Aprendizado de Máquina: Trabalho Prático 2 (Boosting)

João Mateus de Freitas Veneroso

Departamento de Ciência da Computação da Universidade Federal de Minas Gerais

June 21, 2017

Introdução

Este relatório descreve a implementação do trabalho prático 2 da disciplina Aprendizado de Máquina. O trabalho consistiu em implementar um algoritmo de Boosting e treiná-lo no *dataset* Tic-Tac-Toe, que consiste em todas as combinações de jogadas possíveis no Jogo-da-Velha. A avaliação da eficácia do modelo foi feita por meio da análise do erro simples, utilizando a metodologia de *K-Fold Cross-Validation* com 5 partições.

Modelo

Os algoritmos de *Boosting* consistem em um agrupamento de múltiplos *weak learners*, cujo desempenho individual é ruim, para formar um *strong learner*, cujo desempenho é muito melhor. Os *weak learners* são modelos de aprendizado com uma taxa alta de erro (próxima de 50% no caso de uma classificação binária) e uma baixa variância. A combinação de *weak learners* com taxas de erro independentes produz um modelo mais robusto, que, no entanto, preserva a propriedade de baixa variância, prevenindo o fenômeno de *Overfitting*.

O algoritmo de *Boosting* implementado neste trabalho foi o *AdaBoost*. Os *weak learners* utilizados foram os *Decision Stumps*, que consistem em árvores de decisão com apenas 1 nível.

O *Ada Boost* consiste em um processo iterativo que atribui um peso α_i para cada *weak learner* e classifica o dado com base em uma função de classificação binária $H(x)$, cujo valor (-1 ou 1), representa a classe correspondente à entrada x. A definiÇão de $H(x)$ é:

$$H(x) = sign(\alpha_1 h_1(x) + \alpha_2 h_2(x) + ... + \alpha_n h_n(x))$$

O processo de treinamento do nosso algoritmo consiste em ajustar os valores α_t para diminuir o erro empírico de $H(x)$. À medida que cresce o número n de classificadores fracos, o erro empírico tende a diminuir, convergindo para zero no limite da capacidade do modelo. À cada iteração t, selecionamos o classificador fraco com o menor erro empírico e calculamos o seu peso α_t por meio da seguinte expressão:

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

onde ϵ_t é o erro empírico do classificador selecionado. No entanto, o erro empírico não consiste apenas na classificação e avaliação simples do *dataset* M. Cada caso de treino x_i contribui com um peso w_i ao erro empírico ϵ_t, de forma que:

$$\epsilon_t = \sum_{i \in M} w_{i,t} E(x_i)$$

onde $E(x_i)$ é o erro no caso de treino i. O cálculo de $w_{i,t}$ se dá pela expressão:

$$w_{i,t+1} = \frac{w_{i,t}}{z} e^{-\alpha_t h_t(x) y(x)}$$

Finalmente, o processo pode ser repetido n vezes, adicionando um novo classificador a cada iteração para que, assintoticamente, o erro empírico tenda à zero. Contudo, apesar do erro empírico tender à zero, o erro ponderado individual do classificador adicionado tende a crescer a cada iteração. Pois, à medida que prosseguimos com o treinamento, o peso dos casos mais difíceis tendem a aumentar e os padrões começam a se tornar cada vez mais complicados de discernir, de forma que, no limite, os novos classificadores começam a errar 50% das vezes.

Cross-Validation

O *dataset* utilizado nos experimentos possui 958 exemplos de configurações de tabuleiros do Jogo-da-Velha e a informação se o jogador "X" ganhou ou perdeu o jogo. Para avaliar a eficácia dos modelos utilizamos a metodologia de *K-Fold Cross-Validation* com 5 partições. Primeiramente, os dados foram embaralhados de forma aleatória e divididos em cinco partições com: 232, 232, 232, 231 e 231 exemplos. Para cada número m de iterações do *Ada Boost*, o modelo foi treinado com 4 partições e o erro de teste foi calculado com a partição excluída. O processo foi repetido para as 5 combinações possíveis, excluindo cada uma das partições do conjunto de treinamento, e o erro de treino para aquele número de iterações m foi calculado por meio da média simples dos valores obtidos em cada uma das combinações.

Dataset

força bruta descrito abaixo é o mesmo implementado no Trabalho Prático 1.

O algoritmo 1 implementa uma solução de força bruta para o problema de compartilhamento de viagens. Primeiro inicializa-se o benefício máximo b^* com um valor negativo, dessa forma qualquer configuração válida vai proporcionar um benefício maior. A partir daí, no loop das linhas 5-10, para cada configuração válida, calcula-se o benefício total e atualiza-se o valor de b^* se este benefício for maior do que qualquer um encontrado até então. Além disso, a variável G_p^* guarda a configuração que proporcionou o maior benefício. Ao final do algoritmo, b^* guarda o valor do benefício máximo para o grafo G e G_p^* guarda a configuração que proporcionou este benefício.

A complexidade deste algoritmo depende do número de configurações G_p diferentes e do custo da função $ConstraintsAreValid$. O número de configurações G_p diferentes é 2^m para m igual ao número de arestas no grafo G. Pois, cada aresta pode estar presente ou não em G_p e nós queremos as combinações possíveis para todas as arestas m. O custo da função $ConstraintsAreValid$ depende do número de arestas, pois, para cada aresta (u, v), temos de verificar se ela é a única aresta que sai do vértice u, se o vértice v é um motorista e se v possui espaço para acomodar todos os passageiros de u. Como todas estas operações tem custo constante, a função $ConstraintsAreValid$ tem custo $O(m)$. Por último, a linha 7 calcula a soma dos pesos de todas as arestas também com custo $O(m)$. Logo, a complexidade total do algoritmo é $2m2^m$ e o algoritmo é $O(m2^m)$.

Algorithm 1 MaximizeBenefit

1: **procedure** MaximizeBenefit(G(V,A))
2: $\quad b^* \leftarrow -1$
3: $\quad G_p^* \leftarrow \emptyset$
4:
5: $\quad$ **for all** $G_p = (V, A_p) : A_p \subseteq G.A$ **do**
6: $\quad\quad$ **if** *ConstraintsAreValid(G_p)* **then**
7: $\quad\quad\quad b \leftarrow \sum_{(v_i,v_j)\in A_p} B(v_i, v_j)$
8: $\quad\quad\quad$ **if** $b > b^*$ **then**
9: $\quad\quad\quad\quad b^* \leftarrow b$
10: $\quad\quad\quad\quad G_p^* \leftarrow G_p$

Conclusão

Este relatório descreveu a implementação do trabalho prático 2 da disciplina Projeto e Análise de Algoritmos. Entre as três abordagens propostas, a abordagem gulosa apresentou claramente o melhor desempenho chegando a uma aproximação média de 94.27% da resposta ótima em um tempo muitas ordens de magnitude menor do que as abordagens exatas. A programação dinâmica mostrou um ganho muito grande de desempenho em relação ao algoritmo de força bruta, no entanto, a abordagem gasta uma quantidade muito maior de memória.