Trabalho Prático 2 - Aprendizado de Máquina
Implementação ADABoost

Daniel Souza de Campos – 2018054664

UFMG – 2022/1

Sumário

1 Introdução 1
2 Ferramentas 2
3 Implementação 2
3.1 Como executar 2
3.2 Main.py 3
3.3 ADABoost.py 3
4 Experimentos e Resultados 5
5 Conclusão 7

1 Introdução

Em interações no mundo real, todas as observações de um fenômeno não físico e, portanto, não previsíveis por meio de uma função bem definida, possuem um ruído. Esse ruído faz com que os dados não sigam uma função bem definida, mas, ainda assim, podem pertencer a uma mesma classe lógica. A tarefa de classificação em Aprendizado de Máquina possui o objetivo de treinar um modelo que consiga prever, com maior acurácia possível, as classes de observações de um fenômeno qualquer. Para tal, naturalmente, pode-se empregar modelos com capacidade alta e que, portanto, conseguem se adequar muito bem aos dados de treino. Esse tipo de modelo possui baixo viés de previsão, mas alta variabilidade quando aplicado a dados nunca vistos. Por outro lado, pode-se aplicar modelos com capacidade baixa. Esses modelos são mais generalistas e, portanto, mais enviesados, mas possuem baixa variabilidade. A escolha de um modelo correto deve considerar essa questão descrita que é conhecida como *trade-off* entre o viés e a variância.

A balanceio entre viés e variância também está atrelado com outros dois conceitos de *machine learning* (ML): *overfitting* e *underfitting*. Ao aplicar um modelo com alta capacidade, deve-se garantir que ele possua dados suficientes para que, ainda assim, seja generalista o bastante para prever dados futuros que fujam do padrão dos dados de treino. Caso isso não aconteça, provavelmente ocorrerá o chamado *overfitting* que é quando o modelo se ajusta muito bem aos dados de treino, praticamente produzindo erro empírico igual a zero. Entretanto,

ao receber dados nunca vistos, ele irá errar consideravelmente, ou seja, erro esperado alto. Já ao aplicar um modelo com baixa capacidade, ele não conseguirá se adequar suficientemente aos dados de treino. Isso produzirá um erro empírico e, provavelmente, erro esperado alto. Isso significa que se pode estimar muito bem o erro esperado a partir do erro empírico, entretanto, com ambos altos.

Uma das melhores propostas para a área é treinar um modelo formado de vários modelos com baixa capacidade, mas que, juntos, produzem erro empírico baixo. Essa ideia é conhecida como *Boosting* e ela possui algumas variantes. O *Adaptive Boost* (ADABoost) é uma dessas variantes que, iterativamente, treina modelos com baixa capacidade, mas que tentam prever melhor onde os modelos treinados anteriormente erraram. Dessa forma, mesmo que alguns modelos errem a classe de um dado de treino, a maioria irá acertar a sua classe. Isso segue uma lógica democrática onde a vontade da maioria dos modelos será a predição final para uma entrada.

Esse trabalho possui o objetivo de explicar a implementação do algoritmo ADABoost e discutir dificuldades e resultados.

2 Ferramentas

O trabalho foi desenvolvido na linguagem Python 3.8 no editor de códigos *Visual Studio Code*. Para controle de versões e armazenamento, foi utilizado o GitHub e o trabalho ficará disponível publicamente em um repositório próprio após a data de entrega limite do trabalho 18/07/2022. A máquina para desenvolvimento é um notebook DELL 8 GB RAM com processador Intel core i5 7° geração e HDD de 1 TB.

3 Implementação

3.1 Como executar

O trabalho foi feito de forma que o principal arquivo seja o *main.py*. Para executar o ADABoost implementado em cima do *dataset* específico, deve ser passado o endereço do dataset como parâmetro da seguinte forma:

```
python main.py --data_path <dataset_path>
```

O programa também aceita alguns outros parâmetros que podem ser vistos com:

```
python main.py -h
```

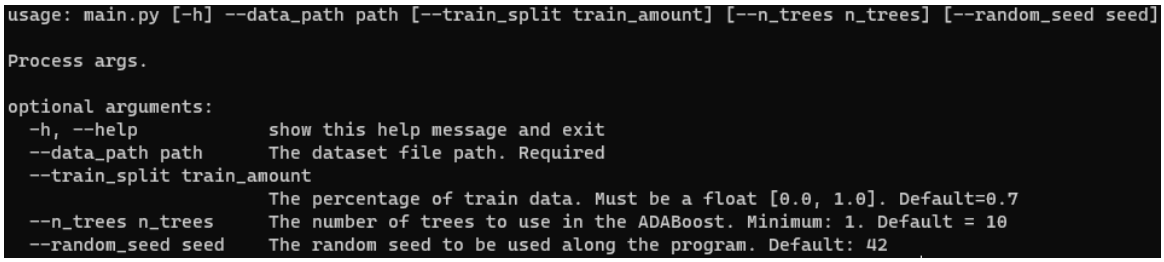

Figura 1: Argumentos aceitos pelo programa

3.2 Main.py

O arquivo main.py possui uma função principal chamada *predict*. É nela que ocorre a lógica principal do programa. Primeiramente, os dados de entrada são transformados na função *treat_data* da seguinte forma:

- Valores iguais a ‘x’ são transformados em 1
- Valores iguais a ‘o’ são transformados em 0
- Valores iguais a ‘b’ são transformados em -1
- Valores iguais a ‘positive’ são transformados em 1
- Valores iguais a ‘negative’ são transformados em -1

Logo depois, os dados são divididos em treino e teste com uma proporção indicada pelo argumento de entrada *--train_split* utilizando a função *get_train_test_data*. Essa proporção possui o valor padrão de 0.7.

Depois disso, é realizada a validação cruzada com 5 *folds* em cima do dado de treino na função *get_kfold_mean_score*. Essa função, como o próprio nome já diz, retorna a acurácia média do modelo nesse processo como um valor entre 0 e 1. O modelo utilizado terá o número de árvores informado no argumento *--n_trees* com valor padrão igual a 10.

Uma vez com o erro empírico em mãos, que pode ser calculado simplesmente como 1 menos a acurácia média, o modelo é treinado com o dado de treino completo e, então, prevê sobre o dado de teste. Dessa forma, teremos o erro esperado do modelo.

Ao final da execução, será impresso na saída padrão a acurácia média na validação cruzada e a acurácia no dado de teste.

3.3 ADABoost.py

O arquivo onde o ADABoost está, de fato, implementado é o ADABoost.py. Sua primeira função é a *fit*. Ela é responsável por treinar as árvores de decisão utilizadas em cima dos dados de treino providos.

Primeiramente, os pesos de cada entrada são definidos como 1 dividido pelo número de entradas na função *_get_starting_weights*. Depois, serão

treinadas, no máximo, *--n_trees* árvores. O único caso onde não serão treinadas todas as árvores será quando for detectado que alguma árvore obteve erro 0 de predição. Nesse caso, o programa para de treinar novas árvores e mantém as que já existentes.

Cada árvore a ser treinada T_i é retornada pela função *_get_new_tree*. Como as árvores são *DecisionTreeClassifier's* da biblioteca *sklearn*, elas possuem a capacidade de serem treinadas utilizando a função *fit*. Os pesos de cada ponto de entrada também são fornecidos. Uma vez com a árvore atual treinada, é realizada uma predição sobre os próprios dados de treino para que possamos recuperar em quais pontos a árvore atual erra a predição. O erro da árvore é calculado na função *_get_tree_error* que usa a função *accuracy_score* do *sklearn.metrics* também levando em consideração os pesos das entradas.

Uma vez calculado o erro da árvore atual, pode-se calcular o seu valor α_i da seguinte forma:

$$\alpha_i = \frac{1}{2} \times \log\left(\frac{1 - \epsilon_{\mathrm{i}}}{\epsilon_{\mathrm{i}}}\right)$$

Uma questão que pode surgir é se o erro da árvore atual for igual a 0. Nesse caso, ficou definido que o valor do α_i será igual a 1. Outra questão que poderia atrapalhar essa conta seria o caso em que o erro é igual a 1, entretanto, seria necessário que a árvore de decisão errasse todas as suas predições, o que não é possível de acontecer.

Agora, com o erro ϵ_i e o α_i, é possível atualizar os pesos das entradas na função *_update_weights*. A atualização dos pesos pode é realizada da seguinte forma:

$$W_{i+1} = \frac{W_i}{Z} \times e^{-1 \times \alpha_i \times \hat{y} \times y}$$

Sendo W_i os pesos atuais da entrada, W_{i+1} os próximos pesos, α_i o peso da árvore T_i treinada, $\hat{y}$ as classes preditas, y as classes reais e Z um normalizador para que a soma de todos os pesos seja igual a 1.

Para tal, primeiramente, é gerado o vetor de constante de Euler elevado a $-1 \times \alpha_i \times \hat{y} \times y$ na função *_euler_exponents*. Depois, esse vetor multiplica o vetor de pesos atuais das entradas. Por fim, esse vetor resultante é normalizado para que a sua soma seja igual a 1.

A segunda principal função da classe ADABoost é a função *predict*. Essa função apenas chama outra de nome *_predict_from_trees*. Essa última realiza a predição dos pontos de entrada em todas as árvores treinadas atualmente. O resultado final de predição para cada ponto é gerado na função *_weighted_most_frequent* que retorna o valor previsto associado a maior soma de α's relativas às árvores que previram esse resultado. Por exemplo, se a soma dos α's das árvores que previram o valor 1 for igual a 0.4 e a soma dos α's das árvores que previram o valor -1 for igual a 0.6, o valor retornado será -1.

4 Experimentos e Resultados

O primeiro experimento realizado foi variar o número de árvores utilizadas no modelo e examinar a sua “acurácia empírica” e “acurácia esperada”. Os resultados podem ser vistos no gráfico a seguir.

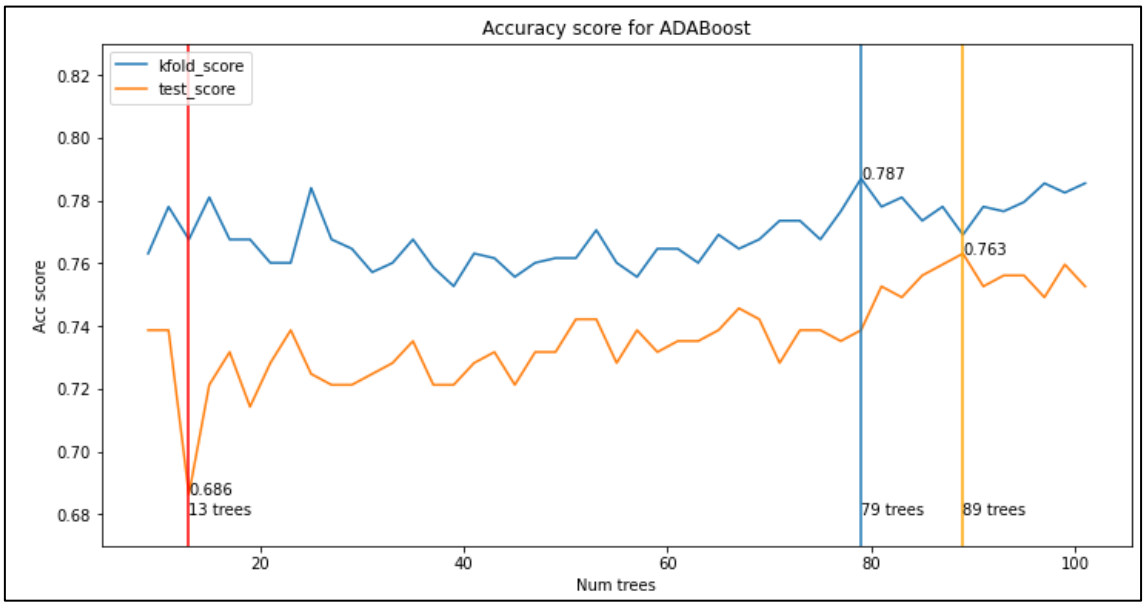

Gráfico 1: Acurácia para o modelo variando a quantidade de árvores

Pode-se perceber que a “acurácia empírica” sempre se manteve otimista com relação à esperada, o que é um fenômeno comum. O modelo que performou melhor na validação cruzada foi aquele com 79 árvores alcançando acurácia de 78.7%, entretanto, o melhor modelo com relação à “acurácia esperada” foi aquele com 89 árvores. O pior modelo foi aquele com 13 árvores.

A próxima comparação foi realizada anotando o erro a cada iteração de treinamento para cada árvore do melhor e pior modelos descritos anteriormente.

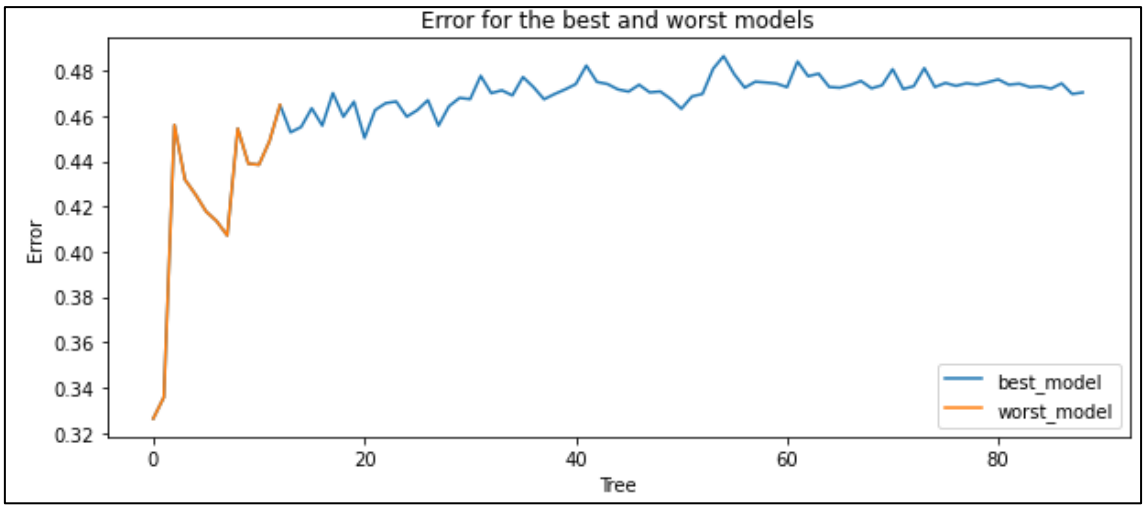

Gráfico 2: Erro por árvore para o melhor e pior modelos

Como dito anteriormente, o melhor modelo foi aquele com 89 árvores e o pior foi aquele com 13 árvores. Para ambos modelos, pode-se perceber que o erro para cada árvore treinada teve tendência de crescimento até estabilizar em certa de 0.47. Uma observação de algo que já era de se esperar é que os erros para o pior modelo são exatamente iguais àqueles do melhor modelo para as suas primeiras 13 árvores. Isso faz sentido pois ambos sempre tentam prever sobre os mesmos pontos e pesos de entrada a cada iteração. Dessa forma, o que diferencia um modelo de ADABoosting com árvores para o outro é, simplesmente, o seu número de árvores. Assim, é possível pegar um modelo com *i* árvores já treinadas e, então, adicionar ou remover árvores de acordo com a necessidade sem ser preciso treiná-la do zero para um mesmo conjunto de dados de entrada.

Uma última análise a ser feita é sobre o efeito da proporção de dados de treino sobre o erro empírico e esperado para o melhor modelo.

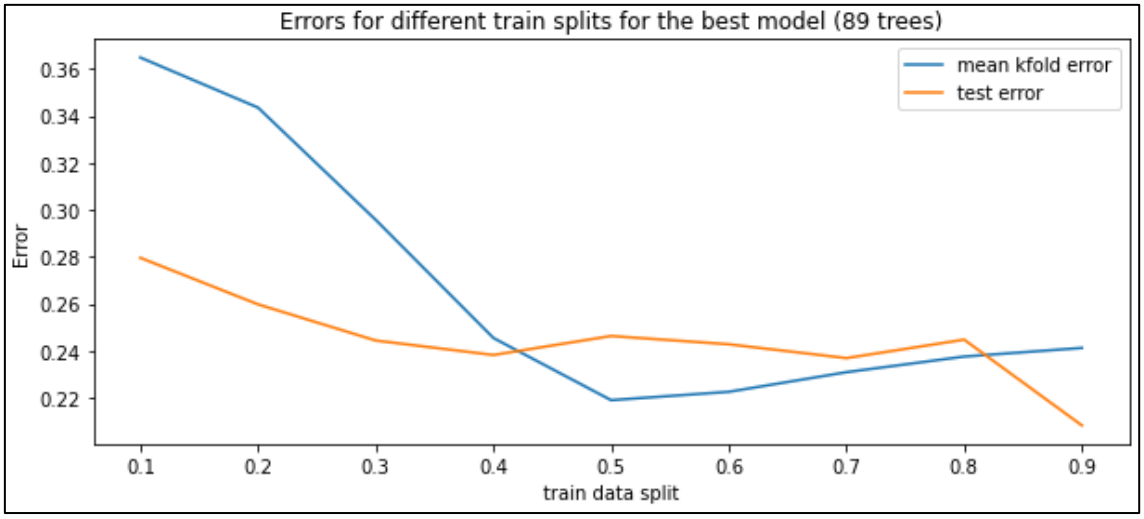

Gráfico 3: Efeito da proporção de dados de treino sobre os erros

Como é de se esperar, quanto mais dados de treino são oferecidos ao modelo, menor é o seu erro esperado. Algo inesperado é que entre a proporção de treino de 0.1 e 0.4 e para 0.9, o erro empírico é maior do que o esperado.

Lembrando do Gráfico 1, parece existir uma pequena tendência de aumento de acurácia quanto mais árvores são permitidas ao modelo. A ideia de adicionar mais árvores a um modelo pode levar a crer que o modelo possuirá capacidade maior e, portanto, irá ocasionar em *overfitting* nos dados de treino se esse não for suficientemente grande. Entretanto, a diferença observada para o erro esperado entre o modelo com 10 árvores e o modelo com 100 árvores é pequena, ainda que maior para o último. Dessa forma, somos levados a crer que o *overfitting* não ocorreu exatamente por uma característica dos modelos de *boosting*: Não é possível ocorrer *overfitting* quanto mais árvores são adicionadas pois a capacidade das árvores individuais é baixíssima e cada uma delas é treinada sobre uma versão alterada dos dados de entrada. Assim sendo, não é possível gravar os resultados dos dados de entrada.

5 Conclusão

Uma das tarefas mais difíceis em ML é o de obter um *trade-off* satisfatório entre o viés e a variância do modelo. Para tal, existem os modelos de *Boosting* que empregam uma forma diferente de aprendizado ao combinar vários modelos de capacidade baixa para formar um modelo que se sai igualmente bem a um modelo que foi tunnado para a tarefa. Assim sendo, o objetivo inicial desse trabalho era implementar uma versão do ADABoost que pudesse prever classes em cima de um dataset especificado de antemão.

Pode-se dizer que esse objetivo foi alcançado visto todo o programa que foi desenvolvido e explicado ao longo desse documento. A dificuldade de implementação foi relativamente baixa visto que a ideia do modelo é bastante simples ainda que eficaz. Entretanto, a atualização dos pesos das entradas e a forma de escolha da predição final para uma entrada foram alguns dos pontos com maior dificuldade de implementação.

Com os resultados dos experimentos realizados, foi possível obter *insights* significativos sobre o modelo além de confirmar conceitos explicitados na matéria ao longo do curso.

Por fim, foi possível colocar em prática conceitos aprendidos em sala de aula além de conseguir um pouco de experiência em como implementar um modelo de ML do início ao fim, ainda que com ajuda de bibliotecas com vários recursos prontos.