importância, uma vez que são eles que guiarão os classificadores ruins a fim de que, quando combinados, alcancem um resultado melhor. A cada iteração do boosting devemos atualizar o peso de todas as instâncias da seguinte forma:

$$w_{t+1} = \frac{w_t e^{(-\alpha_t y h_t(x))}}{Z_t}$$

Onde w_t é o vetor de pesos da iteração corrente, α_t é o peso do classificador escolhido nessa iteração, y é o vetor de saída para o conjunto de treinamento, $h_t(x)$ é a hipótese(predição) do modelo escolhido na iteração atual e w_{t+1} corresponde ao conjunto de pesos a ser usado na próxima iteração.

A variável Z_t é um normalizador que faz com que o somatório do novo conjunto de pesos w_{t+1} seja sempre igual a 1. Ela equivale ao somatório de todos os novos pesos, e portanto, dividindo o peso de cada instância por ela, encontramos o resultado esperado. Essa

condição é mantida como verdadeira porque queremos que o conjunto de pesos represente uma distribuição, com o valor de cada peso significando a chance de aquela instância de treinamento ser "escolhida" pelo modelo.

2.5 Classificador final

O classificador final, conhecido na literatura como **ensemble**(conjunto), é a junção de todos os classificadores multiplicados por seus respectivos pesos. O resultado é extremamente melhor do que o de cada classificador individual e vai melhorando a cada iteração. A hipótese do ensemble para uma instância x é dada por:

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Onde *sign* é a função de sinal que retorna +1 se o parâmetro for positivo e -1 se o parâmetro for negativo, T é o número total de iterações de boosting feitas até o momento, α_t e $h_t(x)$ são respectivamente o peso do modelo encontrado na iteração t e a hipótese do mesmo para a instância x.

2.6 Validação cruzada com 5 partições

A validação cruzada(cross-validation) é um técnica de validação de modelos preditivos que tende a limitar problemas de overfitting e dar uma estimativa real da capacidade de generalização dos mesmos. Uma de suas variações mais utilizadas é a validação cruzada com k partições(k-fold cross-validation), sendo esta a utilizada nesse trabalho.

Escolhemos um valor de k igual a 5, e isso significa que vamos dividir a base de dados em 5 partições, realizando 5 iterações em cima de diferentes combinações dessas partições. Na primeira iteração, a primeira partição é escolhida como a partição de teste e as restantes são concatenadas para comporem a partição de treino. Na segunda iteração, a segunda partição será a de teste, e assim por diante. Ao fim das 5 iterações, podemos calcular o erro da validação cruzada:

$$CV_{error} = \frac{1}{K}\sum_{k=1}^{K} error_k$$

Onde CV_{error} é o erro no cross-validation e K é o número de partições.A variável $error_k$ é a taxa de erro simples gerada em cima da partição de teste a cada rodada da validação cruzada. O cálculo desse erro é:

$$error = \frac{correct}{total} \cdot 100$$

Onde *correct* é o número de predições corretas do ensemble e *total* é o número total de instâncias na partição de testes. Multiplicamos o erro por 100 para encontrarmos a porcentagem.

2.7 Execução

Como o código foi escrito em **Python 3**, para rodá-lo é necessário que a máquina tenha o mesmo instalado, assim como a biblioteca **numpy**. Após satisfeitos esses requisitos, basta que, estando na mesma parta do arquivo, se execute o seguinte comando no terminal:

```
$ python3 main.py
```

3 Análise dos experimentos

Os experimentos foram rodados com 300 iterações de boosting. Esse número foi encontrado empiricamente ao longo do desenvolvimento e se mostrou um bom limite para que o erro convergisse.

3.1 Evolução do conjunto

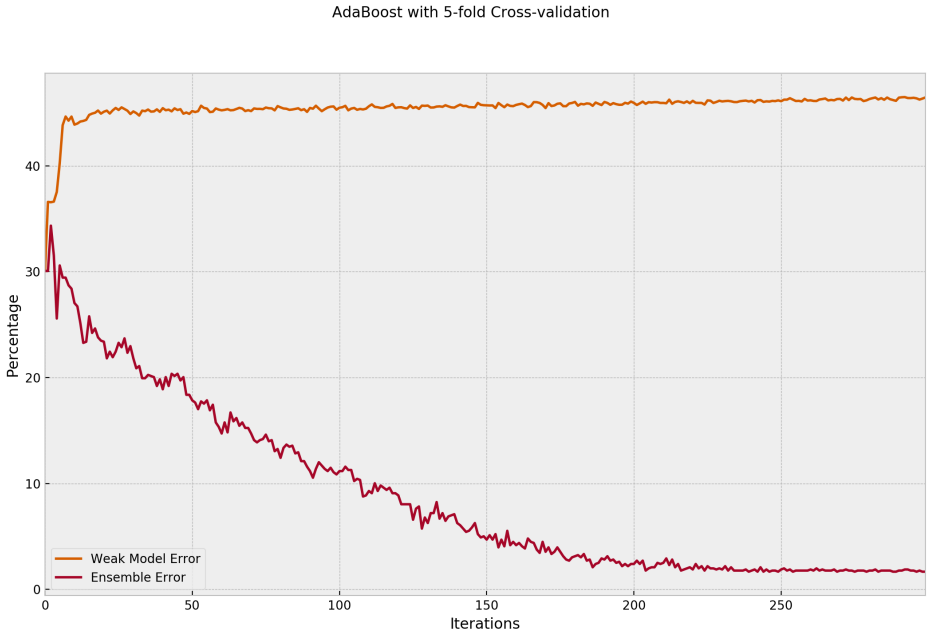

Figura 2: Execução de boosting com 300 iterações e 5-fold cross-validation.

Na figura 2 podemos ver a evolução do conjunto de preditores evoluindo ao longo do tempo. Apesar de ser um gráfico simples, ele guarda muita informação. Temos três pontos interessantíssimos a notar:

1. Na primeira iteração os erros de ambos são muito próximos, mas isso era algo a se esperar. No entanto, podemos nos perguntar o porque do valor do classificador fraco estar tão baixo nas primeiras iterações quando comparado ao restante da execução. Isso ocorre devido a dois fatores: A distribuição dos valores de saída na base e a distribuição uniforme dos pesos na primeira iteração. Se olharmos para quantidade de instâncias com atributo de saída "positive" na base, veremos 626 instâncias, compreendendo 65.34% das 958 instâncias presentes no total. Sendo assim, é fácil ver o porque do erro inicial ser por volta de 30%, pois além dessa grande discrepância na distribuição dos labels, os pesos serem todos iguais não cria nenhum viés para evitar instâncias já acertadas por modelos prévios. Rapidamente o erro dos stumps sobe, ao mesmo tempo que o peso para as instâncias "fáceis" começa a decrescer mais do que os outros.

2. A medida que o boosting progride, a redistribuição dos pesos faz com que o erro do classificador fraco tenda a aleatoriedade(50%). Mesmo assim, o ensemble continua melhorando, pois, por mais que pareçam aleatórios, os modelos fracos estão sendo ajustados para instâncias que tem sido erradas com frequência.

3. O erro estaciona por volta 1.9%. Ao observarmos mais de perto os dados retornados pelo programa, foi possível identificar que isso acontecia porque o ensemble tendia a errar as 8 últimas instâncias sempre, variando, sem nenhum padrão, os erros restantes nas outras instâncias. Isso se deve a natureza intrínseca da base de dados que tem uma mudança brusca no padrão de entrada exatamente nas últimas 8 instâncias, impossibilitando que os modelos fracos acertem todas de uma vez.

3.2 Influência da validação cruzada

Pensando na real influência da validação cruzada em cima dos resultados, resolvemos criar uma visualização dos dados que comparasse a execução com 5-fold cross-validation e a execução em cima de toda a base de dados, tanto no treino quanto no teste, a cada iteração. A visualização resultante corresponde à figura 3.

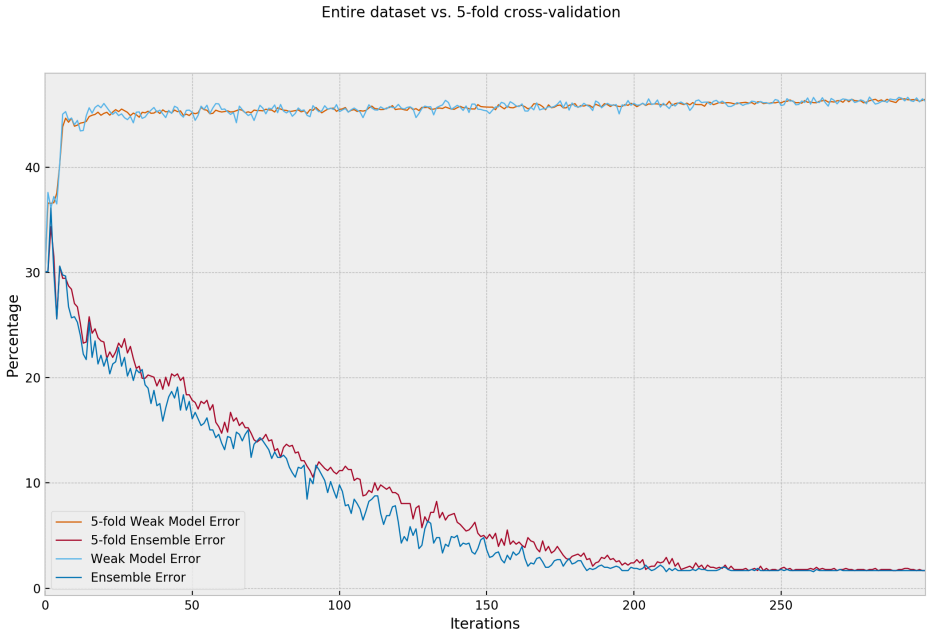

Figura 3: Execução com todo o dataset como treino e teste vs. 5-fold cross-validation.

Os pontos que valem ser notados nesse gráfico são:

1. O erro do classificador fraco para execução com validação cruzada é bem menos irregular que sua contraparte. Isso obviamente se deve ao fato de ele ser uma média de 5 execuções, ficando assim bem mais estável.

2. A execução sem validação cruzada convergiu muito mais rápido porque ficou "viciada" nos dados, o que causaria overfitting se não fosse a barreira imposta pela própria base de dados, já explicada na seção anterior. Como essa barreira existe, ambos os erros ficaram por volta de 1.67%

4 Conclusão

AdaBoost foi o primeiro mas hoje é apenas mais um dos algoritmos de boosting que existem por aí. Outros exemplos seriam Gradient Boosting e Extreme Gradient Boosting(xgboost), sendo esse último a tecnologia estado-da-arte para diversas aplicações hoje em dia.

Foi ótimo aprender o básico de como funciona o boosting, pois esse é um conhecimento que pode ser aplicado a uma gama quase infinita de problemas, por ser tão genérico. Apesar disso, sua implementação não foi fácil, tendo sindo especialmente complicada na hora de decidir como os modelos de predição fracos funcionariam, por uma falta de literatura direcionada especificamente a dados no formato da base Tic-Tac-Toe. Mesmo assim, o código foi muito bem implementado e documentado, podendo ser reutilizado com facilidade no futuro.