Relatório: Resolvedor de Cadeias de Markov para Probabilidades de Jogos de Futebol

1. Introdução

Este relatório detalha a implementação de um resolvedor de cadeias de Markov utilizando a linguagem de programação Julia, com foco na aplicação para prever probabilidades de resultados em jogos de futebol. O projeto aborda a modelagem da dinâmica de um jogo como um processo estocástico, onde a probabilidade de um evento futuro (como a ocorrência de um gol ou a mudança no placar) depende apenas do estado atual do jogo, e não de como esse estado foi alcançado. O objetivo principal é demonstrar a aplicabilidade das cadeias de Markov em um cenário prático, fornecendo tanto uma solução manual para validação quanto uma ferramenta computacional para cálculos mais complexos.

2. Situação Problema: Previsão de Resultados em Jogos de Futebol

A previsão de resultados em jogos de futebol é um desafio complexo devido à natureza dinâmica e imprevisível do esporte. Fatores como o desempenho das equipes, táticas, eventos inesperados (lesões, cartões) e a aleatoriedade inerente ao jogo tornam a previsão um campo fértil para a aplicação de modelos probabilísticos. As cadeias de Markov oferecem uma estrutura matemática robusta para modelar sistemas que evoluem através de uma sequência de estados, onde as transições entre esses estados são governadas por probabilidades. No contexto do futebol, podemos considerar os estados como a diferença de gols entre os times ou o placar exato em um determinado momento da partida.

2.1. Modelagem com Cadeias de Markov

Para este projeto, a situação problema foi modelada considerando os estados da cadeia de Markov com base na diferença de gols entre os dois times. Esta abordagem simplifica

a complexidade do placar exato, tornando o modelo mais gerenciável para demonstração e validação manual. Os estados definidos são:

- **E-2:** Time B está ganhando por 2 ou mais gols de diferença.
- **E-1:** Time B está ganhando por 1 gol de diferença.
- **E0:** Jogo empatado.
- **E1:** Time A está ganhando por 1 gol de diferença.
- **E2:** Time A está ganhando por 2 ou mais gols de diferença.

A transição entre esses estados ocorre a cada gol marcado. As probabilidades de transição entre os estados são representadas por uma matriz de transição, onde cada elemento `P(i, j)` denota a probabilidade de transitar do estado `i` para o estado `j`. Para fins de ilustração e validação, foram utilizadas probabilidades hipotéticas, mas em uma aplicação real, estas seriam derivadas de dados históricos de jogos. O estado inicial do jogo é sempre o empate (E0), com um vetor de estado inicial `π₀ = [0, 0, 1, 0, 0]`.

2.2. Solução Aplicada

A solução aplicada envolve o cálculo das probabilidades de estado da cadeia de Markov após um determinado número de transições (gols). A probabilidade de estar em cada estado após `n` transições é dada pela multiplicação do vetor de estado inicial pela matriz de transição elevada à potência `n` (`π_n = π₀ * P^n`). A partir dessas probabilidades de estado, é possível inferir as probabilidades de resultado final do jogo (vitória do Time A, empate, vitória do Time B) somando as probabilidades dos estados correspondentes a cada resultado.

Para a validação, foram realizados cálculos manuais para 1 e 2 transições, demonstrando o passo a passo da evolução das probabilidades. Esses cálculos manuais servem como base para verificar a correção da implementação computacional em Julia.

3. Código-Fonte da Aplicação

O resolvedor de cadeias de Markov foi implementado em Julia, utilizando a biblioteca `LinearAlgebra` para operações com matrizes. O código é estruturado em uma função principal (`main`) que interage com o usuário e uma função auxiliar (`calculate_markov_probabilities`) que realiza os cálculos da cadeia de Markov.

3.1. `markov_solver.jl`

```
using LinearAlgebra

"""
Calcula as probabilidades de estado de uma cadeia de Markov após
n passos.

Argumentos:
  P: Matriz de transição (NxN, onde N é o número de estados).
  pi0: Vetor de estado inicial (1xN).
  n: Número de passos (transições).

Retorna:
  Vetor de probabilidades de estado após n passos.
"""
function calculate_markov_probabilities(P::Matrix{Float64},
pi0::Vector{Float64}, n::Int)
    if size(P, 1) != size(P, 2)
        error("A matriz de transição (P) deve ser quadrada.")
    end
    if length(pi0) != size(P, 1)
        error("O vetor de estado inicial (pi0) deve ter o mesmo
número de elementos que o número de estados na matriz P.")
    end
    if n < 0
        error("O número de passos (n) deve ser não negativo.")
    end

    # Calcula P^n
    P_n = P^n

    # Multiplica o vetor de estado inicial por P^n
    pi_n = pi0' * P_n

    return vec(pi_n)
end

function main()

println("Resolvedor de Cadeias de Markov para Probabilidades de
Jogos de Futebol")

println("-----------------------------------------------------------------

    # Definindo a matriz de transição P (exemplo da solução
manual)

# Estados: E-2 (Time B +2), E-1 (Time B +1), E0 (Empate), E1
(Time A +1), E2 (Time A +2)
    P = [
```

```
        0.90  0.05  0.05  0.00  0.00; # De E-2
        0.05  0.80  0.10  0.05  0.00; # De E-1
        0.00  0.10  0.80  0.10  0.00; # De E0
        0.00  0.05  0.10  0.80  0.05; # De E1
        0.00  0.00  0.05  0.05  0.90  # De E2
    ]

    # Definindo o vetor de estado inicial pi0 (jogo começa em
empate)
    pi0 = [0.0, 0.0, 1.0, 0.0, 0.0]

    println("\nMatriz de Transição (P):")
    display(P)
    println("\nVetor de Estado Inicial (pi0): ", pi0)

    while true

print("\nDigite o número de transições (n) para calcular as
probabilidades (ou 'sair' para encerrar): ")
        input = readline()

        if lowercase(input) == "sair"
            println("Encerrando o programa.")
            break
        end

        try
            n = parse(Int, input)
            if n < 0
                println("Por favor, digite um número não
negativo.")
                continue
            end

            pi_n = calculate_markov_probabilities(P, pi0, n)

            println("\nProbabilidades de estado após ", n, "
transições:")
            println("  E-2 (Time B +2): ", round(pi_n[1],
digits=4))
            println("  E-1 (Time B +1): ", round(pi_n[2],
digits=4))
            println("  E0 (Empate):     ", round(pi_n[3],
digits=4))
            println("  E1 (Time A +1):  ", round(pi_n[4],
digits=4))
            println("  E2 (Time A +2):  ", round(pi_n[5],
digits=4))

            # Calculando as probabilidades de resultado final
            prob_time_a_vence = round(pi_n[4] + pi_n[5],
digits=4)
```

```
            prob_empate = round(pi_n[3], digits=4)
            prob_time_b_vence = round(pi_n[1] + pi_n[2],
digits=4)

            println("\nProbabilidades de Resultado Final após
", n, " transições:")
            println("  Vitória Time A: ", prob_time_a_vence)
            println("  Empate:         ", prob_empate)
            println("  Vitória Time B: ", prob_time_b_vence)

        catch e
            println("Entrada inválida. Por favor, digite um
número inteiro.")
        end
    end
end

# Executa a função principal apenas quando o script é executado
diretamente
if abspath(PROGRAM_FILE) == @__FILE__
    main()
end
```

4. Instruções de Instalação da Aplicação

Para instalar e executar a aplicação em Julia, siga os passos abaixo:

4.1. Instalação do Julia

1. **Download do Binário:** Baixe a versão mais recente do Julia para Linux (x64) do site oficial. Você pode usar o comando `wget` no terminal: `bash wget https://julialang-s3.julialang.org/bin/linux/x64/1.11/julia-1.11.5-linux-x86_64.tar.gz` (Verifique a versão mais recente no site oficial do Julia e ajuste o link, se necessário).

2. **Extrair o Arquivo:** Descompacte o arquivo baixado: `bash tar -zxvf julia-1.11.5-linux-x86_64.tar.gz` Isso criará uma pasta `julia-1.11.5` (ou similar) no seu diretório atual.

3. **Adicionar Julia ao PATH (Opcional, mas recomendado):** Para executar o Julia de qualquer diretório, adicione o diretório `bin` do Julia ao seu PATH. Você pode criar um link simbólico: `bash sudo ln -s /caminho/para/sua/pasta/julia-1.11.5/bin/julia /usr/local/bin/julia` Substitua `/caminho/para/sua/pasta/julia-1.11.5` pelo caminho real onde você extraiu o Julia.

4. **Verificar a Instalação:** Abra um novo terminal e digite: `bash julia -v` Você deverá ver a versão do Julia instalada.

4.2. Execução da Aplicação

1. **Navegar até o Diretório:** Abra um terminal e navegue até o diretório onde você salvou o arquivo `markov_solver.jl`. `bash cd /caminho/para/o/diretorio/do/projeto`
2. **Executar o Script:** Execute o script Julia usando o comando: `bash julia markov_solver.jl`
3. **Interagir com a Aplicação:** O programa solicitará que você digite o número de transições para calcular as probabilidades. Digite um número inteiro e pressione Enter. Para sair, digite `sair`.

5. Validação e Testes

Para garantir a correção da implementação, foram realizados testes unitários utilizando o módulo `Test` do Julia. O script `test_markov_solver.jl` verifica a função `calculate_markov_probabilities` com diferentes cenários, incluindo:

- Cálculo para `n=0`, `n=1` e `n=2` transições, comparando com os resultados da solução manual.
- Verificação de entradas inválidas (matriz não quadrada, vetor inicial de tamanho incorreto, número de passos negativo).
- Verificação da soma das probabilidades resultantes, que deve ser aproximadamente 1.

5.1. `test_markov_solver.jl`

```
using Test
include("markov_solver.jl")

@testset "Markov Chain Solver Tests" begin
    # Teste 1: Matriz de transição e vetor inicial válidos, n =
0
    P1 = [
        0.90  0.05  0.05  0.00  0.00;
        0.05  0.80  0.10  0.05  0.00;
        0.00  0.10  0.80  0.10  0.00; # Corrigido: 0.70 para
0.80
        0.00  0.05  0.10  0.80  0.05;
        0.00  0.00  0.05  0.05  0.90
```

```
    ]
    pi0_1 = [0.0, 0.0, 1.0, 0.0, 0.0]
    @test calculate_markov_probabilities(P1, pi0_1, 0) ≈ pi0_1

    # Teste 2: n = 1 (comparar com cálculo manual)
    expected_pi1 = [0.0, 0.1, 0.8, 0.1, 0.0] # Corrigido: 0.7
para 0.8
    @test calculate_markov_probabilities(P1, pi0_1, 1) ≈
expected_pi1

    # Teste 3: n = 2 (comparar com cálculo manual)
    expected_pi2_corrected = [0.005, 0.165, 0.66, 0.165, 0.005]
    @test calculate_markov_probabilities(P1, pi0_1, 2) ≈
expected_pi2_corrected


# Teste 4: Matriz de transição com soma das linhas diferente de
1 (deve ser tratada pelo usuário, mas a função deve funcionar)
    P2 = [
        0.5  0.5;
        0.6  0.4
    ]
    pi0_2 = [1.0, 0.0]
    @test calculate_markov_probabilities(P2, pi0_2, 1) ≈ [0.5,
0.5]

    # Teste 5: Matriz de transição não quadrada (deve lançar
erro)
    P_non_square = [
        0.5  0.5  0.0;
        0.6  0.4  0.0
    ]
    pi0_non_square = [1.0, 0.0]
    @test_throws ErrorException("A matriz de transição (P) deve
ser quadrada.") calculate_markov_probabilities(P_non_square,
pi0_non_square, 1)

    # Teste 6: Vetor inicial com tamanho diferente da matriz
(deve lançar erro)
    P_valid = [
        0.5  0.5;
        0.6  0.4
    ]
    pi0_wrong_size = [1.0, 0.0, 0.0]
    @test_throws
ErrorException("O vetor de estado inicial (pi0) deve ter o mesmo
número de elementos que o número de estados na matriz P.")
calculate_markov_probabilities(P_valid, pi0_wrong_size, 1)

    # Teste 7: n negativo (deve lançar erro)
    @test_throws
ErrorException("O número de passos (n) deve ser não negativo.")
```

```
calculate_markov_probabilities(P1, pi0_1, -1)

    # Teste 8: n grande para verificar convergência (opcional,
dependendo do modelo)
    pi_large_n = calculate_markov_probabilities(P1, pi0_1, 100)
    @test sum(pi_large_n) ≈ 1.0
    @test all(x -> x >= 0, pi_large_n)
end
```

6. Conclusão

Este projeto demonstrou a aplicação de cadeias de Markov para a previsão de probabilidades de resultados em jogos de futebol. Através da modelagem dos estados com base na diferença de gols e da utilização de uma matriz de transição, foi possível calcular a evolução das probabilidades ao longo do tempo. A implementação em Julia forneceu uma ferramenta eficiente para realizar esses cálculos, e a validação com a solução manual e testes unitários confirmou a correção do modelo e do código. Embora o modelo apresentado seja simplificado, ele serve como uma base sólida para futuras expansões, como a incorporação de dados reais de jogos, a consideração de mais estados ou a utilização de cadeias de Markov em tempo contínuo para uma representação mais precisa da dinâmica do jogo.