PUCRS

ESCOLA POLITÉCNICA
CURSO DE ENGENHARIA DE SOFTWARE

GRUPO 4:
LEONARDO VARGAS SOARES,
LUCAS GOMES MARTINS,
VÍTOR FELIPE LEITENSKI DELELA.

Disciplina: Inteligência Artificial.

Professora: Silvia Moraes.

Trabalho 1 - ML: Tic Tac Toe.

Porto Alegre
2023

GRADUAÇÃO

Pontifícia Universidade Católica
do Rio Grande do Sul

Sumário

1. Trabalho 1 ML: *Tic Tac Toe*

1.1 Descrevendo o Problema

O trabalho consistiu no desenvolvimento de um sistema de IA para atuar em cima de um jogo da velha clássico, onde o mesmo realizava a verificação do resultado do jogo a cada jogada.

Para isso, solicitava-se a utilização de três algoritmos diferentes de IA e a realização da comparação entre os resultados obtidos a partir de cada uma delas.

2. Dataset

O *dataset* foi obtido no link disponibilizado no enunciado do trabalho, contendo cerca de 900 possíveis entradas/configurações de jogo e seus respectivos resultados.

Inicialmente, havia somente dois rótulos existentes para indicar os resultados, sendo eles: *positive* e *negative*. Após analisar uma certa quantia de dados, foi percebido que havia uma inconsistência, uma vez que os dados rotulados com *positive* estavam indicando uma vitória do jogador X, enquanto os dados rotulados com *negative* estavam indicando tanto vitória do jogador O quanto empates.

Assim, foi adicionado um novo rótulo ao *dataset*, denominado *draw*, a fim de cobrir os possíveis cenários de empate no jogo.

Ao longo da elaboração das soluções descritas abaixo, também foi percebida a necessidade de realizar algumas outras alterações no *dataset*. Primeiramente também foi realizada a inclusão de um resultado “continue” visando identificar jogadas em que não há nem um vencedor nem empate, uma vez que as soluções se mostraram bastante errôneas nas primeiras jogadas inseridas pelos usuários, quando sequer havia posições suficientes no tabuleiro para algum resultado final.

Além disso, visando justamente auxiliar na precisão do conjunto de dados, foi desenvolvida um método para providenciar *feedbacks* ao *dataset* no final de cada movimento do jogo. Ou seja, após o algoritmo mostrar a leitura de jogo daquele momento, o mesmo pergunta ao usuário se a leitura está correta e, se não estiver, solicita a inserção do resultado esperado. Isso grava, no *dataset*, o cenário atual do tabuleiro juntamente com o resultado esperado.

Essa última opção foi utilizada praticamente para cobrir as primeiras movimentações do tabuleiro e deixada desativada depois, evitando assim, popular em demasiado o *dataset*.

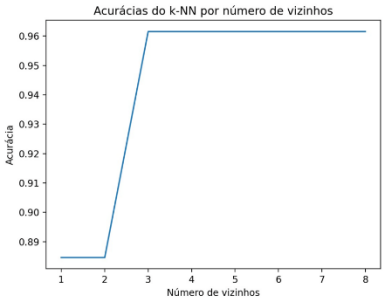
Depois do dataset estar devidamente ajustado e calibrado, foram criados então os *datasets* de treino e teste para os algoritmos. Foram retiradas cerca de 50 entradas do *dataset* original para o teste, contendo os resultados de vitórias e empate, e as demais entradas para o treino.

3. Desenvolvimento Das Soluções

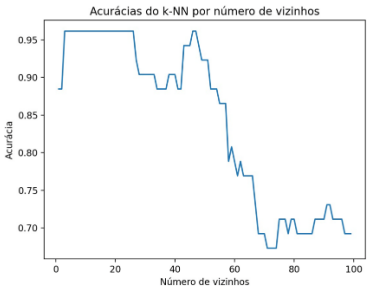
3.1 k-NN

Para a realização do algoritmo *k-NN*, foi usada a biblioteca *sklearn* do *Python*, a qual permite realizar uma instância do modelo. A partir disso, foi trabalhado em cima dos dados puxados do *dataset* para dividi-los nos *arrays* de posições de treino, rótulos de treinos, posições de teste e rótulos de teste. Os dois primeiros puxados do *dataset* de treino e os dois últimos do *dataset* de treino. Essa diferenciação se dá pela última informação de cada dado dos *datasets*, ou seja, o resultado esperado. Assim, temos os *arrays* com os cenários do tabuleiro e os *arrays* com os rótulos separadamente. Através do modelo *Knn* instanciado, foi realizado o treinamento e, em seguida, os testes, ambos utilizando métodos da biblioteca importada. A mesma biblioteca também permitiu o cálculo de acurácia, através da comparação entre os rótulos de teste armazenados no *array* citado anteriormente, e o rótulos gerados a partir da previsão do modelo.

Após o desenvolvimento do algoritmo, foram realizados os testes do modelo através da utilização da biblioteca *matplotlib* do *Python*, a qual permitiu a geração de gráficos comparando a acurácia do sistema pelo valor de vizinhos do k-NN. Foi gerado dois gráficos com a finalidade de encontrar o valor de vizinhos mais assertivo. O primeiro gráfico foi levando em consideração apenas valores razoavelmente baixos de vizinhos (1 até 8), onde, a partir do terceiro, todos demonstraram uma acurácia igual.



Visto isso, foi pensado em gerar um segundo gráfico com valores maiores, para verificar até onde o algoritmo mantém a acurácia. Assim, foi executado o mesmo teste para vizinhos variando de 1 até 100, onde foi obtido o seguinte resultado.



Dessa vez, é possível visualizar que a acurácia constante apresentada anteriormente se mantém linear até pouco antes do número 30 e, após isso, o valor médio começa a ser decrementado. Os gráficos ajudaram a definir o valor de K sendo 6, escolhido aleatoriamente dentro da faixa 3-30, uma vez que qualquer valor entre esses apresentou o mesmo resultado.

3.2 MLP

Assim com o algoritmo K-NN também foi utilizada a biblioteca “Scikit Learn”, instanciando o modelo “MLPClassifier”.

Fazendo uso de dois conjuntos de dados, um denominado treino e outro teste, foi realizado o treinamento do algoritmo, inicialmente utilizando os valores de atributos padrões, então buscando uma melhor precisão foram alterados e testados diversos valores para os parâmetros obtendo uma precisão entre 92% e 98%, valores aproximados.

Como exemplo podemos citar vários testes de valores para a quantidade de camadas, assim como a quantidade de neurônios com valores entre 10 e 1000 para neurônios e entre 1 e 10 para camadas, a forma de ativação, e solucionador. Apesar do solucionador lbfgs ser mais indicado para datasets menores apresentou uma precisão de 96,15% fazendo o adam ser o escolhido por ter tido melhor performance.

Sendo assim a precisão final foi de 98,08% para a topologia definida abaixo.

```
def MlpAlgorithm():
    mlp = MLPClassifier(
        hidden_layer_sizes=(1000,),
        max_iter=600,
        verbose=False,
        learning_rate="constant",
        learning_rate_init=0.01,
        activation="relu",
        solver="adam",
    )
```

3.3 Árvore de Decisão

Através de pesquisas foi possível compreender a **lógica** realizada através de uma arvore binaria (*True* | *False*) de condições, sendo um modelo de aprendizado supervisionado que como próprio nome sugere ser um método apropriado para tomada de decisão.

As árvores de decisão são algoritmos utilizados tanto em tarefas de regressão quanto classificação. Seu objetivo é criar uma estrutura hierárquica de nós que aprendem com os dados por meio de regras básicas. Essa estrutura funciona como um fluxograma, onde cada nó de decisão se relaciona com outros por meio de uma hierarquia. O nó-raiz é o mais importante e os nós-folha representam os resultados. No contexto de *machine learning*, o nó-raiz é um dos atributos da base de dados e o nó-folha é a classe ou o valor que será gerado como resposta.

Vantagens:

- Modelo intuitivo e simples de interpretar.

Desvantagens:

- Sensibilidade a dados de treinamento, pequenas variações nos dados de treinamento podem levar a grandes diferenças na estrutura e precisão da árvore gerada.
- Pode gerar *overfitting*, onde o modelo se ajusta excessivamente aos dados de treinamento e, como resultado, não generaliza bem para novos dados, é um comportamento indesejável de aprendizado de máquina que ocorre quando o modelo de aprendizado de máquina fornece previsões precisas para dados de treinamento, mas não para novos dados.

Motivo da Escolha: Durante a implementação, tanto a Árvore de Decisão quanto o algoritmo Bayesiano foram implementados sem um motivo específico para escolha. A decisão foi baseada na vontade de compreender e avaliar o desempenho de ambos os modelos.

A biblioteca *sklearn* também foi utilizada para aplicar o algoritmo de Árvore de Decisão, assim como nos algoritmos K-NN e MLP. Foi instanciado o modelo 'tree.*DecisionTreeClassifier'*. Referência para biblioteca utilizada sklearn.tree.DecisionTreeClassifier.

Para treinar o algoritmo, foram usados dois conjuntos de dados: um de treinamento e outro de teste. Inicialmente, foram utilizados os valores de atributos padrões. Porém, para obter uma melhor precisão, foram testados diversos valores para os parâmetros, resultando em uma precisão aproximada entre 86% e 88%. Após realizar alguns treinamentos foi possível obter uma melhora:

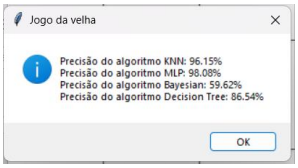


Figura 1 - Anterior ao Treinamento

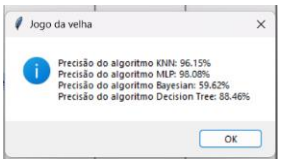


Figura 2 - Posterior ao treinamento

3.4 *Bayesiano*

Sites de referências: Site1 | Site2

Através de pesquisas foi possível compreender a **lógica** realizada através do algoritmo Bayesiano.

O algoritmo de classificação *Naive Bayes* utiliza as teorias de Thomas Bayes para realizar previsões em algoritmos de aprendizagem de máquina. O termo "*naive*" se refere à abordagem do algoritmo para analisar as características de uma base de dados, assumindo que as *features* são independentes entre si.

No entanto, esse algoritmo também pressupõe que todas as variáveis *features* têm a mesma importância para o resultado. Em casos em que essa suposição não se aplica, essa técnica pode não ser a melhor opção.

O algoritmo de *Bernoulli Naive Bayes* utiliza a Distribuição Multivariada de Bernoulli para analisar uma base de dados. Nessa distribuição, as *features* são compostas por valores binários, ou seja, podem ter apenas dois possíveis valores. Caso uma *feature*

não seja binária, o *BernoulliNB()* a transformará em uma *feature* binária, dependendo do valor do parâmetro *binarize*.

Dessa forma, o algoritmo utiliza a presença ou ausência desses valores binários para realizar suas predições. Essa técnica é eficaz em algumas situações, mas pode não ser a opção ideal em casos em que a importância relativa de cada *feature* pode variar.

Vantagens:

- Algoritmo fácil de implementar.
- É um classificador linear enquanto K-NN não é, tende a ser mais rápido quando aplicado a big data.

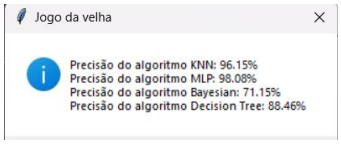
Desvantagens:

- O algoritmo não é adequado para o jogo da velha, ele desempenha melhor funções como exemplo validando o que e *span* ou não em uma rede de e-mails, ou em grande processamento de dados.

Motivo da Escolha: Durante a implementação, tanto a Árvore de Decisão quanto o algoritmo Bayesiano foram implementados sem um motivo específico para escolha. A decisão foi baseada na vontade de compreender e avaliar o desempenho de ambos os modelos.

A biblioteca *sklearn* também foi utilizada para aplicar o algoritmo de Bayes, assim como nos algoritmos K-NN, MLP e Árvore de Decisão. Foi instanciado o modelo '*sklearn.naive_bayes' import BernoulliNB.* Referência para biblioteca utilizada sklearn.naive_bayes.BernoulliNB.

A precisão encontrada foi de 71.15% sendo o pior dos algoritmos que implementamos.



Poderia ser utilizado outras classes de *Naive Bayes* porém a classe que obteve uma melhor acurácia foi *BernoulliNB*. As classes que poderiam ser utilizadas são *GaussianNB, BernoulliNB e MultinomialNB*.

Explicando as classes (baseado na documentação):

BernoulliNB este classificador é adequado para dados discretos. A diferença é que enquanto o *MultinomialNB* trabalha com contagem de ocorrências, o *BernoulliNB* é projetado para recursos binários/booleanos. No caso do problema do *Tic Tac Toe* a abordagem seria usar o *BernoulliNB*, pois os dados são binários (X, O ou vazio) e a distribuição de probabilidade é discreta.

O classificador ***MultinomialNB*** é adequado para classificação com características discretas (por exemplo, contagem de palavras para classificação de texto). A distribuição *multinomial* normalmente requer contagens de características inteiras. No entanto, na prática, contagens fracionárias como tf-idf também podem funcionar.

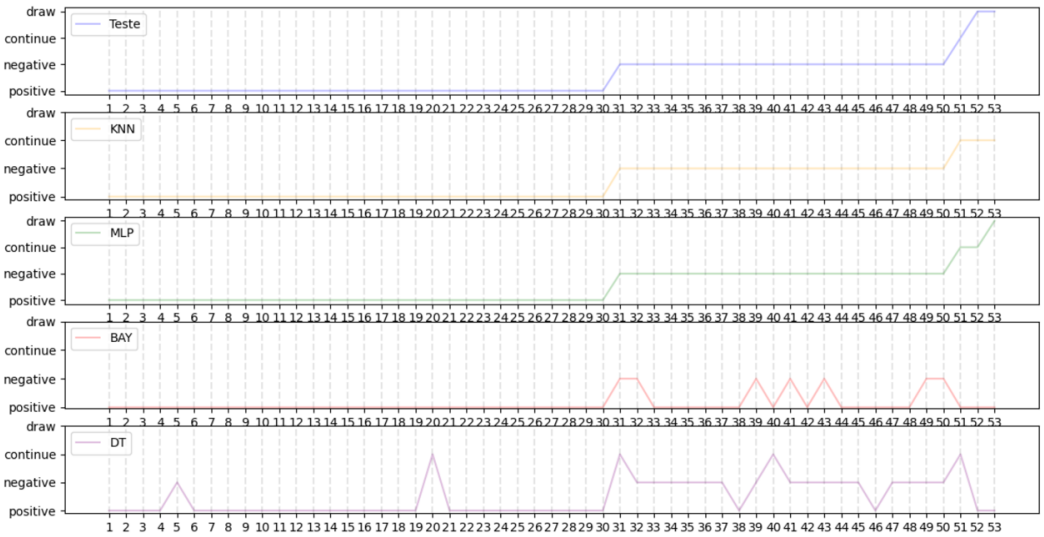
Gaussian Naive Bayes* (*GaussianNB) pode executar atualizações online para modelar parâmetros via *partial_fit*.

4. Conclusões

Este trabalho serviu de grande aprendizado para o grupo. A experiência de aplicar na prática os conhecimentos adquiridos em sala de aula e buscar novas informações foi muito importante para o aprendizado. Mesmo com apenas um dos integrantes possuindo conhecimento prévio em *Python*, todos conseguiram aprender de forma divertida e produtiva.

A aplicação dos algoritmos foi relativamente tranquila, graças às documentações e descrições disponíveis na web. No entanto, a dificuldade veio com o aprendizado da linguagem em contexto de IA e na busca pela melhoria da *accuracy* dos algoritmos no projeto.

Entre os algoritmos testados, o *MLP* se destacou com base nos gráficos e na *accuracy* alcançada. O treinamento da *MLP* foi feito com exemplos de jogos anteriores, onde a rede aprendeu a associar os estados do tabuleiro com as ações específicas que levaram a uma vitória ou empate.



A *MLP* se mostrou uma boa escolha para resolver o jogo da velha, pois foi capaz de aprender e generalizar padrões complexos dos dados de entrada, fornecendo respostas precisas. É importante ressaltar, no entanto, que o desempenho da rede depende do tamanho e w2qualidade do conjunto de dados de treinamento e da estratégia de treinamento utilizada.