Trabalho Prático 2 - Implementação do Algoritmo de Boosting

André Luiz Moreira Dutra - 2019006345
Aprendizado de Máquina

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

cienciandre@ufmg.br

1. Introdução

Este trabalho se propõe a resolver o problema de predição do vencedor do jogo da velha utilizando uma implementação própria do algoritmo AdaBoost, apresentando uma análise da validação dos parâmetros do modelo utilizando validação cruzada.

A base de dados tic-tac-toe constitui uma lista de todas as possíveis configurações do tabuleiro de um jogo da velha, considerando que x jogou primeiro, e o rótulo de predição é uma variável binária indicando se x ganha o jogo. A configuração do tabuleiro é definida como um vetor de 9 posições, uma para cada posição do tabuleiro 3x3, onde cada uma contém uma pode conter “x”, indicando que o jogador “x” marcou o espaço, “o”, indicando que o jogador “o” marcou o espaço, ou “b”, indicando um espaço em branco.

Neste trabalho serão apresentadas todas as escolhas feitas com relação à implementação do modelo, a análise comparativa dos resultados da validação cruzada e os resultados finais do melhor modelo selecionado.

2. Implementação do modelo

2.1. Stump

Sendo um modelo de ensemble, o AdaBoost depende de modelos menores para formar o conjunto de modelos utilizado na classificação. Neste caso, serão usados stumps, que equivalem a árvores de decisão com apenas uma regra. Nesta seção será detalhada a lógica do stump e detalhes de sua implementação.

Usualmente, tratando-se de features contínuas, um stump escolhe uma feature e aplica a ela um limiar de decisão. Se a posição da variável estiver acima do limiar, ele decide como verdadeiro e, caso contrário, ele decide como falso. No entanto, as variáveis tratadas nesse caso são categóricas.

Para resolver este problema, consideramos que um stump, aplicado a uma feature k dos dados, tem como espaço de decisão as três checagens que ele pode aplicar em k quanto aos seus três valores possíveis: “x”, “o” ou “b”. Assim, se o valor de k é “o” ele tem checagens iguais a False, True, False.

A partir disso, o stump pode escolher levar ou não cada um desses valores em consideração. Por exemplo, se ele escolhe levar em consideração apenas um primeiro valor, ele se torna um stump que verifica se a k é igual a “x”. Se ele escolhe levar em consideração apenas os dois últimos, ele se torna um stump que verifica se k não é igual a “x” (pois ele vai retornar positivo se, e somente se, k tiver um dos dois outros valores que não “x”). Se ele escolhe levar os três em consideração, ele se torna um stump que sempre retorna positivo (pois k sempre será algum dos três valores).

Assim, para uma variável, cada stump pode, para cada uma das três decisões, levar ou não ela em consideração (duas opções), totalizando $2^3 = 8$ possíveis stumps para uma variável. Para as nove variáveis da base, temos 9*8 = 72 stumps possíveis.

Os stumps foram implementados na classe TicTacToeStump do módulo stump, e implementam apenas o construtor e o método predict, que realiza a decisão. O construtor recebe o índice da feature sobre a qual ele tomará a decisão e uma máscara de três booleanos. A máscara define quais variáveis aquele stump levará em consideração na decisão. O método predict realiza a previsão conforme explicado. Primeiro ele recebe o vetor de features e escolhe o valor da feature k no índice definido no construtor. Em seguida, ele checa se k é igual a “x”, “o” e “b”. Por fim, ele passa as três decisões pela máscara com um and (para jogar as variáveis não consideradas pelo stump para False e manter as demais intactas) e retorna o or do resultado (positivo se qualquer dos valores levados em consideração for verdadeiro).

Vale ressaltar que o código realiza um any no resultado, que consiste em um or aplicado a todos os elementos de uma lista. Além disso, o stump retorna 1 para True e -1 para False, por conveniência da implementação do módulo de boosting.

2.2. Boosting

O módulo de boosting implementa o algoritmo de treinamento das importâncias dos stumps. A lógica de funcionamento dele é idêntica à lógica de treinamento do AdaBoost vista em sala, e por isso focaremos em explicar como ela foi implementada, junto a outras escolhas de implementação.

O algoritmo está implementado na classe TicTacToeAdaBoost, no módulo adaboost, e recebe no construtor dois hiper-parâmetros: o número de stumps a ser utilizado e o número de iterações. O número de iterações é trivial, equivale a quantas iterações o algoritmo executará. Já o número de stumps é um número de 0 a 72 (considerado como 72 se um valor maior for passado) que determina a quantidade de stumps a ser usada no algoritmo. A escolha dos stumps é feita gerando os 72 stumps possíveis e retirando uma amostra aleatória sem reposição de stumps dessa lista equivalente ao número estipulado. Note que um número de stumps igual a 72 resultará num conjunto contendo todos os possíveis stumps. Os stumps são guardados

Além do construtor, são implementadas as funções fit, predict e score. Na função fit, o treinamento é implementado segundo o algoritmo AdaBoost. A função recebe X e y como features e labels do modelos. Antes do loop, ela inicia o vetor de pesos dos exemplos com pesos iguais para todos, e calcula a predição de cada stump para os dados de X (note que o valor de X e os stumps são fixos, então as predições também são. O que varia são os pesos dos exemplos, que são aplicados posteriormente no cálculo do erro de predição). A função ponderada de predição está implementada em uma função à parte, prediction_error, que recebe as labels reais, as predições e os pesos, e retorna a soma os pesos dos exemplos onde a predição foi errada (diferente do valor real).

Em seguida, o algoritmo entra em um loop, para o número de iterações definido, no qual ele:

1. Escolhe, para os pesos atuais, o stump com menor erro ponderado de predição.
2. Atualiza a importância desse stump segundo uma função log aplicada ao erro obtido (quanto menor o erro, maior a importância).
3. Atualiza os pesos dos exemplos segundo uma função exponencial aplicada à nova importância calculada (exemplos cuja predição errou recebem mais peso, e exemplos de acerto recebem menos, quanto quanto maior a importância do modelo).
4. Os pesos são normalizados para que somem 1.

Assim, ele a cada iteração escolhe e dá importância aos stumps que tiveram um bom desempenho nos exemplos que o stump anterior não conseguiu prever, enquanto ajusta os pesos dos exemplos para que o próximo stump seja escolhido para

os exemplos que ele não conseguiu resolver. As importâncias de cada modelo são salvas em uma variável global à classe, que será usada para a predição.

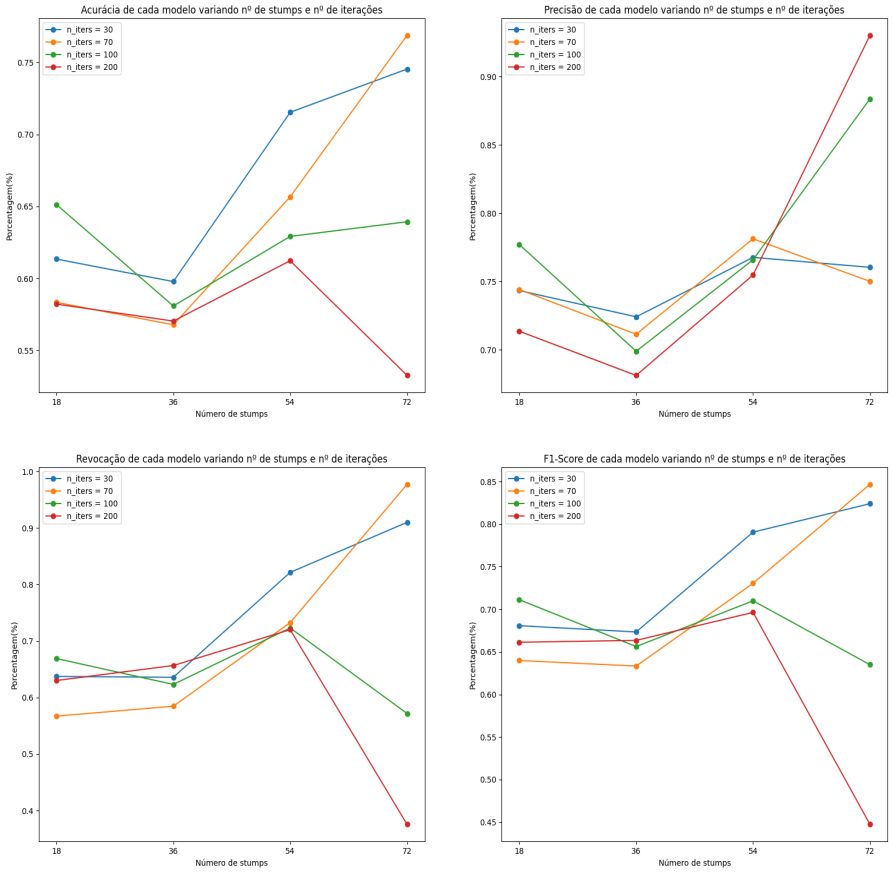
O método predict recebe um único vetor de features x e aplica nele a predição, após o treino feito no fit. Para isso, ele soma a predição de cada stump para a label de x (1 ou -1), ponderando pela importância de cada modelo. Se o valor for positivo, ele retorna True e, caso contrário, retorna False.

O método score recebe X e y e realiza a validação ou teste do modelo. Para isso, com o modelo já treinado, ele chama predict para cada vetor de features x em X e, tendo o vetor de previsões de y, retorna a acurácia, precisão, recall e f1 entre elas e o vetor real de y dado como entrada.

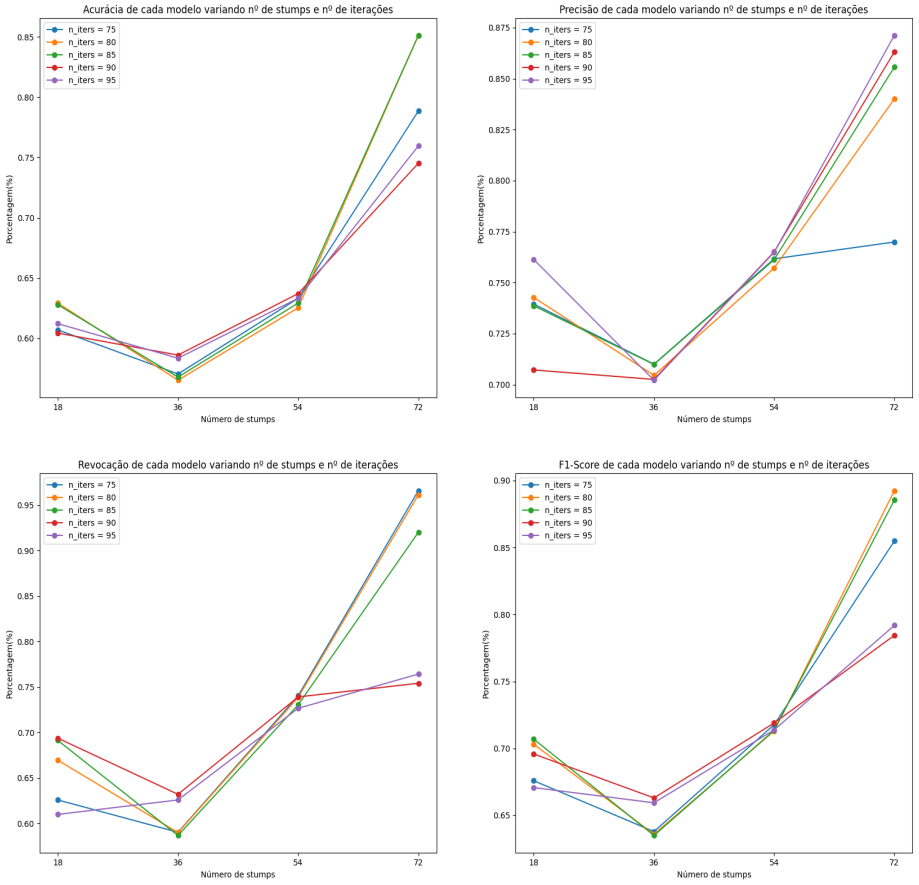
3. Validação do Modelo

O modelo tem como hiperparâmetros o número de stumps e o número de iterações executadas. A base foi dividida em 80% treino e 20% teste e, para o treino, fizemos a validação desses dois parâmetros por meio de uma validação cruzada de 5 folds. A validação foi feita usando o método KFolds do scikit-learn, onde para cada fold um modelo foi treinado e os scores coletados, e ao fim foi feita a média dos cinco valores de cada score. Isso foi repetido para todas as combinações de valores de parâmetros analisadas.

Inicialmente, testamos o número de stumps variando entre 18, 36, 54 e 72, e o número de iterações variando entre 30, 70, 100 e 200. Observe os resultados:



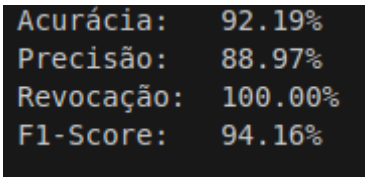
Como podemos ver, a acurácia e f1 do modelo num geral começam a cair quanto mais iterações são executadas, principalmente para valores maiores de stumps, indicando um overfitting do modelo. Podemos observar que, a partir de certo ponto, ele passa a favorecer a precisão em detrimento do recall, pois a precisão é a única métrica em que estes modelos se saíram bem. Podemos ver que o modelo de melhor desempenho geral foi o de 72 stumps rodados por 70 iterações. Vamos agora refazer a validação para passos menores de iterações. Assim, vamos checar modelos com o mesmo range de número de stumps, mas um range de número de iterações variando em 75, 80, 85, 90 e 95:



Para estas configurações, o uso de 72 stumps mostra uma tendência claramente maior que os demais em todos os casos. Além disso, há muito menos variância entre os dados, a maioria segue em torno dos demais sem quedas ou subidas muito bruscas. Podemos ver um padrão com relação ao número de iterações em quase todas as métricas: números de iteração muito altos e muito baixos não se desempenharam muito bem. A única exceção foi a precisão com relação aos modelos de mais iterações, para os quais, novamente, mais iterações tiveram um desempenho melhor.

Observando o gráfico, podemos ver que os modelos com 80 e 85 iterações tiveram resultados muito parecidos, mas aquele com 80 iterações e 72 stumps venceu por muito pouco. Com isso, fica definido que o melhor modelo tem como parâmetros 80 iterações e 72 stumps.

Por fim, juntamos novamente treino e validação, treinamos o modelo com todos os dados de treino e testamos com os 20% dos dados separados para o teste. As métricas obtidas foram as seguintes:



Finalizando em 92% de acurácia, 89% de precisão, 100% de recall e 94% de F1.

4. Conclusão

Este trabalho permitiu a consolidação do aprendizado sobre algoritmos de boosting, por meio da implementação manual do algoritmo AdaBoost. A implementação permitiu a compreensão do algoritmo em diversos aspectos, desde a modelagem dos stumps à interpretação dos cálculos dos pesos e das importâncias envolvidos no modelo. Por fim, com a validação cruzada foi possível compreender na prática como esta técnica é implementada, além de obter uma métrica mais segura de predição do erro do modelo. A análise dos resultados também contribuiu para o conhecimento crítico quanto aos passos necessários para calibrar os hiperparâmetros do modelo, um conhecimento essencial no desenvolvimento de qualquer modelo.

Bibliografia

VELOSO, Adriano. *Slides da disciplina Aprendizado de Máquina. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2023.*