UNIVERSIDADE FEDERAL DO PARANÁ
MARIA GABRIELA DO AMARAL SCAPIN

ASPECTOS COMPUTACIONAIS DA MODELAGEM DE INTERAÇÃO DE CARACTERÍSTICAS MORFOLÓGICAS ENTRE PLANTAS E POLINIZADORES

CURITIBA
2024

UNIVERSIDADE FEDERAL DO PARANÁ
MARIA GABRIELA DO AMARAL SCAPIN

ASPECTOS COMPUTACIONAIS DA MODELAGEM DE INTERAÇÃO DE CARACTERÍSTICAS MORFOLÓGICAS ENTRE PLANTAS E POLINIZADORES

Monografia apresentada como requisito parcial à conclusão do curso de graduação em Matemática Industrial pela Universidade Federal do Paraná.
Orientador: Prof. Dr. Lucas Garcia Pedroso.

CURITIBA
2024

Resumo

A relação entre as características ou traços das espécies e suas interações em redes ecológicas tem sido amplamente discutida, especialmente no contexto de redes de polinização. A hipótese de que interações são mais prováveis quando os traços das espécies se complementam – por exemplo, quando a proporção entre a língua de um beija-flor e o formato de uma flor coincidem – tem motivado diversas investigações. No entanto, estimativas empíricas sobre a importância do pareamento de características variam significativamente entre diferentes tipos de redes ecológicas. Este trabalho explora essa variação com modelos de Aprendizagem de Máquina mais flexíveis, incluindo Florestas Aleatórias, Redes Neurais Profundas, Redes Neurais Convolucionais, *Naive Bayes* e KNN. Utilizando dados simulados e reais, investigamos a capacidade preditiva desses modelos para identificar interações planta-polinizador com base em traços.

Palavras-chave: *Modelo preditivo, Polinização, Aprendizagem de Máquina, Redes de interação.*

Abstract

The relationship between species traits and their interactions in ecological networks has been widely discussed, especially in the context of pollination networks. The hypothesis that interactions are more likely when species traits complement each other, for example, when the proportion between a hummingbird's tongue and the shape of a flower aligns, it has motivated various investigations. However, empirical estimates of the importance of trait-matching vary significantly across different types of ecological networks. This study explores this variation with more flexible Machine Learning (ML) models, including Random Forest, Gradient Boosting, Deep Neural Networks, Convolutional Neural Networks, Naive Bayes, and K-Nearest-Neighbors. Using simulated and real data, we investigate the predictive capacity of these models to identify plant-pollinator interactions based on traits.

Keywords: *Predictive model, Pollination, Machine Learning, Interaction networks.*

Sumário

Introdução

Este trabalho tem como objetivo replicar e expandir o estudo apresentado no artigo *"Machine Learning Algorithms to Infer Trait-Matching and Predict Species Interactions in Ecological Networks"* (PICHLER et al., 2020). Nesse estudo, os autores simulam diferentes características morfológicas de plantas e polinizadores para identificar os principais traços responsáveis por suas interações. Por exemplo, flores com corolas mais longas podem estar associadas à morfologia do bico e da língua de beija-flores, facilitando o acesso ao néctar.

Figura 1: Beija-flor realizando polinização. Fonte: Sergio Gregorio da Silva/Acervo Pessoal

Esse fenômeno, denominado *trait-matching* (ou correspondência de características), sugere que interações entre espécies ocorrem principalmente quando há compatibilidade física ou funcional entre suas características. Essa correspondência é um pré-requisito do conceito mais amplo de síndromes de polinização, que postula a coevolução de plantas e seus polinizadores ao longo do tempo (FAEGRI; PIJL, 1979; FENSTER et al., 2004; OLLERTON et al., 2009; ROSAS-GUERRERO et al., 2014).

A importância ecológica desse tipo de estudo é evidente, sobretudo em um cenário de rápida mudança climática e degradação ambiental. A extinção de uma espécie pode provocar um efeito cascata, afetando várias outras espécies interligadas na rede ecológica, o que compromete o equilíbrio do ecossistema (POTTS et al., 2010). Além disso, os polinizadores, essenciais para a reprodução de muitas plantas, estão sob crescente ameaça, seja pelas alterações climáticas, seja pelo uso intensivo de pesticidas em sistemas agrícolas (VANBERGEN et al., 2013; KLEIN et al., 2007). Essas pressões podem resultar na perda de serviços ecossistêmicos cruciais, como a polinização, o que acentua a urgência em compreender e proteger essas interações ecológicas. Com isso, é possível desenvolver

políticas de conservação mais eficazes, voltadas para a preservação de habitats e a redução do risco de extinção em massa.

O uso de ferramentas computacionais, como os modelos de aprendizagem de máquina, tem se mostrado fundamental na identificação e mensuração dessas interações complexas (SCHLEDER; FAZZIO, 2021). As técnicas de aprendizado de máquina permitem a análise de grandes quantidades de dados, ajudando a prever padrões de interações baseados nas características das espécies. No presente estudo, utilizaremos uma variedade de modelos de aprendizagem de máquina para prever as correspondências de características entre plantas e polinizadores em cenários simulados. Após a fase de simulação, validaremos os resultados utilizando dados empíricos de um banco de dados mundial de interações entre beija-flores e plantas.

Diferente do estudo original, que utilizou a linguagem `R`, nossa implementação será feita em `Python`, aproveitando a vasta gama de bibliotecas otimizadas para aprendizagem de máquina disponíveis nessa linguagem. A escolha de `Python` visa melhorar a eficiência computacional e aprimorar os resultados obtidos na replicação do estudo. Utilizaremos uma diversidade de algoritmos, incluindo *K-Nearest Neighbors* (KNN), *Naive Bayes*, Florestas Aleatórias, *Gradient Boosting* , Redes Neurais Profundas e Redes Neurais Convolucionais para comparar o desempenho dos diferentes modelos.

Capítulo 1

Simulação

Este capítulo apresenta o desenvolvimento de uma simulação computacional baseada na replicação de um estudo que utiliza técnicas de aprendizagem de máquina para analisar as interações entre características morfológicas de plantas e polinizadores (PICHLER et al., 2020). Inicialmente implementado em `R` pelos autores do estudo, o código `TraitMatching` foi adaptado e replicado em `Python`, com foco nas três funções essenciais desta biblioteca para a modelagem. A tradução priorizou a funcionalidade e a eficiência, garantindo a consistência entre as abordagens das duas linguagens.

Para isso, foram desenvolvidas e otimizadas as funções mencionadas. A primeira função é destinada à geração de características específicas das espécies, abrangendo tanto atributos categóricos, como a cor das pétalas das plantas, quanto atributos numéricos, como o tamanho corporal dos insetos polinizadores. Adicionalmente, a segunda função foi criada para simular as interações ecológicas entre essas espécies. Por fim, a terceira função cria uma comunidade de espécies simuladas que possibilita a obtenção de um conjunto abrangente de dados, consolidando uma tabela de interações que serve como base para a aplicação de modelos preditivos de aprendizagem de máquina no próximo capítulo.

1.1 Função `createSpecies`

A função `createSpecies` foi desenvolvida para simular o processo de criação de espécies, considerando seus traços, abundância e variação genética. Essa função, originalmente concebida em `R` e posteriormente adaptada para `Python`, desempenha um papel crucial na modelagem ecológica. Ela permite gerar dados sobre populações de espécies, levando em consideração aspectos como especialização, interações ecológicas e características específicas, como comportamento alimentar, habitat ou preferências reprodutivas.

1.1.1 Conceito de Espécie e Especialização

Na biologia, uma espécie é definida como um grupo de organismos que podem gerar descendentes férteis e que estão reprodutivamente isolados de outros grupos. Desse modo, a função `createSpecies` leva em consideração dois grupos de espécies, denotados por A e B, que podem representar diferentes tipos de organismos dentro de um ecossistema.

Esses dois grupos podem ter características distintas que influenciam suas interações no ambiente.

A especialização de uma espécie refere-se à adaptação a um nicho ecológico específico, o que pode ser modelado pelo parâmetro `specialist`, permitindo criar espécies altamente especializadas ou mais generalistas. Nesse sentido, este conceito é importante para o nosso trabalho, pois duas espécies altamente especializadas também significa dependência entre elas, o que pode ser uma questão no contexto em que uma destas está ameaçada de extinção.

Dessa forma, no nosso caso, ao criar duas espécies diferentes, o objetivo é simular as interações ecológicas entre elas, como o mutualismo entre plantas e polinizadores, no qual a planta oferece néctar ou pólen como recurso alimentar para o polinizador. Paralelamente, o polinizador, ao transportar o pólen de uma flor para outra, facilita a fertilização e a reprodução da planta, beneficiando ambas as partes envolvidas. Assim, desenvolveremos funções com a finalidade de gerar características de espécies.

1.1.2 Descrição da Função

A função recebe como entrada os seguintes parâmetros:

- `NumberA` e `NumberB`: Quantidades de indivíduos das espécies A e B, respectivamente;
- `traitsA` e `traitsB`: Vetores que representam o número de traços discretos e contínuos para as espécies A e B;
- `rangeDiscrete`: Intervalo de níveis discretos a serem utilizados na geração de traços morfológicos;
- `abundance`: Parâmetro de abundância utilizado para gerar a quantidade de indivíduos de cada espécie;
- `speciesClass`: Objeto da classe `SpeciesClass`, que permite reutilizar as características de outra espécie previamente criada;
- `specialist`: Booleano que define se as espécies B são especializadas em suas interações;
- `coLin`: Colunas que devem ser consideradas colineares e tratadas por meio de variáveis colineares;
- `sampling`: Função de amostragem usada para gerar traços contínuos;
- `specRange`: Intervalo de valores para a especialização das espécies B.

A função inicia a criação das espécies definindo dois conjuntos de dados, `A` e `B`, que armazenam os traços morfológicos das espécies A e B, respectivamente. Em seguida, são preenchidos os traços contínuos por meio da função de amostragem `sampling`, que gera valores de uma distribuição uniforme entre 0 e 1, ou aplica para o conjunto de dados uma outra distribuição imputada pelo usuário. Posteriormente, são preenchidos os traços discretos, utilizando a função auxiliar `create_discrete`, que gera amostras de níveis discretos com base em uma distribuição de probabilidades normalizada.

A abundância de cada indivíduo é determinada a partir de uma distribuição estatística, em geral no artigo estudado foi utilizado Poisson ou Exponencial. Finalmente, um vetor de especialização é gerado para as espécies B, baseado no valor booleano de `specialist`. A função retorna um objeto da classe `SpeciesClass`, contendo os traços, abundâncias e funções de distribuição de valores discretos para cada traço que serão usados na função de simular interações.

1.1.3 Aspectos Computacionais

Os aspectos computacionais da função envolvem a geração de dados simulados, a manipulação de distribuições probabilísticas e a aplicação de normalização em amostras discretas, além de operações sobre estruturas de dados. Abaixo estão os principais cálculos e fórmulas utilizados.

Distribuição de Poisson para Abundância

Por *default*, a abundância de indivíduos em cada espécie é modelada usando uma distribuição de Poisson, que é frequentemente utilizada para modelar o número de eventos que ocorrem em um intervalo de tempo fixo, quando esses eventos são independentes. A fórmula para a distribuição de Poisson é dada por

$$P(k, \lambda) = \frac{\lambda^k e^{-\lambda}}{k!},$$

onde λ é a média da distribuição e k é o número de eventos (número de indivíduos de uma espécie). O parâmetro `abundance` determina o valor de λ, que é usado para gerar a abundância `A_abund` e `B_abund` das espécies A e B, respectivamente:

$$\texttt{A\_abund} = \text{np.random.poisson}(abundance, NumberA) + 1$$
$$\texttt{B\_abund} = \text{np.random.poisson}(abundance, NumberB) + 1.$$

Geração de Traços Contínuos

Para a geração de traços contínuos, como a tolerância a temperaturas, por exemplo, a função de amostragem fornecida pelo parâmetro `sampling` é utilizada para gerar valores entre 0 e 1 com uma distribuição uniforme. A amostragem contínua é feita por meio da fórmula

`x = sampling(n)`,

onde x é o vetor gerado de n valores contínuos, representando os traços de uma espécie.

Geração de Traços Discretos

Os traços discretos são gerados utilizando a função auxiliar `create_discrete`, que cria uma distribuição de probabilidades normalizada para os diferentes níveis de um traço. Primeiro, a função escolhe um número de níveis $Nlevels$ de forma aleatória dentro do intervalo `rangeDiscrete`, e então gera as probabilidades para cada nível, normalizando-as

$$\text{prob\_normalized} = \frac{\text{prob}}{\text{sum(prob)}}.$$

A amostragem dos valores discretos é então feita com a fórmula

```
discreteV = np.random.choice(range(1, Nlevels + 1),
            size=n, p=prob_normalized, replace=True),
```

onde `Nlevels` representa os níveis discretos para cada traço e prob_normalized são as probabilidades normalizadas.

Especialização

O parâmetro `specialist` controla a especialização dos traços nas espécies B, ou seja, iremos determinar o quão dependentes as espécies serão ao interagirem. Quando definido como `True`, as espécies B têm valores de traços que seguem uma distribuição mais extrema, com valores mais próximos dos limites do intervalo de especialização, significando que são mais dependentes. Se `False`, as espécies serão mais generalistas. A fórmula usada para gerar a especialização é

$$S = \text{uniform}(specRange_{\min}, specRange_{\max}),$$

onde S é o valor de especialização para um traço e $specRange_{\min}$ e $specRange_{\max}$ são os limites do intervalo de especialização.

1.2 Função `simulate_interaction`

A função `simulate_interaction`, originalmente implementada em `R` e replicada em `Python`, foi projetada para calcular a matriz de interações entre duas populações de espécies (A e B). Durante o processo de replicação, priorizou-se a manutenção da lógica central, adaptando-a à sintaxe e às bibliotecas disponíveis no `Python`, garantindo consistência e eficiência computacional.

Essa função considera características morfológicas, variáveis contínuas e discretas, além de fatores relacionados à especialização. Baseada em modelos probabilísticos, ela permite simular tanto interações do tipo contínuo (como a adaptabilidade de traços entre espécies) quanto interações do tipo discreto (como padrões de especialização morfológica), proporcionando uma estrutura robusta para modelar relações ecológicas complexas.

1.2.1 Conceito de Simulação de Interações entre Características Morfológicas de Plantas e Polinizadores

Em ecologia, a simulação de interações entre plantas e polinizadores é fundamental para entender os processos de adaptação e coevolução que moldam a biodiversidade. As características morfológicas das plantas, como a forma e o tamanho das flores, o comprimento dos estames e a produção de néctar, influenciam diretamente a eficiência com que os polinizadores (como abelhas, borboletas e outros insetos) conseguem acessar os recursos da planta.

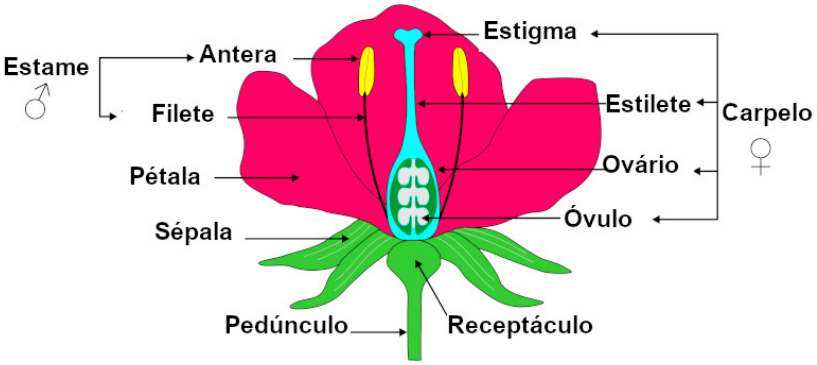


Figura 1.1: Principais partes de uma flor. Fonte: https://escolakids.uol.com.br/ciencias/composicao-da-flor.htm.

Por sua vez, essas interações podem ser um fator crucial na especialização dos polinizadores, que desenvolvem adaptações específicas para explorar essas plantas.

A função `simulate_interaction` foi projetada para simular essas interações morfológicas, levando em consideração não apenas as variáveis genéticas e comportamentais, mas também como as diferenças nos traços morfológicos entre as plantas e os polinizadores afetam o sucesso das interações. Essas interações são importantes porque determinam a eficiência da polinização, o sucesso reprodutivo das plantas e, consequentemente, a estrutura e a estabilidade das comunidades ecológicas.

A simulação dessas interações permite uma compreensão mais profunda de como as características morfológicas das plantas e os comportamentos especializados dos polinizadores influenciam as redes ecológicas, e como mudanças ambientais podem afetar a coevolução entre essas espécies. Além disso, a simulação de interações baseadas em morfologia ajuda a prever como a especialização pode evoluir em diferentes cenários ecológicos, como em ambientes perturbados, onde a competição por recursos ou a disponibilidade de polinizadores pode ser alterada.

Portanto, a função `simulate_interaction` contribui para a modelagem de redes ecológicas complexas, levando em consideração o papel fundamental das interações morfológicas na manutenção da biodiversidade e na adaptação de espécies dentro de seus ecossistemas.

1.2.2 Descrição da Função

A função `simulate_interaction` recebe os seguintes parâmetros de entrada:

- **species**: Refere-se ao objeto que contém as espécies a serem simuladas. O objeto `species` deve ter as propriedades `A` e `B`, que representam as duas espécies envolvidas na simulação. Ele pode também conter as características (`traitsA`, `traitsB`) dessas espécies;

- **main**: Lista de características principais das espécies que são usadas para calcular as interações principais entre elas;

- **inter**: Matriz que descreve as interações possíveis entre os traços das duas espécies. Cada linha da matriz contém pares de traços de espécies A e B que interagem de alguma forma. Por exemplo, a primeira linha pode indicar uma interação entre o traço `a_1` da espécie A e o traço `b_1` da espécie B;

- **weights**: Dicionário que especifica os pesos atribuídos aos traços principais e às interações. A chave `"main"` define o peso dos traços principais (um valor por cada traço em `main`), enquanto a chave `"inter"` define os pesos para as interações entre os traços das espécies A e B, correspondendo ao número de pares em `inter`;

- **re_sim** (default: `None`): Caso seja fornecido, esse parâmetro permite reiniciar a simulação com novos parâmetros. Ele deve conter uma estrutura que define uma nova função para as interações e características (`mainFunc`, `interFunc`) a serem usadas na simulação;

- **set_seed**: Semente para o gerador de números aleatórios, permitindo reproduzir os resultados da simulação;

- **kwargs**: Parâmetros adicionais que podem ser passados para funções internas ou para a criação das espécies. Isso pode incluir configurações específicas para os traços ou outras funcionalidades personalizadas.

Funções Internas e Operações

- **check_weights**: Verifica se os pesos fornecidos para os traços principais e interações estão corretos, com base no número de elementos em `main` e na forma de `inter`;

- **check_species**: Verifica e valida a entrada das espécies, utilizando os parâmetros adicionais passados por `kwargs`, chamando a função descrita na seção anterior;

- **discrete**: Lista de características discretas, baseada nos traços definidos em `species.traitsA` e `species.traitsB`. Se as espécies tiverem traços discretos, esses traços serão usados para calcular as interações;

- **mainTraits**: Dicionário que define as características principais de cada traço, como se é discreto, a média (para traços contínuos) e o peso do traço;

- **mainFunc**: Função usada para calcular o efeito de interações baseadas nos traços principais. Ela calcula a interação de cada traço em `main` utilizando as funções auxiliares;

- **interTraits**: Lista que define as características das interações entre as espécies, incluindo se as interações são entre traços discretos ou contínuos. Dependendo das características, diferentes abordagens são usadas para calcular os efeitos das interações;

- **interFunc**: Função usada para calcular o efeito das interações entre as espécies com base nos traços de `inter` e nas características da interação. Ela leva em consideração se a interação entre os traços de A e B é discreta ou contínua;

- **interMatrix**: Matriz de interações entre as duas espécies.
  A função `calculate_interactions` é usada para calcular os valores das interações para todas as combinações de traços de A e B;
- **create_distribution_func**: Cria uma função de distribuição baseada nas interações, podendo ser de distribuição de Poisson ou binária, dependendo do parâmetro `is_poisson`;
- **Output**: A função retorna um dicionário contendo vários resultados da simulação, incluindo os dados das duas espécies, funções de interação, a matriz de interações calculada e as configurações de simulação.

1.2.3 Cálculos

Os cálculos na função podem ser divididos em três partes principais: cálculo dos traços principais, cálculo das interações e distribuição de interações.

Cálculo dos Traços Principais

A função `mainFunc` calcula o efeito dos traços principais para um par de indivíduos i e j das populações A e B. Para cada traço m, o valor é determinado como

$$T_{ij}^m = \begin{cases} \texttt{mean}[x^m] w_m, & \text{se o traço for discreto,} \\ \mathcal{N}(\texttt{mean}, \sigma) w_m, & \text{se o traço for contínuo,} \end{cases}$$

onde

- $\texttt{mean}[x^m]$ é o valor médio para o traço m do indivíduo i;
- w_m é o peso associado ao traço m;
- $\mathcal{N}(\texttt{mean}, \sigma)$ representa uma distribuição normal com média `mean` e desvio padrão σ.

O produto total dos traços principais é então dado por

$$\texttt{mainFunc}(x, y) = \prod_{m \in \texttt{main}} T_{ij}^m.$$

Este cálculo tem custo computacional baixo na linguagem de programação `Python`.

Cálculo das Interações

Para as interações entre espécies, consideramos a função `interFunc`, que avalia a interação com base em uma matriz de interações e características dos traços. Para cada interação k, o cálculo é

$$I_{ij}^k = \begin{cases} M_{xy} w_k, & \text{se ambos os traços forem discretos,} \\ \phi(y, \mu_x, \sigma) w_k, & \text{se um traço for contínuo,} \\ \phi(\log(x/y), 0, \sigma) w_k, & \text{interação do tipo logarítmica,} \end{cases}$$

onde

- M_{xy} é o valor da matriz de interação discreta para os traços de x e y;
- $\phi(y, \mu_x, \sigma)$ é a função de densidade da distribuição normal

$$\phi(y, \mu_x, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-\mu_x)^2}{2\sigma^2}\right);$$

- w_k é o peso atribuído à interação k.

O efeito total das interações é então dado por

$$\texttt{interFunc}(x, y) = \prod_k I_{ij}^k.$$

Cálculo da Matriz de Interações

A matriz de interações `interMatrix` para as espécies i de A e j de B é então calculada como

$$\texttt{interMatrix}[i, j] = \texttt{mainFunc}(x, y) \times \texttt{interFunc}(x, y) \times A_{abund}[i] \times B_{abund}[j] \times R[i, j],$$

onde

- $A_{abund}[i]$ e $B_{abund}[j]$ são as abundâncias das espécies i e j;
- $R[i, j]$ são valores aleatórios uniformemente distribuídos entre 0.3 e 0.7, simulando a variabilidade natural. Estes valores foram determinados pelos autores do artigo estudado.

1.2.4 Distribuição de Interações

A função `create_distribution_func` gera distribuições com base na matriz de interações calculada. As duas distribuições são:

- **Distribuição Poisson**:

$$P_{ij} = \mathcal{P}(\lambda = \texttt{interMatrix}[i, j] \times x),$$

onde $\mathcal{P}(\lambda)$ é a distribuição Poisson com taxa λ;

- **Distribuição Binária**:

$$B_{ij} = \begin{cases} 1, & \text{se } \mathcal{P}(\lambda) > 0, \\ 0, & \text{caso contrário.} \end{cases}$$

1.3 Função `createCommunity`

A função `createCommunity`, originalmente desenvolvida em R e replicada em `Python`, foi projetada para construir uma comunidade ecológica simulada a partir de dados de espécies e suas interações. Durante o processo de adaptação, buscou-se preservar a lógica e a funcionalidade da implementação original, garantindo que a versão em `Python` mantivesse a eficiência e a flexibilidade do código em R.

Essa função integra os dados gerados sobre as características das espécies e as matrizes de interação para criar comunidades biológicas completas, permitindo a análise de padrões ecológicos e de organização comunitária.

1.3.1 Conceito de Comunidade

Em ecologia, uma **comunidade** é definida como um conjunto de populações de diferentes espécies que coexistem e interagem em uma mesma área geográfica. As interações dentro de uma comunidade podem ser de várias naturezas, como predação, competição, mutualismo, parasitismo, entre outras. Essas interações desempenham um papel crucial na estruturação da comunidade e na dinâmica dos ecossistemas, influenciando o crescimento, a sobrevivência e a distribuição das espécies.

Nesse sentido, a função `createCommunity` utiliza dados de dois grupos de espécies (como presas e predadores ou plantas e polinizadores) e uma matriz de interação para simular essas relações. A matriz de interação é essencial para modelar as conexões entre indivíduos de diferentes grupos e pode incluir informações sobre a intensidade ou a presença/ausência dessas interações.

Além disso, a função incorpora um método de imputação de dados ausentes, utilizando o algoritmo `MissForest`, que preenche lacunas nos dados de interação, com o objetivo de reduzir vieses e melhorar a precisão dos modelos preditivos. Com isso, a função prepara os dados para análise de regressão e classificação, que são usadas para prever padrões de interação entre as espécies.

1.3.2 Descrição da Função

A função `createCommunity` foi projetada para criar uma comunidade ecológica a partir de dois grupos de dados de espécies, com a capacidade de lidar com dados ausentes e realizar análise preditiva. Abaixo, uma descrição detalhada dos parâmetros de entrada dessa função

- **a**: Representa o primeiro grupo de dados de espécies (por exemplo, plantas). Este grupo deve ser fornecido como um *DataFrame* com as características das espécies, onde a primeira coluna contém os identificadores das espécies;
- **b**: Representa o segundo grupo de dados de espécies (por exemplo, polinizadores). Assim como o parâmetro `a`, deve ser fornecido como um *DataFrame* com as características das espécies;
- **z**: Uma matriz de interação entre os grupos `a` e `b`. O *DataFrame* de `z` deve ter como índices os identificadores das espécies em `a` e como colunas os identificadores das espécies em `b`;

- **community**: Um conjunto de dados que representa a comunidade ecológica a ser criada. Pode ser uma lista de pares de grupos de dados (como `a` e `b`) ou diretamente um *DataFrame* contendo as informações da comunidade;
- **response**: O nome da coluna que contém a variável alvo ou resposta no *DataFrame* da comunidade. Essa variável é necessária para a análise preditiva;
- **positive**: Define a classe ou valor que é considerado "positivo"para as interações. Esse parâmetro pode ser usado na construção de um modelo preditivo de classificação, indicando o que se considera uma interação positiva;
- **impute**: Se `True`, a função irá imputar dados ausentes utilizando o algoritmo `MissForest`;
- **log**: Se `True`, aplica a transformação logarítmica nos dados numéricos para a regressão. Isso pode ser útil para normalizar variáveis com distribuições altamente assimétricas.

A função utiliza esses parâmetros para gerar uma comunidade ecológica simulada, considerando as interações entre os grupos e aplicando imputação de dados ausentes quando necessário. Ela também prepara os dados para análise de regressão ou classificação, ajudando a prever padrões de interação e a criar modelos preditivos úteis para diversos estudos ecológicos.

A função `createCommunity` verifica se os dados de entrada são válidos com o auxílio da função `check_input`. Ela também chama a função `create_inter` para gerar a matriz de interações a partir dos dados de entrada, que será usada na criação da comunidade.

1.3.3 Processo de Imputação

A função auxiliar `impute_data` realiza a imputação de valores ausentes utilizando o algoritmo `MissForest`. A imputação é aplicada nos dois grupos de dados (`a` e `b`), substituindo os valores ausentes nas características das espécies. Após a imputação, os dados são combinados, permitindo a criação de uma matriz de interações completa para análises subsequentes, como regressão ou classificação. Isso garante que a análise da comunidade ecológica seja realizada de forma adequada, mesmo com dados incompletos.

1.3.4 Criação das Interações

A função `create_inter` combina os indivíduos dos grupos `A` e `B` utilizando a matriz de interação `z` para formar um conjunto de dados de interação.

Combinação dos Dados

Para cada par de indivíduos i e j dos grupos A e B, o conjunto de características é formado pela anexação dos valores das características de cada espécie envolvida na interação. Por exemplo, se `a_1` interage com `b_25`, a linha correspondente será composta pelas características do indivíduo da espécie A e pelas características do indivíduo da espécie B. Dessa forma, construímos a primeira parte do dicionário `'data'`, ao qual será adicionado o resultado dessa interação, que servirá como nosso alvo.

Cálculo do Alvo (`'target'`)

A coluna alvo `'target'` é definida com base nos valores da matriz de interação `z`, sendo transformada em "positive"se o valor for 1, ou "negative"se for 0, no caso de problemas de classificação. Além disso, para determinar se o problema será de classificação ou regressão, avalia-se a quantidade de valores únicos na coluna `'target'`. Se houver apenas dois valores únicos, o problema será considerado de classificação; caso contrário, será de regressão. Por fim, se o problema for de regressão e o parâmetro *log* estiver ativado, aplica-se o logaritmo natural à coluna alvo.

Durante a replicação do código em `Python`, deu-se especial atenção à otimização dos processos, garantindo uma eficiência computacional comparável à implementação original em `R`. O foco principal esteve na modularização e na otimização de laços de repetição dessas funções.

No próximo capítulo, realizaremos quatro simulações de interações. Utilizaremos o `createSpecies` para criar as espécies, o `simulate_interaction` para simular as interações entre características, e, finalmente, o `createCommunity` para construir o dicionário final. Em seguida, os conjuntos de dados gerados a partir desses dicionários serão utilizados para testar os modelos de aprendizagem de máquina em diferentes cenários.

Capítulo 2

Aprendizagem de Máquina

Neste capítulo, discutiremos os modelos de aprendizagem de máquina aplicados à classificação e regressão, abordando suas respectivas métricas e conceitos fundamentais. Além disso, será realizada a reprodução, em `Python`, do código original `Prediction.Rmd`, desenvolvido originalmente em `R` e utilizado no artigo (PICHLER et al., 2020).

A reprodução será feita utilizando os mesmos parâmetros e métricas, com adição das métricas de precisão e *recall*, permitindo uma análise de desempenho detalhada que será explorada no Capítulo 3. Essas simulações permitirão avaliar a performance dos modelos em um cenário prático, com base em um conjunto de dados real.

Adicionalmente, serão detalhadas as funções criadas para executar os modelos de classificação e regressão. Primeiramente, apresentaremos as quatro simulações geradas de acordo com algumas especificações de característocas, serão utilizadas as chaves 'data' e 'target' para fazer as predições com os modelos de aprendizado de máquina. Após isso, descreveremos a função voltada para os modelos de classificação, seguida da função destinada à execução dos modelos de regressão, além das funções com hiperparâmetros.

2.1 Formando conjuntos de dados com a função `createCommunity`

Após o desenvolvimento e a aplicação de todas as funções mencionadas no Capítulo 1, é executada a função `createCommunity`, que retorna um dicionário denominado `class_community`, contendo as seguintes chaves:

- `'data'`: conjunto de dados contendo os índices X e Y, com as quantidades a_i, $i = 1, \cdots, n$, e b_j, $j = 1, \cdots, m$, que representam o número de animais de cada espécie A e B. Inclui também os valores das características A_u, $u = 1, \cdots, t$, e B_v, $v = 1, \cdots, k$, seguidos pela coluna `'target'`, que será utilizada como variável resposta para a predição nos modelos de aprendizagem de máquina;

- `'target'`: Coluna `target` do conjunto de dados contido em `'data'`, destinada à análise detalhada;

- `'type'`: Identifica se o problema é de classificação (`'Classification'`) ou regressão (`'Regression'`), com base no cálculo dos valores únicos da coluna `target`. A partir

dessa chave, o conjunto de dados é direcionado para a função correspondente ao modelo de aprendizado de máquina;

- `'class'`: Indica se os dados pertencem a uma comunidade específica e qual é o tipo de modelo, determinado pelos valores dos parâmetros X, Y e z inseridos na função `create_community`.

Dessa forma, seguindo alguns parâmetros, iremos descrever doze dicionários que serão úteis para captar a capacidade dos modelos em diferentes tipos de simulações.

Para formar os conjuntos de dados usados para fazer as predições com os modelos, foi implementada a função `simulate_interaction` para diferentes valores de características e números de espécies. Os resultados desta seção demonstram como diferentes características, tempos de observação e quantidade de espécies influenciam diretamente as distribuições de classes para os modelos de classificação e as interações de regressão. Cada especificação oferece um cenário distinto para analisar as interações e ajustar os modelos conforme necessário.

2.1.1 C1 e R1

Neste primeiro modelo, não há efeitos principais ou abundâncias de espécies, e o tamanho da rede base é de 50 por 100, com 50 entidades no conjunto 'A' e 100 entidades no conjunto 'B'. O modelo é caracterizado por efeitos amplos e fracos, definidos por uma distribuição gaussiana com desvio padrão igual a 100.

Para desenvolver os primeiros modelos de classificação e regressão, as características são configuradas conforme segue. O código abaixo simula os dados sem efeitos, com um intervalo de interação específico e sem abundâncias de espécies.

```
simulated_data_1 = simulate_interaction(None, main=[],
                                        inter=np.array([["A1", "B1"]]),
                                        weights= {'main': [], 'inter': [1]},
                                        NumberA=50, NumberB=100,
                                        traitsA=[0, 6], traitsB=[0, 6],
                                        abundance=False, specRange=[100, 100])
```

Explicação dos parâmetros:

- `None`: Sem efeitos principais definidos;
- `main=[]`: Lista vazia, indicando a ausência de interações principais;
- `inter=np.array([["A1", "B1"]])`: Define uma interação específica entre as entidades 'A1' e 'B1';
- `weights='main': [], 'inter': [1]`: Define os pesos das interações principais e binárias. Neste caso, o peso da interação é 1;
- `NumberA = 50, NumberB = 100`: O número de espécies nos conjuntos 'A' e 'B', respectivamente;

- `traitsA=[0, 6], traitsB=[0, 6]`: Os intervalos dos traços para as entidades 'A' e 'B'.
- `abundance = False`: Desativa a abundância das espécies;
- `specRange = [100, 100]`: Define o intervalo de alcance das espécies.

Após a simulação dos dados, as variáveis X e Y são atribuídas, representando as características das entidades 'A' e 'B', respectivamente.

```
X = simulated_data_1['A'],
Y = simulated_data_1['B'].
```

Em seguida, as interações binárias são determinadas, e a comunidade de classificação é criada

```
binary_interactions = minOneInter(simulated_data_1['binar'](3e2)),
class_community_C1 = createCommunity(X, Y, binary_interactions).
```

O dicionário resultante `class_community_C1` contém os seguintes elementos para `['data']`

```
{'data':         X     Y       A1       A2       A3       A4       A5       A6
 0      a1    b1  0.220049  0.392224  0.519877  0.103307  0.020509  0.082303
 1      a1    b2  0.220049  0.392224  0.519877  0.103307  0.020509  0.082303
 2      a1    b3  0.220049  0.392224  0.519877  0.103307  0.020509  0.082303
 3      a1    b4  0.220049  0.392224  0.519877  0.103307  0.020509  0.082303
 4      a1    b5  0.220049  0.392224  0.519877  0.103307  0.020509  0.082303
 ...   ...   ...       ...       ...       ...       ...       ...       ...
 4995  a50   b96  0.744806  0.621427  0.153943  0.252078  0.763151  0.473746
 4996  a50   b97  0.744806  0.621427  0.153943  0.252078  0.763151  0.473746
 4997  a50   b98  0.744806  0.621427  0.153943  0.252078  0.763151  0.473746
 4998  a50   b99  0.744806  0.621427  0.153943  0.252078  0.763151  0.473746
 4999  a50  b100  0.744806  0.621427  0.153943  0.252078  0.763151  0.473746

            B1        B2        B3        B4        B5        B6    target
 0     0.554262  0.201497  0.598168  0.321227  0.937218  0.351307  negative
 1     0.240440  0.052236  0.679376  0.712086  0.511005  0.345725  negative
 2     0.053010  0.927847  0.002979  0.031931  0.824025  0.312266  positive
 3     0.856577  0.078558  0.463723  0.773029  0.167946  0.815044  positive
 4     0.415313  0.309463  0.924364  0.793046  0.128646  0.450695  negative
 ...        ...       ...       ...       ...       ...       ...       ...
 4995  0.459322  0.521029  0.904058  0.972598  0.651525  0.174716  positive
 4996  0.759874  0.746792  0.284996  0.020193  0.779698  0.995294  positive
 4997  0.863722  0.834731  0.153054  0.374362  0.112776  0.157571  negative
 4998  0.013790  0.186785  0.841001  0.203053  0.718653  0.231585  negative
 4999  0.438386  0.595809  0.646948  0.979375  0.477054  0.413430  positive
```

Figura 2.1: Elementos contidos no dicionário `class_community_C1` associados à chave `['data']`.

A Figura 2.1 ilustra os dados presentes no dicionário para a chave `['data']`.
Além disso, um modelo com interações de Poisson é criado

```
poisson_interactions = minOneInter(simulated_data_1['poisson'](3e2))
class_community_R1 = createCommunity(X, Y, poisson_interactions, log=False)
```

{'data':
	X	Y	A1	A2	A3	A4	A5	A6
0	a1	b1	0.220049	0.392224	0.519877	0.103307	0.020509	0.082303
1	a1	b2	0.220049	0.392224	0.519877	0.103307	0.020509	0.082303
2	a1	b3	0.220049	0.392224	0.519877	0.103307	0.020509	0.082303
3	a1	b4	0.220049	0.392224	0.519877	0.103307	0.020509	0.082303
4	a1	b5	0.220049	0.392224	0.519877	0.103307	0.020509	0.082303
...	...	...	...	...	...	...	...	...
4995	a50	b96	0.744806	0.621427	0.153943	0.252078	0.763151	0.473746
4996	a50	b97	0.744806	0.621427	0.153943	0.252078	0.763151	0.473746
4997	a50	b98	0.744806	0.621427	0.153943	0.252078	0.763151	0.473746
4998	a50	b99	0.744806	0.621427	0.153943	0.252078	0.763151	0.473746
4999	a50	b100	0.744806	0.621427	0.153943	0.252078	0.763151	0.473746

	B1	B2	B3	B4	B5	B6	target
0	0.554262	0.201497	0.598168	0.321227	0.937218	0.351307	0.0
1	0.240440	0.052236	0.679376	0.712086	0.511005	0.345725	0.0
2	0.053010	0.927847	0.002979	0.031931	0.824025	0.312266	2.0
3	0.856577	0.078558	0.463723	0.773029	0.167946	0.815044	2.0
4	0.415313	0.309463	0.924364	0.793046	0.128646	0.450695	0.0
...	...	...	...	...	...	...	...
4995	0.459322	0.521029	0.904058	0.972598	0.651525	0.174716	1.0
4996	0.759874	0.746792	0.284996	0.020193	0.779698	0.995294	1.0
4997	0.863722	0.834731	0.153054	0.374362	0.112776	0.157571	0.0
4998	0.013790	0.186785	0.841001	0.203053	0.718653	0.231585	0.0
4999	0.438386	0.595809	0.646948	0.979375	0.477054	0.413430	1.0

Figura 2.2: Elementos contidos no dicionário `class_community_R1` associados à chave `['data']`.

O dicionário `['data']` resultante para `class_community_R1` é estruturado da seguinte forma

A Figura 2.2 apresenta a estrutura dos dados resultantes para a chave `['data']` no modelo de Poisson. Neste contexto, o tipo de tarefa é 'Regression', e o `target` é numérico, representando a saída do modelo com valores multiclasse. O modelo com interações de Poisson é utilizado para ajustar os dados de forma adequada.

2.1.2 C2 e R2

Nesta seção, os modelos incorporam abundâncias de espécies, mas ainda não apresentam efeitos principais. As principais características deste modelo com abundância incluem efeitos amplos e fracos, definidos por uma distribuição gaussiana com desvio padrão igual a 100 e a introdução de abundâncias de espécies. O tamanho da rede base permanece como 50 por 100, com 50 entidades no conjunto 'A' e 100 entidades no conjunto 'B'.

Para desenvolver os modelos de classificação e regressão desta configuração, os dados são simulados utilizando um intervalo de interação específico, enquanto as abundâncias de espécies seguem uma distribuição exponencial. O código abaixo detalha a simulação dos dados

```
simulated_data_2 = simulate_interaction(None,
                                        main=[],
                                        inter=np.array([["A1", "B1"]]),
                                        weights= {'main': [],
                                        'inter': [1]},
                                        NumberA=50,
                                        NumberB=100,
                                        traitsA=[0, 5],
```

```
traitsB=[0, 5],
abundance= lambda a, b, rng:
np.random.exponential(2, size=a),
specRange=[100, 100])
```

Explicação dos parâmetros:

- `None`: Sem efeitos principais definidos;
- `main=[]`: Lista vazia, indicando a ausência de interações principais;
- `inter=np.array([["A1", "B1"]])`: Define uma interação específica entre as entidades 'A1' e 'B1';
- `weights='main': [], 'inter': [1]`: Define os pesos das interações principais e binárias. Neste caso, o peso da interação é 1;
- `NumberA=50, NumberB=100`: O número de espécies nos conjuntos 'A' e 'B', respectivamente;
- `traitsA=[0, 5], traitsB=[0, 5]`: Os intervalos dos traços para as entidades 'A' e 'B';
- `abundance=lambda a, b, rng: np.random.exponential(10, size=a)`: Define as abundâncias de espécies seguindo uma distribuição exponencial com valor médio de 10;
- `specRange=[100, 100]`: Define o intervalo de alcance das espécies.

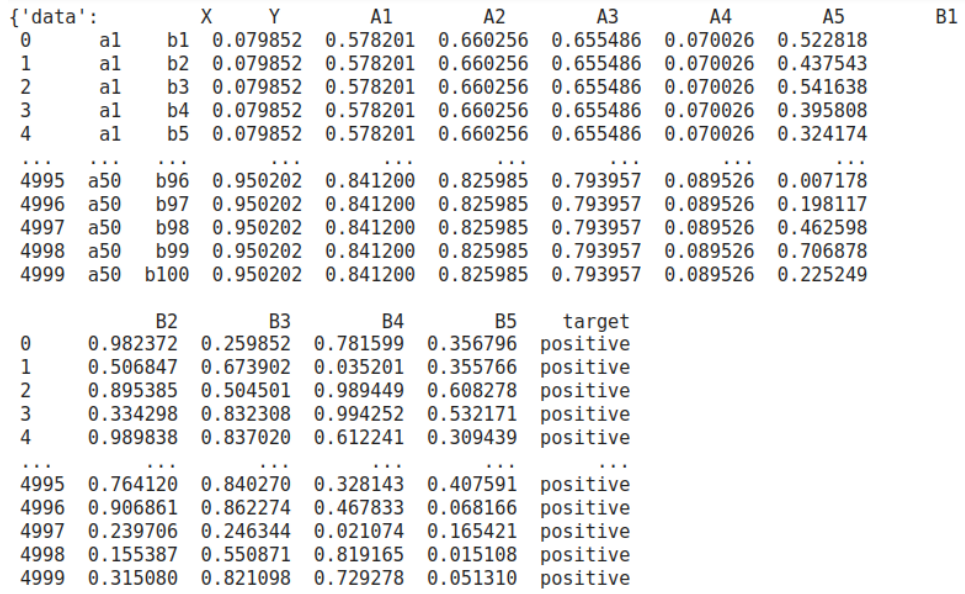
Após a simulação dos dados, as variáveis X e Y são atribuídas, representando as características das entidades 'A' e 'B', respectivamente.

```
X = simulated_data_2['A'],
Y = simulated_data_2['B'].
```

Para o modelo de classificação, as interações binárias são determinadas, e a comunidade correspondente é criada

```
binary_interactions_2 = minOneInter(simulated_data_2['binar'](4e3)),
class_community_C2 = createCommunity(X, Y, binary_interactions_2).
```

O dicionário `['data']` resultante para `class_community_C2` é ilustrado na Figura 2.3



{'data':	X	Y	A1	A2	A3	A4	A5	B1
0	a1	b1	0.079852	0.578201	0.660256	0.655486	0.070026	0.522818
1	a1	b2	0.079852	0.578201	0.660256	0.655486	0.070026	0.437543
2	a1	b3	0.079852	0.578201	0.660256	0.655486	0.070026	0.541638
3	a1	b4	0.079852	0.578201	0.660256	0.655486	0.070026	0.395808
4	a1	b5	0.079852	0.578201	0.660256	0.655486	0.070026	0.324174
...	...	...	...	...	...	...	...	...
4995	a50	b96	0.950202	0.841200	0.825985	0.793957	0.089526	0.007178
4996	a50	b97	0.950202	0.841200	0.825985	0.793957	0.089526	0.198117
4997	a50	b98	0.950202	0.841200	0.825985	0.793957	0.089526	0.462598
4998	a50	b99	0.950202	0.841200	0.825985	0.793957	0.089526	0.706878
4999	a50	b100	0.950202	0.841200	0.825985	0.793957	0.089526	0.225249

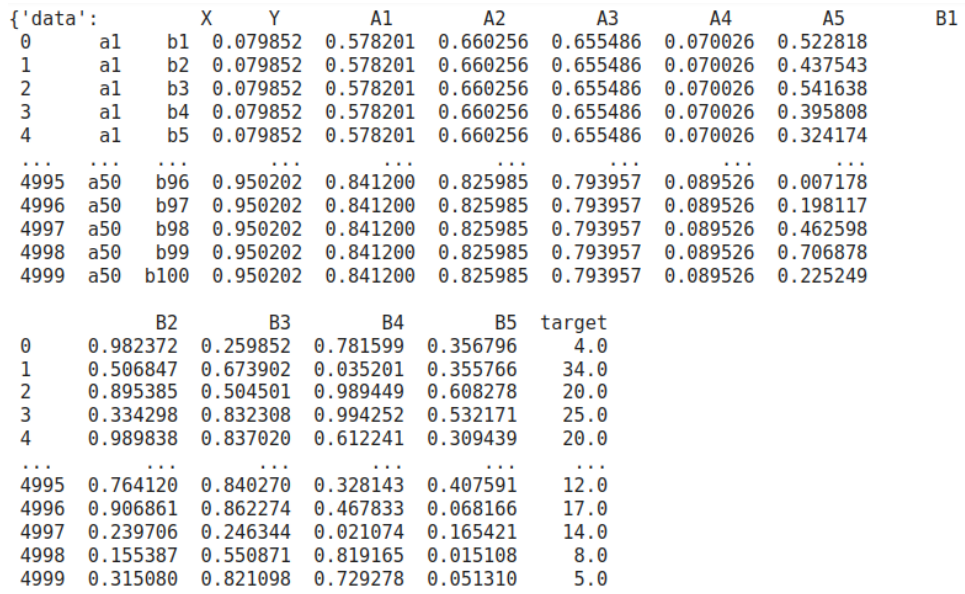
	B2	B3	B4	B5	target
0	0.982372	0.259852	0.781599	0.356796	positive
1	0.506847	0.673902	0.035201	0.355766	positive
2	0.895385	0.504501	0.989449	0.608278	positive
3	0.334298	0.832308	0.994252	0.532171	positive
4	0.989838	0.837020	0.612241	0.309439	positive
...	...	...	...	...	...
4995	0.764120	0.840270	0.328143	0.407591	positive
4996	0.906861	0.862274	0.467833	0.068166	positive
4997	0.239706	0.246344	0.021074	0.165421	positive
4998	0.155387	0.550871	0.819165	0.015108	positive
4999	0.315080	0.821098	0.729278	0.051310	positive

Figura 2.3: Elementos contidos no dicionário `class_community_C2` associados à chave `['data']`.

Para o modelo de regressão, as interações de Poisson são utilizadas, e a comunidade correspondente é criada:

```
poisson_interactions_2 = simulated_data_2['poisson'](4e3),
class_community_R2 = createCommunity(X, Y, poisson_interactions_2,
log=False).
```

O dicionário `['data']` resultante para `class_community_R2` é ilustrado na Figura 2.4



{'data':	X	Y	A1	A2	A3	A4	A5	B1
0	a1	b1	0.079852	0.578201	0.660256	0.655486	0.070026	0.522818
1	a1	b2	0.079852	0.578201	0.660256	0.655486	0.070026	0.437543
2	a1	b3	0.079852	0.578201	0.660256	0.655486	0.070026	0.541638
3	a1	b4	0.079852	0.578201	0.660256	0.655486	0.070026	0.395808
4	a1	b5	0.079852	0.578201	0.660256	0.655486	0.070026	0.324174
...	...	...	...	...	...	...	...	...
4995	a50	b96	0.950202	0.841200	0.825985	0.793957	0.089526	0.007178
4996	a50	b97	0.950202	0.841200	0.825985	0.793957	0.089526	0.198117
4997	a50	b98	0.950202	0.841200	0.825985	0.793957	0.089526	0.462598
4998	a50	b99	0.950202	0.841200	0.825985	0.793957	0.089526	0.706878
4999	a50	b100	0.950202	0.841200	0.825985	0.793957	0.089526	0.225249

	B2	B3	B4	B5	target
0	0.982372	0.259852	0.781599	0.356796	4.0
1	0.506847	0.673902	0.035201	0.355766	34.0
2	0.895385	0.504501	0.989449	0.608278	20.0
3	0.334298	0.832308	0.994252	0.532171	25.0
4	0.989838	0.837020	0.612241	0.309439	20.0
...	...	...	...	...	...
4995	0.764120	0.840270	0.328143	0.407591	12.0
4996	0.906861	0.862274	0.467833	0.068166	17.0
4997	0.239706	0.246344	0.021074	0.165421	14.0
4998	0.155387	0.550871	0.819165	0.015108	8.0
4999	0.315080	0.821098	0.729278	0.051310	5.0

Figura 2.4: Elementos contidos no dicionário `class_community_R2` associados à chave `['data']`.

Aqui, o tipo de tarefa é 'Regression', e o `target` é numérico, representando a saída do modelo. O modelo com interações de Poisson é utilizado para ajustar os dados com base nas abundâncias de espécies simuladas.

2.1.3 C3 e R3

Nesta seção, os modelos introduzem efeitos principais fortes e interações de alto peso, mas não incluem abundâncias de espécies. As características principais incluem uma rede de tamanho ampliado, com 100 entidades no conjunto 'A' e 200 entidades no conjunto 'B', e interações fortes definidas por uma distribuição gaussiana com pequeno desvio padrão e alto peso de interação.

Os dados são simulados conforme o código abaixo, incluindo várias interações específicas e sem abundâncias de espécies

```
simulated_data_3 = simulate_interaction(None, main=[],
                                        inter=np.array([["A1", "B1"],
                                        ["A2", "B2"],
                                        ["A3", "B3"]]),
                                        weights={'main': [],
                                        'inter': [10, 10, 10]},
                                        NumberA=100,
                                        NumberB=200,
                                        traitsA=[0, 6],
                                        traitsB=[0, 6],
                                        abundance=False,
                                        specRange=[0.5, 1.2])
```

Explicação dos parâmetros:

- `main=[]`: Lista vazia, indicando a ausência de interações principais;
- `inter=np.array([["A1", "B1"], ["A2", "B2"], ["A3", "B3"]])`: Define interações específicas entre pares de entidades ('A1-B1', 'A2-B2', 'A3-B3');
- `weights='main': [], 'inter': [10, 10, 10]`: Define pesos altos para as interações especificadas;
- `NumberA = 100, NumberB = 200`: O número de espécies nos conjuntos 'A' e 'B', respectivamente;
- `traitsA=[0, 6], traitsB=[0, 6]`: Intervalos dos traços para as entidades 'A' e 'B';
- `abundance=False`: Sem abundâncias de espécies;
- `specRange=[0.5, 1.2]`: Intervalo para os alcances específicos das espécies.

Após a simulação dos dados, as variáveis X e Y são atribuídas, representando as características das entidades 'A' e 'B', respectivamente.

```
X = simulated_data_3[‘A’],
Y = simulated_data_3[‘B’].
```

Para o modelo de classificação, as interações binárias são determinadas, e a comunidade correspondente é criada

```
binary_interactions_3 = simulated_data_3[‘binar’](1.2e-1),
class_community_C3 = createCommunity(X, Y, binary_interactions_3).
```

O dicionário `[‘data’]` resultante para `class_community_C3` é ilustrado na Figura 2.5:

```
{'data':          X     Y       A1       A2       A3       A4       A5       A6
 0       a1    b1  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 1       a1    b2  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 2       a1    b3  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 3       a1    b4  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 4       a1    b5  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 ...    ...   ...       ...       ...       ...       ...       ...       ...
 19995  a100  b196  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149
 19996  a100  b197  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149
 19997  a100  b198  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149
 19998  a100  b199  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149
 19999  a100  b200  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149

              B1        B2        B3        B4        B5        B6    target
 0      0.905394  0.627141  0.004124  0.495460  0.677577  0.208311  negative
 1      0.224414  0.228480  0.016449  0.815453  0.078746  0.322951  negative
 2      0.973814  0.019673  0.702359  0.935972  0.218404  0.329397  negative
 3      0.201055  0.989632  0.955579  0.835887  0.658748  0.769800  negative
 4      0.226728  0.434467  0.392817  0.756376  0.728210  0.284380  negative
 ...         ...       ...       ...       ...       ...       ...       ...
 19995  0.101379  0.985636  0.209741  0.109177  0.153338  0.845952  negative
 19996  0.749521  0.252868  0.423237  0.166023  0.371363  0.610882  negative
 19997  0.014416  0.248335  0.593786  0.911374  0.425073  0.918091  negative
 19998  0.228933  0.846219  0.739485  0.118318  0.019194  0.367549  negative
 19999  0.083377  0.793793  0.002202  0.738356  0.321049  0.847341  negative
```

Figura 2.5: Elementos contidos no dicionário `class_community_C3` associados à chave `[‘data’]`.

Para o modelo de regressão, as interações de Poisson são utilizadas, e a comunidade correspondente é criada

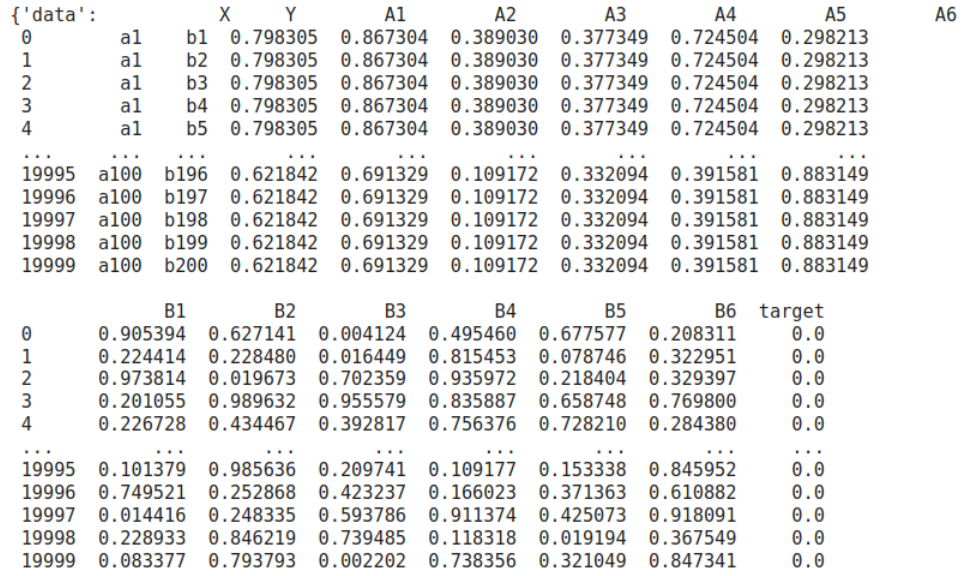
```
poisson_interactions_3 = simulated_data_3[‘poisson’](1.2e-1)
class_community_R3 = createCommunity(X, Y, poisson_interactions_3,
log=False)
```

O dicionário `[‘data’]` resultante para `class_community_R3` é ilustrado na Figura 2.6

2.1.4 C4 e R4

Nesta seção, os modelos introduzem efeitos principais e consideram diferentes tempos de observação, mas sem abundâncias de espécies. A variação na observação resulta em diferentes distribuições de classes no contexto de classificação. Três proporções específicas de classes foram escolhidas para interações positivas: 10%, 25% e 50%.

A simulação dos dados segue a configuração descrita abaixo, com interações específicas e sem abundâncias de espécies



{'data':	X	Y	A1	A2	A3	A4	A5	A6
0	a1	b1	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
1	a1	b2	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
2	a1	b3	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
3	a1	b4	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
4	a1	b5	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
...	...	...	...	...	...	...	...	...
19995	a100	b196	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19996	a100	b197	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19997	a100	b198	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19998	a100	b199	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19999	a100	b200	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149

	B1	B2	B3	B4	B5	B6	target
0	0.905394	0.627141	0.004124	0.495460	0.677577	0.208311	0.0
1	0.224414	0.228480	0.016449	0.815453	0.078746	0.322951	0.0
2	0.973814	0.019673	0.702359	0.935972	0.218404	0.329397	0.0
3	0.201055	0.989632	0.955579	0.835887	0.658748	0.769800	0.0
4	0.226728	0.434467	0.392817	0.756376	0.728210	0.284380	0.0
...	...	...	...	...	...	...	...
19995	0.101379	0.985636	0.209741	0.109177	0.153338	0.845952	0.0
19996	0.749521	0.252868	0.423237	0.166023	0.371363	0.610882	0.0
19997	0.014416	0.248335	0.593786	0.911374	0.425073	0.918091	0.0
19998	0.228933	0.846219	0.739485	0.118318	0.019194	0.367549	0.0
19999	0.083377	0.793793	0.002202	0.738356	0.321049	0.847341	0.0

Figura 2.6: Elementos contidos no dicionário `class_community_R3` associados à chave `['data']`.

```
simulated_data_4 = simulate_interaction(None, main=[],
                                        inter=np.array([["A1", "B1"],
                                        ["A2", "B2"],
                                        ["A3", "B3"]]),
                                        weights={'main': [],
                                        'inter': [10, 10, 10]},
                                        NumberA=100,
                                        NumberB=200,
                                        traitsA=[0, 6],
                                        traitsB=[0, 6],
                                        abundance=False,
                                        specRange=[0.5, 1.2])
```

Os níveis de observação são ajustados para gerar proporções de classes específicas. Os valores de interação binária correspondentes são calculados

```
obsC = {
    'low': simulated_data_4['binar'](0.7e-2),
    'mid': simulated_data_4['binar'](3.2e-2),
    'high': simulated_data_4['binar'](1.2e-1)
}
```

A comunidade de classificação é então criada para cada observação

```
class_community_C4_low = createCommunity(X, Y, obsC['low']),
class_community_C4_mid = createCommunity(X, Y, obsC['mid']),
class_community_C4_high = createCommunity(X, Y, obsC['high']).
```

A seguir, são apresentadas as visualizações dos elementos contidos nos dicionários `class_community_C4_low`, `class_community_C4_mid` e `class_community_C4_high`, associados à chave `['data']`.

A Figura 2.7 mostra os elementos do dicionário `class_community_C4_low`, representando as interações e proporções de classes para o nível baixo de observação (10%)

{'data':

	X	Y	A1	A2	A3	A4	A5	A6
0	a1	b1	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
1	a1	b2	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
2	a1	b3	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
3	a1	b4	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
4	a1	b5	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
...	...	...	...	...	...	...	...	...
19995	a100	b196	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19996	a100	b197	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19997	a100	b198	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19998	a100	b199	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19999	a100	b200	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149

	B1	B2	B3	B4	B5	B6	target
0	0.905394	0.627141	0.004124	0.495460	0.677577	0.208311	negative
1	0.224414	0.228480	0.016449	0.815453	0.078746	0.322951	negative
2	0.973814	0.019673	0.702359	0.935972	0.218404	0.329397	negative
3	0.201055	0.989632	0.955579	0.835887	0.658748	0.769800	negative
4	0.226728	0.434467	0.392817	0.756376	0.728210	0.284380	negative
...	...	...	...	...	...	...	...
19995	0.101379	0.985636	0.209741	0.109177	0.153338	0.845952	negative
19996	0.749521	0.252868	0.423237	0.166023	0.371363	0.610882	negative
19997	0.014416	0.248335	0.593786	0.911374	0.425073	0.918091	negative
19998	0.228933	0.846219	0.739485	0.118318	0.019194	0.367549	negative
19999	0.083377	0.793793	0.002202	0.738356	0.321049	0.847341	negative

Figura 2.7: Elementos do dicionário `class_community_C4_low` associados à chave `['data']`, para o nível de observação baixo (10%).

A Figura 2.8 ilustra os elementos do dicionário `class_community_C4_mid` para o nível médio de observação (25%)

{'data':

	X	Y	A1	A2	A3	A4	A5	A6
0	a1	b1	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
1	a1	b2	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
2	a1	b3	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
3	a1	b4	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
4	a1	b5	0.798305	0.867304	0.389030	0.377349	0.724504	0.298213
...	...	...	...	...	...	...	...	...
19995	a100	b196	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19996	a100	b197	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19997	a100	b198	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19998	a100	b199	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149
19999	a100	b200	0.621842	0.691329	0.109172	0.332094	0.391581	0.883149

	B1	B2	B3	B4	B5	B6	target
0	0.905394	0.627141	0.004124	0.495460	0.677577	0.208311	negative
1	0.224414	0.228480	0.016449	0.815453	0.078746	0.322951	negative
2	0.973814	0.019673	0.702359	0.935972	0.218404	0.329397	negative
3	0.201055	0.989632	0.955579	0.835887	0.658748	0.769800	negative
4	0.226728	0.434467	0.392817	0.756376	0.728210	0.284380	negative
...	...	...	...	...	...	...	...
19995	0.101379	0.985636	0.209741	0.109177	0.153338	0.845952	negative
19996	0.749521	0.252868	0.423237	0.166023	0.371363	0.610882	negative
19997	0.014416	0.248335	0.593786	0.911374	0.425073	0.918091	negative
19998	0.228933	0.846219	0.739485	0.118318	0.019194	0.367549	negative
19999	0.083377	0.793793	0.002202	0.738356	0.321049	0.847341	negative

Figura 2.8: Elementos do dicionário `class_community_C4_mid` associados à chave `['data']`, para o nível de observação médio (25%).

A Figura 2.9 apresenta os elementos do dicionário `class_community_C4_high` para o nível alto de observação (50%), significando que a coluna 'target' terá mais proporções desta classe.

```
{'data':           X     Y        A1        A2        A3        A4        A5        A6
 0        a1    b1  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 1        a1    b2  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 2        a1    b3  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 3        a1    b4  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 4        a1    b5  0.798305  0.867304  0.389030  0.377349  0.724504  0.298213
 ...     ...   ...       ...       ...       ...       ...       ...       ...
 19995  a100  b196  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149
 19996  a100  b197  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149
 19997  a100  b198  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149
 19998  a100  b199  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149
 19999  a100  b200  0.621842  0.691329  0.109172  0.332094  0.391581  0.883149

              B1        B2        B3        B4        B5        B6    target
 0      0.905394  0.627141  0.004124  0.495460  0.677577  0.208311  negative
 1      0.224414  0.228480  0.016449  0.815453  0.078746  0.322951  negative
 2      0.973814  0.019673  0.702359  0.935972  0.218404  0.329397  negative
 3      0.201055  0.989632  0.955579  0.835887  0.658748  0.769800  negative
 4      0.226728  0.434467  0.392817  0.756376  0.728210  0.284380  negative
 ...         ...       ...       ...       ...       ...       ...       ...
 19995  0.101379  0.985636  0.209741  0.109177  0.153338  0.845952  negative
 19996  0.749521  0.252868  0.423237  0.166023  0.371363  0.610882  negative
 19997  0.014416  0.248335  0.593786  0.911374  0.425073  0.918091  negative
 19998  0.228933  0.846219  0.739485  0.118318  0.019194  0.367549  negative
 19999  0.083377  0.793793  0.002202  0.738356  0.321049  0.847341  negative
```

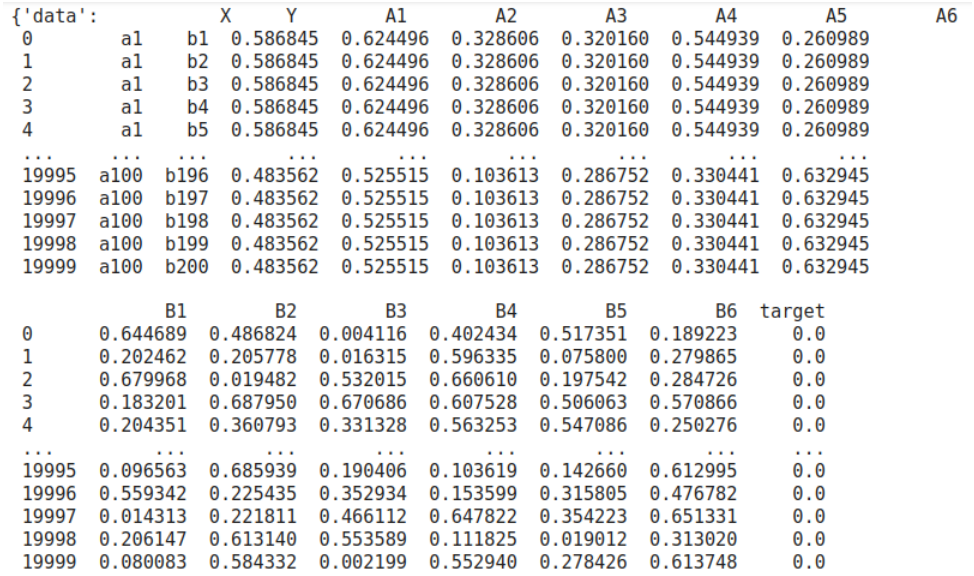
Figura 2.9: Elementos do dicionário `class_community_C4_high` associados à chave `['data']`, para o nível de observação alto (50%).

Para o modelo de regressão, os valores de interação de Poisson são ajustados para os mesmos níveis de tempo de observação. A comunidade de regressão é criada para cada nível

```
class_community_R4_low = createCommunity(X, Y, obsR['low'])
class_community_R4_mid = createCommunity(X, Y, obsR['mid'])
class_community_R4_high = createCommunity(X, Y, obsR['high'])
```

As imagens a seguir apresentam os elementos dos dicionários `class_community_R4_low`, `class_community_R4_mid` e `class_community_R4_high`, associados à chave `['data']` para os diferentes níveis de tempo de observação.

A Figura 2.10 mostra os elementos do dicionário `class_community_R4_low` para o nível baixo de observação (10%)

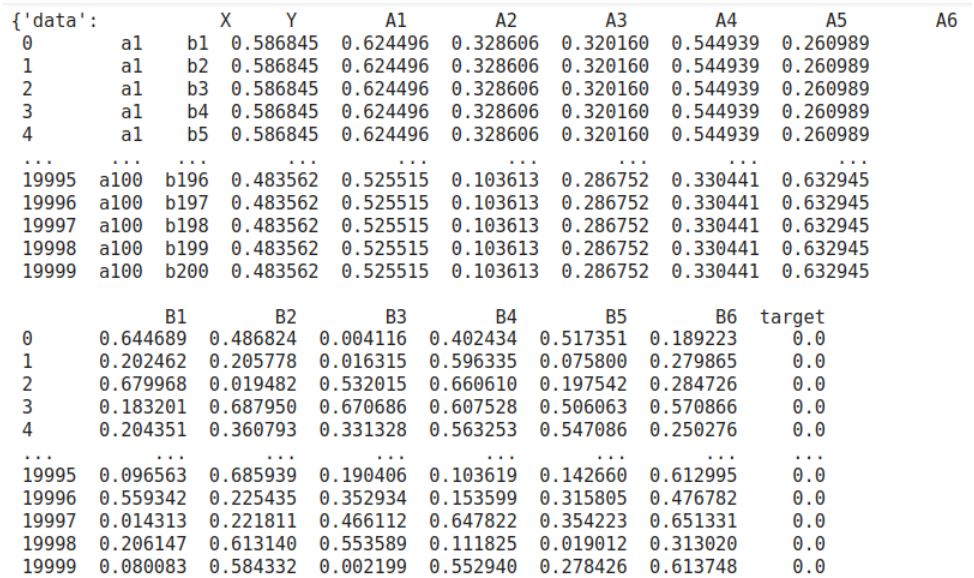


```
{'data':           X     Y        A1        A2        A3        A4        A5        A6
 0        a1    b1  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 1        a1    b2  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 2        a1    b3  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 3        a1    b4  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 4        a1    b5  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 ...     ...   ...       ...       ...       ...       ...       ...       ...
 19995  a100  b196  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945
 19996  a100  b197  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945
 19997  a100  b198  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945
 19998  a100  b199  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945
 19999  a100  b200  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945

              B1        B2        B3        B4        B5        B6  target
 0      0.644689  0.486824  0.004116  0.402434  0.517351  0.189223     0.0
 1      0.202462  0.205778  0.016315  0.596335  0.075800  0.279865     0.0
 2      0.679968  0.019482  0.532015  0.660610  0.197542  0.284726     0.0
 3      0.183201  0.687950  0.670686  0.607528  0.506063  0.570866     0.0
 4      0.204351  0.360793  0.331328  0.563253  0.547086  0.250276     0.0
 ...         ...       ...       ...       ...       ...       ...     ...
 19995  0.096563  0.685939  0.190406  0.103619  0.142660  0.612995     0.0
 19996  0.559342  0.225435  0.352934  0.153599  0.315805  0.476782     0.0
 19997  0.014313  0.221811  0.466112  0.647822  0.354223  0.651331     0.0
 19998  0.206147  0.613140  0.553589  0.111825  0.019012  0.313020     0.0
 19999  0.080083  0.584332  0.002199  0.552940  0.278426  0.613748     0.0
```

Figura 2.10: Elementos do dicionário `class_community_R4_low` associados à chave `['data']`, para o nível de observação baixo (10%).

A Figura 2.11 ilustra os elementos do dicionário `class_community_R4_mid` para o nível médio de observação (25%)



```
{'data':           X     Y        A1        A2        A3        A4        A5        A6
 0        a1    b1  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 1        a1    b2  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 2        a1    b3  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 3        a1    b4  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 4        a1    b5  0.586845  0.624496  0.328606  0.320160  0.544939  0.260989
 ...     ...   ...       ...       ...       ...       ...       ...       ...
 19995  a100  b196  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945
 19996  a100  b197  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945
 19997  a100  b198  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945
 19998  a100  b199  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945
 19999  a100  b200  0.483562  0.525515  0.103613  0.286752  0.330441  0.632945

              B1        B2        B3        B4        B5        B6  target
 0      0.644689  0.486824  0.004116  0.402434  0.517351  0.189223     0.0
 1      0.202462  0.205778  0.016315  0.596335  0.075800  0.279865     0.0
 2      0.679968  0.019482  0.532015  0.660610  0.197542  0.284726     0.0
 3      0.183201  0.687950  0.670686  0.607528  0.506063  0.570866     0.0
 4      0.204351  0.360793  0.331328  0.563253  0.547086  0.250276     0.0
 ...         ...       ...       ...       ...       ...       ...     ...
 19995  0.096563  0.685939  0.190406  0.103619  0.142660  0.612995     0.0
 19996  0.559342  0.225435  0.352934  0.153599  0.315805  0.476782     0.0
 19997  0.014313  0.221811  0.466112  0.647822  0.354223  0.651331     0.0
 19998  0.206147  0.613140  0.553589  0.111825  0.019012  0.313020     0.0
 19999  0.080083  0.584332  0.002199  0.552940  0.278426  0.613748     0.0
```

Figura 2.11: Elementos do dicionário `class_community_R4_mid` associados à chave `['data']`, para o nível de observação médio (25%).

A Figura 2.12 apresenta os elementos do dicionário `class_community_R4_high` para o nível alto de observação (50%), significando que a coluna 'target' terá mais proporções desta classe.

{'data':

	X	Y	A1	A2	A3	A4	A5	A6
0	a1	b1	0.586845	0.624496	0.328606	0.320160	0.544939	0.260989
1	a1	b2	0.586845	0.624496	0.328606	0.320160	0.544939	0.260989
2	a1	b3	0.586845	0.624496	0.328606	0.320160	0.544939	0.260989
3	a1	b4	0.586845	0.624496	0.328606	0.320160	0.544939	0.260989
4	a1	b5	0.586845	0.624496	0.328606	0.320160	0.544939	0.260989
...	...	...	...	...	...	...	...	...
19995	a100	b196	0.483562	0.525515	0.103613	0.286752	0.330441	0.632945
19996	a100	b197	0.483562	0.525515	0.103613	0.286752	0.330441	0.632945
19997	a100	b198	0.483562	0.525515	0.103613	0.286752	0.330441	0.632945
19998	a100	b199	0.483562	0.525515	0.103613	0.286752	0.330441	0.632945
19999	a100	b200	0.483562	0.525515	0.103613	0.286752	0.330441	0.632945

	B1	B2	B3	B4	B5	B6	target
0	0.644689	0.486824	0.004116	0.402434	0.517351	0.189223	0.0
1	0.202462	0.205778	0.016315	0.596335	0.075800	0.279865	0.0
2	0.679968	0.019482	0.532015	0.660610	0.197542	0.284726	0.0
3	0.183201	0.687950	0.670686	0.607528	0.506063	0.570866	0.0
4	0.204351	0.360793	0.331328	0.563253	0.547086	0.250276	0.0
...	...	...	...	...	...	...	...
19995	0.096563	0.685939	0.190406	0.103619	0.142660	0.612995	0.0
19996	0.559342	0.225435	0.352934	0.153599	0.315805	0.476782	0.0
19997	0.014313	0.221811	0.466112	0.647822	0.354223	0.651331	0.0
19998	0.206147	0.613140	0.553589	0.111825	0.019012	0.313020	0.0
19999	0.080083	0.584332	0.002199	0.552940	0.278426	0.613748	0.0

Figura 2.12: Elementos do dicionário `class_community_R4_high` associados à chave `['data']`, para o nível de observação alto (50%).

Explicação dos Parâmetros:

- `low, mid, high`: Níveis de observação ajustados para gerar diferentes proporções de classes (10%, 25%, 50%);
- `binar()`: Função para calcular interações binárias para classificação;
- `poisson()`: Função para calcular interações de Poisson para regressão;
- `createCommunity(X, Y, obs)`: Criação da comunidade a partir dos dados simulados (X e Y) e das interações observadas (`obs`).

2.1.5 Dados reais

Para validar as simulações apresentadas no Capítulo 1 e avaliar o desempenho dos modelos de aprendizado de máquina com dados reais, utilizamos uma base de dados disponível publicamente na *internet*, intitulada `plant_pollinator_database`. Durante este processo, realizamos transformações necessárias, incluindo a imputação de valores nulos utilizando a técnica `MissForest`, e dividimos o conjunto de dados em dois novos subconjuntos: um contendo características morfológicas de plantas e outro com características morfológicas de polinizadores.

Adicionalmente, empregamos a função `crosstab` da biblioteca `pandas` para gerar uma matriz binária. Nesta matriz, o valor 1 indica a interação entre duas espécies, enquanto o valor 0 indica a ausência de interação. A partir dessa matriz, obtemos os parâmetros `X`, `Y` e `binary_interaction`, mencionados nas seções anteriores, que são utilizados na análise dos dados reais.

O resultado dessa transformação foi a geração do dicionário `class_community`, com a chave `['data']`, representando as interações entre as espécies.

A seguir, apresentamos as figuras que ilustram os elementos contidos no dicionário `dados`, associados à chave `['data']`.

A Figura 2.14 apresenta uma visualização geral dos elementos do dicionário `class_community`, mostrando os dados binários relacionados às interações entre plantas e polinizadores

```
{'data':                          X                         Y       type   season  \
 0      Vaccinium_corymbosum          Andrena_wilkella  arboreous  sprisum
 1      Vaccinium_corymbosum       Andrena_barbilabris  arboreous  sprisum
 2      Vaccinium_corymbosum         Andrena_cineraria  arboreous  sprisum
 3      Vaccinium_corymbosum          Andrena_flavipes  arboreous  sprisum
 4      Vaccinium_corymbosum           Andrena_gravida  arboreous  sprisum
 ...                     ...                       ...        ...      ...
 20475        Coffea_arabica  Dolichovespula_norwegica  arboreous   spring
 20476        Coffea_arabica   Dolichovespula_saxonica  arboreous   spring
 20477        Coffea_arabica        Bembecinus_tridens  arboreous   spring
 20478        Coffea_arabica          Vespula_vulgaris  arboreous   spring
 20479        Coffea_arabica     Philanthus_triangulum  arboreous   spring

         diameter  corolla  colour  nectar      b.system s.pollination  \
 0      20.560050      0.0     4.0     1.0       insects            no
 1      20.560050      0.0     4.0     1.0       insects            no
 2      20.560050      0.0     4.0     1.0       insects            no
 3      20.560050      0.0     4.0     1.0       insects            no
 4      20.560050      0.0     4.0     1.0       insects            no
 ...          ...      ...     ...     ...           ...           ...
 20475  19.023909      2.0     4.0     1.0  wind/insects            no
 20476  19.023909      2.0     4.0     1.0  wind/insects            no
 20477  19.023909      2.0     4.0     1.0  wind/insects            no
 20478  19.023909      2.0     4.0     1.0  wind/insects            no
 20479  19.023909      2.0     4.0     1.0  wind/insects            no
```

Figura 2.13: Elementos do dicionário `class_community` associados à chave `['data']`, mostrando as interações binárias entre plantas e polinizadores.

A Figura 2.14 é complementada pela segunda imagem, que fornece uma visão detalhada dos dados resultantes da análise de interações entre as espécies, após as transformações aplicadas no conjunto de dados real

	inflorescence	composite	guild	tongue	body	sociality	\
0	yes	no	ANDRENIDAE	5.419557	10.5	0.0	
1	yes	no	ANDRENIDAE	5.584353	10.5	0.0	
2	yes	no	ANDRENIDAE	10.704976	12.0	0.0	
3	yes	no	ANDRENIDAE	6.035945	11.0	0.0	
4	yes	no	ANDRENIDAE	9.057516	13.0	0.0	
...	...	...	...	...	...	...	
20475	yes	no	WASPS	8.693142	14.5	1.0	
20476	yes	no	WASPS	5.138402	13.0	1.0	
20477	yes	no	WASPS	4.585817	9.0	0.0	
20478	yes	no	WASPS	7.557467	12.5	1.0	
20479	yes	no	WASPS	5.259339	15.0	0.0	

	feeding	target
0	2.000000	negative
1	0.000000	negative
2	0.000000	negative
3	0.000000	negative
4	0.000000	negative
...	...	...
20475	0.000000	negative
20476	0.000000	positive
20477	0.217665	negative
20478	0.000000	negative
20479	0.000000	negative

Figura 2.14: Visualização detalhada dos dados processados, com as interações binárias entre as espécies de plantas e polinizadores.

2.2 Caso 'Classification'

A função `main_classification` é responsável por orquestrar o processo de treinamento e avaliação de diversos modelos de aprendizado de máquina para problemas de classificação binária. Esta seção descreve as partes constituintes da função, incluindo funções auxiliares, os modelos utilizados, e as técnicas de pré-processamento. Esta adaptação foi feita com base no código original `Prediction.Rmd`, contido no repositório do GitHub do artigo, com modificações para replicar os modelos selecionados pelos autores no artigo utilizando `Python`.

2.2.1 Biblioteca `scikit-learn`

A biblioteca `scikit-learn`, desenvolvida por (PEDREGOSA et al., 2011), é uma das ferramentas mais populares de algoritmos de aprendizado de máquina em `Python`. Ela será utilizada neste trabalho para aplicar uma variedade de técnicas de pré-processamento de dados, como normalização e codificação de variáveis, além de treinar, testar e avaliar os modelos de aprendizado de máquina. Esta biblioteca oferece uma interface simples e eficiente, permitindo a construção de pipelines completos para análise e modelagem de dados.

2.2.2 Funções Auxiliares

Função `train_and_evaluate_with_cv`

Esta função realiza o treinamento e avaliação de um modelo utilizando validação cruzada estratificada (*Stratified K-Fold*). O *Stratified K-Fold* garante que a divisão dos dados mantenha a mesma distribuição de classes em cada uma das k partições. Esse processo é especialmente importante em problemas de classificação desbalanceada, onde as classes têm distribuições diferentes, para garantir que todas as classes estejam representadas de forma proporcional em cada partição.

Os passos principais incluem:

- **Divisão dos dados**: Os dados são divididos em k partições estratificadas, garantindo que a proporção de classes seja mantida em cada divisão;
- **Treinamento do modelo**: Para cada partição, o modelo é treinado no conjunto de treinamento ($k-1$ partições) e avaliado no conjunto de validação (1 partição);
- **Predições**: Dependendo do modelo, as predições podem ser probabilísticas (`predict_proba`) ou diretas (`predict`);
- **Cálculo de métricas**: As métricas acurácia, precisão, *recall*, e *f1-score* são calculadas para cada divisão. A média dessas métricas é retornada ao final. Essas métricas são usadas para avaliar a performance do modelo em termos de sua capacidade de prever corretamente as classes. A acurácia mede a porcentagem de previsões corretas, *f1-score* indica a exatidão das previsões positivas, *recall* reflete a capacidade do modelo de identificar todas as instâncias positivas e *f1-score* é a média harmônica entre precisão e *recall*, fornecendo uma medida balanceada de performance. Mais detalhes sobre as métricas serão fornecidos adiante.

Parâmetros:

- `model`: O modelo de aprendizado de máquina a ser avaliado;
- `X_train`: As características (*features*) do conjunto de treinamento;
- `y_train`: Os rótulos (*labels*) do conjunto de treinamento;
- `cv`: O número de divisões (*folds*) na validação cruzada. Valor padrão: 5.

Retorno: Um dicionário contendo as médias das métricas calculadas.

Métricas de Avaliação

As métricas de avaliação são usadas para medir a performance de um modelo de aprendizado de máquina. As principais métricas para problemas de classificação são:

- **Acurácia**: A acurácia é a proporção de previsões corretas (verdadeiros positivos e verdadeiros negativos) em relação ao total de exemplos. A fórmula é

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN}.$$

onde

  - TP (*True Positives*) = Verdadeiros Positivos: casos em que o modelo previu a classe positiva corretamente;
  - TN (*True Negatives*) = Verdadeiros Negativos: casos em que o modelo previu a classe negativa corretamente;
  - FP (*False Positives*) = Falsos Positivos: casos em que o modelo previu a classe positiva incorretamente;
  - FN (*False Negatives*) = Falsos Negativos: casos em que o modelo previu a classe negativa incorretamente.

- **Precisão**: A precisão mede a exatidão das previsões positivas. Ou seja, de todas as instâncias classificadas como positivas, qual a proporção que realmente são positivas

$$\text{Precisão} = \frac{TP}{TP + FP}.$$

- ***Recall* (Sensibilidade)**: O *recall*, também chamado de sensibilidade, mede a capacidade do modelo de identificar todas as instâncias positivas. Ou seja, de todas as instâncias que são realmente positivas, qual a proporção que o modelo conseguiu identificar corretamente

$$\textit{Recall} = \frac{TP}{TP + FN}.$$

- ***F1-Score***: O *f1-score* é a média harmônica entre precisão e *recall*. Ele é usado para obter uma medida balanceada de desempenho, especialmente quando as classes estão desbalanceadas

$$\textit{F1-Score} = 2 \times \frac{\text{precisão} \times \textit{recall}}{\text{precisão} + \textit{recall}} = \frac{2TP}{2TP + FP + FN}.$$

  Essas métricas são fundamentais para avaliar a performance de um modelo, especialmente em cenários de classificação desbalanceada, onde a acurácia por si só pode ser enganosa.

Funções de Redes Neurais

A seguir, detalhamos as funções utilizadas para definir arquiteturas de redes neurais.

Função `build_dnn`: Define uma Rede Neural Profunda (*Deep Neural Network*, DNN) para classificação binária. Veja a Seção 2.2.5 para detalhes.

Função `build_cnn`: Define uma Rede Neural Convolucional (*Convolutional Neural Network*, CNN) para classificação de dados sequenciais. Veja a Seção 2.2.5 para detalhes.

Função `build_negbin_dnn`: Implementa uma DNN configurada para tarefas de regressão com distribuição de Poisson. Veja a Seção 2.5.5 para detalhes.

2.2.3 Técnicas de Pré-Processamento

A função `main_classification` realiza uma série de passos de pré-processamento:

Divisão dos Dados (`X` e `y`)

Os dados são divididos em `X` (características) e `y` (rótulos). Para este projeto:

- `X`: Todas as colunas, exceto a última;
- `y`: A última coluna, com os rótulos convertidos para valores binários (1 para `positive`, 0 para `negative`).

Normalização e Codificação

O pré-processamento das características (`X`) é realizado com as seguintes técnicas utilizando a biblioteca `scikit-learn`.

- **Codificação *One-Hot***: A codificação *One-Hot* é utilizada para transformar variáveis categóricas em uma representação binária. Cada categoria é representada por uma coluna separada, onde o valor da variável para uma categoria é representado por 1, enquanto as outras categorias são representadas por 0. Isso é necessário, pois muitos algoritmos de aprendizado de máquina não conseguem trabalhar diretamente com dados categóricos. O `OneHotEncoder` do `scikit-learn` é utilizado para realizar essa transformação. Por exemplo, se uma variável categórica tiver as categorias {'A', 'B', 'C'}, ela será transformada em três colunas:

  $$\text{Categoria 'A'} \quad \text{Categoria 'B'} \quad \text{Categoria 'C'}.$$

  Para um exemplo com a categoria 'A', a codificação será: $[1, 0, 0]$.

- **Normalização**: A normalização é aplicada a variáveis numéricas para escalá-las de forma que todas as variáveis tenham uma distribuição com média zero e desvio padrão igual a um, o que facilita a convergência dos modelos de aprendizado de máquina. Isso é importante para algoritmos que dependem da distância entre as observações, como regressão logística, k-vizinhos mais próximos (KNN) e máquinas de vetores de suporte (SVM). O `StandardScaler` do `scikit-learn` realiza a normalização, que é dada pela seguinte fórmula:

  $$x_{\text{normalizado}} = \frac{x - \mu}{\sigma},$$

  onde

  - x é o valor original da variável;
  - μ é a média da variável;
  - σ é o desvio padrão da variável.

Aumento de Dados com SMOTE

O *Synthetic Minority Oversampling Technique* (SMOTE) é uma técnica usada para lidar com o desbalanceamento de classes em problemas de aprendizado de máquina. Ele gera exemplos sintéticos da classe minoritária por meio da interpolação entre instâncias existentes e seus k-vizinhos mais próximos, equilibrando a distribuição das classes no conjunto de treinamento. Isso reduz a tendência dos modelos de favorecer a classe majoritária, melhorando a performance em classes menos representadas.

2.3 Função `main_classification`

A função `main_classification` é responsável por aplicar diferentes modelos de classificação a um conjunto de dados, realizando pré-processamento, treinamento, avaliação e coleta de resultados. Abaixo está uma descrição detalhada de cada parte do código, os parâmetros utilizados, e os modelos aplicados.

2.3.1 Parâmetros da Função

A função recebe dois parâmetros principais:

- `class_community`: Um dicionário que contém o tipo do problema (`'type'`) e os dados (`'data'`). A chave `'type'` deve ser `'Classification'` para que o processo de treinamento de modelos seja realizado. O valor de `'data'` é um conjunto de dados com os dados de entrada;
- `sampling_strategy`: Define a estratégia de amostragem para balanceamento das classes, caso o conjunto de dados esteja desequilibrado. Se `None`, o `SMOTE` não será aplicado.

2.3.2 Pré-processamento dos Dados

Primeiramente, o conjunto de dados é dividido em variáveis independentes (`X`) e dependentes (`y`). A variável `X` contém todas as colunas, exceto a última (que será a variável de saída), e a variável `y` contém a última coluna, que é convertida para valores numéricos (`'positive'` é mapeado para 1.0 e `'negative'` para 0.0).

Depois, as colunas categóricas e numéricas são identificadas, permitindo que as transformações adequadas sejam aplicadas. As colunas categóricas são transformadas utilizando a codificação `OneHotEncoder`, e as colunas numéricas são normalizadas com o `StandardScaler`.

A transformação é realizada por meio de um `ColumnTransformer`, que aplica o `OneHotEncoder` às colunas categóricas e o `StandardScaler` às numéricas. A função `fit_transform` é utilizada para aplicar as transformações, gerando o conjunto de dados transformado (`X_transformed`).

Divisão em Treinamento e Teste

Após o pré-processamento, os dados são divididos em conjuntos de treinamento e teste utilizando a função `train_test_split`. O conjunto de teste corresponde a 30% dos dados, com a semente de aleatoriedade definida para garantir resultados reprodutíveis.

Aplicação do SMOTE

Se a estratégia de amostragem (`sampling_strategy`) for fornecida, a técnica de balanceamento `SMOTE` é aplicada aos dados de treinamento para os casos em que a classe majoritária representa mais de 95% das amostras, utilizando um `sampling_strategy` de 0.75. O SMOTE gera exemplos sintéticos da classe minoritária para balancear as classes. O número de vizinhos (`k_neighbors`) é definido como 2.

2.3.3 Treinamento e Avaliação dos Modelos

A função aplica seis modelos diferentes de aprendizado de máquina e avalia seu desempenho utilizando validação cruzada. Os resultados de cada modelo são armazenados em um conjunto de dados. Os modelos serão descritos a seguir.

1. Redes Neurais Profundas

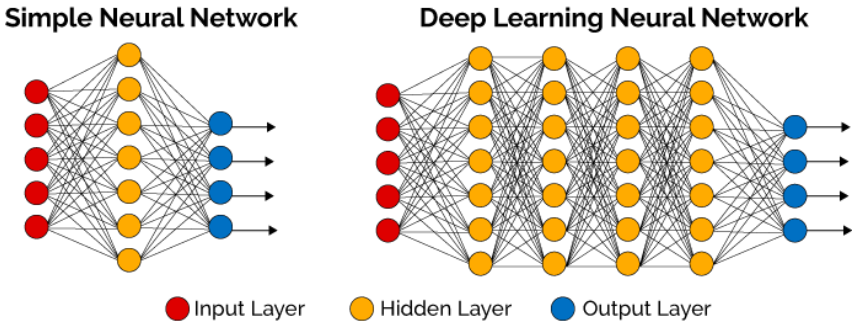


Figura 2.15: Arquitetura das Redes Neurais Profundas. Fonte: https://www.deeplearningbook.com.br/o-que-sao-redes-neurais-artificiais-profundas/

As Redes Neurais Profundas (*Deep Neural Networks*, DNN) são modelos de aprendizado profundo compostos por múltiplas camadas de neurônios. Em uma arquitetura típica de DNN, cada camada é densamente conectada à próxima, o que significa que cada neurônio de uma camada está ligado a todos os neurônios da camada subsequente. Essa conectividade permite que a rede aprenda representações complexas e hierárquicas dos dados, tornando as DNNs particularmente eficazes em tarefas como classificação, regressão e outros problemas que envolvem grandes volumes de dados ou padrões não-lineares.

A saída de uma camada l em uma DNN pode ser calculada de forma semelhante às equações descritas no 6º capítulo do livro *Deep Learning* de (GOODFELLOW; BENGIO; COURVILLE, 2016), que detalham as operações de camadas em redes neurais profundas. A fórmula é expressa como

$$\mathbf{h}^{(l)} = g^{(l)}\left(\mathbf{W}^{(l)T}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right),$$

onde

- $\mathbf{h}^{(l)}$ é o vetor de ativações da camada l, ou seja, as saídas dos neurônios dessa camada após a aplicação da função de ativação $g^{(l)}(\cdot)$. Esse vetor possui uma dimensão igual ao número de neurônios na camada l;
- $\mathbf{W}^{(l)}$ é a matriz de pesos que conecta a camada $l-1$ à camada l. Cada elemento $w_{ij}^{(l)}$ da matriz representa o peso da conexão entre o j-ésimo neurônio da camada $l-1$ e o i-ésimo neurônio da camada l;
- $\mathbf{h}^{(l-1)}$ é o vetor de ativações da camada $l-1$, ou seja, as saídas dos neurônios da camada anterior. Esse vetor é multiplicado pela matriz de pesos $\mathbf{W}^{(l)}$;

- $\mathbf{b}^{(l)}$ é o vetor de vieses associado à camada l, com um valor de viés para cada neurônio da camada. Os vieses são adicionados ao produto $\mathbf{W}^{(l)T}\mathbf{h}^{(l-1)}$ antes de ser aplicada a função de ativação;

- $g^{(l)}(\cdot)$ é a função de ativação da camada l, que introduz não-linearidade ao modelo. Exemplos comuns incluem ReLU (*Rectified Linear Unit*), sigmoide e Tanh.

O processo de treinamento de uma DNN consiste na minimização de uma função de perda que mede a discrepância entre as previsões do modelo e os valores reais dos dados. Para problemas de classificação binária, a função de perda mais utilizada é a entropia cruzada binária (*binary crossentropy*), definida como

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right],$$

onde

- N é o número total de exemplos no conjunto de treinamento;

- y_i é o rótulo real para o i-ésimo exemplo, com $y_i \in \{0, 1\}$;

- $\hat{y}_i$ é a probabilidade prevista pelo modelo para o i-ésimo exemplo.

A arquitetura da DNN é definida com o uso de ferramentas como `KerasClassifier`, que utiliza uma função personalizada, `build_dnn`, para especificar o número de camadas, a quantidade de neurônios por camada, a função de ativação e outros parâmetros importantes. Durante o treinamento, o modelo foi configurado para treinar por 15 épocas, com um tamanho de lote (*batch size*) de 32, o que significa que os pesos do modelo são atualizados após o processamento de cada grupo de 32 exemplos.

2. Redes Neurais Convolucionais

As Redes Neurais Convolucionais (*Convolutional Neural Networks*, CNNs) são modelos de aprendizado profundo especialmente projetados para capturar padrões espaciais e temporais em dados estruturados, como imagens, vídeos ou séries temporais. O principal diferencial das CNNs em relação às redes neurais tradicionais é o uso de operações de convolução, que permitem ao modelo aprender a identificar características locais e hierárquicas nos dados, como bordas, texturas ou padrões de movimento.

A operação de convolução, que é o núcleo das CNNs, pode ser expressa pela seguinte fórmula, adaptada de (GOODFELLOW; BENGIO; COURVILLE, 2016)

$$z_{i,j}^{(l)} = \sum_{m=1}^{M}\sum_{n=1}^{N}\mathbf{W}^{(l)}[m, n] \cdot \mathbf{a}^{(l-1)}[i + m, j + n] + b^{(l)},$$

onde

- $\mathbf{W}^{(l)}$ é o filtro convolucional (ou kernel), que é um pequeno bloco de pesos responsáveis por detectar características locais na entrada. O filtro tem dimensões $M \times N$, onde M e N são o número de linhas e colunas do filtro. A ideia é que o filtro "deslize"pela imagem ou pela ativação da camada anterior para extrair padrões espaciais, como bordas ou texturas;

- $\mathbf{a}^{(l-1)}$ é a entrada da camada anterior. Pode ser uma imagem original ou as ativações de uma camada anterior. Em uma imagem 2D, $\mathbf{a}^{(l-1)}$ é uma matriz bidimensional que representa os valores de intensidade de cada pixel ou ativação de um mapa de características;

- $b^{(l)}$ é o viés associado à camada convolucional. O viés é adicionado à soma ponderada das ativações locais para permitir flexibilidade no modelo, ajudando a rede a aprender padrões que não dependem de uma origem específica dos dados;

- M e N são as dimensões do filtro convolucional. O filtro é aplicado de forma iterativa sobre a entrada, produzindo uma nova matriz de ativações ($z_{i,j}^{(l)}$) que representa as características extraídas pela camada convolucional.

A operação de convolução varre a entrada ($\mathbf{a}^{(l-1)}$) com um filtro ($\mathbf{W}^{(l)}$), calculando somas ponderadas seguidas pela adição de um viés ($b^{(l)}$). O resultado, $z_{i,j}^{(l)}$, representa uma característica detectada em uma região específica da entrada, permitindo à rede aprender padrões espaciais.

Além disso, após as camadas convolucionais, operações de *pooling*, como *max pooling*, são usadas para reduzir a dimensionalidade das ativações, resumindo informações locais (e.g., valores máximos) e tornando o modelo mais eficiente e menos propenso ao *overfitting*. A saída convolucional é então transformada em um vetor unidimensional (*flattening*), preparando os dados para camadas densas que realizam tarefas como classificação ou regressão.

Nesse sentido, a arquitetura da CNN é definida com o `KerasClassifier`, usando a função personalizada `build_cnn` para especificar parâmetros como número de camadas, tamanho e número de filtros, e funções de ativação. Durante o pré-processamento, o conjunto de dados é remodelado para garantir que a entrada tenha a forma correta, como incluir dimensões de canais em imagens RGB.

3. *Naive Bayes*

O *Naive Bayes* é um modelo probabilístico simples, mas eficaz, que se baseia no Teorema de Bayes para realizar classificações. Esse teorema descreve a probabilidade de uma classe y dada uma entrada $\mathbf{x}$ das variáveis preditoras e é expresso pela seguinte fórmula

$$P(y \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid y)P(y)}{P(\mathbf{x})},$$

onde

- $P(y \mid \mathbf{x})$ é a probabilidade posterior da classe y dado o vetor de características $\mathbf{x}$;

- $P(\mathbf{x} \mid y)$ é a verossimilhança, ou a probabilidade de observar as características $\mathbf{x}$ dado que a classe é y;

- $P(y)$ é a probabilidade a priori da classe y, ou seja, a probabilidade de que uma instância pertença à classe y antes de observar os dados;

- $P(\mathbf{x})$ é a probabilidade marginal das características $\mathbf{x}$, que atua como um fator normalizador, garantindo que a soma das probabilidades de todas as classes seja igual a 1.

A principal suposição do Naive Bayes é a independência condicional entre as variáveis preditoras $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, ou seja, as variáveis são consideradas independentes umas das outras, dado o valor da classe y. Essa suposição simplifica a formulação do modelo, pois a probabilidade conjunta $P(\mathbf{x} \mid y)$ pode ser decomposta como o produto das probabilidades condicionais de cada variável preditora, como

$$P(\mathbf{x} \mid y) = \prod_{i=1}^{n} P(x_i \mid y),$$

onde

- $P(x_i \mid y)$ é a probabilidade condicional de uma variável preditora x_i dado que a classe é y.

Para o caso de variáveis contínuas, como aquelas com valores reais, o modelo `GaussianNB` assume que as variáveis preditoras seguem uma distribuição normal (Gaussiana) para cada classe y. A função de densidade de probabilidade de uma variável x_i condicional à classe y é dada pela fórmula da distribuição normal

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right),$$

onde

- x_i é o valor da variável preditora i-ésima para uma instância de dados.
- μ_y é a média da variável x_i para a classe y, ou seja, o valor central ao redor do qual as observações se agrupam para a classe y;
- σ_y^2 é a variância da variável x_i para a classe y; que mede a dispersão ou espalhamento dos valores de x_i em torno da média μ_y;

O modelo de *Naive Bayes* é treinado utilizando o `GaussianNB`, que calcula as médias e variâncias de cada variável para cada classe e usa essas informações para estimar as probabilidades condicionais $P(x_i \mid y)$. Durante a classificação, o modelo calcula as probabilidades de cada classe y para um conjunto de dados de entrada $\mathbf{x}$ e escolhe a classe com a maior probabilidade posterior $P(y \mid \mathbf{x})$, ou seja, a classe mais provável dado os dados observados.

4. *Gradient Boosting*

O *Gradient Boosting* é um método de aprendizado em conjunto (*ensemble learning*) que visa melhorar a performance de modelos fracos ao combiná-los em um modelo forte. Esse processo de combinação é feito de maneira iterativa, em que cada modelo subsequente tenta corrigir os erros (ou resíduos) cometidos pelo modelo anterior. O modelo é ajustado ao minimizar uma função de perda $L(y, \hat{y})$, que quantifica a discrepância entre as previsões

$\hat{y}$ e os valores reais y. Esse ajuste é feito utilizando o gradiente descendente, de forma a otimizar a função de perda.

A fórmula geral do *Gradient Boosting* para a m-ésima iteração, adaptada de (FRIEDMAN, 2001) é dada por

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \gamma_m h_m(\mathbf{x}),$$

onde

- $F_{m-1}(\mathbf{x})$ é o modelo acumulado até a iteração $m-1$, ou seja, a previsão total considerando todos os modelos anteriores;
- $h_m(\mathbf{x})$ é o modelo fraco (geralmente uma árvore de decisão) ajustado na m-ésima iteração, que visa corrigir os erros do modelo anterior;
- γ_m é o peso otimizado para o modelo h_m, que controla a magnitude da atualização, ajustando a contribuição do modelo h_m no modelo final.

Durante o treinamento, o algoritmo de *Gradient Boosting* adiciona, iterativamente, modelos fracos ao modelo acumulado. Cada modelo é treinado para minimizar a função de perda utilizando um método de gradiente descendente.

O objetivo é ajustar os resíduos do modelo anterior de forma a melhorar o desempenho da previsão. Assim, cada modelo subsequente tenta corrigir os erros cometidos pelos modelos anteriores, sendo, portanto, treinado sobre os erros residuais do modelo anterior.

Os principais parâmetros do `GradientBoostingClassifier` no `scikit-learn` incluem: `n_estimators`, que define o número de estimadores (geralmente árvores de decisão); `max_depth`, que controla a profundidade máxima das árvores para evitar *overfitting* e `learning_rate`, que ajusta o tamanho dos passos em cada iteração.

5. Florestas Aleatórias

As Florestas Aleatórias (*Random Forest*) são um modelo de aprendizado em conjunto (*ensemble learning*) que combina várias árvores de decisão para melhorar a precisão da predição e reduzir a variância do modelo. A ideia central por trás do *Random Forest* é a construção de múltiplas árvores de decisão, cada uma treinada em um subconjunto aleatório dos dados e das características. Isso ajuda a reduzir o risco de *overfitting*, que é comum em modelos individuais de árvore de decisão, proporcionando uma maior generalização do modelo.

A predição final das Florestas Aleatórias é obtida por meio de votação majoritária para problemas de classificação ou pela média das predições das árvores para problemas de regressão. A fórmula para a predição de um modelo de classificação, apresentada no 15º capítulo do livro "*The elements of statistical learning: data mining, inference, and prediction (2nd ed.).*"de (HASTIE; TIBSHIRANI; FRIEDMAN, 2009), é dada por

$$\hat{C}_{rf}^{B}(x) = voto\_majoritário\left\{\hat{C}_b(x)\right\}_1^B,$$

onde

- $\hat{C}_b(x)$ é a classe de predição da b-ésima árvore do *Random Forest* para a entrada $\mathbf{x}$;

- $\hat{C}_{rf}^{B}(x)$ é a predição final da Árvore Aleatória para a instância x, considerando o conjunto de B árvores;
- B é o número total de árvores no modelo;
- $voto\_majoritário\left\{\hat{C}_b(x)\right\}_1^B$ é a função que determina a classe mais frequente entre as predições das B árvores.

Cada árvore nas Floresta Aleatórias é construída utilizando um subconjunto aleatório dos dados de treinamento e das características (*features*). Essa técnica é conhecida como *bagging* (*bootstrap aggregating*), onde os dados de treinamento são amostrados com reposição e as árvores são treinadas de forma independente.

Além disso, em cada divisão de um nó da árvore, um subconjunto aleatório de características é considerado, o que aumenta a diversidade entre as árvores e diminui o risco de *overfitting*.

O modelo utiliza múltiplas árvores de decisão criadas com amostras e subconjuntos de características diferentes, promovendo diversidade e melhorando a robustez e precisão.

No `RandomForestClassifier` do `scikit-learn`, os parâmetros principais incluem: `n_estimators`, `max_depth` e `min_samples_split`. O *Random Forest* é robusto, generaliza bem em dados desbalanceados e fornece uma medida de importância das variáveis, permitindo identificar características relevantes para a predição.

6. *K-Nearest Neighbors* (KNN)

O *K-Nearest Neighbors* (KNN) é um algoritmo de classificação baseado em instâncias, no qual a predição de uma nova observação é feita com base nas classes dos k vizinhos mais próximos desta, medidos por uma métrica de distância. A mais comum dessas métricas é a distância euclidiana, dada pela fórmula

$$d(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{i=1}^{n}(x_i - x_i')^2},$$

onde

- $\mathbf{x}$ e $\mathbf{x}'$ são dois vetores de características;
- x_i e x_i' representam os valores das i-ésimas características dos vetores $\mathbf{x}$ e $\mathbf{x}'$;
- n é o número total de características (ou dimensões) das observações.

A ideia do KNN é simples: para classificar uma nova observação $\mathbf{x}$, o algoritmo calcula a distância entre $\mathbf{x}$ e todas as observações de treinamento, selecionando os k vizinhos mais próximos. A classe da amostra é determinada pela votação majoritária entre os k vizinhos mais próximos. Em caso de empate, diferentes estratégias podem ser adotadas, como escolher a classe com maior número de ocorrências em uma região de maior densidade ou usar uma métrica ponderada pelas distâncias dos vizinhos. A predição de uma nova observação x_0 é baseada nas k observações mais próximas de x_0 no conjunto de treinamento.

A fórmula para determinar se uma observação x está entre os k vizinhos mais próximos de x_0, descrita no 2º capítulo do livro "*The elements of statistical learning: data mining, inference, and prediction (2nd ed.).*"de (HASTIE; TIBSHIRANI; FRIEDMAN, 2009), é dada por

$$K_k(x, x_0) = I\left(\|x - x_0\| \leq \|x_{(k)} - x_0\|\right),$$

onde

- x é uma observação qualquer do conjunto de treinamento;
- x_0 é a nova observação para a qual queremos fazer uma predição;
- $\|x - x_0\|$ é a distância entre a observação x e a nova observação x_0, usualmente calculada com a distância euclidiana;
- $x_{(k)}$ é a k-ésima observação mais próxima de x_0 no conjunto de treinamento;
- $I(S)$ é a função indicadora, que vale 1 se a condição S for verdadeira e 0 caso contrário.

Para maiores dimensões, é necessário fazer alterações para evitar a maldição da dimensionalidade, que se refere à deterioração do desempenho de algoritmos baseados em distâncias, como o KNN, em espaços de alta dimensão, onde as distâncias entre os pontos tornam-se semelhantes, dificultando a distinção entre elementos próximos e distantes. Para mitigar esse problema, são utilizadas estratégias como redução de dimensionalidade (e.g., PCA ou t-SNE), escolha de métricas de distância mais adequadas (e.g., Manhattan ou Mahalanobis), aumento do número de dados para cobrir melhor o espaço e normalização das variáveis para equilibrar as escalas. Essas técnicas ajudam a melhorar o desempenho do KNN em cenários de alta dimensionalidade.

O modelo de KNN é treinado e avaliado utilizando o `KNeighborsClassifier` da biblioteca `scikit-learn`. Os principais parâmetros do modelo incluem:

- `n_neighbors`: O número de vizinhos a serem considerados para cada predição. Este valor é geralmente determinado automaticamente, mas pode ser ajustado conforme necessário.
- `metric`: A métrica de distância a ser utilizada. A distância euclidiana é a métrica padrão, mas outras métricas, como a distância de Manhattan, podem ser usadas, dependendo da natureza do problema e das características dos dados.

Exibição e Retorno dos Resultados

Após a realização do treinamento e avaliação de todos os modelos, os resultados de desempenho são coletados e organizados em um conjunto de dados denominado `results_df`. Este conjunto de dados contém as métricas de avaliação de cada modelo, incluindo acurácia, precisão, *recall* e *f1-score*, permitindo uma comparação clara entre os diferentes algoritmos utilizados.

Cada linha do conjunto de dados corresponde a um modelo distinto, enquanto as colunas representam as métricas de desempenho calculadas para cada um. As métricas

fornecem uma visão abrangente do comportamento de cada modelo em relação aos dados de teste. O formato do conjunto de dados facilita a visualização e análise dos resultados, permitindo identificar qual modelo apresenta o melhor desempenho para o problema em questão.

2.4 Caso 'Regression'

A função `main_regression` é responsável por aplicar diferentes modelos de regressão a um conjunto de dados, realizando pré-processamento, treinamento, avaliação e coleta de resultados. O código descrito aqui é uma adaptação do código em `R Prediction.Rmd`, proposto em (PICHLER et al., 2020), com modificações para replicar os modelos de regressão selecionados pelos autores do artigo.

2.4.1 Funções Auxiliares

Função `reg_train_and_evaluate`

A função `reg_train_and_evaluate` é responsável pelo treinamento e avaliação de um modelo de regressão utilizando validação cruzada (*K-Fold*). Para cada divisão dos dados, o modelo é treinado e avaliado em termos de correlação de Spearman entre as previsões e os valores reais. Os resultados são retornados como a média das correlações de Spearman.

Parâmetros:

- `model`: O modelo de aprendizado de máquina a ser treinado;
- `X_train`: As características (*features*) do conjunto de treinamento;
- `y_train`: Os valores de resposta do conjunto de treinamento;
- `cv`: O número de divisões (*folds*) para validação cruzada. Valor padrão: 5.

Retorno: Um dicionário contendo o valor médio da correlação de Spearman.

2.4.2 Métrica de Avaliação Spearman

A métrica de avaliação Spearman mede a força e direção da relação monotônica entre duas variáveis, baseada nas classificações das variáveis, ao invés de seus valores absolutos. Diferente da correlação de Pearson, que avalia relações lineares, o coeficiente de Spearman (ρ) é calculado pela fórmula:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)},$$

onde d_i é a diferença nas classificações das variáveis e n é o número de observações.

A métrica é usada para avaliar previsões em regressão sem assumir relação linear. O coeficiente ρ varia de -1 a +1:

- $\rho = +1$: Correlação monotônica perfeita positiva;
- $\rho = -1$: Correlação monotônica perfeita negativa;
- $\rho = 0$: Nenhuma correlação monotônica;
- $0 < \rho < 1$: Correlação monotônica positiva;
- $-1 < \rho < 0$: Correlação monotônica negativa.

Para avaliar a performance de um modelo, as previsões e os valores reais são ordenados para calcular as classificações. Em seguida, a diferença entre as classificações é utilizada para calcular o coeficiente ρ em cada partição de validação, e o valor médio de ρ é calculado ao longo de todas as partições.

2.5 Função `main_regression`

2.5.1 Parâmetros da Função

A função recebe dois parâmetros principais:

- `class_community`: Um dicionário que contém o tipo do problema (`'type'`) e os dados (`'data'`). A chave `'type'` deve ser `'Regression'` para que o processo de treinamento de modelos seja realizado. O valor de `'data'` é um conjunto de dados com os dados de entrada;
- `sampling_strategy`: Define a estratégia de amostragem para balanceamento das classes, caso o conjunto de dados esteja desequilibrado. Se `None`, o `SMOTE` não será aplicado.

2.5.2 Pré-processamento dos Dados

Semelhante ao pré-processamento realizado no Caso 'Classification' na seção 2.3.2, seguiremos a mesma abordagem, com a adição da transformação da variável dependente (`y`). Para isso, utilizaremos o `RobustScaler` para escalonar `y`, especialmente no caso de haver *outliers* nos dados. Essa técnica, que se baseia na mediana e no intervalo interquartil, é menos sensível a valores extremos. Em problemas de regressão, onde os *outliers* podem ter um impacto significativo no modelo, essa transformação contribui para aumentar a robustez e melhorar a qualidade das previsões.

Divisão em Treinamento e Teste

Após o pré-processamento, da mesma forma que os casos de classificação, os dados são divididos em conjuntos de treinamento e teste utilizando a função `train_test_split`. O conjunto de teste corresponde a 30% dos dados, com a semente de aleatoriedade definida para garantir resultados reprodutíveis.

Aplicação do SMOTE

De maneira análoga à abordagem utilizada em `main_classification`, o parâmetro `sampling_strategy` foi configurado como um dicionário para equilibrar as classes menos representadas em cenários de múltiplas classes. Esse procedimento evidencia um desafio importante a ser aprimorado neste projeto, uma vez que a abordagem atual não se mostra suficientemente robusta ou generalizável.

2.5.3 Treinamento e Avaliação dos Modelos

A função aplica diferentes modelos de regressão e avalia seu desempenho utilizando validação cruzada. Os resultados de cada modelo são armazenados em um conjunto de dados. Os modelos de regressão utilizados serão descritos a seguir.

1. Regressão *Gradient Boosting*

O *Gradient Boosting Regression* (GBR) é uma técnica de aprendizado supervisionado que utiliza uma sequência de modelos fracos, geralmente árvores de regressão, para criar um modelo robusto que consegue realizar previsões precisas. De acordo com o artigo de (FRIEDMAN, 2001), "*Greedy Function Approximation: A Gradient Boosting Machine*", Cada modelo individual no Gradient Boosting é uma árvore de decisão de regressão com J nós terminais. Essas árvores são usadas para representar funções aditivas, onde cada árvore contribui com uma parte do valor final da predição. A função preditiva para uma árvore de regressão é dada por

$$h(x) = \sum_{j=1}^{J} b_j \cdot 1(x \in R_j).$$

Aqui, R_j são as regiões disjuntas que cobrem o espaço das variáveis preditoras x. A função indicadora $1(x \in R_j)$ vale 1 quando x está na região R_j, e 0 caso contrário. Os valores de b_j são os coeficientes da árvore que representam a previsão para cada região R_j.

Em cada iteração do algoritmo de boosting, o modelo previamente ajustado, $F_{m-1}(x)$, é atualizado com a adição de uma nova árvore de regressão que tenta corrigir os erros do modelo anterior. A fórmula de atualização do modelo é dada por

$$F_m(x) = F_{m-1}(x) + \rho_m \sum_{j=1}^{J} b_{jm} \cdot 1(x \in R_{jm}),$$

onde

- R_{jm} representa as regiões definidas pelos nós terminais da árvore na iteração m;
- b_{jm} são os coeficientes ajustados da árvore na iteração m, calculados pela técnica de mínimos quadrados;
- ρ_m é um fator de escala encontrado por meio de uma busca de linha, que determina a contribuição da árvore para o modelo final.

No `GradientBoostingRegressor` do `scikit-learn`, cada árvore é ajustada para prever as pseudo-respostas ($\tilde{y}_i$), que representam os resíduos entre os valores reais (y_i) e as previsões do modelo anterior ($F_{m-1}(x_i)$), minimizando o erro quadrático. O parâmetro ρ_m, calculado por busca de linha, ajusta a contribuição de cada árvore no modelo final.

Os principais parâmetros incluem: `n_estimators`, que define o número de árvores e controla a capacidade do modelo, e `max_depth`, que limita a profundidade das árvores para evitar *overfitting* ou *underfitting*.

2. Florestas Aleatórias para Regressão

As Florestas Aleatórias para regressão são uma técnica de aprendizado de máquina baseada em um *ensemble* de árvores de decisão. De acordo com o Capítulo 15 do livro "*The elements of statistical learning: data mining, inference, and prediction (2nd ed.)*.", a previsão final é dada pela média de acordo com a expressão

$$\hat{f}_{rf}^{B}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x;\Theta_b),$$

onde

- $\hat{f}_{rf}^{B}(x)$ é a previsão do modelo;
- $T_b(x;\Theta_b)$ é a previsão da b-ésima árvore, com Θ_b representando seus parâmetros (variáveis de divisão, pontos de corte e valores dos nós terminais).

Cada árvore é treinada com um subconjunto aleatório dos dados e seleciona um subconjunto aleatório de características para cada divisão de nó. Isso aumenta a diversidade entre as árvores e reduz a correlação entre elas. A média das previsões reduz a variância do modelo, tornando-o mais robusto e menos propenso ao *overfitting*.

O `RandomForestRegressor` do `scikit-learn` possui parâmetros que influenciam a performance e a complexidade do modelo. Entre os principais estão: `n_estimators`, que define o número de árvores na floresta, `max_depth`, que limita a profundidade das árvores para evitar *overfitting*, e critérios mínimos para divisão de nós e folhas, que controlam a simplicidade e generalização do modelo.

Ajustes desses parâmetros, como o critério de divisão (`criterion`) e o número de características (`max_features`), ajudam a equilibrar desempenho e custo computacional.

3. *K-Nearest Neighbors* (KNN)

No contexto de regressão, a previsão do modelo *K-Nearest Neighbors* (KNN) é obtida pela média dos valores dos k vizinhos mais próximos. Para um novo ponto de teste x, o algoritmo KNN calcula a distância entre x e os pontos de treinamento, geralmente utilizando a distância euclidiana.

Após calcular a distância entre x e todos os pontos de treinamento, o algoritmo seleciona os K vizinhos mais próximos e, para cada um deles, obtém o valor da variável dependente y. A previsão para x é então dada pela média (ou uma outra medida estatística, como a mediana) dos valores y dos K vizinhos

$$\hat{y}(x) = \frac{1}{K} \sum_{i \in N(x,K)} y,$$

onde

- $N(x, K)$ representa o conjunto dos K vizinhos mais próximos de x;
- K é o número de vizinhos a serem considerados na previsão.

O modelo `KNeighborsRegressor` do `scikit-learn` é treinado e avaliado por validação cruzada, utilizando a métrica de Spearman para medir a correlação monotônica entre as previsões e os valores reais, especialmente útil em relações não lineares. Parâmetros importantes incluem a métrica de distância, o peso dos vizinhos e a estratégia de busca. Esses ajustes permitem otimizar a performance do modelo e sua sensibilidade aos dados.

4. Redes Neurais Profundas com *loss function* Binomial Negativa

A distribuição Binomial Negativa é uma generalização da distribuição Poisson, permitindo que a variância seja maior que a média, o que a torna adequada para modelar dados com dispersão. A fórmula da probabilidade de uma variável aleatória Y seguindo uma distribuição Binomial Negativa é dada por

$$P(Y = y) = \binom{y + r - 1}{y} \left(\frac{p}{1+p}\right)^y \left(\frac{1}{1+p}\right)^r,$$

onde

- y é o número de eventos de contagem;
- r é o parâmetro de dispersão (número de falhas até o sucesso);
- p é a probabilidade de sucesso.

No contexto de Redes Neurais Profundas, descritas em (GOODFELLOW; BENGIO; COURVILLE, 2016), a regressão Binomial Negativa pode ser modelada para prever o parâmetro de taxa de ocorrência (λ) da distribuição de Poisson, ou os parâmetros da distribuição Binomial Negativa, como a taxa de ocorrência e o parâmetro de dispersão r.

A rede neural é treinada para mapear as variáveis preditoras $\mathbf{x}$ para uma estimativa de $\hat{y}$, que é usada para modelar a taxa de contagem. A previsão pode ser ajustada para obter tanto a taxa λ quanto o parâmetro de dispersão r da distribuição Binomial Negativa.

Neste caso, a função de perda (*loss function*) que irá quantificar o erro entre as previsões do modelo e os valores reais é baseada na verossimilhança da distribuição Binomial Negativa. A verossimilhança (L) é dada por

$$L = \prod_{i=1}^{N} P(Y_i = y_i | \hat{y}_i, r) = \prod_{i=1}^{N} \binom{y_i + r - 1}{y_i} \left(\frac{p(\hat{y}_i)}{1 + p(\hat{y}_i)}\right)^{y_i} \left(\frac{1}{1 + p(\hat{y}_i)}\right)^r,$$

onde $p(\hat{y}_i)$ é a probabilidade de sucesso estimada pela rede neural para a amostra i e N é o número total de exemplos no conjunto de treinamento. A rede neural será otimizada

para minimizar a função de perda, ajustando os parâmetros da rede para maximizar a verossimilhança dos dados observados.

Os principais parâmetros da rede neural incluem o número de camadas e unidades, as funções de ativação (como ReLU para não linearidade) e a função de perda baseada na verossimilhança da distribuição Binomial Negativa, que avalia o ajuste aos dados de contagem. A arquitetura proposta consiste em uma primeira camada com 64 neurônios e $ReLU$, seguida por uma camada de 32 neurônios também com $ReLU$, e uma camada final com 1 neurônio e função sigmoide, adequada para prever a média da distribuição ou probabilidades.

Exibição e Retorno dos Resultados

Análogo ao Caso de Classificação, após a realização do treinamento e avaliação de todos os modelos, os resultados de desempenho são coletados e organizados em um conjunto de dados denominado `results_df`. A função retorna um DataFrame contendo os resultados de cada modelo, com as colunas `Model` e Spearman, onde a última coluna armazena a média das correlações de Spearman obtidas na validação cruzada.

2.6 Hiperparâmetros

Nesta seção, iremos abordar a nossa inovação para o trabalho estudado, em que adicionamos o uso de hiperparâmetros. Com o objetivo de otimizar o desempenho dos modelos, introduzimos a seleção e ajuste de hiperparâmetros utilizando o `GridSearch` do `scikit-learn`. Esse processo envolve a busca exaustiva por combinações de hiperparâmetros, testadas com validação cruzada para garantir robustez. O `GridSearch` permite identificar a combinação ideal de parâmetros, maximizando o desempenho do modelo. Além disso, nos casos de classificação em que a proporção das classes binárias são extremamente desbalanceadas, com a classe majoritária representando mais de 95% das amostras, aplicou-se o método `SMOTE`, utilizando um `sampling_strategy` de 0.75. Para isso, criamos as funções
`main_classification_hiper` e `main_regression_hiper` para implementar essa abordagem, cujos resultados serão comparados no próximo capítulo.

2.6.1 Função `main_classification_hyper`

Descrição dos Hiperparâmetros

O dicionário `param_grids` contém os hiperparâmetros a serem ajustados para diferentes modelos de aprendizado de máquina. Abaixo, detalhamos cada chave presente nesse dicionário e discutimos as implicações dos valores testados.

- **Florestas Aleatórias:**
  - `n_estimators`: Número de árvores na floresta.
    Valores testados: 10, 25, 50, 100.

    Um valor maior tende a melhorar o desempenho do modelo, pois o modelo

tem mais árvores para aprender, mas também aumenta o custo computacional. Valores muito baixos podem levar a um modelo com alto viés, enquanto valores mais altos podem reduzir o viés e aumentar a variabilidade.

- `max_depth`: Profundidade máxima das árvores.
  Valores testados: 2, 5, 7, 10, `None` (sem limite de profundidade).

  Limitar a profundidade das árvores pode ajudar a evitar *overfitting*, enquanto árvores mais profundas podem capturar mais complexidade, mas também aumentar o risco de *overfitting*. Valores muito altos (como `None`) podem resultar em modelos muito complexos, enquanto valores baixos podem fazer com que o modelo subajuste os dados.

- `min_samples_split`: Número mínimo de observações necessárias para dividir um nó.
  Valores testados: 2, 5, 10, 15.

  Um valor pequeno permite que o modelo aprenda com mais detalhes, mas pode resultar em um modelo mais propenso a *overfitting*. Valores mais altos podem melhorar a generalização ao exigir uma maior quantidade de dados para realizar uma divisão.

- ***Gradient Boosting*:**

  - `n_estimators`: Número de estimadores (ou árvores) a serem usados.
    Valores testados: 25, 50, 100, 200.

    Um número maior de estimadores pode melhorar a performance do modelo, mas também aumenta o risco de *overfitting* e o custo computacional. Valores entre 50 e 100 são frequentemente eficazes para a maioria dos problemas.

  - `learning_rate`: Taxa de aprendizado, que controla o impacto de cada árvore.
    Valores testados: 0.01, 0.1, 0.2.

    Uma taxa de aprendizado menor pode resultar em uma convergência mais lenta, mas é menos provável de causar *overfitting*. Valores mais altos fazem com que o modelo aprenda mais rápido, mas aumentam o risco de *overfitting*.

  - `max_depth`: Profundidade máxima de cada árvore.
    Valores testados: 2, 3, 5, 7, 10, `None` (sem limite de profundidade).

    Um valor de profundidade menor ajuda a evitar *overfitting* e melhora a generalização. Árvores mais profundas capturam mais detalhes, mas também podem se tornar mais complexas e suscetíveis a *overfitting*.

- **KNN:**

  - `n_neighbors`: Número de vizinhos a serem considerados para a classificação.
    Valores testados: 2, 3, 5, 10.

Um valor menor de `n_neighbors` pode fazer o modelo ser mais sensível ao ruído nos dados, enquanto valores maiores tornam a decisão mais robusta, mas podem suavizar as fronteiras de decisão e aumentar o viés.

- `weights`: Função de ponderação para os vizinhos.
  Valores testados: `'uniform'` (todos os vizinhos têm o mesmo peso), `'distance'` (os vizinhos mais próximos têm maior peso).

  Usar `uniform` pode ser mais simples e rápido, mas usar `distance` pode melhorar a precisão, dando mais importância a vizinhos próximos. Isso pode ser vantajoso em situações onde a proximidade dos pontos de dados é relevante para a classificação.

- `metric`: Métrica utilizada para calcular a distância entre pontos.
  Valores testados: `'euclidean'` (distância euclidiana), `'manhattan'` (distância de Manhattan).

  A escolha da métrica pode impactar o desempenho dependendo da natureza dos dados. A `euclidean` é comumente usada em dados contínuos, enquanto a `manhattan` pode ser mais apropriada para dados discretos ou com variáveis de escala diferente.

- ***Naive Bayes:***

  - `Naive Bayes` não possui hiperparâmetros ajustáveis relevantes para o modelo `GaussianNB`.

- **Rede Neural Profunda:**

  - `model_neurons`: Número de neurônios na camada oculta da rede neural.
    Valores testados: 32, 64, 128.

    Mais neurônios permitem que a rede aprenda representações mais complexas, mas também aumentam o risco de *overfitting* e a complexidade computacional. O valor ideal depende do problema e dos dados disponíveis.

  - `model_activation`: Função de ativação para as camadas da rede neural.
    Valores testados: `'relu'` (*Rectified Linear Unit*), `'tanh'` (tangente hiperbólica).

    A função `relu` é geralmente mais eficiente e ajuda a evitar o problema do gradiente evanescente, tornando o treinamento mais rápido. Já a função `tanh` pode ser útil em alguns casos, mas tende a sofrer com o problema do gradiente evanescente em redes profundas.

  - `model_dropout_rate`: Taxa de *dropout* para evitar *overfitting*.
    Valores testados: 0.2, 0.3.

    A taxa de *dropout* ajuda a regularizar o modelo, forçando-o a aprender representações mais gerais. Valores muito altos podem dificultar o aprendizado, enquanto valores muito baixos podem resultar em *overfitting*.

- `model_optimizer`: Algoritmo de otimização utilizado.
  Valores testados: `'adam'` (Adaptive Moment Estimation), `'sgd'` (Stochastic Gradient Descent).

  `Adam` é frequentemente mais rápido e eficaz em muitos problemas, especialmente com redes profundas. `SGD` pode ser mais simples e eficiente em problemas menores ou quando combinada com técnicas como o momentum.
- `epochs`: Número de épocas (iterações) para o treinamento.
  Valores testados: 10, 15.

  Um número maior de épocas pode melhorar o desempenho, mas também pode levar ao *overfitting*. O número ideal depende da taxa de aprendizado e da complexidade do problema.
- `batch_size`: Tamanho do lote para o treinamento.
  Valores testados: 16, 32.

  Lotes menores podem proporcionar uma atualização mais frequente dos parâmetros, o que pode melhorar a convergência, mas também aumenta o custo computacional. Lotes maiores tendem a ser mais eficientes em termos de memória, mas podem resultar em uma convergência mais lenta.

- **Rede Neural Convolucional:**
  - `model_filters`: Número de filtros na camada convolucional.
    Valores testados: 32, 64.

    O número de filtros afeta a capacidade da rede de aprender representações mais complexas. Filtros maiores podem ser vantajosos para capturar padrões mais ricos, mas também aumentam o custo computacional.
  - `model_kernel_size`: Tamanho do kernel da convolução.
    Valores testados: 2, 3.

    O tamanho do kernel afeta a resolução das características que a rede pode extrair. Kernels menores são mais detalhados, enquanto kernels maiores podem capturar padrões mais globais.
  - `model_activation`: Função de ativação para as camadas convolucionais.
    Valores testados: `'relu'`, `'tanh'`.

    A função `relu` é amplamente utilizada devido à sua eficiência e rapidez na convergência. Já a função `tanh` pode ser mais adequada para determinados tipos de redes e problemas.
  - `model_optimizer`: Algoritmo de otimização utilizado.
    Valores testados: `'adam'`, `'sgd'`.

    Similar ao DNN, `adam` é geralmente a escolha preferida devido à sua eficiência, enquanto `sgd` pode ser útil em algumas redes e combinações de parâmetros.

– `epochs`: Número de épocas para o treinamento.
Valores testados: 10, 20.

O número de épocas deve ser ajustado para equilibrar entre *overfitting* e underfitting.

– `batch_size`: Tamanho do lote para o treinamento.
Valores testados: 16, 32.

Tamanhos de lote maiores são mais eficientes, mas podem prejudicar a precisão de aprendizado, enquanto tamanhos menores podem melhorar a qualidade, mas aumentam o tempo de treinamento.

Explicação do Processo de Treinamento e Avaliação

Nesta seção, descrevemos o que ocorre no trecho de código responsável por treinar e avaliar os modelos, utilizando a técnica de busca de hiperparâmetros (`GridSearchCV`) e `StratifiedKFold`. O objetivo desse código é encontrar a melhor configuração de parâmetros para cada modelo e avaliar seu desempenho de maneira robusta.

- **Iteração sobre os modelos:** O código itera por meio de um dicionário `models`, onde cada chave representa o nome de um modelo (como `RandomForest` ou `KNN`) e o valor associado é uma instância do modelo correspondente.

- **Busca de hiperparâmetros com `GridSearchCV`:** Para cada modelo, verifica-se se há um conjunto de hiperparâmetros associado no dicionário `param_grids`. Caso existam parâmetros:

  – A classe `GridSearchCV`, do `scikit-learn`, é utilizada para realizar uma busca exaustiva entre as combinações de hiperparâmetros definidos.

  – O método `fit` treina o modelo utilizando validação cruzada (`cv=3`), otimizando o desempenho com base na métrica *f1-score*.

  – O melhor modelo (com os parâmetros ideais) é armazenado em `best_model`, e os hiperparâmetros selecionados são armazenados em `best_params`.

  Caso o modelo não possua hiperparâmetros ajustáveis (como o `GaussianNB`), ele é treinado diretamente usando o conjunto de dados de treinamento.

- **Validação cruzada e avaliação do desempenho:** Após o treinamento, o modelo selecionado é avaliado utilizando uma função personalizada, `train_and_evaluate_with_cv`, que realiza validação cruzada para medir várias métricas de desempenho. Os hiperparâmetros selecionados pela função `GridSearch` (`Best_Params`) são incluídos nos resultados. Cada modelo é avaliado e seus resultados são armazenados em um conjunto de dados, `results_df`, que inclui as métricas de desempenho e os hiperparâmetros selecionados. Esse processo automatiza a busca pelos melhores modelos e parâmetros, com o `GridSearchCV` testando combinações de hiperparâmetros de forma sistemática e a validação cruzada garantindo uma avaliação robusta, minimizando o risco de *overfitting*.

2.6.2 Função `main_regression_hyper`

Descrição dos Hiperparâmetros para Modelos de Regressão

O dicionário `param_grids` para modelos de regressão contém os hiperparâmetros que serão ajustados para cada modelo. Abaixo, detalhamos cada chave do dicionário e discutimos as implicações dos Valores testados.

- ***Gradient Boosting*:**
  - `n_estimators`: Número de estimadores (ou árvores) a serem usados.
    Valores testados: 50, 100, 200.

    Um valor maior tende a melhorar o desempenho do modelo, pois o modelo tem mais árvores para aprender, mas também aumenta o custo computacional. Valores muito baixos podem levar a um modelo com alto viés, enquanto valores mais altos ajudam a reduzir o viés e aumentar a variabilidade.
  - `learning_rate`: Taxa de aprendizado, controlando o impacto de cada árvore.
    Valores testados: 0.01, 0.1, 0.2.

    A taxa de aprendizado mais baixa resulta em uma convergência mais suave, mas mais lenta, enquanto valores maiores aceleram a convergência, mas podem aumentar o risco de *overfitting*.
  - `max_depth`: Profundidade máxima de cada árvore.
    Valores testados: 3, 5, 7.

    Limitar a profundidade das árvores ajuda a evitar *overfitting*. Árvores mais profundas podem aprender mais padrões complexos, mas com o risco de não generalizar bem.
  - `min_samples_split`: Número mínimo de observações necessárias para dividir um nó.
    Valores testados: 2, 5, 10.

    Valores menores permitem que o modelo aprenda mais detalhes, mas podem resultar em *overfitting*. Valores maiores tornam o modelo mais robusto e generalizável.
  - `min_samples_leaf`: Número mínimo de observações necessárias em um nó folha.
    Valores testados: 1, 3, 5.

    Um valor menor permite maior flexibilidade para aprender detalhes dos dados, enquanto valores maiores ajudam a reduzir o *overfitting*, suavizando as previsões.
- **Florestas Aleatórias:**
  - `n_estimators`: Número de árvores na floresta.
    Valores testados: 50, 100, 200.

Um número maior de árvores melhora a robustez e o desempenho, mas também aumenta o custo computacional.

- `max_depth`: Profundidade máxima de cada árvore.
  Valores testados: 5, 10, 20.

  Limitar a profundidade pode evitar *overfitting*, enquanto árvores muito profundas podem capturar padrões excessivamente específicos que não se generalizam bem para dados não vistos.

- `min_samples_split`: Número mínimo de amostras para dividir um nó.
  Valores testados: 2, 5, 10.

  Um valor menor pode resultar em árvores muito complexas, enquanto valores maiores ajudam a suavizar o modelo e evitar o *overfitting*.

- `min_samples_leaf`: Número mínimo de amostras necessárias em um nó folha.
  Valores testados: 1, 3, 5.

  Valores menores permitem um modelo mais flexível, mas podem aumentar o risco de *overfitting*. Valores maiores tendem a melhorar a generalização.

- **KNN:**

  - `n_neighbors`: Número de vizinhos a serem considerados.
    Valores testados: 3, 5, 10.

    Valores menores tornam o modelo mais sensível ao ruído, enquanto valores maiores suavizam as decisões, mas podem aumentar o viés. Valores típicos variam entre 3 e 5.

  - `weights`: Função de ponderação dos vizinhos.
    Valores testados: `'uniform'` (todos os vizinhos têm o mesmo peso), `'distance'` (vizinho mais próximo tem maior peso).

    Usar `'uniform'` pode ser mais simples e rápido, enquanto `'distance'` pode melhorar a precisão ao considerar vizinhos mais próximos como mais relevantes.

  - `metric`: Métrica usada para calcular a distância entre pontos.
    Valores testados: `'euclidean'` (distância euclidiana), `'manhattan'` (distância de Manhattan).

    A métrica escolhida afeta o desempenho do modelo. Desse modo, `'euclidean'` é comum para dados contínuos, enquanto `'manhattan'` pode ser mais adequado para dados discretos ou em espaços com escalas diferentes.

- **Redes Neurais Profundas com *loss function* Binomial Negativa:**

  - `neurons`: Número de neurônios na camada oculta.
    Valores testados: 32, 64, 128.

Um número maior de neurônios permite ao modelo aprender representações mais complexas, mas também pode aumentar o risco de *overfitting* e o custo computacional.

- `activation`: Função de ativação para as camadas ocultas.
  Valores testados: `'relu'` (*Rectified Linear Unit*), `'tanh'` (tangente hiperbólica).

  A função `relu` é mais eficiente para redes profundas, enquanto a função `tanh` pode ser útil, mas sofre com o problema de gradiente evanescente em redes profundas.

- `optimizer`: Algoritmo de otimização utilizado.
  Valores testados: `'adam'` (Adaptive Moment Estimation), `'sgd'` (Stochastic Gradient Descent).

  `Adam` é mais popular devido à sua eficiência e capacidade de adaptação das taxas de aprendizado. `SGD` pode ser útil em algumas situações específicas, mas geralmente tem uma convergência mais lenta.

- `epochs`: Número de épocas (iterações) para o treinamento.
  Valores testados: 10, 20.

  O número de épocas deve ser ajustado para evitar *overfitting*. Valores menores podem resultar em underfitting, enquanto valores maiores podem causar *overfitting*.

- `batch_size`: Tamanho do lote para o treinamento.
  Valores testados: 16, 32.

  Tamanhos de lote menores permitem uma atualização mais frequente dos parâmetros, enquanto tamanhos maiores são mais eficientes em termos de memória e tempo de computação, mas podem reduzir a qualidade do aprendizado.

Explicação do Processo de Treinamento e Avaliação

Assim como no processo de classificação, utilizamos o `GridSearchCV` para realizar a busca pelos melhores hiperparâmetros dos modelos de regressão. A validação cruzada é conduzida com a função auxiliar `reg_train_and_evaluate`, que implementa o método `K-Fold` para dividir os dados de maneira estratificada e avaliar o desempenho dos modelos de forma consistente.

A combinação do `GridSearchCV` com a validação cruzada permite identificar os parâmetros que maximizam as métricas de desempenho dos modelos, garantindo que a avaliação seja robusta e generalizável.

No próximo capítulo, serão apresentados os resultados do treinamento dos modelos descritos até o momento. Faremos uma análise detalhada do desempenho de cada modelo, incluindo uma comparação entre os resultados obtidos antes e após o ajuste de hiperparâmetros.

Capítulo 3

Resultados, Comparações e Discussões

Nesta seção, apresentamos os resultados obtidos após a execução dos modelos de aprendizado de máquina aplicados a cada dicionário. Primeiramente, exibiremos os resultados gerados pelo código em `Python` sem a aplicação de hiperparâmetros, seguidos pelos resultados com ajustes de hiperparâmetros. Para a execução dos experimentos, utilizamos um equipamento com as seguintes especificações de hardware: modelo Acer Aspire A315-53, processador Intel® Core™ i3-7020U com 2 núcleos, memória RAM de 4,0 GiB e capacidade de disco de 480,1 GB. O código utilizado no desenvolvimento deste trabalho, juntamente com os resultados obtidos, pode ser acessado no repositório GitHub disponível em: https://github.com/mariagscapin/PythonTraitMatching-Prediction/tree/main.

3.1 Resultados da Classificação

Os resultados apresentados a seguir incluem as métricas de desempenho dos modelos de classificação aplicados aos diferentes conjuntos de dados nas quatro simulações realizadas. Essas simulações consideram os modelos descritos no capítulo anterior, tanto sem quanto com o ajuste de hiperparâmetros. Neste contexto, analisamos os resultados e a performance de cada modelo em relação a cada conjunto de dados, avaliando, de forma detalhada, se o uso de hiperparâmetros contribuiu de maneira significativa para a melhoria do desempenho. Além disso, destacaremos em negrito os modelos que obtiveram melhores resultados em cada contexto.

3.1.1 Desempenho em `C1`

Nesta subseção, comparamos o desempenho dos modelos treinados no conjunto de dados `C1`, considerando os resultados sem ajustes de hiperparâmetros (Tabela 3.1) e com ajustes de hiperparâmetros (Tabela 3.2). Essa análise nos permite avaliar o impacto do ajuste de hiperparâmetros no desempenho dos modelos

Modelos	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	***AUC***
Florestas Aleatórias	**0.5514**	0.1035	0.4760	0.0584	0.5175
Gradient Boosting	0.5349	0.3725	0.4654	0.3113	0.5151
KNN	0.4934	**0.5560**	0.4560	**0.7125**	**0.5186**
Naive Bayes	0.5180	0.4909	0.4631	0.5225	0.5165
DNN	0.5414	0.2507	**0.4874**	0.1843	0.5140
CNN	0.5237	0.4228	0.4586	0.3960	0.5092

Tabela 3.1: Desempenho dos modelos no conjunto de dados `C1`.

Modelos com Hiperparâmetros	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.5066	0.3801	0.4141	0.3515	0.4877
Gradient Boosting	0.5094	0.4363	0.4317	0.4417	0.5063
KNN	0.5077	0.4160	0.4255	0.4071	0.5002
Naive Bayes	0.5131	**0.4881**	**0.4461**	**0.5391**	**0.5204**
DNN	**0.5323**	0.3683	0.4381	0.3250	0.5075
CNN	0.5209	0.4267	0.4400	0.4151	0.5057

Tabela 3.2: Desempenho dos modelos com hiperparâmetros ajustados em `C1`.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': None, 'min_samples_split': 5, 'n_estimators': 10}
- ***Gradient Boosting*:** {'learning_rate': 0.2, 'max_depth': None, 'n_estimators': 25}
- **KNN:** {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'uniform'}
- **DNN:** {'batch_size': 16, 'epochs': 15, 'model_activation': 'relu', 'model_dropout_rate': 0.2, 'model_neurons': 128, 'model_optimizer': 'adam'}
- **CNN:** {'batch_size': 32, 'epochs': 20, 'model_activation': 'relu', 'model_filters': 32, 'model_kernel_size': 3, 'model_optimizer': 'adam'}

O modelo de otimização Adam (*Adaptive Moment Estimation*) é um método de otimização usado para ajustar os pesos de redes neurais. Ele combina as vantagens de dois outros métodos: *momentum*, que ajuda a suavizar atualizações, e *adaptive learning rate*, que ajusta a taxa de aprendizado para cada peso da rede. Adam é muito usado

em redes neurais convolucionais (CNNs) porque é eficiente, rápido e funciona bem com grandes quantidades de dados. A prática de ajustar hiperparâmetros é essencial para otimizar o desempenho de algoritmos de aprendizado de máquina. Contudo, os resultados indicaram efeitos variáveis entre os modelos analisados.

No *Naive Bayes*,houve uma leve melhora no *recall* de 0.5225 para 0.5391 embora esse tipo de modelo não se utiliza hiperparâmetros, este resultado oriunda de eficiência computacional. Em contrapartida, modelos como Floresta Aleatória e *Gradient Boosting* apresentaram quedas em acurácia e *AUC*, sugerindo ajustes subótimos para o conjunto `C1`. No caso das Florestas Aleatórias, mesmo com ajustes em parâmetros como `max_depth` e `n_estimators`, a acurácia caiu de 0.5514 para 0.5066, indicando que os ajustes não capturaram adequadamente a estrutura dos dados. Para o *Gradient Boosting*, pequenos ajustes em `learning_rate` resultaram em ganhos marginais, destacando a sensibilidade do modelo a mudanças. O KNN, mesmo após ajustes em `n_neighbors`, apresentou uma acurácia baixa (0.5077), refletindo limitações em cenários complexos. Por outro lado, as Redes Neurais, com ajustes em número de épocas e taxa de *dropout*, alcançaram ganhos modestos, mas demonstraram maior robustez em métricas críticas.

Os resultados mostram que a eficácia dos ajustes depende das características dos modelos e dos dados. Estratégias como validação cruzada e *Grid Search* podem ter sido insuficientes para capturar configurações ideais. Estudos futuros poderiam explorar outras técnicas para ajustes mais eficientes, além de aumentar o valor da validação cruzada de três para cinco, por exemplo.

3.1.2 Desempenho em `C2`

No conjunto de dados `C2`, os modelos foram avaliados e os resultados detalhados são apresentados nas Tabelas 3.3 e 3.4, com destaque para as métricas de desempenho obtidas tanto com quanto sem a otimização dos hiperparâmetros

Modelos	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.9673	0.9717	0.9631	**0.9804**	0.9946
Gradient Boosting	0.9552	0.9609	0.9598	0.9621	0.9906
KNN	0.9298	0.9346	**0.9993**	0.8778	0.9587
Naive Bayes	0.7109	0.6636	0.9900	0.4992	0.9432
DNN	0.9636	0.9675	0.9853	0.9504	0.9910
CNN	**0.9785**	**0.9810**	0.9935	0.9687	**0.9958**

Tabela 3.3: Desempenho dos modelos no conjunto de dados `C2`.

Modelos com Hiperparâmetros	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.8667	0.8908	0.8381	**0.9508**	0.9559
Gradient Boosting	0.9443	0.9508	0.9592	0.9426	**0.9857**
KNN	0.9384	0.9432	**0.9962**	0.8956	0.9469
Naive Bayes	0.7330	0.6970	0.9908	0.5378	0.9396
DNN	0.9425	0.9488	0.9661	0.9323	0.9797
CNN	**0.9597**	**0.9641**	0.9820	0.9470	0.9854

Tabela 3.4: Desempenho dos modelos com hiperparâmetros ajustados em `C2`.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 50}
- ***Gradient Boosting*:** {'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 100}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'distance'}
- **DNN:** {'batch_size': 16, 'epochs': 15, 'model_activation': 'relu', 'model_dropout_rate': 0.2, 'model_neurons': 64, 'model_optimizer': 'adam'}
- **CNN:** {'batch_size': 16, 'epochs': 20, 'model_activation': 'relu', 'model_filters': 64, 'model_kernel_size': 2, 'model_optimizer': 'adam'}

No conjunto de dados `C2`, os modelos foram avaliados com base em várias métricas, como acurácia, *f1_score*, precisão, *recall* e AUC. Entre eles, a CNN obteve o melhor desempenho, com acurácia de 0.9785, *f1_score* de 0.9810 e AUC de 0.9958, destacando-se por sua capacidade de capturar padrões hierárquicos e interações complexas nas variáveis.

O modelo Florestas Aleatórias, apesar de uma boa acurácia inicial de 0.9673 e AUC de 0.9946, apresentou uma queda no desempenho após ajustes, com acurácia reduzida para 0.8667, indicando que a configuração inicial foi mais eficaz do que a otimizada. O *Gradient Boosting* obteve resultados consistentes, com acurácias de 0.9552 e 0.9443 antes e depois do ajuste, respectivamente, mantendo um bom equilíbrio entre precisão e *Recall*.

O modelo KNN teve uma excelente precisão (0.9993), mas um *recall* inferior (0.8778), e sua acurácia aumentou de 0.9298 para 0.9384 com ajustes, ressaltando a importância da escolha adequada dos hiperparâmetros. Por outro lado, o *Naive Bayes* foi o de pior desempenho, com uma acurácia de apenas 0.7109 e *recall* reduzido, sugerindo que sua simplicidade não foi suficiente para lidar com a complexidade do conjunto de dados. Entretanto, o modelo ajustado foi melhor comparado ao mesmo não hiperparametrizado.

O modelo DNN apresentou um bom desempenho, com acurácias de 0.9636 e 0.9425, além de AUC de 0.9910 e 0.9797, demonstrando sua capacidade de aprender padrões não lineares e separar globalmente as classes de forma eficaz. Em geral, os modelos baseados em redes neurais, como a CNN e DNN, além do *Gradient Boosting*, se destacaram por sua adaptação ao conjunto de dados `C2`, especialmente em cenários de alta complexidade e interações não lineares.

Embora o KNN e o *Naive Bayes* apresentem vantagens em termos de simplicidade, não foram tão eficazes quanto os outros modelos, especialmente quando se considera o *recall* e o *f1_score*. Esses resultados ressaltam a importância do ajuste de hiperparâmetros para otimizar o desempenho dos modelos e garantir a escolha mais apropriada para cada cenário específico.

3.1.3 Desempenho em C3

As Tabelas 3.5 e 3.6 apresentam os resultados dos modelos aplicados ao conjunto de dados C3, que corresponde a um cenário intermediário. Esse conjunto de dados foi escolhido por refletir características típicas de um ambiente com complexidade moderada, permitindo uma avaliação abrangente das diferentes abordagens utilizadas

Modelos	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.5818	0.1301	0.6729	0.0723	0.6994
Gradient Boosting	0.7091	0.6493	0.6809	0.6211	0.7782
KNN	0.5390	0.5877	0.4803	0.7570	0.5839
Naive Bayes	0.6681	0.6988	0.5766	**0.8871**	0.7280
DNN	0.7491	0.7111	0.7125	0.7121	0.8198
CNN	**0.7719**	**0.7264**	**0.7578**	0.6991	**0.8472**

Tabela 3.5: Desempenho dos modelos no conjunto de dados C3.

Modelos com Hiperparâmetros	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.7231	0.6776	0.6855	0.6698	0.7877
Gradient Boosting	0.7184	0.6806	0.6709	0.6909	0.7772
KNN	0.6234	0.5382	0.5754	0.5057	0.6636
Naive Bayes	0.6830	0.7062	0.5909	**0.8774**	0.7390
DNN	0.7781	0.7347	**0.7632**	0.7103	0.8469
CNN	**0.7853**	**0.7498**	0.7591	0.7407	**0.8590**

Tabela 3.6: Desempenho dos modelos com hiperparâmetros ajustados em C3.

Parâmetros selecionados pelo GridSearch:

- **Florestas Aleatórias:** {'max_depth': None, 'min_samples_split': 25, 'n_estimators': 100}

- ***Gradient Boosting*:** {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 100}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'distance'}
- **DNN:** {'batch_size': 16, 'epochs': 15, 'model_activation': 'relu', 'model_dropout_rate': 0.2, 'model_neurons': 32, 'model_optimizer': 'adam'}
- **CNN:** {'batch_size': 32, 'epochs': 10, 'model_activation': 'relu', 'model_filters': 32, 'model_kernel_size': 3, 'model_optimizer': 'adam'}

No caso das Florestas Aleatórias, o ajuste trouxe uma melhoria considerável, com a acurácia subindo de 0.5818 para 0.7231 e o *f1_score* subindo de 0.1301 para 0.6776. Isso indica que os ajustes em parâmetros como `max_depth` e `n_estimators` ajudaram o modelo a se adaptar melhor aos dados, superando as limitações do modelo inicial. Em contraste, o Gradient Boosting apresentou uma melhora marginal, com a acurácia subindo de 0.7091 para 0.7184 e o *f1_score* de 0.6493 para 0.6806. Embora os ajustes em `learning_rate` e `max_depth` tenham levado a um pequeno ganho, o modelo demonstrou ser relativamente insensível a essas mudanças.

O KNN, por outro lado, experimentou uma queda no desempenho após os ajustes, com a acurácia subindo de 0.5390 para 0.6234, mas o *f1_score* caiu de 0.5877 para 0.5382. A escolha dos parâmetros de `metric` e `n_neighbors` não conseguiu capturar adequadamente a complexidade do conjunto de dados, refletindo as limitações do modelo para cenários mais desafiadores.

Já o *Naive Bayes* apresentou uma leve melhora na acurácia (de 0.6681 para 0.6830) e uma leve melhoria no *f1_score* (de 0.6988 para 0.7062), mantendo um bom *recall* (de 0.8871 para 0.8774). Os ajustes de parâmetros, como o aumento do número de estimadores, permitiram que o modelo se mantivesse robusto na identificação das classes positivas, sem causar grandes variações em outras métricas.

Em relação às Redes Neurais, o DNN obteve um aumento considerável na acurácia, subindo de 0.7491 para 0.7781, e no *f1_score*, de 0.7111 para 0.7347, com os ajustes nos parâmetros de número de neurônios e *dropout*. Isso demonstra que o modelo se beneficiou da redução de *overfitting* e melhor capacidade de aprendizado. O CNN teve uma melhoria ainda mais expressiva, com a acurácia subindo de 0.7719 para 0.7853 e o *f1_score* indo de 0.7264 para 0.7498. A configuração otimizada de filtros e número de épocas contribuiu para um modelo mais robusto e capaz de capturar padrões complexos de forma mais eficaz.

Os resultados indicam que as Redes Neurais, especificamente o DNN e o CNN, foram os melhores modelos para o conjunto de dados `C3`. Ambos apresentaram melhorias consideráveis nas métricas de desempenho, com o CNN alcançando o maior aprimoramento em acurácia (de 0.7719 para 0.7853) e *f1_score* (de 0.7264 para 0.7498), indicando que a combinação de otimizações, como o ajuste de filtros e o número de épocas, ajudou o modelo a se ajustar de maneira mais eficiente aos dados complexos. O DNN também se beneficiou dos ajustes, com melhorias na acurácia e *f1_score*, demonstrando a capacidade das Redes Neurais em capturar padrões mais complexos e sua maior robustez em comparação com outros modelos.

Embora o modelo de Florestas Aleatórias tenha apresentado uma melhoria significativa após os ajustes, ele ainda ficou atrás das Redes Neurais, especialmente em métricas como o *f1_score*. O Gradient Boosting, embora tenha apresentado um pequeno aumento,

não teve um desempenho tão expressivo quanto as Redes Neurais, e o KNN sofreu uma queda no desempenho após a otimização dos parâmetros, indicando que esse modelo não foi adequado para a complexidade do conjunto de dados.

Portanto, para o conjunto de dados `C3`, os melhores desempenhos foram alcançados com Redes Neurais, destacando-se pela capacidade de aprender padrões complexos e se adaptar adequadamente aos dados.

3.1.4 Desempenho em `C4_high`

O conjunto de dados `C4_high` foi utilizado para avaliar o desempenho dos modelos em um cenário com dados de alta proporção, caracterizados por uma estrutura bem definida e baixa presença de ruídos. A seguir, analisamos as métricas de cada modelo para determinar suas forças e limitações utilizando as Tabelas 3.7 e 3.8.

Modelos	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.5796	0.0827	0.7252	0.0445	0.7004
Gradient Boosting	0.7151	0.6551	0.6840	0.6286	0.7811
KNN	0.5497	0.5899	0.4853	0.7520	0.5933
Naive Bayes	0.6711	0.6992	0.5769	**0.8875**	0.7300
DNN	0.7434	0.6953	0.7121	0.6805	0.8131
CNN	**0.7721**	**0.7332**	**0.7396**	0.7271	**0.8480**

Tabela 3.7: Desempenho dos modelos no conjunto de dados `C4_high`.

Modelos com Hiperparâmetros	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.7250	0.6708	0.6948	0.6485	0.7919
Gradient Boosting	0.7165	0.6703	0.6738	0.6668	0.7828
KNN	0.6344	0.5478	0.5884	0.5124	0.6698
Naive Bayes	0.6828	0.7034	0.5901	**0.8705**	0.7424
DNN	**0.7827**	0.7356	**0.7764**	0.6996	**0.8513**
CNN	0.7811	**0.7430**	0.7539	0.7325	0.8507

Tabela 3.8: Desempenho dos modelos com hiperparâmetros ajustados em `C4_high`.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': None, 'min_samples_split': 25, 'n_estimators': 100}

- ***Gradient Boosting*:** {'learning_rate': 0.2, 'max_depth': 5, 'n_estimators': 100}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'distance'}
- **DNN:** {'batch_size': 32, 'epochs': 15, 'model_activation': 'relu', 'model_dropout_rate': 0.2, 'model_neurons': 32, 'model_optimizer': 'adam'}
- **CNN:** {'batch_size': 16, 'epochs': 10, 'model_activation': 'relu', 'model_filters': 32, 'model_kernel_size': 3, 'model_optimizer': 'adam'}

Ao analisar o desempenho das Florestas Aleatórias, foi observada uma melhoria significativa após os ajustes. A acurácia aumentou de 0.5796 para 0.7250, e o *f1_score* subiu de 0.0827 para 0.6708. Esse avanço pode ser atribuído aos ajustes em parâmetros como `max_depth` e `n_estimators`, que permitiram uma melhor adaptação do modelo aos dados, superando suas limitações iniciais. Por outro lado, o modelo de *Gradient Boosting* apresentou um ganho mais modesto, com a acurácia subindo de 0.7151 para 0.7165 e o *f1_score* passando de 0.6551 para 0.6703. Embora ajustes no `learning_rate` e `max_depth` tenham sido realizados, o impacto foi limitado, sugerindo que o modelo já estava em um estado relativamente bom.

O KNN, por sua vez, não obteve os resultados esperados após os ajustes. Apesar de uma leve melhoria na acurácia (de 0.5497 para 0.6344), o *f1_score* diminuiu de 0.5899 para 0.5478. Essa queda nas métricas de desempenho indica que o modelo não foi capaz de melhorar sua capacidade de classificar adequadamente as instâncias positivas, refletindo as dificuldades do KNN em dados com uma alta proporção de exemplos.

Em contrapartida, o *Naive Bayes* apresentou uma melhora discreta, com a acurácia passando de 0.6711 para 0.6828 e o *f1_score* subindo de 0.6992 para 0.7034, enquanto manteve um excelente *recall* (de 0.8875 para 0.8705). As modificações, como o aumento no número de estimadores, ajudaram a manter o modelo robusto para a classificação das classes positivas, sem provocar grandes mudanças nas demais métricas.

Em relação às Redes Neurais, o DNN se destacou ao apresentar uma melhoria significativa, com a acurácia passando de 0.7434 para 0.7827 e o *f1_score* subindo de 0.6953 para 0.7356. Esses avanços foram possibilitados pelos ajustes nos parâmetros, como o número de neurônios e o *dropout*, que reduziram o *overfitting* e melhoraram a capacidade do modelo de generalizar para dados desconhecidos. O CNN obteve um desempenho ainda mais notável, com a acurácia subindo de 0.7721 para 0.7811 e o *f1_score* indo de 0.7332 para 0.7430. O ajuste nos filtros e no número de épocas contribuiu para um modelo mais robusto e capaz de captar padrões complexos com maior precisão.

Logo, para o conjunto `C4_high`, as Redes Neurais, particularmente o DNN e o CNN, foram os modelos mais eficazes, alcançando os melhores resultados devido à sua capacidade de capturar padrões complexos e melhorar a performance nas métricas chave.

3.1.5 Desempenho em `C4_mid`

Os dados moderadamente limpos do conjunto oferecem uma oportunidade de avaliar a robustez dos modelos em um contexto intermediário. As métricas apresentadas nas Tabelas 3.9 e 3.10 fornecem intuições sobre o equilíbrio entre precisão e sensibilidade.

Modelos	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.6093	0.2069	0.8045	0.1192	0.7311
Gradient Boosting	0.7586	0.6675	**0.8146**	0.5658	0.8466
KNN	0.7114	0.7412	0.6020	**0.9640**	0.8525
Naive Bayes	0.6546	0.6973	0.5584	0.9283	0.7072
DNN	0.7492	0.7172	0.6950	0.7434	0.8259
CNN	**0.8303**	**0.8075**	0.7851	0.8323	**0.9009**

Tabela 3.9: Desempenho dos modelos no conjunto de dados `C4_mid`.

Modelos com Hiperparâmetros	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	**0.8598**	0.8219	**0.9016**	0.7552	**0.9272**
Gradient Boosting	0.7672	0.7280	0.7295	0.7265	0.7885
KNN	0.8301	**0.8258**	0.7365	**0.9398**	0.8917
Naive Bayes	0.6484	0.6926	0.5539	0.9239	0.7031
DNN	0.7966	0.7585	0.7721	0.7472	0.8754
CNN	0.8377	0.8176	0.7892	0.8485	0.9051

Tabela 3.10: Desempenho dos modelos com hiperparâmetros ajustados em `C4_mid`.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 100}
- ***Gradient Boosting*:** {'learning_rate': 0.1, 'max_depth': None, 'n_estimators': 50}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'distance'}
- **DNN:** {'batch_size': 32, 'epochs': 15, 'model_activation': 'relu', 'model_dropout_rate': 0.2, 'model_neurons': 128, 'model_optimizer': 'adam'}
- **CNN:** {'batch_size': 32, 'epochs': 20, 'model_activation': 'relu', 'model_filters': 64, 'model_kernel_size': 3, 'model_optimizer': 'adam'}

Em relação às Florestas Aleatórias, o modelo apresentou um desempenho sólido após os ajustes, com a acurácia subindo de 0.6093 para 0.8598 e o *f1_score* aumentando de 0.2069 para 0.8219. A otimização dos parâmetros como `max_depth` e `n_estimators`

contribuiu para uma melhora substancial, permitindo que o modelo tivesse um desempenho superior na captura das classes positivas, além de equilibrar bem a precisão e o *Recall*. Por outro lado, o *Gradient Boosting* apresentou um desempenho relativamente bom, mas sem grandes variações após os ajustes. A acurácia aumentou de 0.7586 para 0.7672 e o *f1_score* foi de 0.6675 para 0.7280. Embora os ajustes em parâmetros como `learning_rate` e `max_depth` tenham sido aplicados, a melhoria foi moderada, sugerindo que o modelo já estava razoavelmente ajustado.

O KNN, apesar de uma boa acurácia inicial de 0.7114, apresentou um desempenho notável após os ajustes, com a acurácia subindo para 0.8301 e o *f1_score* indo de 0.7412 para 0.8258. Esses ajustes, particularmente no número de vizinhos e a métrica de distância, permitiram ao modelo melhorar a captura das classes positivas, enquanto ainda mantinha uma boa capacidade de generalização. No entanto, o modelo de *Naive Bayes* não apresentou grandes melhorias, mantendo uma acurácia de 0.6546 para 0.6484, com o *f1_score* indo de 0.6973 para 0.6926, o que sugere que o modelo não foi tão sensível aos ajustes quanto outros.

O DNN demonstrou uma melhoria significativa após a otimização, com a acurácia passando de 0.7492 para 0.7966 e o *f1_score* subindo de 0.7172 para 0.7585. Os ajustes no número de neurônios e o *dropout* ajudaram a melhorar a generalização do modelo, reduzindo o *overfitting* e aumentando sua capacidade de identificar padrões mais complexos. O CNN, por sua vez, obteve os melhores resultados em termos de acurácia e *f1_score*, alcançando uma acurácia de 0.8377 e um *f1_score* de 0.8176, sendo o modelo mais eficaz para este conjunto de dados. A configuração otimizada de filtros e o número de épocas de treinamento contribuíram para um desempenho superior na captura de características mais complexas e no aprimoramento da sensibilidade e precisão.

Com base nas métricas de desempenho, os modelos de Redes Neurais, particularmente o CNN, demonstraram a melhor performance no conjunto de dados `C4_mid`. O CNN, com uma acurácia de 0.8377 e um *f1_score* de 0.8176, superou os outros modelos em termos de precisão e sensibilidade. A otimização de parâmetros, como o número de filtros e o aumento no número de épocas, contribuiu para a eficácia do modelo em capturar padrões mais complexos nos dados.

3.1.6 Desempenho em `C4_low`

Neste caso, avaliamos os modelos em um ambiente com dados `C4_low` de baixa proporção, apresentados nas Tabelas 3.11 e 3.12. A análise do desempenho permite identificar quais técnicas mantêm a eficácia nesse tipo de conjunto.

Modelos	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.6512	0.3389	0.9016	0.2088	0.8163
Gradient Boosting	0.8648	0.8208	**0.9495**	0.7229	0.9415
KNN	0.8498	0.8508	0.7407	**0.9993**	0.9279
Naive Bayes	0.6327	0.6934	0.5399	0.9689	0.6842
DNN	0.9088	0.8953	0.8811	0.9101	0.9656
CNN	**0.9463**	**0.9391**	0.9145	0.9651	**0.9837**

Tabela 3.11: Desempenho dos modelos no conjunto de dados `C4_low`.

Modelos com Hiperparâmetros	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	**0.9718**	**0.9660**	**0.9986**	0.9356	**0.9943**
Gradient Boosting	0.9513	0.9409	0.9813	0.9036	0.9776
KNN	0.8795	0.8768	0.7807	**0.9998**	0.9542
Naive Bayes	0.6564	0.7061	0.5574	0.9632	0.7059
DNN	0.9477	0.9402	0.9214	0.9598	0.9880
CNN	0.9631	0.9582	0.9324	0.9854	0.9901

Tabela 3.12: Desempenho dos modelos com hiperparâmetros ajustados em `C4_low`.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 100}
- ***Gradient Boosting*:** {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 100}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'uniform'}
- **DNN:** {'batch_size': 32, 'epochs': 15, 'model_activation': 'relu', 'model_dropout_rate': 0.3, 'model_neurons': 128, 'model_optimizer': 'adam'}
- **CNN:** {'batch_size': 16, 'epochs': 20, 'model_activation': 'relu', 'model_filters': 32, 'model_kernel_size': 3, 'model_optimizer': 'adam'}

Os resultados obtidos no conjunto de dados `C4_low` revelam desafios claros associados ao baixo número de representações das classes. No cenário inicial, sem o ajuste dos hiperparâmetros, o desempenho dos modelos foi consideravelmente inferior. O modelo de Florestas Aleatórias, por exemplo, teve uma acurácia de apenas 0.6512, com um *f1_score*

de 0.3389, evidenciando um desempenho ruim na identificação da classe de menor proporção. A precisão de 0.9016 sugere que o modelo foi relativamente bom em identificar corretamente a classe mais prevalente, mas a sua baixa taxa de *recall* (0.2088) indica que não conseguiu capturar adequadamente a classe minoritária.

O modelo *Gradient Boosting* se destacou com uma acurácia de 0.8648 e um *f1_score* de 0.8208, demonstrando boa performance na tarefa de classificação, apesar da dificuldade imposta pela baixa proporção de classe. No entanto, o *recall* de 0.7229 indicou que o modelo teve dificuldades em identificar os verdadeiros positivos, embora sua capacidade de discriminação fosse boa, como evidenciado pelo AUC de 0.9415.

O *K-Nearest Neighbors* (KNN) obteve uma acurácia de 0.8498 e um *f1_score* de 0.8508, apresentando um desempenho satisfatório, mas a baixa precisão (0.7407) sugere que o modelo teve dificuldades em classificar corretamente a proporção de previsões positivas, apesar de um *recall* quase perfeito (0.9993), refletindo uma capacidade de detectar os verdadeiros positivos.

O modelo *Naive Bayes*, com a menor performance, obteve uma acurácia de 0.6327 e um *f1_score* de 0.6934, destacando-se apenas no *recall* (0.9689), mas com uma precisão muito baixa (0.5399). Esse comportamento é esperado devido à natureza simplista do modelo, que não conseguiu lidar bem com as características do conjunto de dados.

Em contraste, após o ajuste de hiperparâmetros, todos os modelos mostraram uma melhoria significativa no desempenho. A Floresta Aleatória, por exemplo, alcançou uma acurácia de 0.9718 e um *f1_score* de 0.9660, com uma precisão de 0.9986 e um AUC de 0.9943, destacando-se como o modelo mais robusto neste cenário. A melhoria foi notável devido à maior capacidade de ajuste aos dados desbalanceados.

O *Gradient Boosting*, com os parâmetros otimizados, manteve uma boa performance com acurácia de 0.9513 e um *f1_score* de 0.9409, embora sua precisão (0.9813) tenha sido ligeiramente inferior à da Floresta Aleatória. O KNN, embora tenha mostrado uma melhoria na acurácia (0.8795) e no *f1_score* (0.8768), ficou inferior a outros modelos mais complexos. O DNN e o CNN também se destacaram após o ajuste de hiperparâmetros, com o DNN alcançando uma acurácia de 0.9477 e o CNN de 0.9631.

Em resumo, o cenário de baixa proporção de classes no conjunto `C4_low` demonstrou a importância do ajuste de hiperparâmetros para melhorar o desempenho dos modelos. A Floresta Aleatória se destacou como a opções mais robusta, mostrando a melhor capacidade de ajuste aos dados desbalanceados.

3.1.7 Desempenho em um caso real

Neste cenário, avaliamos os modelos em um conjunto de `dados reais`, apresentados nas Tabelas 3.13 e 3.14. Os resultados obtidos demonstram a eficácia dos modelos antes e após o ajuste de hiperparâmetros, permitindo uma análise detalhada de seu desempenho.

Modelos	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.7185	0.5400	0.9010	0.3858	0.8844
Gradient Boosting	0.9344	0.9191	**0.9747**	0.8696	0.9782
KNN	0.8881	0.8846	0.7930	**1.0000**	0.9305
Naive Bayes	0.6055	0.6826	0.5209	0.9898	0.6618
DNN	0.9655	0.9601	0.9503	0.9702	0.9931
CNN	**0.9847**	**0.9824**	0.9686	0.9965	**0.9979**

Tabela 3.13: Desempenho dos modelos no conjunto de dados reais.

Modelos com Hiperparâmetros	**Acurácia**	***F1_Score***	**Precisão**	***Recall***	**AUC**
Florestas Aleatórias	0.9816	0.9781	0.9969	0.9600	0.9980
Gradient Boosting	0.9758	0.9713	**0.9902**	0.9531	0.9885
KNN	0.9481	0.9429	0.8919	**1.0000**	0.9821
Naive Bayes	0.6055	0.6826	0.5209	0.9898	0.6618
DNN	0.9781	0.9748	0.9631	0.9869	0.9969
CNN	**0.9868**	**0.9849**	0.9704	0.9997	**0.9985**

Tabela 3.14: Desempenho dos modelos com ajustes de hiperparâmetros no conjunto de dados reais.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': None, 'min_samples_split': 5, 'n_estimators': 100}
- ***Gradient Boosting*:** {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 100}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'uniform'}
- **DNN:** {'batch_size': 32, 'epochs': 15, 'model_activation': 'relu', 'model_dropout_rate': 0.2, 'model_neurons': 64, 'model_optimizer': 'adam'}
- **CNN:** {'batch_size': 32, 'epochs': 20, 'model_activation': 'relu', 'model_filters': 32, 'model_kernel_size': 2, 'model_optimizer': 'adam'}

No cenário inicial, os modelos apresentaram variações significativas em seus desempenhos. O modelo de Florestas Aleatórias, por exemplo, obteve uma acurácia de 0.7185 e um *f1_score* de 0.5400, evidenciando dificuldades em capturar adequadamente a classe

minoritária, refletida no *recall* de apenas 0.3858. Apesar disso, sua alta precisão (0.9010) sugere um bom desempenho na identificação correta da classe majoritária.

O *Gradient Boosting* apresentou um desempenho consistente, com uma acurácia de 0.9344 e um *f1_score* de 0.9191, indicando sua eficácia na classificação geral. Seu AUC de 0.9782 demonstra alta capacidade de discriminação, embora seu *recall* de 0.8696 revele alguma dificuldade em capturar todos os verdadeiros positivos.

O KNN destacou-se com um *recall* perfeito (1.0000) e um *f1_score* de 0.8846, mas a precisão de 0.7930 sugere uma propensão a falsos positivos. O *Naive Bayes*, como esperado devido à sua simplicidade, apresentou o menor desempenho geral, com acurácia de 0.6055 e baixa precisão (0.5209), apesar do *recall* elevado (0.9898).

Os modelos de redes neurais se sobressaíram. O DNN alcançou uma acurácia de 0.9655 e um *f1_score* de 0.9601, enquanto o CNN teve o melhor desempenho geral, com acurácia de 0.9847, *f1_score* de 0.9824 e AUC de 0.9979, evidenciando sua robustez e capacidade de generalização.

Após o ajuste de hiperparâmetros (Tabela 3.14), todos os modelos apresentaram melhorias substanciais. A Floresta Aleatória alcançou uma acurácia de 0.9816 e um *f1_score* de 0.9781, com uma precisão excepcional de 0.9969, destacando-se como o modelo mais robusto após o ajuste.

O *Gradient Boosting* também manteve um desempenho elevado, com acurácia de 0.9758 e um *f1_score* de 0.9713. O KNN, embora tenha mostrado melhorias, continuou abaixo de modelos mais avançados, como o DNN e o CNN, que alcançaram acurácias de 0.9781 e 0.9868, respectivamente.

O CNN foi o modelo mais eficaz, combinando alta precisão (0.9704), *recall* (0.9997) e AUC (0.9985), demonstrando sua superioridade em tarefas de classificação complexa.

Em resumo, o caso real reforça a importância do ajuste de hiperparâmetros para melhorar o desempenho dos modelos. O CNN e a Floresta Aleatória destacaram-se como os mais robustos, oferecendo excelentes resultados tanto no cenário inicial quanto após o ajuste.

Análise da Importância das Variáveis

Os resultados da análise de importância das características utilizadas nos modelos de Florestas Aleatórias e *Gradient Boosting* demonstram que algumas variáveis possuem um impacto significativo na predição da variável alvo. Como mostrado na Tabela 3.15, a variável **guild** apresentou a maior importância (0.261), indicando que a categorização funcional é um fator determinante para o modelo, ou seja, o grupo de polinizadores que usam os mesmos recursos de forma similar (abelhas, vespas, borboletas, moscas) é um fator importante para a interação com as plantas. Além disso, a variável **season** (0.181) também teve um peso relevante, sugerindo que a estação do ano influencia fortemente a interação entre espécies. Outras variáveis, como **body** (0.114) e **diameter** (0.100), também contribuíram para a predição, embora com um peso menor, mostra-se que o tamanho do corpo do polinizador e o diâmetro das plantas é relevante para a interação. Essas descobertas reforçam a necessidade de considerar aspectos sazonais e morfológicos ao estudar o fenômeno em questão.

Característica	Importância
'guild'	0.261216
'season'	0.180965
'body'	0.114084
'diameter'	0.100052
'type'	0.070951
'corolla'	0.053666
'inflorescence'	0.052258
'tongue'	0.046120
's.pollination'	0.030973
'colour'	0.028949

Tabela 3.15: Importância das Características nos modelos Floresta Aleatória e *Gradient Boosting*

3.2 Resultados da Regressão

Nesta seção, realizamos uma avaliação detalhada do desempenho de diferentes modelos de regressão aplicados a diversos conjuntos de dados. O principal objetivo foi analisar a correlação de Spearman, utilizada como métrica para os problemas de regressão no artigo de (PICHLER et al., 2020). Cada subseção é dedicada a um conjunto de dados específico, apresentando uma tabela de resultados e uma análise criteriosa do desempenho de cada modelo.

3.2.1 Desempenho em `R1`

O conjunto de dados `R1` apresenta características que permitem avaliar a habilidade dos modelos em capturar relações monotônicas em um contexto com complexidade moderada. A Tabela 3.16 exibe os resultados dos modelos de regressão, com e sem a otimização de hiperparâmetros, utilizando a correlação de Spearman como métrica de desempenho.

Modelo	Spearman	Spearman - Modelos com Hiperparâmetros
Gradient Boosting	**0.6545**	0.6698
Florestas Aleatórias	0.5381	0.6586
KNN	0.6449	**0.7101**
Função de Perda Binomial Negativa	0.4528	0.3760

Tabela 3.16: Desempenho de modelos de regressão no conjunto de dados `R1`, comparando a correlação de Spearman sem a otimização de hiperparâmetros e com os melhores hiperparâmetros.

Parâmetros selecionados pelo `GridSearch`:

- ***Gradient Boosting*:** {'learning_rate': 0.2, 'max_depth': 7, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 200}
- **Florestas Aleatórias:** {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'distance'}

- **Função de Perda Binomial Negativa:** {'activation': 'relu', 'batch_size': 16, 'epochs': 20, 'neurons': 128, 'optimizer': 'adam'}

Ao analisar o desempenho para os conjuntos de dados de Regressão, o *Gradient Boosting* apresentou resultados consistentes, com a correlação de Spearman aumentando de 0.6545 para 0.6698 após a otimização dos hiperparâmetros. Essa melhoria, embora modesta, indica que ajustes como maior profundidade das árvores e número de estimadores contribuíram para um modelo mais refinado na captura de padrões monotônicos.

As Florestas Aleatórias demonstraram uma significativa melhoria de desempenho, com a correlação de Spearman subindo de 0.5381 para 0.6586. O ajuste dos parâmetros, como a profundidade máxima e o número de estimadores, foi crucial para ampliar a capacidade do modelo de explorar as relações complexas nos dados, tornando-o mais competitivo em relação a outros modelos.

O KNN foi o modelo que mais se beneficiou dos ajustes. Sua correlação de Spearman aumentou de 0.6449 para 0.7101, evidenciando que o uso de métricas de distância otimizadas e a redução do número de vizinhos próximos resultaram em um modelo mais adaptado às características do conjunto de dados `R1`.

Por outro lado, o DNN com Função de Perda Binomial Negativa apresentou uma queda de desempenho, com a correlação de Spearman diminuindo de 0.4528 para 0.3760 após os ajustes. Isso sugere que a combinação de hiperparâmetros aplicados pode não ter sido ideal para este conjunto de dados, ou que o modelo possui limitações intrínsecas ao lidar com relações monotônicas mais complexas.

Os resultados mostram que o KNN e as Florestas Aleatórias destacaram-se no conjunto de dados `R1`, especialmente após a otimização dos hiperparâmetros. O *Gradient Boosting* também apresentou um desempenho sólido, mas sua melhoria foi menos expressiva. Em contrapartida, a Função de Perda Binomial Negativa foi a menos eficiente, indicando que modelos baseados em abordagens distintas podem ser mais adequados para esse tipo de tarefa.

3.2.2 Desempenho em `R2`

O conjunto de dados `R2` é caracterizado por uma estrutura mais complexa, onde as relações monotônicas podem variar em força entre diferentes variáveis. Nesta subseção, avaliamos o desempenho dos modelos de regressão para capturar essas relações. A Tabela 3.17 apresenta os resultados dos modelos de regressão com a correlação de Spearman, tanto sem como com a otimização de hiperparâmetros.

Modelo	**Spearman**	**Spearman - Modelos com Hiperparâmetros**
Gradient Boosting	0.6865	0.7175
Florestas Aleatórias	0.4775	0.6882
KNN	0.4666	0.5169
Função de Perda Binomial Negativa	**0.7631**	**0.7734**

Tabela 3.17: Desempenho de modelos de regressão no conjunto de dados `R2`, comparando a correlação de Spearman sem a otimização de hiperparâmetros e com os melhores hiperparâmetros.

Parâmetros selecionados pelo `GridSearch`:

- **Gradient Boosting:** {'learning_rate': 0.1, 'max_depth': 7, 'min_samples_leaf': 1, 'min_samples_split': 10, 'n_estimators': 100}
- **Florestas Aleatórias:** {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 10, 'n_estimators': 50}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'uniform'}
- **Função de Perda Binomial Negativa:** {'activation': 'relu', 'batch_size': 16, 'epochs': 10, 'neurons': 32, 'optimizer': 'adam'}

Ao analisar os resultados no conjunto de dados `R2`, observamos que o modelo DNN com Função de Perda Binomial Negativa se destacou, com uma correlação de Spearman inicial de 0.7631, que foi levemente aprimorada para 0.7734 após a otimização dos hiperparâmetros. Este modelo mostrou uma boa capacidade de capturar as relações complexas do conjunto de dados, o que é indicativo de sua robustez para problemas de regressão com estruturas variáveis. O modelo de *Gradient Boosting* apresentou uma melhora considerável de 0.6865 para 0.7175 após o ajuste dos parâmetros, sendo eficaz em capturar relações mais fortes entre variáveis. A escolha de uma taxa de aprendizado menor e um número adequado de estimadores contribuíram para esse aumento no desempenho. O modelo Florestas Aleatórias, embora tenha apresentado uma correlação de 0.4775 sem os ajustes, conseguiu melhorar significativamente para 0.6882 com a otimização dos hiperparâmetros. A profundidade maior e o aumento no número de estimadores contribuíram para o aumento da correlação, indicando que o modelo se adaptou melhor ao comportamento do conjunto de dados. O modelo *K-Nearest Neighbors* (KNN) teve um desempenho um pouco inferior, com correlações de 0.4666 e 0.5169, respectivamente, sem e com ajustes. Embora a melhoria tenha sido modesta, os parâmetros de vizinhos e a métrica de distância manhattan ajudaram a capturar relações mais sutis, mas o desempenho não foi tão competitivo quanto os outros modelos.

Em resumo, os modelos ajustados apresentaram uma melhoria significativa na captura das relações monotônicas do conjunto de dados `R2`, especialmente o modelo DNN de Função de Perda Binomial Negativa, que superou os demais em termos de desempenho. A otimização dos parâmetros foi crucial para melhorar a capacidade preditiva dos modelos de *Gradient Boosting* e Florestas Aleatórias, enquanto o KNN não conseguiu alcançar o mesmo nível de aprimoramento.

3.2.3 Desempenho em `R3`

O conjunto de dados `R3` apresenta um cenário desafiador, caracterizado por relações monotônicas fracas e a presença de potenciais ruídos nos dados, o que torna a tarefa de modelagem de regressão mais difícil. Nesse contexto, as interações entre as variáveis são relativamente pequenas, com um modelo mais simples de interações entre as entidades de conjunto 'A' e 'B', o que contribui para a complexidade do problema. O desempenho dos modelos é apresentado na Tabela 3.18, comparando a correlação de Spearman sem e com a otimização dos hiperparâmetros.

Modelo	Spearman	Spearman - Modelos com Hiperparâmetros
Gradient Boosting	0.4031	0.2650
Florestas Aleatórias	0.2131	0.2385
KNN	0.2094	**0.5240**
Função de Perda Binomial Negativa	**0.5245**	0.4036

Tabela 3.18: Correlação Spearman de diferentes modelos de regressão no conjunto de dados `R3`, comparando a correlação sem otimização de hiperparâmetros e com hiperparâmetros.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': 10, 'min_samples_leaf': 5, 'min_samples_split': 15, 'n_estimators': 100}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'uniform'}
- **DNN:** {'activation': 'tanh', 'batch_size': 16, 'epochs': 20, 'neurons': 128, 'optimizer': 'sgd'}
- ***Gradient Boosting*:** {'learning_rate': 0.2, 'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 100}

Analisando os resultados, é possível observar que, devido à estrutura desbalanceada do conjunto de dados `R3` e à presença de relações mais fracas, nenhum dos modelos conseguiu alcançar um desempenho excepcional. O modelo de *Gradient Boosting*, por exemplo, obteve uma correlação de Spearman inicial de 0.4031, que caiu ainda mais para 0.2650 após a otimização dos hiperparâmetros. Esse desempenho modesto pode ser atribuído à dificuldade do modelo em capturar interações significativas em um cenário com dados ruidosos e relações fracas entre as variáveis.

O modelo de Florestas Aleatórias teve uma correlação de 0.2131 sem a otimização dos parâmetros e melhorou um pouco para 0.2385 com os melhores parâmetros, mas o aumento foi insignificante, o que pode indicar que a complexidade do problema foi muito alta para que esse modelo fosse eficaz. O *K-Nearest Neighbors* (KNN) inicialmente apresentou um desempenho semelhante, com uma correlação de 0.2094, mas foi o modelo que obteve a maior melhoria ao aplicar a otimização de hiperparâmetros, alcançando 0.5240, o que sugere que o ajuste adequado do número de vizinhos e da métrica de distância foi útil para capturar melhor as interações do conjunto de dados.

Finalmente, o modelo DNN de Função de Perda Binomial Negativa obteve a maior correlação inicial (0.5245), mas também experimentou uma diminuição após a otimização dos hiperparâmetros, chegando a 0.4036. Isso demonstra que, apesar de ter sido um bom modelo para o conjunto de dados inicialmente, ele não conseguiu melhorar muito com os ajustes realizados.

Em resumo, os resultados indicam que, em cenários com dados desbalanceados e relações fracas, mesmo a otimização de hiperparâmetros não foi suficiente para garantir um desempenho robusto de todos os modelos. O KNN se destacou entre os modelos, sendo o que apresentou maior melhoria, embora os demais modelos, como o *Gradient Boosting* e Florestas Aleatórias, não conseguiram mostrar grandes avanços, refletindo a dificuldade inerente ao problema.

3.2.4 Desempenho em R4_high

O conjunto de dados R4_high apresenta uma alta proporção de classes, com diferentes tempos de observação e interações entre as variáveis, mas sem abundâncias de espécies. As classes são distribuídas em proporções específicas, variando entre 10%, 25% e 50%, o que cria um desafio adicional para os modelos, especialmente no contexto de classificação. Este cenário exige que os modelos lidem com desequilíbrios nas classes e capturem as variações ao longo do tempo para entender corretamente as relações entre as variáveis. A Tabela 3.19 apresenta o desempenho dos modelos de regressão no conjunto R4_high, comparando a correlação de Spearman sem otimização de hiperparâmetros e com hiperparâmetros.

Modelo	Spearman	Spearman - Modelos com Hiperparâmetros
Gradient Boosting	0.4740	0.3825
Florestas Aleatórias	0.3273	0.2826
KNN	0.2279	0.6117
Função de Perda Binomial Negativa	**0.5527**	**0.4768**

Tabela 3.19: Correlação Spearman de diferentes modelos de regressão no conjunto de dados R4_high, comparando a correlação sem otimização de hiperparâmetros e com hiperparâmetros.

Parâmetros selecionados pelo GridSearch:

- **Florestas Aleatórias:** {'max_depth': 10, 'min_samples_leaf': 5, 'min_samples_split': 2, 'n_estimators': 100}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'uniform'}
- **DNN:** {'activation': 'tanh', 'batch_size': 16, 'epochs': 20, 'neurons': 128, 'optimizer': 'adam'}
- ***Gradient Boosting*:** {'learning_rate': 0.1, 'max_depth': 7, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}

A análise dos resultados para o conjunto de dados R4_high revela que os modelos enfrentaram desafios devido à alta proporção de classes e à distribuição variável das classes ao longo do tempo. O modelo *Gradient Boosting* obteve uma correlação de Spearman inicial de 0.4740, que diminuiu para 0.3825 após a otimização dos hiperparâmetros. Isso indica que, apesar da tentativa de ajuste, o modelo teve dificuldades em lidar com a alta proporção de classes e a variação temporal dos dados.

O modelo de Florestas Aleatórias apresentou uma correlação inicial de 0.3273, que caiu para 0.2826 com os melhores parâmetros. Isso sugere que, embora o modelo tenha mostrado algum valor, sua performance foi limitada por sua capacidade de capturar variações em classes desbalanceadas.

O *K-Nearest Neighbors* teve a menor correlação inicial (0.2279), mas alcançou uma melhora significativa após a otimização, chegando a 0.6117. Isso destaca a importância do ajuste adequado de hiperparâmetros, como o número de vizinhos e a métrica de distância, especialmente em cenários com distribuições desbalanceadas.

Por fim, o modelo DNN de Função de Perda Binomial Negativa obteve uma correlação de 0.5527 inicialmente e apresentou uma diminuição para 0.4768 após a otimização. Embora não tenha sido o modelo com o maior desempenho, ele ainda manteve uma boa capacidade de lidar com as interações entre as variáveis e a distribuição das classes.

Em resumo, os resultados indicam que, no cenário de alta proporção de classes com variação temporal, o ajuste de hiperparâmetros desempenhou um papel importante, mas o desempenho geral dos modelos foi afetado pela dificuldade em capturar as variações nas classes e pelo desbalanceamento dos dados. O *K-Nearest Neighbors* se destacou como o modelo mais robusto, demonstrando que, com ajustes adequados, é possível melhorar significativamente a performance em contextos desafiadores como o de `R4_high`.

3.2.5 Desempenho em `R4_mid`

O conjunto de dados `R4_mid` caracteriza-se por uma proporção intermediária, oferecendo um cenário equilibrado para a avaliação do desempenho dos modelos de regressão. A seguir, na Tabela 3.20, analisamos como cada modelo se comportou neste contexto, considerando as métricas obtidas.

Modelo	**Spearman**	**Spearman - Modelos com Hiperparâmetros**
Gradient Boosting	0.3005	0.2547
Florestas Aleatórias	0.2185	0.1725
KNN	0.1217	0.3927
Função de Perda Binomial Negativa	**0.3537**	**0.3176**

Tabela 3.20: Correlação Spearman de diferentes modelos de regressão no conjunto de dados `R4_mid`, comparando a correlação sem otimização de hiperparâmetros e com hiperparâmetros.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': 10, 'min_samples_leaf': 5, 'min_samples_split': 15, 'n_estimators': 100}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'uniform'}
- **DNN:** {'activation': 'tanh', 'batch_size': 32, 'epochs': 20, 'neurons': 128, 'optimizer': 'adam'}
- ***Gradient Boosting*:** {'learning_rate': 0.2, 'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}

A análise dos resultados para o conjunto `R4_mid` evidencia que os modelos enfrentaram desafios para capturar relações moderadas entre as variáveis. O modelo *Gradient Boosting* apresentou uma correlação inicial de 0.3005, que diminuiu para 0.2547 após a otimização de hiperparâmetros, indicando dificuldades em ajustar os parâmetros ao cenário intermediário.

O modelo de Florestas Aleatórias começou com uma correlação de 0.2185 e reduziu para 0.1725 após os ajustes, demonstrando limitações no tratamento da proporção intermediária e das variações entre classes.

Por outro lado, o modelo *K-Nearest Neighbors* se destacou com uma correlação inicial de 0.1217, que aumentou significativamente para 0.3927 após a otimização. Isso sugere que ajustes precisos, como a escolha adequada da métrica de distância e do número de vizinhos, podem melhorar consideravelmente o desempenho, mesmo em cenários desafiadores.

Por fim, o modelo DNN baseado na Função de Perda Binomial Negativa apresentou uma correlação inicial de 0.3537, reduzindo ligeiramente para 0.3176 após a otimização. Apesar dessa diminuição, o modelo demonstrou uma robustez relativa em capturar padrões nos dados intermediários.

Em resumo, o conjunto `R4_mid` revelou um desempenho geral fraco para a maioria dos modelos. No entanto, o *K-Nearest Neighbors* destacou-se como o mais adaptável, demonstrando a importância de ajustes específicos para lidar com cenários equilibrados e de proporção moderada. Nesse sentido, melhores ajustes de hiperparâmetros e técnicas para lidar com dados desbalanceados são necessários para melhorer a performance nesse caso.

3.2.6 Desempenho em `R4_low`

O conjunto de dados `R4_low` é caracterizado por baixa proporção. A seguir, avaliamos o desempenho dos modelos nesse cenário desafiador, apresentado na Tabela 3.21.

Modelo	**Spearman**	**Spearman - Modelos com Hiperparâmetros**
Gradient Boosting	0.1303	0.0823
Florestas Aleatórias	0.0900	0.0704
KNN	0.0402	**0.1988**
Função de Perda Binomial Negativa	**0.1609**	0.1385

Tabela 3.21: Correlação Spearman de diferentes modelos de regressão no conjunto de dados `R4_low`, comparando a correlação sem otimização de hiperparâmetros e com hiperparâmetros.

Parâmetros selecionados pelo `GridSearch`:

- **Florestas Aleatórias:** {'max_depth': 2, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50}
- **KNN:** {'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'uniform'}
- **DNN:** {'activation': 'tanh', 'batch_size': 32, 'epochs': 10, 'neurons': 32, 'optimizer': 'adam'}
- ***Gradient Boosting*:** {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 100}

Os resultados para o cenário `R4_low` foram os mais fracos entre todos os conjuntos avaliados, destacando o impacto severo de trabalhar com baixa proporção de classes. O *Gradient Boosting*, que geralmente apresenta boa robustez em cenários variados, teve um desempenho inicial muito limitado (correlação de 0.1303), que piorou ainda mais após a otimização, caindo para 0.0823. Esse comportamento evidencia uma dificuldade do modelo em ajustar-se aos padrões escassos do conjunto.

O modelo de Florestas Aleatórias apresentou uma correlação inicial de apenas 0.0900, reduzida para 0.0704 após a otimização. Esses valores mostram que o modelo foi praticamente incapaz de capturar qualquer padrão significativo, sendo altamente penalizado pela baixa variabilidade das classes.

O *K-Nearest Neighbors*, que obteve a menor correlação inicial de todos os modelos (0.0402), conseguiu uma melhoria considerável após o ajuste de hiperparâmetros, alcançando 0.1988. Apesar dessa recuperação, o desempenho permanece muito abaixo do aceitável para um problema de regressão, destacando as limitações impostas pelo conjunto de dados.

Por fim, o DNN com Função de Perda Binomial Negativa apresentou o melhor desempenho inicial (0.1609), mas não conseguiu sustentar uma melhora significativa após a otimização, registrando uma correlação reduzida de 0.1385. Ainda que tenha superado outros modelos em termos absolutos, o resultado é insuficiente para aplicações práticas.

Em resumo, o desempenho no conjunto `R4_low` foi substancialmente ruim para todos os modelos, com correlações de Spearman baixíssimas, mesmo após ajustes. Esse cenário expôs as limitações intrínsecas dos modelos ao lidar com proporções extremamente reduzidas e padrões escassos, sugerindo que abordagens alternativas ou pré-processamentos específicos podem ser necessários para melhorar os resultados em situações similares.

Conclusão

No Capítulo 1, exploramos as funções implementadas em `Python`, onde foram realizadas otimizações no código original, reduzindo parâmetros desnecessários nos laços de repetição. Além disso, promovemos a modularização do código, tornando-o mais limpo e enfatizando as funções principais da biblioteca `TraitMatching`. Dessa forma, obtivemos um código mais eficiente em uma linguagem moderna, expandindo a aplicabilidade e universalização desta biblioteca.

No Capítulo 2, implementamos o código `Prediction`, no qual exploramos quatro simulações, além de um conjunto de dados de espécies reais. Utilizamos os mesmos modelos de aprendizagem de máquina do código proposto por (PICHLER et al., 2020) e, indo além, realizamos otimizações com hiperparâmetros, identificando a teoria e a estrutura por meio das funções `main_classification` e `main_regression`. Também propusemos funções específicas para a otimização desses modelos.

No Capítulo 3, apresentamos uma série de tabelas com o desempenho dos conjuntos de dados mencionados anteriormente, tanto com quanto sem ajustes de hiperparâmetros. Observamos que, de modo geral, os modelos de Florestas Aleatórias, Redes Neurais Profundas (DNN) e Redes Neurais Convolucionais (CNN) apresentaram um desempenho superior nos casos de classificação, em relação aos outros modelos. Para os problemas de regressão, os modelos *Gradient Boosting*, Florestas Aleatórias e DNN com função de perda Binomial Negativa tiveram os melhores resultados.

Nesse sentido, o ajuste de hiperparâmetros foi essencial na maioria dos casos para melhorar o desempenho dos modelos em identificar as classes. Além disso, a proporção e o desbalanceamento dos dados contribuíram para um desempenho inferior, especialmente nos problemas de regressão, evidenciando a necessidade de técnicas adequadas de balanceamento para lidar com esses casos.

Em resumo, o uso de `Python` demonstrou-se eficaz na replicação das principais funções da biblioteca original em `R`. As otimizações realizadas mostraram-se poderosas para melhorar o desempenho na detecção das interações entre as características morfológicas das espécies. O próximo passo deste projeto será utilizar os conjuntos de dados das funções originais em `R` para comparar diretamente a eficiência das linguagens e das otimizações aplicadas nos modelos de aprendizagem de máquina em `Python`. Isso se faz necessário, uma vez que a execução da semente em `Python` gerou um conjunto de dados diferente do original, o que exige uma investigação mais detalhada para identificar onde ocorreu essa diferença. Cabe destacar que a tradução para `Python` foi conduzida com rigor, e as otimizações foram cuidadosamente avaliadas para garantir que os parâmetros removidos realmente contribuíssem para a melhoria da execução.

Concluímos que, apesar das questões levantadas, este projeto já se mostra útil e plenamente capaz de processar dados reais em `Python`, evidenciando sua relevância e

aplicabilidade. Ele não apenas demonstrou a viabilidade do uso de `Python` em aplicações práticas, como também abriu caminho para avanços contínuos na otimização de modelos e na análise de dados ecológicos. Os resultados alcançados são promissores e reforçam o potencial deste trabalho para fundamentar pesquisas futuras e aplicações que exijam ferramentas adaptáveis a diferentes contextos e demandas, como a modelagem da interação entre abelhas e flores na polinização de culturas agrícolas, auxiliando na conservação e no manejo sustentável dos ecossistemas.

Referências Bibliográficas

FAEGRI, K.; PIJL, L. van der. *The principles of pollination ecology*. Oxford: Pergamon Press, 1979. ID - 19790561579.

FENSTER, C. B. et al. Pollination syndromes and floral specialization. *Annual Review of Ecology Evolution and Systematics*, v. 35, p. 375–403, 2004.

FRIEDMAN, J. H. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, v. 29, n. 5, p. 1189–1232, 2001.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. MIT Press, 2016. Capítulo 6. Disponível em: ⟨http://www.deeplearningbook.org⟩.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd. ed. [S.l.]: Springer, 2009.

KLEIN, A. M. et al. Importance of pollinators in changing landscapes for world crops. *Proceedings of the Royal Society B: Biological Sciences*, v. 274, n. 1608, p. 303–313, 2007.

OLLERTON, J. et al. A global test of the pollination syndrome hypothesis. *Annals of Botany*, v. 103, p. 1471–1480, 2009.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, v. 12, p. 2825–2830, 2011.

PICHLER, M. et al. Machine learning algorithms to infer trait-matching and predict species interactions in ecological networks. *Methods in Ecology and Evolution*, v. 11, p. 281–293, 2020.

POTTS, S. G. et al. Global pollinator declines: trends, impacts and drivers. *Trends in Ecology & Evolution*, v. 25, n. 6, p. 345–353, 2010.

ROSAS-GUERRERO, V. et al. A quantitative review of pollination syndromes: Do floral traits predict effective pollinators? *Ecology Letters*, v. 17, p. 388–400, 2014.

SCHLEDER, G. R.; FAZZIO, A. Machine learning na física, química, e ciência de materiais: Descoberta e design de materiais. *Revista Brasileira de Ensino de Física*, v. 43, p. e20200407, 2021.

VANBERGEN, A. J. et al. Threats to an ecosystem service: pressures on pollinators. *Frontiers in Ecology and the Environment*, v. 11, n. 5, p. 251–259, 2013.