Trabalho 1 - Inteligência Artificial

Isabella Leal Becker[*], Leticia Brasil Flores[†]
Escola Politécnica — PUCRS

16 de outubro de 2023

Resumo

Este relatório propõe uma solução para o jogo da velha, utilizando o conjunto de dados Tic-Tac-Toe como base. Foram empregados três algoritmos de inteligência artificial para uma análise comparativa. O documento inclui uma análise aprofundada do dataset, detalhando os ajustes necessários para sua utilização, explicações sobre o funcionamento de cada algoritmo e a apresentação de uma interface frontend que permite a interação e a realização de simulações.

1 Descrição do problema

Este trabalho envolve a criação de um sistema de IA para o Jogo da velha em um tabuleiro 3x3. O sistema foi desenvolvido para analisar o status do jogo a cada movimento, indicando se alguém ganhou. se houve empate ou se ainda há jogo. Muito além da funcionalidade básica de jogar, o principal objetivo foi lidar com o dataset disponibilizado, para que fosse possível utilizá-lo para finalidade de fazer o treinamento e predições utilizando diferentes algoritmos de Machine learning, sendo que para esse trabalho utilizamos os algoritmos K-NN, Árvore de decisão e Naive Bayes.

O problema foi dividido em quatro etapas, sendo a última dedicada a conclusões:

1. Pré-processamento dos dados: Etapa que envolveu a preparação dos dados, garantindo que estivessem prontos para serem usados nos algoritmos de machine learning.
2. Execução nos 3 algoritmos selecionados: Implementação e execução do dataset nos três algoritmos mencionados: K-NN, Árvore de Decisão e Naive Bayes.
3. Elaboração de um Front End: Criação de uma interface básica para permitir ao usuário interagir com o sistema.
4. Conclusão: Análise sobre os resultados obtidos com cada algoritmo, as dificuldades encontradas e o aprendizado obtido com o trabalho.

As próximas seções detalham de forma completa as ações realizadas em cada uma dessas etapas.

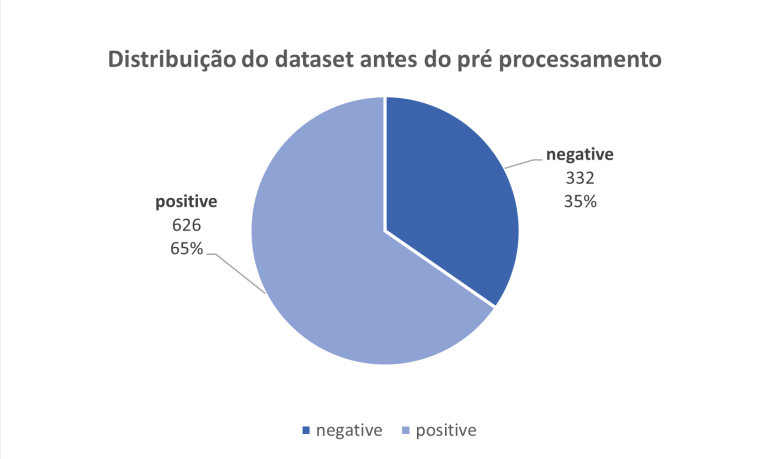
[*]isabella.becker@edu.pucrs.br
[†]leticia.flores@edu.pucrs.br

2 Sobre o dataset e seu pré-processamento

O dataset inicial do Jogo da velha que foi disponibilizado era composto por 10 colunas e 958 linhas. Representando uma matriz com as informações do tabuleiro e a última coluna que representa o resultado. Sendo 626 casos positivos 332 casos negativos.

A Figura 1 representa a distribuição do dataset em cada classe antes do pré-processamento. Tendo somente as duas classes positive e negative.

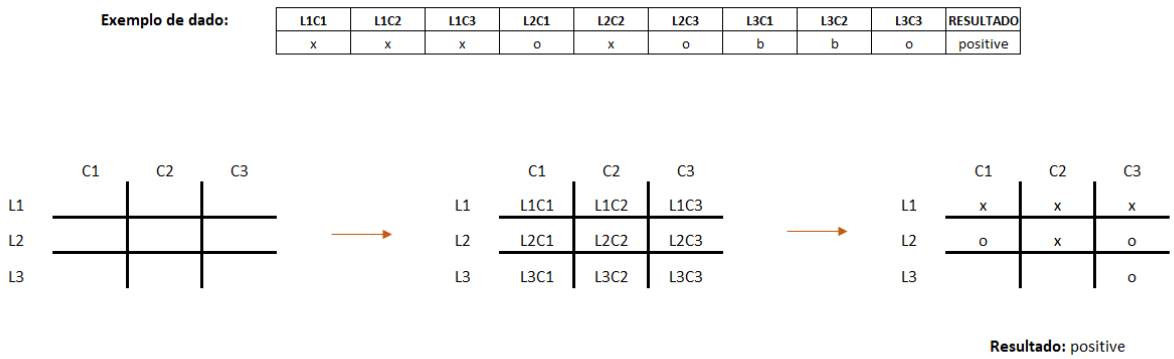
Figura 1:



Conforme analisado, o resultado positivo, significa que o jogador 'x' ganhou a partida, e o resultado negativo significa que o jogador 'o' ganhou, ou teve um empate ou ainda há jogo.

A Figura 2 foi elaborada para demonstrar a interpretação feita do dataset. Onde cada linha do dataset é considerada como uma matriz do jogo da velha, sendo os dados de entrada de L1C1 até L3C3 e a última coluna como o resultado do jogo.

Figura 2:



Dessa forma, para possibilitar que o algoritmo de Machine Learning faça previsões indicando se alguém ganhou, se houve empate ou se o jogo ainda está em andamento, é fundamental efetuar ajustes no dataset. Estes ajustes englobam a inclusão de dados adicionais, a definição de novas categorias, o balanceamento das informações e a transformação dos tipos de dados. Somente após essas adaptações, poderemos avançar para a aplicação dos algoritmos.

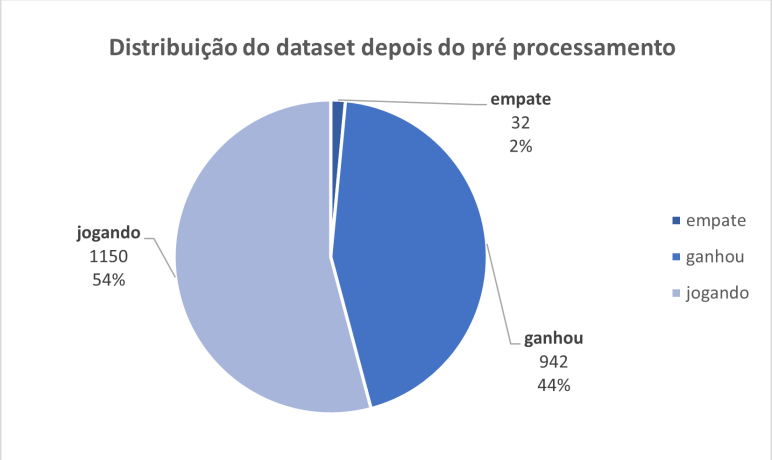
Devido à extensão do dataset e à natureza dos resultados do Jogo da Velha que ele representa, decidimos realizar a etapa de pré-processamento com o suporte de um programa desenvolvido em linguagem Java. Esse programa está disponível em diretório no GitHub:
https://github.com/IsabellaBeckerES/t1-ia-tic-tac-toe

Os principais ajustes que esse programa realiza no dataset são:

1. Identificar os casos em que alguém ganhou o jogo, seja jogador ’x’ ou ’o’, e casos de empate.
2. Haviam poucos casos de empate, então foi feita a criação de mais casos, substituindo ’x’ por ’o’. Dessa forma conseguimos dobrar os casos te empate.
3. Como a quantidade de empates seguia muito inferior ao das demais classes, tentamos criar empates utilizando o algoritmo que desenvolvemos em Java, mas ao excluir dados duplicados o número de empates sempre voltava para 32.
4. Não haviam casos de jogo em andamento então foi feito um algoritmo para criar casos de jogo em andamento, que basicamente pega os casos de jogo ganho e substitui de forma aleatória os ’x’ e ’o’ em ’b’ nos tabuleiros, depois verifica se esse tabuleiro gerado é válido e se não tem ganhadores, se passar nessas validações ele vai para o dataset.
5. A definição das três classes necessárias para resolver o problema proposto: Ganhou, Empate e Jogando.

   Após o pré-processamento, chegamos a um dataset que pode ter seus dados representado na Figura 3.

Figura 3:



Com isso, chegamos em um dataset contendo as classes necessárias para executar os algoritmos e obtivemos os dados que contemplam todas as situações possíveis para o jogo.

Um último ajuste necessário é a transformação dos tipos de dados, importante para mapear esses dados em um formato adequado para treinamento de algoritmos de aprendizado, esse passo foi realizado utilizando a ferramenta Google Colab. Assim as letras constantes no dataset foram mapeadas da seguinte forma: letras ’x’, ’o’ e ’b’ para 1, -1, 0 respectivamente, e os resultados ’empate’, ’ganhou’ e ’jogando’ para 0, 1, -1.

3 Soluções IA

Após a conclusão do pré-processamento, obtivemos um dataset contendo as informações essenciais para o problema em apresentado. Nessa fase de divisão do dataset em conjuntos de treino e teste, bem como na aplicação dos algoritmos, optamos por utilizar o Google Colab devido à facilidade oferecida pela biblioteca scikit-learn. Os passos estão disponíveis no seguinte link: Link Google Colab

Primeiramente foi feita a separação dos dados de entrada e dos rótulos de classes.

```
x = data.drop(columns=['RESULTADO'])
y = data['RESULTADO']
```

Dessa forma, x representa as características dos dados que o modelo usará para fazer previsões, e y representa as classes associadas a esses dados que o modelo deverá prever.

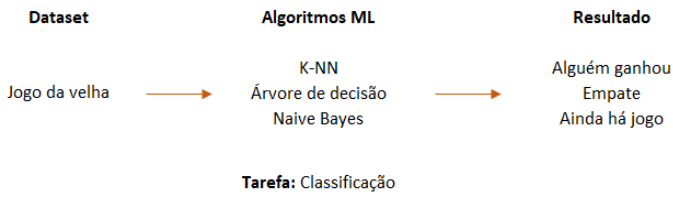
Feito isso a separação de dados de treino e teste foi elaborada da seguinte forma:

```
X_train , X_test , y_train , y_test = train_test_split(x , y , test_size=0.2 , random_state=42)
```

A função train_test_ split da biblioteca Pandas divide aleatoriamente os dados, o parâmetro test_size, indica que 20% dos dados serão de teste, e o parâmetro randon_state igual a 42 garante que a divisão produza os mesmos resultados cada vez que o código for executado.

De acordo com o tipo de problema que temos e com a natureza dos dados, o paradigma de aprendizado empregado aqui é o Supervisionado, onde a partir de exemplos rotulados, usando atributos de entrada se chega a uma saída correspondente. A tarefa (Figura 4) a ser realizada pelos algoritmos é a de Classificação, que é o processo de atribuir rótulos automáticamente.

Figura 4:



Nessa seção descrevemos o funcionamento de cada um desses algoritmo, os parâmetros necessários para execução e a acurácia obtida com os dados de teste.

3.1 KNN

O K-NN é um dos algoritmos mais simples para aprendizado de máquina, tem como princípio o método de aprendizado de vizinhos mais próximos, que tem como objetivo encontrar amostras de treinamento mais próximas em distância a um ponto e prever o rótulo a partir delas.

3.1.1 Como funciona

Para prever a classe de um novo dado o K-NN encontra os K exemplos mais próximos desse novo dado nos dados de treino, verifica qual a classe mais comum entre esses dados mais próximos e atribui essa classe ao novo dado. O valor de K passado como parâmetro no algoritmo representa o número

de vizinhos mais próximos que serão considerados ao fazer uma previsão. Valores de K mais baixos tornam o modelo mais sensível a variações dos dados.

Pseudocódigo do K-NN:

```
1  Inicialização
2      Dados de conjunto de treino e de teste
3      Valor de K
4
5  Para cada nova amostra do conjunto de teste faça:
6      Calcula a distância para todas as amostras do conjunto de treino
7      Determina o conjunto das K amostras mais próximas
8      Determina o rótulo mais representativo entre os K vizinhos
9
10 Retorna o conjunto de testes rotulados
```

3.1.2 Detalhes da implementação

Para utilizar o K-NN foi feita a importação da classe KNeighborsClassifier da biblioteca sklearn.neighbors.

```
1  from sklearn.neighbors import KNeighborsClassifier
2  from sklearn.metrics import accuracy_score
3
4  knn_clf = KNeighborsClassifier(n_neighbors=2)
5
6  knn_clf.fit(X_train , y_train)
7
8  knn_predicao = knn_clf.predict(X_test)
9
10 knn_accuracia = accuracy_score(y_test , Knn_predicao)
```

Criamos uma instância da classe KNeighborsClassifier (linha 4) chamada knn_clf e utilizamos o método fit() para calcular e organizar os dados para que possa ser feitas as classificações. Em seguida, linha 8, utilizamos o método predict() para aplicar o modelo knn_clf nos dados de teste e retornar as previsões que serão armazenadas em knn_predicao.

A seleção do valor de k foi realizada por meio de uma análise exploratória. Ao utilizar k = 3, conseguimos alcançar uma acurácia superior nos dados de teste. No entanto, durante a simulação do jogo, notamos que o algoritmo foi mais preciso com um valor de k = 2.

Após realizar esses passos utilizamos o método accuracy_score() para verificar a taxa de acerto em comparação com os dados de teste (linha 10), que são os resultados esperado (y_test) e os rótulos previstos pelo modelo, que estão em knn_predicao. O resultado obtido foi de uma acurácia de 0.74, o que significa que em 74% dos casos o modelo fez previsões corretas.

3.2 Árvore de decisão

O Algoritmo Árvore de decisão possui a característica de criar uma estrutura de árvore onde cada nó representa um atributo e os ramos representam uma decisão possível, até chegar a um nó folha que representa a classe prevista.

3.2.1 Como funciona

A construção da árvore é feita recursivamente, dividindo os dados em subconjuntos menores e criando nós para cada subconjunto.

Pseudocódigo de uma Árvore de decisão, exemplo baseado no livro de referência da disciplina "Inteligência Artificial - Uma Abordagem de Aprendizado de Máquina":

```
Inicialização
    Dados de conjunto de treino

Função ConstroiArvore(dados de treino):
    Se critério de parada éatingido:
        Retorna um nó folha que representa a classe
    Encontra o melhor atributo para dividir os dados
    Divide os dados em subconjuntos com base no atributo escolhido

    Para cada subconjunto faça:
        Crie um nó na árvore
        Chama recursivamente ConstroiArvore(subconjunto)

Retorna árvore contendo um nó de decisão
```

A entrada da função ConstroiArvore são os dados de treino e a função é chamada recursivamente recebendo como parâmetro os subconjuntos, resultados das divisões feitas pelo algoritmo com base no critério de parada.

3.2.2 Detalhes da implementação

Para utilizar o algoritmo árvore decisão foi necessário fazer a importação da classe DecisionTreeClassifier da biblioteca sklearn.tree.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

decisionTree_clf = DecisionTreeClassifier()

decisionTree_clf.fit(X_train , y_train)

decisionTree_predictions = decisionTree_clf.predict(X_test)

decisionTree_accuracy = accuracy_score(y_test , decisionTree_predictions)
```

Semelhante aos passos feitos no algoritmo anterior, primeiramente foi criado uma instância da classe DecisionTreeClassifier, que nomeamos como decisionTree_clf e utilizamos o método fit() passando os dados de treino como parâmetro. Em seguida, utilizamos a função predict() para aplicar o modelo decisionTree_clf nos dados de teste, armazenando o resultado em decisionTree_predictions. Utilizando o método accuracy_score() passando como parâmetro os rótulos esperados (y_test) e o retorno da predição (decisionTree_predictions), chegamos em uma acurácia de 0.92, ou 92% de previsoes corretas.

3.3 Naive Bayes

O Naive Bayes é um algoritmo de classificação que se baseia no Teorema de Bayes, este teorema é utilizado para calcular a probabilidade de um evento acontecer. O nome "Naive"(ingênuo) dado ao algoritmo vem da sua característica de assumir que os atributos dos dados são independentes entre si e que todos têm igual importância para gerar o resultado.

3.3.1 Como funciona

Para realizar a classificação Naive Bayes primeiro cria uma tabela de frequência, quando utilizamos o algoritmo para descobrir qual sua classe é utilizando essa tabela e dados de entrada aplicando o Teorema de Bayes para descobrir a que classe o dado de entrada pertence.

```
Inicialização
    Dados de conjunto de treino

Função criaTabela(dados de treino):
    Cria um contador para cada valor de atributo por classe

    Para cada linha do dado de treino faça:
        Incrementa contadores
    Para cada contador faça:
        Calcula a frequência relativa de cada contador

Retorna tabela de frequência
```

3.3.2 Detalhes da implementação

Para utilizar o algoritmo Naive Bayes importamos a classe GaussianNB da biblioteca sklearn.naive_bayes.

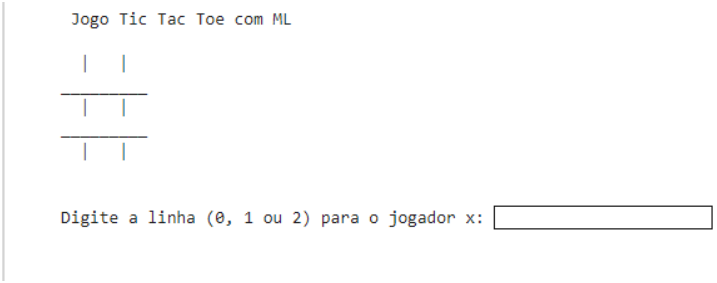
```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

bayes_clf = GaussianNB()

bayes_clf.fit(X_train , y_train)

bayes_predictions = bayes_clf.predict(X_test)

bayes_accuracy = accuracy_score(y_test , bayes_predictions)
```

Criamos uma instância da classe GaussianNB, chamada de bayes_clf, sendo esse o nosso classificador. Após isso, chamamos o método fit() passando os dados de treino como parâmetro. Em seguida, utilizamos a função predict() para aplicar o modelo bayes_clf nos dados de teste, armazenando o resultado em bayes_predictions. Utilizando o método accuracy_score() passando como parâmetro os rótulos esperados (y_test) e o retorno da predição (bayes_predictions), chegamos em uma acurácia de 0.90, ou 90% de previsões corretas.

4 Front End

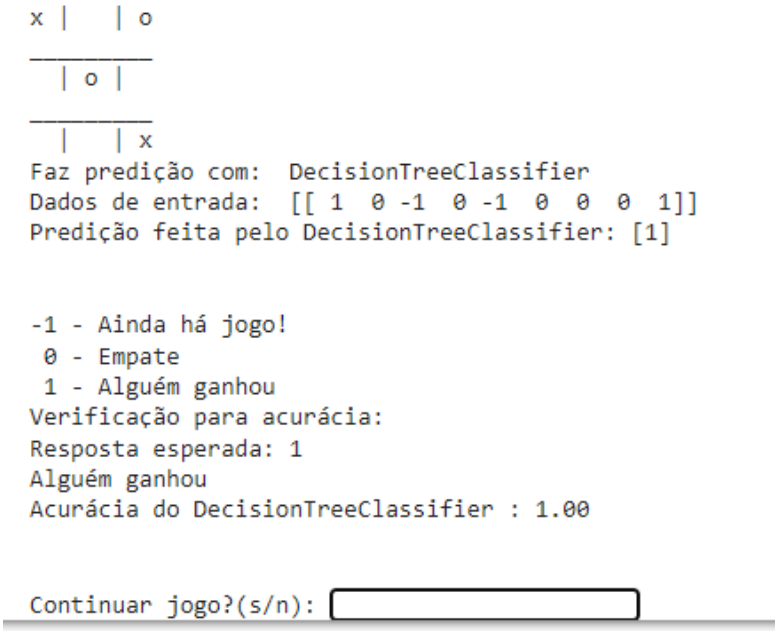
A interface para interação com o usuário também foi escrita em Python e pode ser executada na página do Google Colab. O usuário deverá executar os passos para treinar o modelo de cada algoritmo e em seguida poderá executar o jogo. A tela do jogo está exemplificada na Figura 5.

Figura 5:



A cada rodada o programa irá verificar a predição feita pelo algoritmo e solicitar ao usuário que informe qual o resultado esperado, para que seja possível calcular a acurácia da resposta.

Figura 6:



Após receber o resultado esperado, a acurácia será calculada levando em conta todas as jogadas feitas até agora. Se o resultado for 1 (indicando que alguém ganhou) ou 0 (indicando empate), será perguntado se deseja continuar o jogo ou não.

5 Conclusão

Relembrando, a acurácia de cada algoritmo na etapa de treino e teste foi de:
- K-NN = 0.74
- Árvore de decisão = 0.92
- Naive Bayes = 0.90

Nesta análise, comparamos os resultados das execuções do jogo para avaliar o desempenho dos diferentes algoritmos e as acurácias alcançadas em cada jogada. A Tabela apresentada na Figura 7 inclui três cenários de jogo: uma vitória para "x", outra para "o"e um empate.

Conduzimos o teste repetindo as mesmas jogadas na mesma ordem para cada algoritmo, acompanhando a precisão a cada jogada.

Figura 7:

	C0	C1	C2
L0	x		o
L1	x	o	
L2	x	o	x

Jogadas para x ganha			Resultado esperado	Acertou?		
				Knn	Árvore de decisão	Naive Bayes
1º	L0C0	x	-1	sim	sim	sim
2º	L1C1	o	-1	sim	sim	não
3º	L2C2	x	-1	sim	sim	não
4º	L0C2	o	-1	sim	alguém ganhou	alguém ganhou
5º	L2C0	x	1	não	-	-
6º	L2C1	o	1	não	-	-
7º	L1C0	x	1	sim	-	-
Acurácia:				0.71	1.0	0.50

	C0	C1	C2
L0	x		o
L1		x	o
L2	x		o

Jogadas para o ganha			Resultado esperado	Acertou?		
				Knn	Árvore de decisão	Naive Bayes
1º	L0C0	x	-1	sim	sim	sim
2º	L0C2	o	-1	sim	sim	sim
3º	L1C1	x	-1	sim	não	não
4º	L2C2	o	-1	sim	não	não
5º	L2C0	x	1	sim	alguém ganhou	alguém ganhou
6º	L1C2	o	1	-	-	-
Acurácia:				1.0	0.75	0.60

	C0	C1	C2
L0	x	o	x
L1	o	o	x
L2	x	x	o

Jogadas para empate			Resultado esperado	Acertou?		
				Knn	Árvore de decisão	Naive Bayes
1º	L0C0	x	-1	sim	sim	sim
2º	L1C0	o	-1	sim	sim	não
3º	L0C2	x	-1	sim	sim	não
4º	L0C1	o	-1	sim	sim	não
5º	L2C1	x	-1	sim	sim	não
6º	L2C2	o	-1	sim	sim	não
7º	L1C2	x	0	sim	não	não
8º	L1C1	o	0	-	sim	não
9º	L2C0	x	0	-	-	não
Acurácia:				1.0	0.88	0.11

Apesar de apresentar uma acurácia inferior durante as fases de treino e teste, o algoritmo K-NN demonstrou maior precisão durante o jogo. Entretanto, no cenário em que "x"vence, ele não conseguiu antecipar o resultado como os outros modelos. No entanto, na simulação em que "o"ganha ou ocorre um empate, o K-NN alcançou uma precisão de 100%.

O algoritmo de Árvore de Decisão mostrou resultados satisfatórios quando "x"ganha, prevendo corretamente o resultado já na quarta jogada, indicando "Alguém ganhou", seja "x"ou "o". Entretanto, no cenário em que "o"ganha, o desempenho não foi tão consistente, resultando em algumas respostas incorretas e alcançando uma acurácia de 75%. No caso de empate, o algoritmo teve um desempenho melhor, acertando 88% das jogadas.

O algoritmo Naive Bayes teve o pior desempenho entre os três avaliados. Embora tenha alcançado uma acurácia de 90% nos dados de treino e teste, teve um desempenho muito ruim ao prever os resultados das jogadas

As principais dificuldades encontradas no desenvolvimento do trabalho foram:

- Pré-processamento de Dados: A etapa de pré-processamento foi complexa, envolvendo várias modificações até chegarmos a um dataset adequado para resolver o problema. Isso levanta a questão de que um dataset maior poderia demandar ainda mais dedicação nessa etapa.
- Ajuste dos Parâmetros do Algoritmo: O ajuste dos parâmetros do algoritmo foi uma tarefa delicada. Por exemplo, no caso do K-NN, foi necessário sacrificar um pouco da acurácia nos testes para obter resultados mais precisos nas jogadas. Este ponto ainda requer uma análise mais profunda para entender completamente os motivos por trás dessas diferença.

O trabalho proporcionou aprendizados valiosos, incluindo a compreensão da importância de um dataset com dados significativos, além da realização de um pré-processamento eficaz para alcançar resultados melhores. Também adquirimos conhecimento sobre a implementação de diversos algoritmos, nos familiarizamos com a plataforma Google Colab e aplicamos os conceitos aprendidos em aula na prática.