MACHINE LEARNING Ciclo 2 - Capacitação SOFTEX IA/IoT
Professor: Thiago José Marques Moura

DATASET
CROP RECOMMENDATION

—

Ronaldo Urquiza Herculano Filho

INTRODUÇÃO

Esse documento trata da análise e usabilidade do dataset de classificação “Crop Recommendation” criado pelo MIT e disponibilizado na plataforma Kaggle.

Acesso:

https://www.kaggle.com/datasets/varshitanalluri/crop-recommendation-dataset/data

Diretório de todos os códigos presentes no projeto:

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation

```
   Nitrogen  Phosphorus  Potassium  Temperature   Humidity  pH_Value    Rainfall  Crop
0        90          42         43    20.879744  82.002744  6.502985  202.935536  Rice
1        85          58         41    21.770462  80.319644  7.038096  226.655537  Rice
2        60          55         44    23.004459  82.320763  7.840207  263.964248  Rice
3        74          35         40    26.491096  80.158363  6.980401  242.864034  Rice
4        78          42         42    20.130175  81.604873  7.628473  262.717340  Rice

Tamanho do dataset (linhas, colunas) -> (2200, 8)
```

Dataset

Printando as 5 primeiras linhas do DS usando o head() e o 'display_max_columns'.

Features

As features desse dataset são elementos químicos (Nitrogênio, Fósforo e Potássio [NPK]) e grandezas físico-químicas (Temperatura, Umidade, pH e Quantidade de chuva) presentes em cada amostra de solo que definem o tipo de grão que melhor se adaptará às condições daquela amostra.

Foram executados uma série de comandos (métodos) em uma estrutura que vou chamar de "comandos de verificação" para analisar os dados de cada coluna de feature, são eles:

```
DS."nome da coluna".dtype #Verificar o tipo de dados das colunas

DS."nome da coluna".isnull().sum() #Verficar valores vazios na coluna

DS."nome da coluna".isna().sum() #Verificar valores NaN na coluna

DS."nome da coluna".mean() #Verificar média dos valores da coluna

DS."nome da coluna".max() #Verificar o valor máximo da coluna

DS."nome da coluna".min() #Verificar o valor mínimo da coluna

DS."nome da coluna".value_counts() #Verificar as maiores
ocorrências de valores específicos na coluna
```

Elementos químicos:

1. Nitrogênio

A principal função do nitrogênio para as plantas é promover o crescimento e desenvolvimento ao ser um componente essencial na síntese de proteínas, clorofila e ácidos nucleicos. Isso é vital para a fotossíntese, a formação de estruturas celulares e a reprodução genética.

Em termos estatísticos do dataset:

Essa coluna lista valores proporcionais de nitrogênio no solo a cada linha

Ex.: "22": O valor 22 É a proporção do conteúdo de Nitrogênio naquela amostra de solo (cada amostra de solo é representada por 1 linha no dataset).

- **Tipo dos valores da feature: int64;**
- **Ocorrências de valores vazio: 0;**
- **Ocorrências de valores NaN (not a number): 0;**
- **Média: 50.551;**
- **Valor máximo: 140;**
- **Valor mínimo: 0;**
- **Ocorrência de valores: 1º lugar -> valor 22 (44 ocorrências);**
  2º lugar -> valor 40 (44 ocorrências);
  3º lugar -> valor 27 (42 ocorrências).

```
Feature Nitrogênio:

Tipo dos valores que aparecem nessa feature: int64
Valores NaN (Not a Number) da feature: 0
Valores vazios da feature: 0
Média dos valores da feature: 50.551818181818184
Valor máximo da feature: 140
Valor mínimo da feature: 0
```

```
Contagem de cada valor de Nitrogen
22     44
40     44
27     42
39     41
31     41
       ..
136     2
139     1
135     1
130     1
46      1
Name: count, Length: 137, dtype: int64
```

Imagens do PyCharm com os comandos de verificação

2. Fósforo

A principal função do fósforo para as plantas é promover o desenvolvimento de raízes, a floração e a frutificação, além de ser crucial para o armazenamento e transferência de energia. O fósforo é um componente essencial do ATP (adenosina trifosfato), fundamental para a transferência de energia dentro das células, e também é parte dos ácidos nucleicos e das membranas celulares.

Em termos estatísticos do dataset:

Essa coluna lista valores proporcionais de fósforo no solo a cada linha

Ex.: "60": O valor 60 É a proporção do conteúdo de Fósforo naquela amostra de solo (cada amostra de solo é representada por 1 linha no dataset).

- **Tipo dos valores da feature: int64;**
- **Ocorrências de valores vazio: 0;**
- **Ocorrências de valores NaN (not a number): 0;**
- **Média: 53.3627;**
- **Valor máximo: 145;**
- **Valor mínimo: 5;**
- **Ocorrência de valores: 1º lugar -> valor 60 (56 ocorrências);**
  2º lugar -> valor 58 (48 ocorrências);
  3º lugar -> valor 56 (46 ocorrências).

```
Feature Fósforo:

Tipo dos valores que aparecem nessa feature: int64
Valores NaN (Not a Number) da feature: 0
Valores vazios da feature: 0
Média dos valores da feature: 53.36272727272727
Valor máximo da feature: 145
Valor mínimo da feature: 5
```

```
Contagem de cada valor de Phosphorus
60    56
58    48
56    46
55    44
57    42
      ..
83     2
82     2
90     2
93     1
84     1
```

Imagens do PyCharm com os comandos de verificação

3. Potássio

A principal função do potássio para as plantas é regular processos fisiológicos essenciais, incluindo a abertura e fechamento dos estômatos, que controlam a troca de gases e a transpiração. O potássio também é crucial para a ativação de enzimas envolvidas na síntese de proteínas e carboidratos, contribuindo para o crescimento geral da planta, resistência a doenças e tolerância a estresses ambientais, como seca e frio.

Em termos estatísticos do dataset:

Essa coluna lista valores proporcionais de potássio no solo a cada linha

Ex.: "17": O valor 17 É a proporção do conteúdo de Potássio naquela amostra de solo (cada amostra de solo é representada por 1 linha no dataset).

- **Tipo dos valores da feature: int64;**
- **Ocorrências de valores vazio: 0;**
- **Ocorrências de valores NaN (not a number): 0;**
- **Média: 48.149;**
- **Valor máximo: 205;**
- **Valor mínimo: 5;**
- **Ocorrência de valores: 1º lugar -> valor 17 (90 ocorrências);**
  2º lugar -> valor 22 (87 ocorrências);
  3º lugar -> valor 15 (86 ocorrências).

```
Feature Potássio:

Tipo dos valores que aparecem nessa feature: int64
Valores NaN (Not a Number) da feature: 0
Valores vazios da feature: 0
Média dos valores da feature: 48.14909090909091
Valor máximo da feature: 205
Valor mínimo da feature: 5
```

```
Contagem de cada valor de Potassium
17    90
22    87
15    86
20    80
25    78
      ..
5      8
11     8
13     7
7      5
80     4
```

Imagens do PyCharm com os comandos de verificação

Grandezas físico-químicas:

1. Temperatura

A temperatura é crucial para as plantas porque influencia diretamente processos vitais como a fotossíntese e a respiração, afetando a taxa de crescimento. Ela é importante para a germinação das sementes, o desenvolvimento de raízes, folhas e flores, e para a absorção de nutrientes. Além disso, a temperatura adequada ajuda a plantas a serem mais resistentes a pragas e doenças, garantindo um crescimento saudável e produtivo.

Em termos estatísticos do dataset:

Essa coluna lista valores de temperatura, em graus celsius, de cada amostra de solo a cada linha

Ex.: "20.87": O valor 20.87 é a temperatura, em graus celsius, naquela amostra de solo (cada amostra de solo é representada por 1 linha no dataset).

- **Tipo dos valores da feature: float64;**
- **Ocorrências de valores vazio: 0;**
- **Ocorrências de valores NaN (not a number): 0;**
- **Média: 25.61ºC;**
- **Valor máximo: 43.6ºC;**
- **Valor mínimo: 8.82ºC;**
- **Ocorrência de valores: Valores sempre únicos.**

```
Feature Temperatura:

Tipo dos valores que aparecem nessa feature: float64
Valores NaN (Not a Number) da feature: 0
Valores vazios da feature: 0
Média dos valores da feature: 25.616243851779544
Valor máximo da feature: 43.67549305
Valor mínimo da feature: 8.825674745
```

```
Contagem de cada valor de Temperature
20.879744    1
29.480699    1
29.943492    1
28.033065    1
29.884305    1
            ..
25.365861    1
28.568406    1
30.284966    1
27.325421    1
23.603016    1
```

Imagens do PyCharm com os comandos de verificação

2. Umidade

A umidade é essencial para as plantas porque influencia a absorção de água e nutrientes pelas raízes, mantém a turgidez das células, e regula a transpiração, que é crucial para a troca gasosa e a temperatura interna das plantas. Além disso, a umidade adequada ajuda a prevenir o estresse hídrico, promove a germinação das sementes, e contribui para o desenvolvimento saudável das plantas, aumentando sua resistência a pragas e doenças.

Em termos estatísticos do dataset:

Essa coluna lista valores de umidade, em porcentagem, de cada local de amostra de solo a cada linha.

Ex.: "82.00": O valor 82.00 é a umidade percentuada naquela amostra de solo (cada amostra de solo é representada por 1 linha no dataset).

- **Tipo dos valores da feature: float64;**
- **Ocorrências de valores vazio: 0;**
- **Ocorrências de valores NaN (not a number): 0;**
- **Média: 71.48%;**
- **Valor máximo: 99.98%;**
- **Valor mínimo: 14.25%;**
- **Ocorrência de valores: Valores sempre únicos.**

```
Feature Umidade:

Tipo dos valores que aparecem nessa feature: float64
Valores NaN (Not a Number) da feature: 0
Valores vazios da feature: 0
Média dos valores da feature: 71.48177921778637
Valor máximo da feature: 99.98187601
Valor mínimo da feature: 14.25803981
```

```
Contagem de cada valor de Humidity
82.002744    1
90.336987    1
93.907412    1
91.473558    1
94.037115    1
            ..
66.637972    1
61.532786    1
61.692951    1
69.090478    1
60.396475    1
```

Imagens do PyCharm com os comandos de verificação

3. Índice de Acidez e Basicidade (pH)

O pH do solo é importante para as plantas porque afeta a disponibilidade de nutrientes essenciais e a atividade microbiana no solo. Um pH inadequado pode limitar a absorção de nutrientes, levando a deficiências e afetando o crescimento e a saúde das plantas. Cada planta tem uma faixa de pH ideal; fora dessa faixa, certos nutrientes se tornam menos disponíveis, enquanto outros podem se tornar tóxicos. Ajustar o pH do solo para a faixa ideal das plantas cultivadas é fundamental para uma nutrição equilibrada e um desenvolvimento saudável.

Em termos estatísticos do dataset:

Essa coluna lista valores de pH, na escala de pH(0-14), de cada amostra de solo a cada linha.

Ex.: "6.50": O valor 6.50 é o valor de acidez ou basicidade, na escala pH, naquela amostra de solo (cada amostra de solo é representada por 1 linha no dataset).

- **Tipo dos valores da feature: float64;**
- **Ocorrências de valores vazio: 0;**
- **Ocorrências de valores NaN (not a number): 0;**
- **Média: 6.46 (pH neutro);**
- **Valor máximo: 9.93 (pH ácido);**
- **Valor mínimo: 3.50 (pH básico);**
- **Ocorrência de valores: Valores sempre únicos.**

```
Feature pH:

Tipo dos valores que aparecem nessa feature: float64
Valores NaN (Not a Number) da feature: 0
Valores vazios da feature: 0
Média dos valores da feature: 6.469480065256364
Valor máximo da feature: 9.93509073
Valor mínimo da feature: 3.504752314
```

```
Contagem de cada valor de pH_Value
6.502985    1
6.640471    1
6.251420    1
6.274453    1
6.135996    1
           ..
7.538631    1
7.127064    1
6.628265    1
6.726469    1
6.779833    1
```

Imagens do PyCharm com os comandos de verificação

4. Chuva em mm

A quantidade de chuva, medida em milímetros (mm), é fundamental para as plantas porque fornece a água necessária para a fotossíntese, o transporte de nutrientes e a manutenção da turgidez celular. A quantidade adequada de chuva ajuda a garantir que o solo permaneça úmido, permitindo que as plantas absorvam água e nutrientes de maneira eficiente. Excesso de chuva pode causar encharcamento do solo e problemas de oxigenação das raízes, enquanto a falta de chuva pode levar ao estresse hídrico, afetando negativamente o crescimento e a saúde das plantas.

Em termos estatísticos do dataset:

Essa coluna lista valores de quantidades de chuva, medidas em milímetros, de cada local de amostra de solo a cada linha.

Ex.: "202.93": O valor 202.93 é o valor, medido em milímetros, da quantidade de chuba naquele local da amostra de solo (cada amostra de solo é representada por 1 linha no dataset).

- **Tipo dos valores da feature: float64;**
- **Ocorrências de valores vazio: 0;**
- **Ocorrências de valores NaN (not a number): 0;**
- **Média: 103.46 mm de chuva;**
- **Valor máximo: 298.56 mm de chuva;**
- **Valor mínimo: 20.21 mm de chuva;**
- **Ocorrência de valores: Valores sempre únicos.**

```
Feature Chuva em mm:

Tipo dos valores que aparecem nessa feature: float64
Valores NaN (Not a Number) da feature: 0
Valores vazios da feature: 0
Média dos valores da feature: 103.46365541576817
Valor máximo da feature: 298.5601175
Valor mínimo da feature: 20.21126747
```

```
Contagem de cada valor de Rainfall
202.935536    1
26.036577     1
20.390205     1
21.179248     1
21.000099     1
             ..
65.816559     1
63.497263     1
65.628595     1
61.192509     1
140.937041    1
```

Imagens do PyCharm com os comandos de verificação

Visualização de Dados:

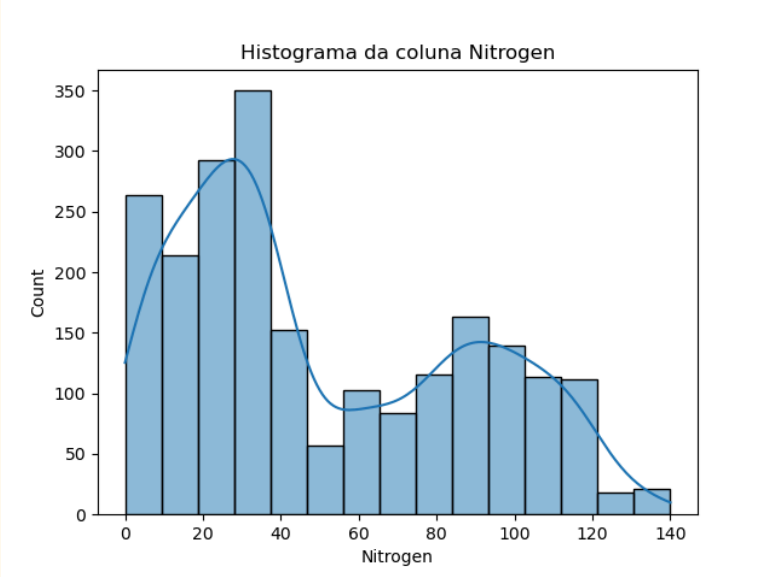
Para ainda melhor visualização dos dados já citados foram empregados os gráficos de histograma e boxplot, através da biblioteca Seaborn com auxílio da biblioetca MatPlotLib.

O código utilizado foi um laço de repetição que contempla todas as features e disponibiliza as imagens, no formato png, dos gráficos na tela da IDE que o usuário está utilizando, sem salvá-las no diretório do código.

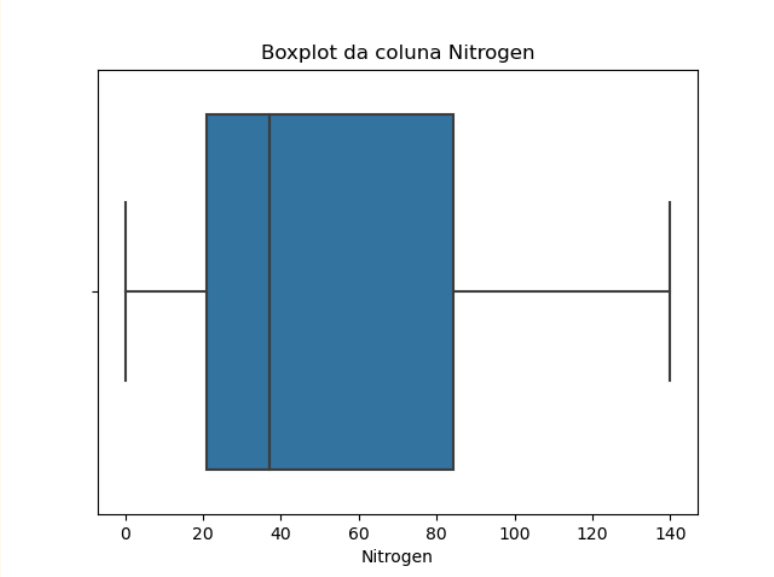
```
# Lista das colunas a serem plotadas

colunas = ['Nitrogen', 'Phosphorus', 'Potassium', 'Temperature', 'Humidity',
'pH_Value', 'Rainfall']

# Loop sobre cada coluna para gerar os histogramas

for coluna in colunas:

   # Plotando o histograma

   sns.histplot(x=coluna, data=DS, kde=True)

   # Adicionando título ao histograma

   plt.title(f'Histograma da coluna {coluna}')

   # Exibindo o histograma

   plt.show()



# Loop sobre cada coluna para gerar os boxplots

for coluna in colunas:

   # Plotando o boxplot

   sns.boxplot(x=coluna, data=DS)

   # Adicionando título ao boxplot

   plt.title(f'Boxplot da coluna {coluna}')

   # Exibindo o boxplot

   plt.show()
```

Nitrogênio:

- Histograma

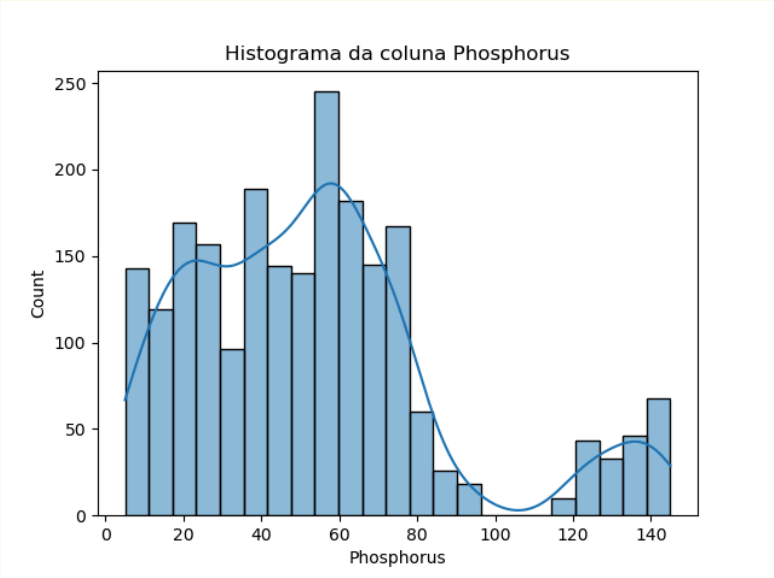


- Boxplot

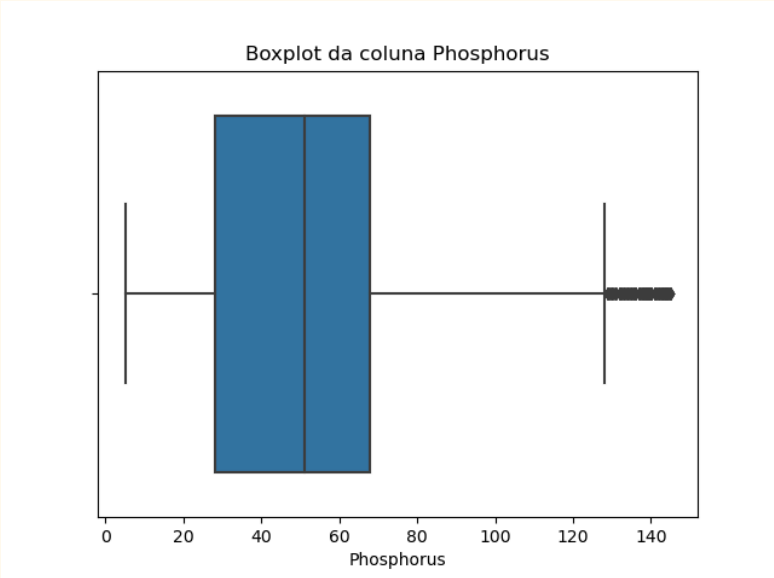


Fósforo:

- Histograma

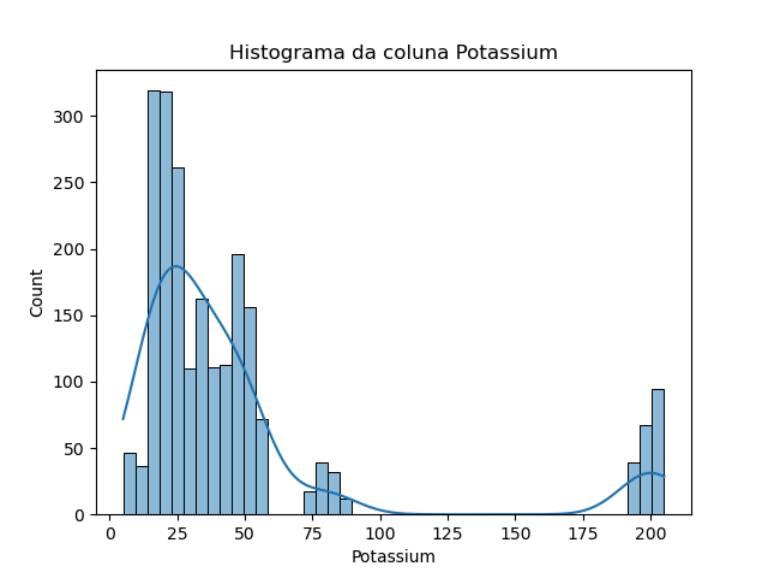


- Boxplot

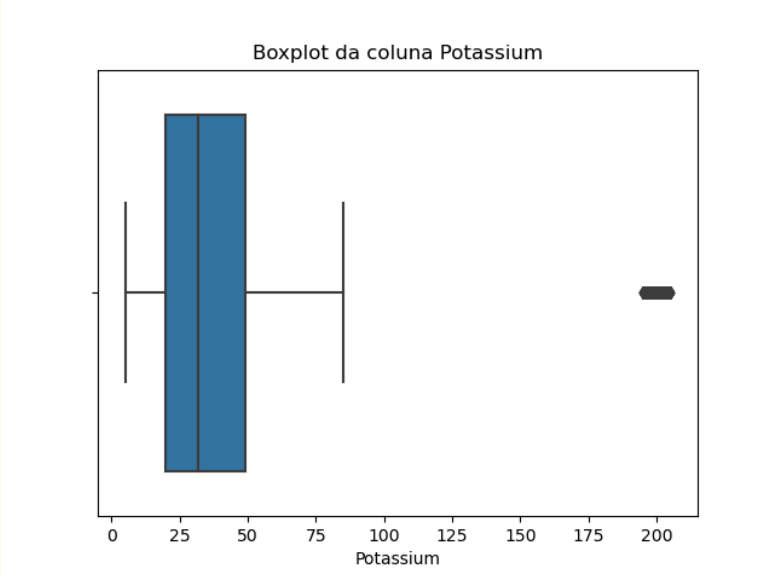


<u>Potássio</u>:

- Histograma

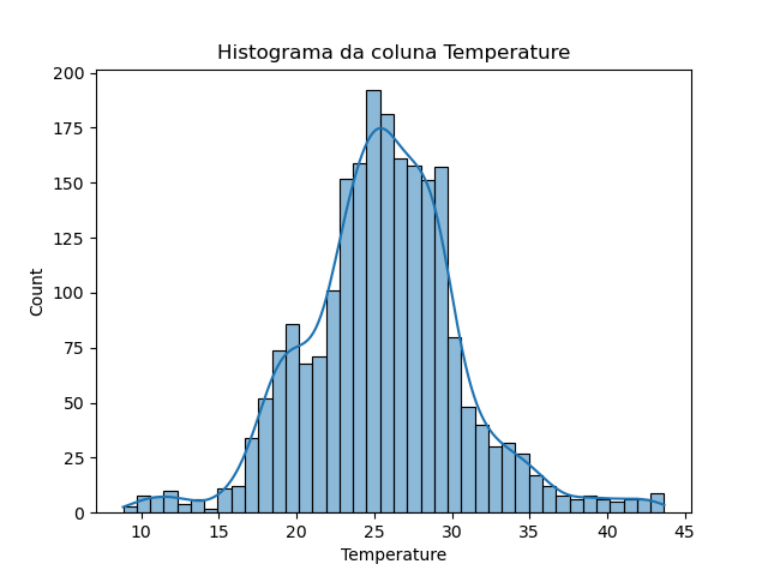


- Boxplot

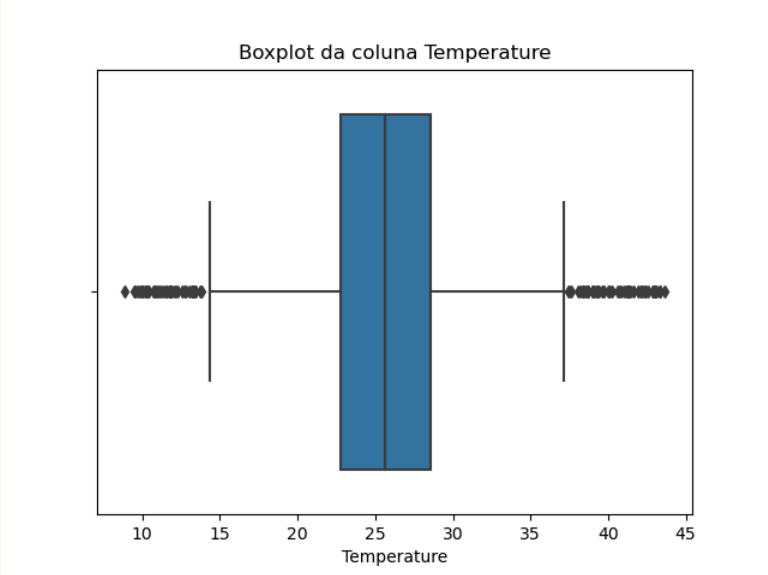


<u>Temperatura</u>:

- Histograma

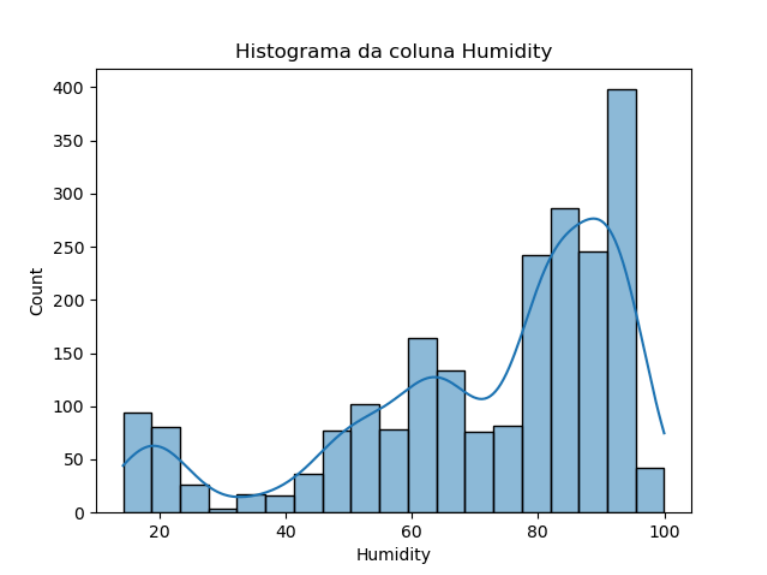


- Boxplot

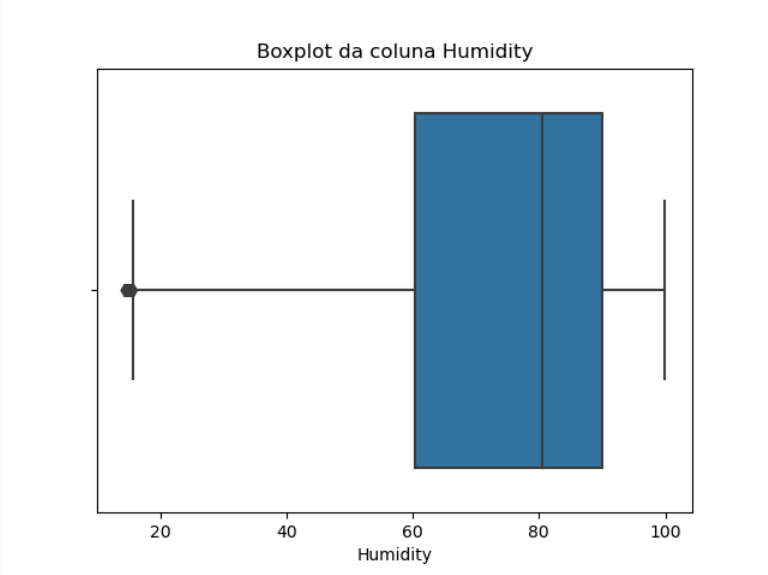


<u>Umidade</u>:

- Histograma

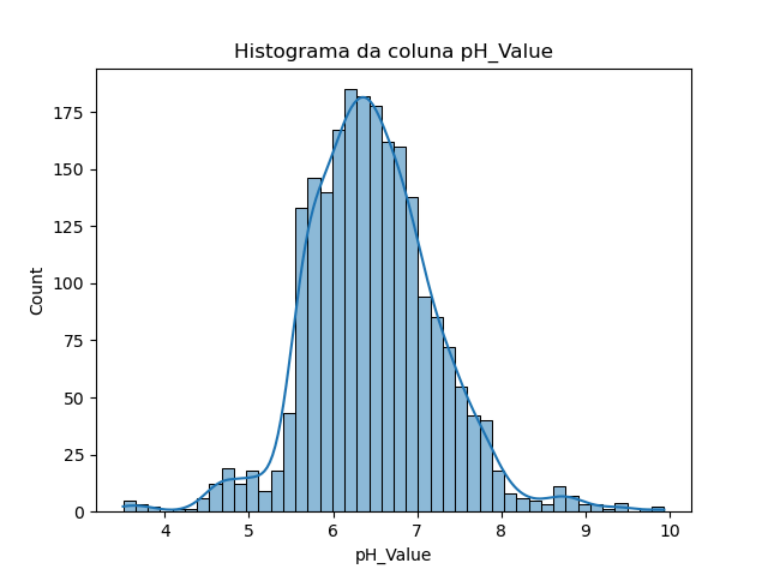


- Boxplot

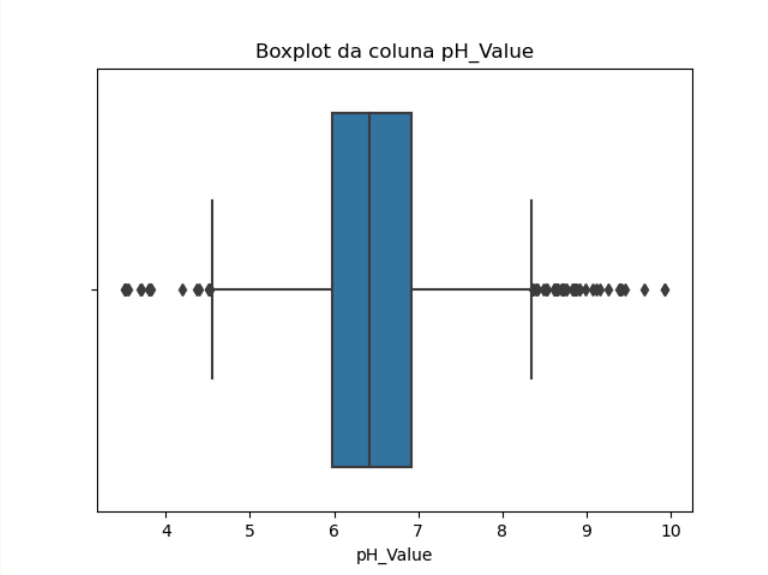


<u>pH</u>:

- Histograma

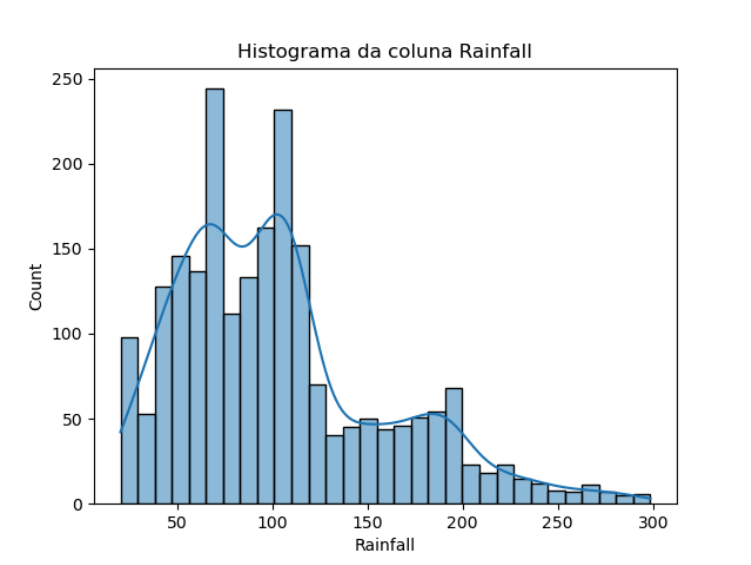


- Boxplot

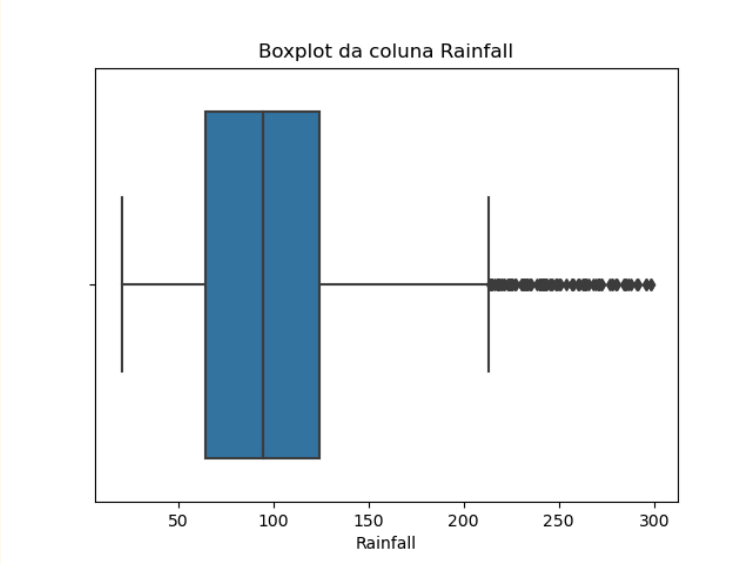


Chuva:

- Histograma



- Boxplot



Acesso dos códigos de gráficos em:
https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/tree/main/Projeto%20-%20IA/01_B%C3%A1sico

Target

Como em todo dataset de classificação o "Crop Recommendation" não é diferente, temos uma target, um alvo, a classe que o modelo tentará inferir baseada em toooodos os valores presentes em cada coluna de features.

Como o nome do dataset já diz, "recomendação de cultivo", a classe será o tipo de planta/grão que sera plantado baseado nas características de cada solo assim como sua localidade, são 22 tipos de plantas com 100 linhas de exemplos de solos próprios para cada uma, esse é o motivo do dataset ter 2200 linhas (22 [22 plantas] * 100 [100 amostras de solos para cada uma]).

Cultivo recomendado:

1. **Rice (Arroz)**

   Nitrogênio: Alto | Fósforo: Médio a Alto | Potássio: Alto | Temperatura: Quente a Moderada | Umidade: Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Alta.

2. Maize (Milho)

Nitrogênio: Alto | Fósforo: Médio | Potássio: Médio a Alto | Temperatura: Quente | Umidade: Moderada a Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Moderada a Alta.

3. Chickpea (Grão-de-bico)

Nitrogênio: Médio a Alto | Fósforo: Médio | Potássio: Médio | Temperatura: Moderada a Quente | Umidade: Moderada | pH do Solo: Neutro a Alcalino | Precipitação (Chuva): Moderada.

4. KidneyBeans (Feijão Roxo)

Nitrogênio: Alto | Fósforo: Médio | Potássio: Médio | Temperatura: Quente | Umidade: Moderada | pH do Solo: Neutro a Alcalino | Precipitação (Chuva): Moderada.

5. PigeonPeas (Feijão-guandu ou Feijão-de-corda)

Nitrogênio: Médio | Fósforo: Médio | Potássio: Médio | Temperatura: Quente | Umidade: Moderada a Alta | pH do Solo: Neutro a Alcalino | Precipitação (Chuva): Moderada a Alta.

6. MothBeans (Feijões-moth ou Feijões-matki)

Nitrogênio: Alto | Fósforo: Médio | Potássio: Médio | Temperatura: Quente | Umidade: Moderada | pH do Solo: Neutro a Alcalino | Precipitação (Chuva): Moderada.

7. MungBean (Feijão-verde ou Feijões-mungo)

Nitrogênio: Alto | Fósforo: Médio | Potássio: Médio | Temperatura: Quente | Umidade: Moderada a Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Moderada a Alta.

8. Blackgram (Feijão-preto ou Feijão-da-china)

Nitrogênio: Alto | Fósforo: Médio | Potássio: Médio | Temperatura: Quente | Umidade: Moderada a Alta | pH do Solo: Neutro a Alcalino | Precipitação (Chuva): Moderada a Alta.

9. Lentil (Lentilha)

Nitrogênio: Médio | Fósforo: Médio | Potássio: Médio | Temperatura: Moderada | Umidade: Moderada a Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Moderada.

10. Pomegranate (Romã)

Nitrogênio: Baixo a Médio | Fósforo: Baixo a Médio | Potássio: Médio | Temperatura: Quente | Umidade: Baixa a Moderada | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Baixa a Moderada.

11. Banana (Banana)

Nitrogênio: Alto | Fósforo: Alto | Potássio: Alto | Temperatura: Quente | Umidade: Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Alta.

12. Mango (Manga)

Nitrogênio: Médio a Alto | Fósforo: Médio a Alto | Potássio: Alto | Temperatura: Quente | Umidade: Moderada a Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Moderada a Alta.

13. Grapes (Uva)

Nitrogênio: Baixo a Médio | Fósforo: Médio | Potássio: Médio | Temperatura: Moderada a Quente | Umidade: Baixa a Moderada | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Moderada.

14. Watermelon (Melancia)

Nitrogênio: Alto | Fósforo: Médio a Alto | Potássio: Alto | Temperatura: Quente | Umidade: Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Alta.

15. Muskmelon (Melão)

Nitrogênio: Alto | Fósforo: Médio a Alto | Potássio: Alto | Temperatura: Quente | Umidade: Moderada a Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Moderada a Alta.

16. Apple (Mação)

Nitrogênio: Baixo a Médio | Fósforo: Baixo a Médio | Potássio: Baixo a Médio | Temperatura: Moderada | Umidade: Baixa a Moderada | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Baixa a Moderada.

17. Orange (Laranja)

Nitrogênio: Baixo a Médio | Fósforo: Baixo a Médio | Potássio: Baixo a Médio | Temperatura: Quente | Umidade: Baixa a Moderada | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Baixa a Moderada.

18. Papaya (Mamão)

Nitrogênio: Alto | Fósforo: Alto | Potássio: Alto | Temperatura: Quente | Umidade: Moderada a Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Moderada a Alta.

19. Coconut (Coco)

Nitrogênio: Baixo a Médio | Fósforo: Baixo a Médio | Potássio: Alto | Temperatura: Quente | Umidade: Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Alta.

20. Cotton (Algodão)

Nitrogênio: Alto | Fósforo: Alto | Potássio: Alto | Temperatura: Quente | Umidade: Moderada a Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Moderada a Alta.

21. Jute (Juta)

Nitrogênio: Alto | Fósforo: Alto | Potássio: Alto | Temperatura: Quente | Umidade: Alta | pH do Solo: Neutro a Ácido | Precipitação (Chuva): Alta.

22. Coffe (Café)

Nitrogênio: Alto | Fósforo: Médio | Potássio: Médio a Alto | Temperatura: Moderada | Umidade: Alta | pH do Solo: Ácido | Precipitação (Chuva): Moderada a Alta.

Visualização de Dados:

Para melhorar ainda mais a visualização das classes já citadas foram empregados os gráficos de barras e setores, através da biblioteca Seaborn com auxílio da biblioetca MatPlotLib e da biblioteca PANDAS.

O código utilizado para criar o gráfico de barras foi o:

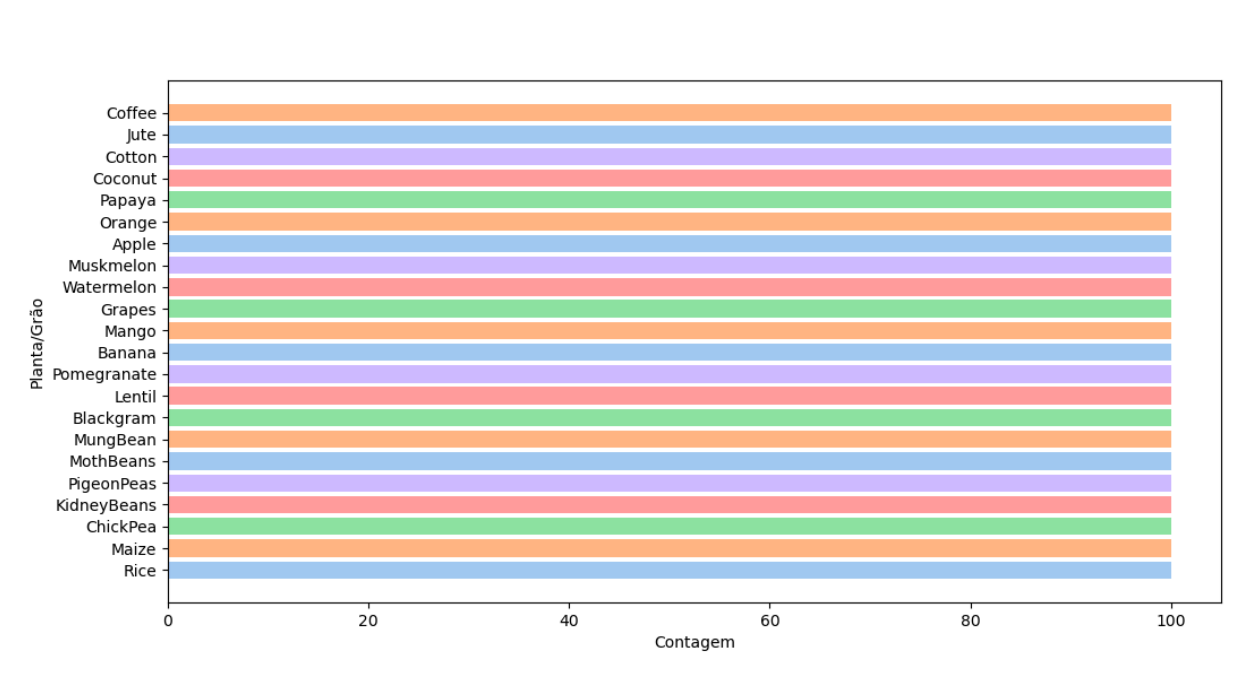
```
# Define a paleta de core através do Seaborn

colors = sns.color_palette('pastel')[0:5]

#Cria gráfico de barras

plt.figure(figsize=(12, 6), facecolor='#FFFFFF')

plt.barh(DS['Crop'].unique(), DS['Crop'].value_counts(), color=colors)

plt.xlabel('Contagem')

plt.ylabel('Planta/Grão')

plt.show()
```

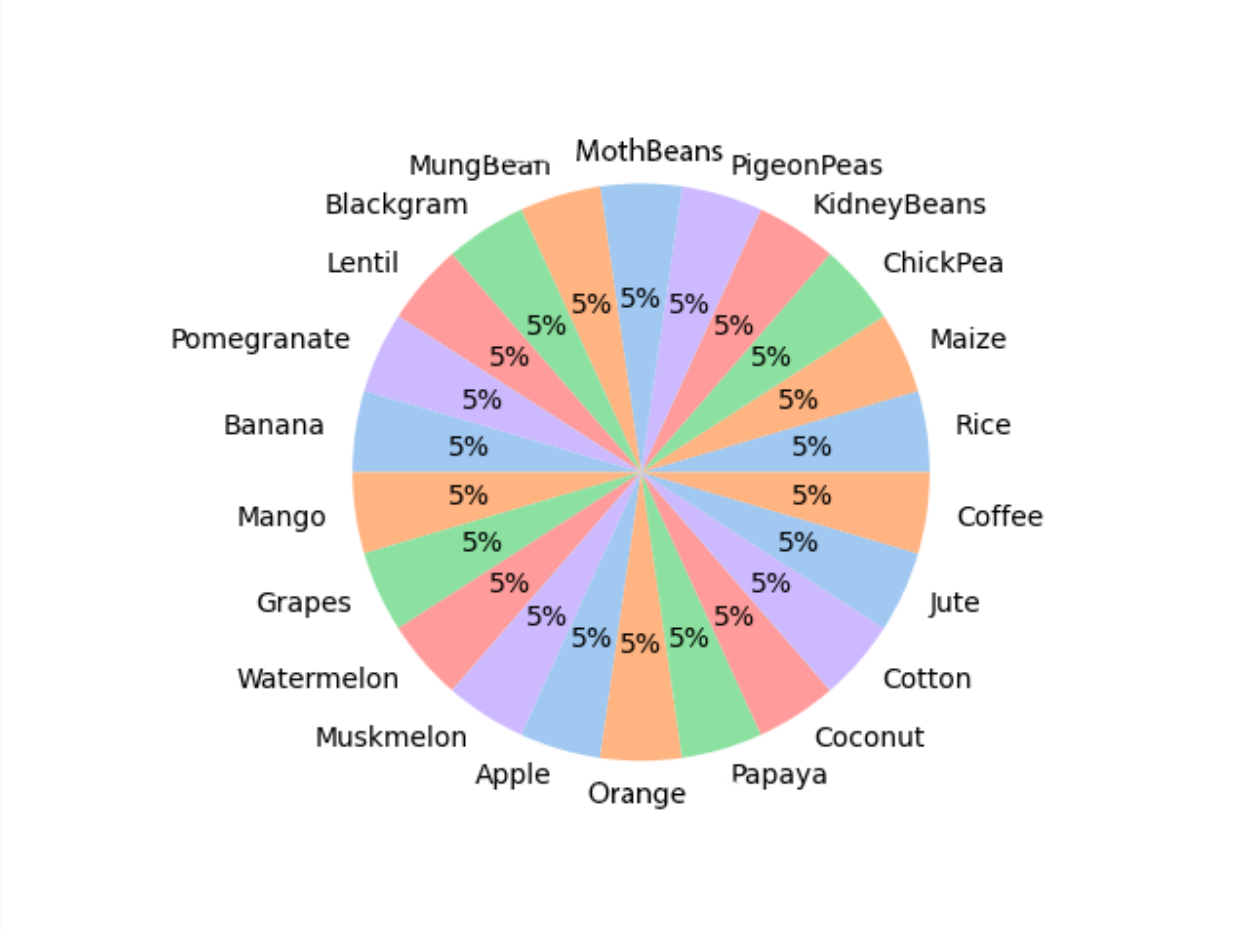
Já o código utilizado para criar o gráfico de setores foi o seguinte:

```
# Define a paleta de core através do Seaborn

colors = sns.color_palette('pastel')[0:5]

# Cria gráfico de setores/pizza

plt.pie(DS['Crop'].value_counts(), labels=DS['Crop'].unique(), colors=colors,
autopct='%.0f%%')

plt.show()
```

Ambos com acesso em:
https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/01_B%C3%A1sico/!%20Visualiza%C3%A7%C3%A3o%20inicial%20do%20Dataset%20(gr%C3%A1ficos).py

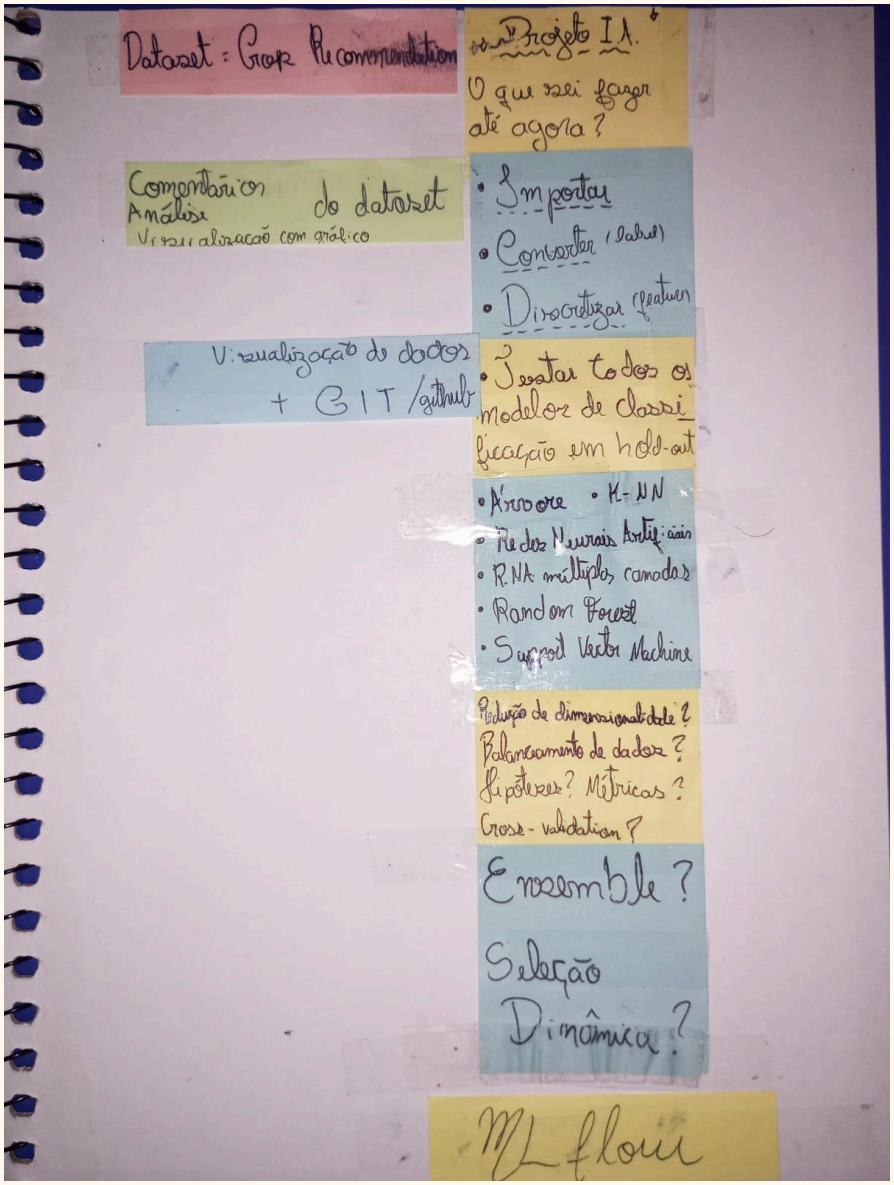
Classes visualizadas nos gráficos





Planejamento seguido:

Esse tópico é para mostar a metodologia, o plano montado e seguido aos poucos para o sucesso total do projeto.



DESENVOLVIMENTO

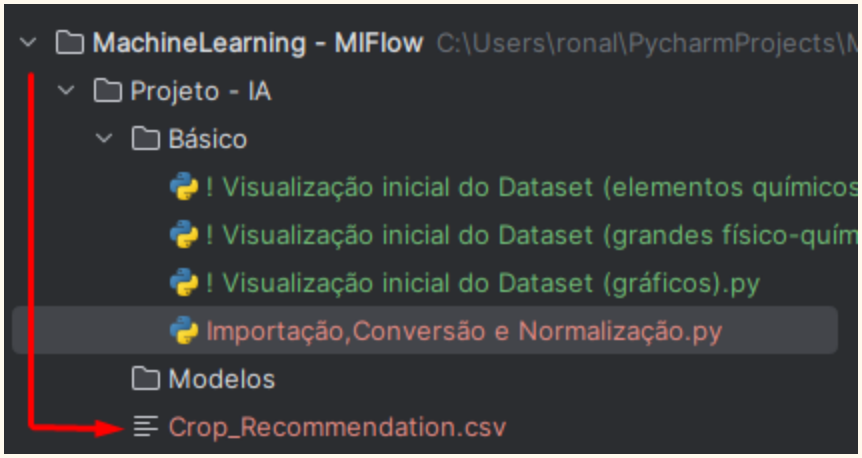
Esse tópico abordará como o os modelos de aprendizado de máquina foram implementados no dataset escolhido, do básico (tratando dados) ao avançado (seleção dinâmica de modelos).

Importação, Conversão e Normalização

Para a importação e implementação dos códigos é necessário importar todas as bibliotecas que são necessárias para os algoritmos de aprendizado de máquina funcionarem, são elas as biblioetecas “Sckit-learn”, a biblioteca “Pandas”, a bibilioteca “Numpy”, entre outras...

Além da importação das bibliotecas necessárias também faz-se necessário colocar o arquivo .csv do dataset no diretório do projeto e utilizá-lo no código atráves de uma variável contendo o caminho especificado de onde o arquivo, especificamente, está.

Exemplos em imagem do processo:




```
#Obtendo caminho onde o dataset está
ds_path = '../Crop_Recommendation.csv'

# Lendo o arquivo CSV
DS = pd.read_csv(ds_path)
```

Para o bom uso e desenvolvimento eficaz do dataset a coluna dos targets/alvos/classes (a coluna ‘Crops’) sofreu o processo de conversão “Label_Encoder”, esse processo é próprio para a conversão numérica de valores categóricos (textos) em valores numéricos perimitindo com que o raciocínio/entedimento, assim como sua escrita em código, do programador seja mais rápida e organizada.

O "Label Encoder" é um método de transformação que converte os valores de texto em números inteiros. Isso é feito atribuindo um número único a cada categoria na coluna de destino. Por exemplo, se tivermos as categorias "maçã", "banana" e "laranja", o Label Encoder atribuiria a elas os números 0, 1 e 2, respectivamente.

O código utilizado foi o seguinte:

```
from sklearn import preprocessing  # Importa o módulo de
pré-processamento do scikit-learn


label_encoder = preprocessing.LabelEncoder()  # Inicializa o Label
Encoder para TARGETS


DS['Crop'] = label_encoder.fit_transform(DS['Crop'])  # Aplica a
codificação de rótulos à coluna de 'Crop' do DataFrame df
```

Além disso todos os dados das features foram normalizados (escala de 0 a 1), essa é uma etapa de pré-processamento de dados antes de alimentar os dados em algoritmos de aprendizado de máquina. Ela ajuda a garantir que os dados estejam em um formato adequado para o modelo (sem perder a compatibilidade com os dados originais) e pode melhorar a convergência e o desempenho geral do modelo.

O código empregado foi o:

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler

DS['Nitrogen'] = MinMaxScaler().fit_transform(np.array(DS['Nitrogen']).reshape(-1,1))
DS['Phosphorus'] = MinMaxScaler().fit_transform(np.array(DS['Phosphorus']).reshape(-1,1))
DS['Potassium'] = MinMaxScaler().fit_transform(np.array(DS['Potassium']).reshape(-1,1))
DS['Temperature'] = MinMaxScaler().fit_transform(np.array(DS['Temperature']).reshape(-1,1))
DS['Humidity'] = MinMaxScaler().fit_transform(np.array(DS['Humidity']).reshape(-1,1))
DS['pH_Value'] = MinMaxScaler().fit_transform(np.array(DS['pH_Value']).reshape(-1,1))
DS['Rainfall'] = MinMaxScaler().fit_transform(np.array(DS['Rainfall']).reshape(-1,1))
```

Implantação de modelos de aprendizado de máquina

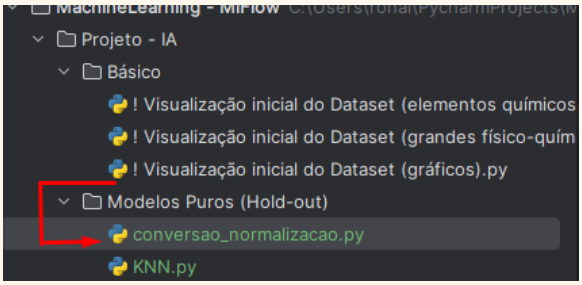
Os modelo que serão empregados aqui são próprios para datasets de classificação onde o objetivo do algoritmo é predizer, baseado nas features, a qual classe os itens (linhas) pertecem.

Por hora os modelos serão empregados em hold-out (única execução), sem qualquer verificação de hipótese, redução de dimensionalidade ou balanceamento de dados.

Para o uso dos modelos e por uma questão de organização de código foi utilizado a importação do código de **Conversão e Normalização,** o código foi alterado para o formato de uma função e movido para o mesmo diretório dos modelos

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/02_Modelos%20Puros%20(Hold-out)/conversao_normalizacao.py

```
2 usages  new *
def converte_normaliza (ds_path):
    DS = pd.read_csv(ds_path) # Lendo o arquivo CSV
    label_encoder = preprocessing.LabelEncoder()  # Inicializa o Label Encoder para TARGETS
    DS['Crop'] = label_encoder.fit_transform(DS['Crop'])  # Aplica a codificação de rótulos à coluna de 'Crop' do DataFrame df

    #-----------------------------------------------------------------------------------------------------------------
    # Configura pandas para exibir o dataset inteiro
    #pd.set_option('display.max_columns', None)

    # Imprime o DataFrame inteiro
    #print(DS.to_string())
    #-----------------------------------------------------------------------------------------------------------------

    DS['Nitrogen'] = MinMaxScaler().fit_transform(np.array(DS['Nitrogen']).reshape(-1,1))
    DS['Phosphorus'] = MinMaxScaler().fit_transform(np.array(DS['Phosphorus']).reshape(-1,1))
    DS['Potassium'] = MinMaxScaler().fit_transform(np.array(DS['Potassium']).reshape(-1,1))
    DS['Temperature'] = MinMaxScaler().fit_transform(np.array(DS['Temperature']).reshape(-1,1))
    DS['Humidity'] = MinMaxScaler().fit_transform(np.array(DS['Humidity']).reshape(-1,1))
    DS['pH_Value'] = MinMaxScaler().fit_transform(np.array(DS['pH_Value']).reshape(-1,1))
    DS['Rainfall'] = MinMaxScaler().fit_transform(np.array(DS['Rainfall']).reshape(-1,1))

    #-----------------------------------------------------------------------------------------------------------------
    # Imprime o DataFrame inteiro
    #print(DS.to_string())
    #-----------------------------------------------------------------------------------------------------------------

    return DS
```



1. **Árvore de Decisão (Tree):**
    - Método de aprendizado supervisionado usado para classificação e regressão.
    - Divide os dados em subconjuntos baseados em valores de atributos, formando uma estrutura de árvore de decisões que leva a uma conclusão ou previsão.

Implementação:

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/Modelos%20Puros%20(Hold-out)/1_Tree.py

Resultado:

```
Árvore de decisão:

Acurácia: 99%
O que o modelo tentou predizer ->   [10, 3, 14, 21, 0, 1, 10, 17, 10, 18, 19, 13, 5, 12, 15,
O que o modelo era para predizer -> [10, 3, 14, 21, 0, 1, 10, 17, 10, 18, 19, 13, 5, 12, 15,
```

Imagem vetorizada da árvore:

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/Modelos%20Puros%20(Hold-out)/1_Tree.svg

2. **K-Nearest Neighbors (K-NN):**
    - Algoritmo simples e intuitivo para classificação e regressão.
    - Classifica um dado ponto com base na maioria das classes de seus k vizinhos mais próximos no espaço de características.

Implementação:

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/Modelos%20Puros%20(Hold-out)/2_KNN.py

Resultado:

```
KNN:

Acurácia: 99%
O que o modelo tentou predizer ->   [20, 18, 20, 6, 14, 9, 18, 19, 0, 2, 18, 15, 9, 4, 21,
O que o modelo era para predizer -> [20, 18, 20, 6, 14, 9, 18, 19, 0, 2, 18, 15, 9, 4, 21,
```

3. **Perceptron de Múltiplas Camadas (MLP):**
    - Modelo de aprendizado inspirado na estrutura do cérebro humano.
    - Composto por camadas de nós (neurônios), onde cada nó está conectado a outros nós das camadas adjacentes. Utiliza funções de ativação para mapear entradas para saídas.

Implementação:

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/Modelos%20Puros%20(Hold-out)/3_MLP.py

Resultado:

```
Multi Layer Perceptron:

Acurácia: 98%
O que o modelo tentou predizer ->   [5, 16, 14, 4, 12, 15, 0, 7, 18, 7, 15, 0, 9, 4, 6, 18, 13, 8,
O que o modelo era para predizer -> [5, 16, 14, 4, 12, 15, 0, 7, 18, 7, 15, 0, 9, 4, 6, 18, 13, 8,
```

4. **Random Forest (RF):**
    - Algoritmo de aprendizado de conjunto para classificação e regressão.
    - Constrói múltiplas árvores de decisão durante o treinamento e usa a média das previsões de todas as árvores para melhorar a precisão e evitar overfitting.

Implementação:

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/Modelos%20Puros%20(Hold-out)/4_RF.py

Resultado:

```
Random Forest:

Acurácia: 100%
O que o modelo tentou predizer ->   [4, 19, 16, 3, 20, 13, 17, 19, 11, 20, 10, 1, 11, 7, 9, 4,
O que o modelo era para predizer -> [4, 19, 16, 3, 20, 13, 17, 19, 11, 20, 10, 1, 11, 7, 9, 4,
```

5. **Support Vector Machine (SVM):**
    - Algoritmo de aprendizado supervisionado para classificação e regressão.
    - Encontra o hiperplano ótimo que separa as classes de dados no espaço de características com a maior margem possível.
    - SVC - Support Vector Classifier, variação do SVM para classificação.

Implementação:

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/Modelos%20Puros%20(Hold-out)/5_SVM.py

Resultado:

```
Suppor Vector Classifier:

Acurácia: 98%
O que o modelo tentou predizer ->   [17, 3, 8, 6, 20, 6, 16, 2, 15, 9, 18, 1, 19,
O que o modelo era para predizer -> [17, 3, 8, 11, 20, 6, 16, 2, 15, 9, 18, 1, 19,
```

Tratamento dos dados aplicados aos modelos de aprendizado de máquina

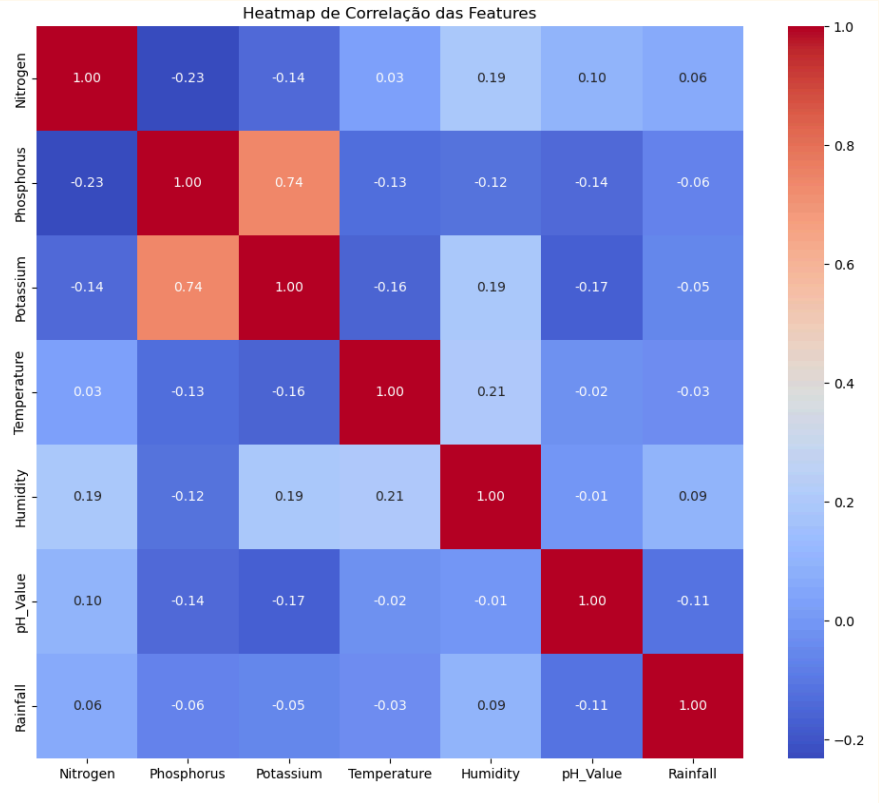
Seguindo o conteúdo programático visto em sala de aula o primeiro passo para realizar uma melhora, um tratamento do dataset que aumente a eficiência dos modelos de aprendizado de máquina seria a **Redução de Dimensionalidade**.

Análise de correlação entre features

Primeiramente geramos um mapa de calor que demonstrará a correlação entre as features para saber se temos que eliminar alguma delas, isso pode ser feito através do código:

```
# Calcula a matriz de correlação
matriz_de_correlacao = X.corr()

# Configura a aparência da matriz
plt.figure(figsize=(12, 10))                    #cores frias     #duas casas decimais
sns.heatmap(matriz_de_correlacao, annot=True, cmap='coolwarm', fmt='.2f')
                                  #valores numéricos de correlação serão exibidos diretamente nas células do gráfico

# Mostra o heatmap
plt.title('Heatmap de Correlação das Features')
plt.show()
```

Resultado:



Conclusão:

Apesar de uma correlação considerável entre a feature Phosphorus e Potassium devemos lembrar que esse dataset necessita de todas as features por serem características essenciais para o desenvolvimento de cada planta, mas para fins de aprendizado vamos executar a Feature Selection através do SelectKBest (método filter), observando sempre a diferença entre os resultados dos modelos anteriores com a situação nova.

Redução de dimensionalidade baseado no método de filter

O método filter é quando um conjunto das features é submetido a um teste de estatística para avaliar cada coluna e ver seu peso na hora de uma classificação, por exemplo.

Em termos de código podemos utilizar o "SelectKBest" com a função "class_if" própria para datasets com features numéricas e target categórico, que é o nosso caso, veja o código:

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/03_Feature%20Selection/!%20Feature%20selection%20(FILTER).py

```
# O SelectKBest pode ser usado com várias funções, aqui usamos a f_classif (dados numéricos e variável alvo categórica)
f_classifteste = SelectKBest(score_func=f_classif,k=7)

modelofeat = f_classifteste.fit(X,y)

# Obtendo os nomes das features
feature_names = modelofeat.get_feature_names_out()
# Obtendo os scores das features
feature_scores = modelofeat.scores_

# Iterando sobre as features e imprimindo os nomes e scores em linhas alternadas para melhor visualização
for name, score in zip(feature_names, feature_scores):
    print(f"Feature: {name}")
    print(f"Score: {score}\n")
```

Resultado:

```
Feature: Nitrogen
Score: 897.5681863257206

Feature: Phosphorus
Score: 1885.6578591841849

Feature: Potassium
Score: 27238.362067066617

Feature: Temperature
Score: 102.18698122995443
```

```
Feature: Humidity
Score: 3103.708891217346

Feature: pH_Value
Score: 60.34403352941339

Feature: Rainfall
Score: 605.5279661441442
```

Agora vamos selecionar as 3 features (contexto tridimensional) com maior pontuação que são, respectivamente:

- **Potassium (27238.36)**
- **Humidity(3103.70)**
- **Phosphorus (1885.65)**

E aplicá-las aos modelos visualizando cada resultado:

- **Árvore de decisão:** https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/03_Feature%20Selection/1_Tree%20(Filter).py

```
Árvore de decisão:

Acurácia: 84%
O que o modelo tentou predizer ->   [7, 8, 12, 3, 5, 5, 6, 8, 16, 4
O que o modelo era para predizer -> [7, 8, 12, 3, 5, 5, 14, 8, 16,
```

- **K-NN:** https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/03_Feature%20Selection/2_KNN%20(Filter).py

```
KNN:

Acurácia: 83%
O que o modelo tentou predizer ->   [10, 17, 17, 2, 14, 0, 6, 7, 20,
O que o modelo era para predizer -> [2, 17, 17, 10, 6, 0, 6, 7, 20,
```

- **MLP:** https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/03_Feature%20Selection/3_MLP%20(Filter).py

```
Multi Layer Perceptron:

Acurácia: 84%
O que o modelo tentou predizer ->   [20, 15, 16, 8, 19, 16, 14, 19, 19, 11
O que o modelo era para predizer -> [20, 15, 16, 8, 19, 16, 14, 19, 19, 11
```

- **Random Forest:** https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/03_Feature%20Selection/4_RF%20(Filter).py

```
Random Forest:

Acurácia: 83%
O que o modelo tentou predizer ->   [0, 0, 6, 4, 6, 20, 7, 13, 2,
O que o modelo era para predizer -> [0, 0, 14, 4, 6, 20, 7, 11, 10
```

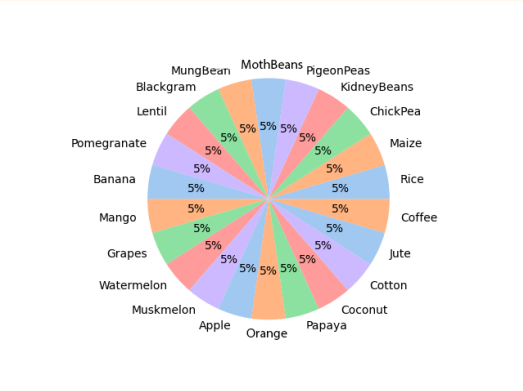
- **Support Vector Machine:** https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/03_Feature%20Selection/5_SVM%20(Filter).py

```
Suppor Vector Classifier:

Acurácia: 81%
O que o modelo tentou predizer ->   [9, 5, 18, 5, 5, 17, 21, 10, 0,
O que o modelo era para predizer -> [9, 5, 18, 5, 5, 17, 21, 10, 0,
```

Como podemos ver, todos os nossos modelos tiveram o seu desempenho piorado comprovado através da métrica "acurácia", e isso era **exatamente** o que se esperava, pois, no contexto do dataset, onde temos 7 features que representam os pilares fundamentais que as plantas possuem para seu desenvolvimento, não se faz necessário e não chega nem a ser certo reduzir o número de features, já que é através de **todas** elas que temos a recomendação exata do plantio em cada amostra de solo.

Balanceamento de dados

Como já demonstrado anteriormente os dados estão balanceados, não existem classes majoritárias ou minoritárias, assim, não necessitando qualquer método de reamostragem.



Validação cruzada

Como todos os modelos sempre estavam em HOLD-OUT (execução única) vamos executá-los 100 vezes e extrair a média de acerto para termos uma resposta mais efetiva, segura e completa de como os modelos de aprendizado de máquina se comportam quando aplicados ao dataset.

Podemos fazer essas 100 execuçãos de cada modelo através da validação cruzada, que em código pode ser aplicada da seguinte forma (o código foi feito em função para poder ser importado e deixar os outros códigos de modelo mais facilmente visíveis):

https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/04_Cross_Validation/k_folds.py

```
import numpy as np
from sklearn.model_selection import StratifiedKFold
from conversao_normalizacao import converte_normaliza  # Importação do código de conversão e normalização
10 usages  new *
def cross_validation(ds_path, folds=100):
    # Carregando e pré-processando os dados
    DS = converte_normaliza(ds_path)  # Conversão e normalização

    # Extrai a última coluna
    y = DS['Crop']  # Extrai a última coluna, que é o label

    # Extrai as características (todas as colunas exceto a última)
    X = DS.iloc[:, :-1]

    # Transforma para Array NumPy
    X = np.array(X)
    y = np.array(y)

    kf = StratifiedKFold(n_splits=folds)

    X_train = []
    y_train = []

    X_test = []
    y_test = []

    for train_index, test_index in kf.split(X, y):
        X_train.append(X[train_index])
        X_test.append(X[test_index])

        y_train.append(y[train_index])
        y_test.append(y[test_index])

    return X_train, X_test, y_train, y_test
```

Feito isso basta chamar a nossa função em cada arquivo de modelo através da linha de comando abaixo:

```
X_train, X_test, y_train, y_test = cross_validation(ds_path,folds)
```

E realizar a estrutra de repetição padrão para trabalhar com validação cruzada (exemplo com o modelo de Árvore de decisão):

```
results = []

for i in range(folds):  # Certifique-se de que "folds" está definido antes deste loop
    model = DecisionTreeClassifier(criterion="entropy")
    model = model.fit(X_train[i], y_train[i])

    result = model.predict(X_test[i])
    acc = metrics.accuracy_score(result, y_test[i]) * 100  # Converte precisão para porcentagem
    acc = round(acc, 2)  # Arredonda a precisão para duas casas decimais
    results.append(acc)

print("Acurácia de cada Árvore de Decisão ->",results)
show = round(np.mean(results), 2)  # Arredonda a média para duas casas decimais
print("Média das acurácias das Árvores de Decisão (100 folds): {}%".format(show))
```

Resultados da validação cruzada aplicada a cada modelo:

- **Árvore de decisão:**

  https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/04_Cross_Validation/1_Tree%20(K-Folds).py

```
Acurácia de cada Árvore de Decisão -> [100.0, 100.0, 100.0, 100.0, 100
Média das acurácias das Árvores de Decisão (100 folds): 98.95%
```

- **KNN:**

  https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/04_Cross_Validation/2_KNN%20(K-Folds).py

```
Acurácia de cada K-Nearest-Neighbour -> [100.0, 100.0, 95.45, 100.0, 1
Média das acurácias dos modelos KNN (100 folds): 98.5%
```

- **MLP (Apenas 10 execuções dado o poder de processamento do modelo e sua demora) :**

  https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/04_Cross_Validation/3_MLP%20(K-Folds).py

```
Acurácia de cada Multi-Layer-Perceptron -> [97.73, 98.18, 99.55, 98.18, 98.18, 97.73, 98.64, 98.18, 98.18, 97.27]
Média das acurácias dos modelos MLP (10 folds): 98.18%
```

- **Random Forest:**

  https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/04_Cross_Validation/4_RF%20(K-Folds).py

```
Acurácia de cada RandomForest -> [100.0, 100.0, 100.0, 100.0, 100.0, 1
Média das acurácias de cada RandomForest (100 folds): 99.45%
```

- **Support Vector Machine:**

  https://github.com/Ronaldo-Urquiza/ProjetoIA-DS.Crop_Recommendation/blob/main/Projeto%20-%20IA/04_Cross_Validation/5_SVM%20(K-Folds).py

```
Acurácia de cada Support Vector Classifier -> [100.0, 100.0, 95.45, 10
Média dos modelos SVC (100 folds): 98.5%
```

Métricas

Já que o dataset conta com 22 classes o objetivo será adaptar os códigos de métricas, que são para classificações binárias (0 ou 1), para o problema atual, um problema de multi-classes já com a validação cruzada implantada.

Através de pesquisa foi encontrado o seguinte resultado:

1º Passo) Criar estruturas de lista que armazenam os valores processados finais.

```
# Inicializa listas para armazenar os resultados de cada fold
accuracies = []
precisions = []
recalls = []
f1_scores = []
roc_aucs = []
```

2º Passo) Dentro da estrura da validação cruzada colocamos cada acurácia na lista de acurácia para depois tirarmos a média:

```
for i in range(folds):  # Certifique-se de que "folds" está definido antes deste loop
    model = DecisionTreeClassifier(criterion="entropy")
    model = model.fit(X_train[i], y_train[i])

    result = model.predict(X_test[i])

    # Acurácia
    acc = accuracy_score(y_test[i], result)
    accuracies.append(acc)
```

3ª Passo) Para gerenciarmos as outras métricas, ainda na estrutura de repetição de validação cruzada, sendo elas, precision e recall (para classes positivas) e f1-score que é (2 * o precision * recall) dividido pela soma do precision e recall, temos que, primeiramente, usar a função classification_report.

```
# Relatório de Classificação
report = classification_report(y_test[i], result, output_dict=True, zero_division=0)
#print(report)
precision_values = [report[str(label)]['precision'] for label in np.unique(y) if str(label) in report]
precisions.append(np.mean(np.array(precision_values)))
recall_values = [report[str(label)]['recall'] for label in np.unique(y) if str(label) in report]
recalls.append(np.mean(np.array(recall_values)))
f1_score_values = [report[str(label)]['f1-score'] for label in np.unique(y) if str(label) in report]
f1_scores.append(np.mean(np.array(f1_score_values)))
```

3.1) função classification_report()

- Esta função gera um relatório com várias métricas (precisão, recall e F1-score) para cada classe.
- Parâmetros:
  - y_test[i]: Lista de rótulos verdadeiros (valores reais) do fold i.
  - result: Lista de rótulos previstos pelo modelo para o fold i.
  - output_dict=True: Especifica que o resultado deve ser retornado como um dicionário, facilitando o acesso aos valores específicos.
  - zero_division=0: Este parâmetro controla o que fazer quando houver uma divisão por zero nas métricas. Isso pode acontecer, por exemplo, quando uma classe nunca é prevista pelo modelo (resultando em zero predições verdadeiras). Definir zero_division=0 faz com que tais situações sejam tratadas definindo a métrica como zero ao invés de lançar um erro ou usar NaN.

❖ Resultado:

O retorno da função vai ser atribuída à variável “report” como descrito no trecho:

```
# Relatório de Classificação
report = classification_report(y_test[i], result, output_dict=True, zero_division=0)
```

Exemplo de saída:

```
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
 '8': {'precision': 0.5, 'recall': 1.0, 'f1-score': 0.6666666666666666, 'support': 1.0},
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
 '8': {'precision': 0.5, 'recall': 1.0, 'f1-score': 0.6666666666666666, 'support': 1.0},
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
support': 1.0}, '8': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 1.0},
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
 '8': {'precision': 1.0, 'recall': 1.0, 'f1-score': 1.0, 'support': 1.0}, '9': {'precisi
```

4º passo) Linhas de comando para extração de valores

- São elas:

```
precision_values = [report[str(label)]['precision'] for label in np.unique(y) if str(label) in report]

recall_values = [report[str(label)]['recall'] for label in np.unique(y) if str(label) in report]

f1_score_values = [report[str(label)]['f1-score'] for label in np.unique(y) if str(label) in report]
```

- Usando a primeira de exemplos:

precision_values = [report[str(label)]['precision'] for label in np.unique(y) if str(label) in report]

- O objetivo dessa linha é extrair os valores de precisão para cada classe presente no conjunto de dados y e armazená-los em uma lista chamada "precision_values".
- Esta linha utiliza uma list comprehension, uma maneira concisa de criar listas no Python. A sintaxe básica de uma list comprehension é

[expressão + for + item + in + iterável + if condição]

- Focando na parte do loop:

```
for label in np.unique(y) if str(label) in report]
```

- **np.unique(y)** retorna uma array com os valores únicos presentes em y, que são as diferentes classes do problema de classificação.
- **for label in np.unique(y)** itera sobre cada classe única.
- **str(label)** converte o valor de label para uma string. Isso é necessário porque as chaves no dicionário report são strings.

- **if str(label) in report** verifica se a string correspondente à classe (str(label)) está presente como uma chave no dicionário report. Isso é importante para garantir que não tentamos acessar uma chave inexistente, o que causaria um erro.

  - Focando na parte do acesso

```
precision_values = [report[str(label)]['precision']
```

- **report[str(label)]['precision']** acessa o valor de precisão para a classe label no dicionário report.
- **report** é um dicionário aninhado, onde as chaves do primeiro nível são as classes (convertidas para strings) e as chaves do segundo nível são as métricas ('precision', 'recall', 'f1-score', etc.).

  ❖ Resultado:

```
precision_values = [report[str(label)]['precision'] for label in np.unique(y) if str(label) in report]
```

- Para cada label em **np.unique(y)**, se **str(label)** estiver presente em **report**, o valor de **report[str(label)]['precision']** é adicionado à lista **precision_values**.

  ❖ Exemplo Prático:

Para ilustrar, vamos considerar um exemplo:

Suponha que y contenha classes [1, 2, 3] e report seja o seguinte dicionário:

```
report = {
    '1': {'precision': 0.9, 'recall': 0.8, 'f1-score': 0.85, 'support': 50},
    '2': {'precision': 0.7, 'recall': 0.75, 'f1-score': 0.72, 'support': 40},
    '3': {'precision': 0.85, 'recall': 0.88, 'f1-score': 0.86, 'support': 60},
```

1. Iteração:
    - np.unique(y) resulta em [1, 2, 3].
    - A iteração ocorre sobre esses valores: 1, 2, 3.
2. Conversão e Verificação:
    - Para label = 1: str(label) resulta em '1', que está presente em report.
    - Para label = 2: str(label) resulta em '2', que está presente em report.
    - Para label = 3: str(label) resulta em '3', que está presente em report.
3. Extração dos Valores de Precisão:
    - report['1']['precision'] é 0.9.
    - report['2']['precision'] é 0.7.
    - report['3']['precision'] é 0.85.
4. Construção da Lista:
    - precision_values será [0.9, 0.7, 0.85].

Essa lista precision_values agora contém as precisões de cada classe, que podem ser usadas para calcular a média ou realizar outras análises.

O funcionamento é o mesmo para as outras linhas que extram as outras 2 métricas:

```
recall_values = [report[str(label)]['recall'] for label in
np.unique(y) if str(label) in report]
```

```
f1_score_values = [report[str(label)]['f1-score'] for label in
np.unique(y) if str(label) in report]
```

5º Passo) Armazenamento da extração

- As linhas abaixo são responsáveis por transformar as listas dos comandos imediatamente anteriores explicados em arrays numpy, gerar médias e armazenar os valores em listas novas, pois cada iteração de um dos comandos anteriores gera o report das classes totalizando 100 valores para **cada** classe, assim, se faz necessário uma média de cada linha que, como explicado anteriormente, engloba um tipo de métrica para todas as 22 classes.

```
precisions.append(np.mean(np.array(precision_values)))
recalls.append(np.mean(np.array(recall_values)))
f1_scores.append(np.mean(np.array(f1_score_values)))
```

- Vamos usar a linha de precisions como exemplo

```
precisions.append(np.mean(np.array(precision_values)))
```

1. Transforma a lista em um array do NumPy:
    - **np.array(precision_values)**: Converte a lista precision_values em um array do NumPy. Isso permite que você utilize funções do NumPy para calcular estatísticas facilmente.
2. Calcula a média dos valores de precisão:
    - **np.mean(...)**: Calcula a média dos valores contidos no array resultante da conversão acima. A função np.mean é usada para calcular a média aritmética ao longo do array.
3. Adiciona a média calculada à lista precisions:
    - **precisions.append(...)**: Adiciona a média calculada dos valores de precisão à lista precisions. Cada iteração do loop adiciona uma nova média de precisão à lista.

6º passo) ROC AUC (Receiver Operating Characteristic Area Under the Curve)

```
# ROC AUC
y_test_bin = label_binarize(y_test[i], classes=np.unique(y))
result_bin = label_binarize(result, classes=np.unique(y))
if y_test_bin.shape == result_bin.shape:  # Verifica se a binarização está correta
    roc_auc = roc_auc_score(y_test_bin, result_bin, average="macro")
    roc_aucs.append(roc_auc)
```

- O código acima calcula a métrica ROC AUC (Receiver Operating Characteristic Area Under the Curve) para um problema de classificação multiclasse, usando validação cruzada. Vamos detalhar cada linha:

- **Linha 1: y_test_bin = label_binarize(y_test[i], classes=np.unique(y))**
    - ➢ Objetivo: Transformar os labels de teste em formato binarizado (one-hot encoding).
        - Exemplo: Se as classes forem ['apple', 'banana', 'cherry'], a one-hot encoding seria:
            - 'apple' -> [1, 0, 0]
            - 'banana' -> [0, 1, 0]
            - 'cherry' -> [0, 0, 1]
    - ➢ **label_binarize**: Função do sklearn que converte rótulos multiclasses em formato binário.
    - ➢ **y_test[i]**: O vetor de rótulos verdadeiros para o i-ésimo fold de teste.
    - ➢ **classes=np.unique(y)**: Garante que todas as classes possíveis estejam presentes no binarizador. np.unique(y) retorna uma lista de todas as classes únicas presentes no conjunto de dados.
    - ➢ Resultado: **y_test_bin** será uma matriz onde cada linha representa um exemplo e cada coluna representa uma classe. Um valor de 1 indica a presença da classe, e 0 a ausência.

- **Linha 2: result_bin = label_binarize(result, classes=np.unique(y))**
    - ➢ Mesmo funcionamento da linha de código passada mas dessa vez com a lista dos labels que o modelo tentou prever.

- **Linha 3: if y_test_bin.shape == result_bin.shape:**
  - ➢ Objetivo: Garantir que a binarização foi feita corretamente e que as formas das matrizes de rótulos binarizados e previsões binarizadas são iguais.
  - ➢ **y_test_bin.shape**: A forma (dimensão) da matriz de rótulos binarizados.
  - ➢ **result_bin.shape**: A forma (dimensão) da matriz de previsões binarizadas.
  - ➢ Verificação: As duas formas devem ser iguais para calcular corretamente a ROC AUC. Se as formas não coincidirem, significa que houve algum problema na binarização.

- **Linha 4: roc_auc = roc_auc_score(y_test_bin, result_bin, average="macro")**
  - ➢ Objetivo: Calcular a métrica ROC AUC.
  - ➢ **roc_auc_score**: Função do sklearn que calcula a área sob a curva ROC.
  - ➢ **y_test_bin**: A matriz binarizada dos rótulos verdadeiros.
  - ➢ **result_bin**: A matriz binarizada das previsões do modelo.
  - ➢ **average="macro"**: Especifica que a média deve ser calculada levando em consideração todas as classes igualmente, independentemente de suas frequências. Isso é útil em problemas de classificação multiclasse.
  - ➢ Resultado: roc_auc será um valor numérico representando a média das áreas sob as curvas ROC para cada classe.

- **Linha 5: roc_aucs.append(roc_auc)**
  - ➢ Objetivo: Armazenar o valor calculado de ROC AUC.
  - ➢ **roc_aucs.append(roc_auc)**: Adiciona o valor de roc_auc à lista roc_aucs. Essa lista acumula os valores de ROC AUC para cada fold da validação cruzada.
  - ➢ Resultado: Ao final do loop, a lista **roc_aucs** conterá os valores de ROC AUC para todos os folds, permitindo calcular a média ou outras estatísticas descritivas.

- **Resumo**

  Essas linhas de código transformam os rótulos e previsões em formato binarizado para calcular a métrica ROC AUC em um problema de classificação multiclasse. A métrica ROC AUC avalia a capacidade do modelo em distinguir entre as classes, sendo uma medida de desempenho globalmente útil, especialmente em problemas multiclasse.

7º passo) Formatação

- Essa etapa é para melhorar a visualização das métricas, o formato escolhido foi a porcentagem, para simbolizar o acerto de cada métrica, sendo assim é realizado uma multiplicação por 100 nas métricas necessárias e é feito uma impressão dos valores, com apenas 2 casas decimais e o símbolo de porcentagem, no terminal.

```
# Média das métricas
mean_accuracy = np.mean(accuracies) * 100
mean_precision = np.mean(precisions) * 100
mean_recall = np.mean(recalls) * 100
mean_f1_score = np.mean(f1_scores) * 100
mean_roc_auc = np.mean(roc_aucs) * 100

print("Média das acurácias das Árvores de Decisão (100 folds): {:.2f}%".format(mean_accuracy))
print("Média das precisões: {:.2f}%".format(mean_precision))
print("Média dos recalls: {:.2f}%".format(mean_recall))
print("Média dos F1-scores: {:.2f}%".format(mean_f1_score))
print("Média das ROC AUCs: {:.2f}%".format(mean_roc_auc))
```

Resultado:

```
Média das acurácias das Árvores de Decisão (100 folds): 98.95%
Média das precisões: 98.48%
Média dos recalls: 98.95%
Média dos F1-scores: 98.64%
Média das ROC AUCs: 99.45%
```

Ensemble

Ensemble Learning, ou **Aprendizado em Conjunto,** é uma técnica que combina múltiplos modelos de aprendizado de máquina para obter melhores resultados de predição. Os principais motivos para usar ensemble são:

1. **Aumento da Precisão**: Modelos combinados frequentemente têm um desempenho melhor do que modelos individuais. A média das previsões de vários modelos pode corrigir os erros individuais.
2. **Redução de Overfitting**: Modelos individuais podem se ajustar excessivamente aos dados de treino (overfitting). Ensembles, especialmente aqueles que combinam modelos com diferentes tipos de viés, tendem a ser mais robustos a esse problema.
3. **Diversidade de Modelos**: Diferentes algoritmos de aprendizado têm diferentes pontos fortes e fracos. Usar um conjunto de modelos permite aproveitar as vantagens de cada um.

No código o ensemble learning foi usado da seguinte forma:

```
knn =
BaggingClassifier(estimator=KNeighborsClassifier(n_neighbors=5),
n_estimators=50)

svc = AdaBoostClassifier(estimator=SVC(kernel='rbf',
probability=True), n_estimators=50, algorithm='SAMME')

rf =
BaggingClassifier(estimator=RandomForestClassifier(n_estimators=100),
n_estimators=50)
```

1. K-Nearest Neighbors (KNN) com Bagging

```
knn =
BaggingClassifier(estimator=KNeighborsClassifier(n_neighbors=5),
n_estimators=50)
```

Explicação:

- **BaggingClassifier**: Bagging, ou Bootstrap Aggregating, é uma técnica de ensemble que treina múltiplos modelos base em diferentes subconjuntos de dados amostrados com

reposição (bootstrap). As previsões finais são feitas pela média ou votação das previsões dos modelos base.

- **KNeighborsClassifier**: O modelo base é o K-Nearest Neighbors (KNN) com n_neighbors=5, o que significa que a classificação de um ponto é determinada pelos 5 vizinhos mais próximos.
- **n_estimators=50**: Cria 50 instâncias do KNN, cada uma treinada em um subconjunto diferente dos dados. As previsões finais são a média das previsões dessas 50 instâncias.

Vantagens:

- **Redução da Variância**: Bagging ajuda a reduzir a variância dos modelos, o que é particularmente útil para modelos como KNN que podem ser sensíveis a ruídos nos dados de treino.
- **Melhoria na Generalização**: Com múltiplas instâncias do modelo base, a combinação das previsões tende a generalizar melhor para novos dados.

2. Support Vector Machine (SVM) com AdaBoost

```
svc = AdaBoostClassifier(estimator=SVC(kernel='rbf',
probability=True), n_estimators=50, algorithm='SAMME')
```

Explicação:

- **AdaBoostClassifier**: Adaptive Boosting (AdaBoost) é uma técnica de ensemble que treina modelos sequencialmente, cada um corrigindo os erros dos anteriores. A cada iteração, os pesos das amostras mal classificadas são aumentados.
- **SVC**: O modelo base é o Support Vector Classifier (SVC) com kernel radial base function (RBF), que é uma função de kernel comum para SVMs que mapeia os dados em uma dimensão maior onde uma separação linear é possível.
- **probability=True**: Calcula probabilidades para cada previsão, o que é necessário para AdaBoost.
- **n_estimators=50**: Cria até 50 instâncias do SVC. Cada nova instância é ajustada para corrigir os erros cometidos pelas instâncias anteriores.
- **algorithm='SAMME'**: O algoritmo SAMME (Stagewise Additive Modeling using a Multiclass Exponential loss function) é uma extensão de AdaBoost para classificação multiclasse.

Vantagens:

- **Aumento da Precisão**: AdaBoost foca em corrigir erros de previsões anteriores, o que pode levar a um aumento significativo na precisão.
- **Combinação Eficiente**: A combinação de SVM com AdaBoost permite capturar tanto relações complexas nos dados quanto melhorar a robustez do modelo.

3. Random Forest (RF) com Bagging

```
rf =
BaggingClassifier(estimator=RandomForestClassifier(n_estimators=100),
n_estimators=50)
```

Explicação:

- **BaggingClassifier**: Utilizamos Bagging para treinar múltiplas instâncias do Random Forest, aumentando ainda mais a robustez do ensemble.
- **RandomForestClassifier**: O modelo base é o Random Forest, que é um ensemble de árvores de decisão treinadas em diferentes subconjuntos de dados com amostragem bootstrap. Cada árvore é treinada com um subconjunto aleatório de características.
- **n_estimators=100**: Cria 100 árvores de decisão em cada instância do Random Forest.
- **n_estimators=50** (no BaggingClassifier): Cria 50 instâncias do Random Forest, cada uma com 100 árvores de decisão. As previsões finais são a média das previsões dessas 50 instâncias de Random Forest.

Vantagens:

- **Redução da Variância**: Bagging e o uso de múltiplas árvores em Random Forest ajudam a reduzir a variância, tornando o modelo mais robusto contra overfitting.
- **Melhoria na Generalização**: Com 50 instâncias de Random Forest, cada uma contendo 100 árvores, o ensemble final tem uma capacidade muito alta de generalização para novos dados.

Conclusão

A combinação dessas técnicas de ensemble permite criar modelos robustos e precisos. Cada técnica traz suas próprias vantagens, como redução de variância, aumento de precisão e melhoria na generalização, resultando em um sistema de predição mais confiável e eficiente.

Dynamic Selection

A **seleção dinâmica** é uma técnica avançada utilizada em sistemas de aprendizado de máquina para melhorar a performance dos modelos de ensemble. Ao invés de utilizar um modelo fixo para todas as instâncias de teste, a seleção dinâmica escolhe, para cada instância, os classificadores mais adequados com base em determinadas heurísticas. A biblioteca **DESlib** (Dynamic Ensemble Selection Library) implementa diversas técnicas de seleção dinâmica, proporcionando flexibilidade e precisão para os sistemas de ensemble.

Motivação para a Seleção Dinâmica

1. **Diversidade nos Dados**: Diferentes regiões do espaço de características podem ser melhor modeladas por diferentes classificadores. A seleção dinâmica reconhece essa diversidade e ajusta o modelo conforme a necessidade.
2. **Melhoria na Performance**: Ao selecionar o melhor conjunto de classificadores para cada instância, a seleção dinâmica pode melhorar a acurácia e reduzir o erro de generalização.
3. **Adaptação às Variações**: Dados reais muitas vezes contêm variações significativas, e a capacidade de adaptação da seleção dinâmica permite lidar melhor com essas variações.

Técnicas de Seleção Dinâmica na DESlib

1. **KNORA-U (K-Nearest Oracles Union)**:
    - **Descrição**: Seleciona todos os classificadores que conseguem corretamente classificar pelo menos uma das k instâncias mais próximas no conjunto de validação.
    - **Vantagem**: Utiliza a força coletiva de múltiplos classificadores para melhorar a robustez.

```
# KNORA-U

knu = KNORAU(pool_classifiers=[knn, svc, rf], k=5)

knu.fit(X_treino, y_treino)
```

2. **KNORA-E (K-Nearest Oracles Eliminate)**:

- **Descrição**: Seleciona apenas os classificadores que conseguem corretamente classificar todas as k instâncias mais próximas no conjunto de validação.
- **Vantagem**: Foca na precisão, utilizando apenas os classificadores mais confiáveis.

```
kne = KNORAE(pool_classifiers=[knn, svc, rf], k=5)

kne.fit(X_treino, y_treino)
```

3. **LCA (Local Class Accuracy)**:
   - **Descrição**: Calcula a precisão local de cada classificador baseado na vizinhança da instância de teste. Os classificadores com maior precisão local são selecionados.
   - **Vantagem**: Combina precisão local com flexibilidade na seleção dos classificadores.

```
lca = LCA(pool_classifiers=[knn, svc, rf], k=5)

lca.fit(X_treino, y_treino)
```

4. **OLA (Overall Local Accuracy)**:
   - **Descrição**: Similar ao LCA, mas considera a acurácia geral na vizinhança ao invés da precisão local por classe.
   - **Vantagem**: Simplicidade e eficácia em selecionar classificadores bem-sucedidos na vizinhança.

```
ola = OLA(pool_classifiers=[knn, svc, rf], k=5)

ola.fit(X_treino, y_treino)
```

5. **MCB (Multiple Classifier Behavior)**:
   - **Descrição**: Analisa o comportamento dos classificadores em várias instâncias para determinar os mais adequados para cada instância de teste.

- **Vantagem**: Flexibilidade e capacidade de capturar comportamentos complexos dos classificadores.

```
mcb = MCB(pool_classifiers=[knn, svc, rf], k=5)

mcb.fit(X_treino, y_treino)
```

Benefícios da Seleção Dinâmica com DESlib

- **Aumento da Performance**: A seleção dinâmica pode levar a melhorias significativas na performance dos modelos ao escolher os classificadores mais apropriados para cada instância.
- **Flexibilidade**: Com diversas técnicas disponíveis, é possível adaptar o sistema às características específicas do problema.
- **Robustez**: A combinação de vários classificadores com seleção dinâmica aumenta a robustez e a resistência a ruídos e variações nos dados.

Conclusão

O uso de técnicas de seleção dinâmica como as fornecidas pela DESlib representa um avanço significativo em relação aos métodos tradicionais de ensemble. Ao selecionar dinamicamente os classificadores mais adequados para cada instância de teste, é possível alcançar uma maior acurácia, melhorar a generalização e lidar eficientemente com a diversidade dos dados.

Implementação do MLflow

O MLflow é uma plataforma de código aberto para gerenciar todo o ciclo de vida do desenvolvimento de machine learning. Ele oferece suporte à experimentação, rastreamento de parâmetros e métricas, reprodução de experimentos, implantação de modelos e monitoramento. Aqui estão alguns dos principais usos e benefícios do MLflow:

1. **Experimentação Reprodutível**: Com o MLflow, você pode registrar todos os detalhes de seus experimentos de machine learning, incluindo os dados, parâmetros do modelo, métricas de desempenho e artefatos do modelo. Isso permite que você reproduza facilmente os experimentos e compreenda o impacto das diferentes escolhas de parâmetros.

2. **Gerenciamento de Versões de Modelos**: O MLflow rastreia automaticamente as versões dos modelos treinados, permitindo que você compare o desempenho de diferentes versões e reverta para versões anteriores, se necessário.
3. **Colaboração e Compartilhamento**: Com o MLflow, equipes de cientistas de dados podem colaborar de forma eficiente em projetos de machine learning, compartilhando código, modelos e resultados de experimentos em um único local.
4. **Implantação de Modelos**: O MLflow oferece integrações com várias plataformas de implantação de modelos, facilitando a transição de modelos do estágio de desenvolvimento para produção. Você pode implantar modelos diretamente do MLflow para serviços em nuvem, contêineres Docker, clusters Kubernetes e outros ambientes.
5. **Monitoramento de Desempenho**: O MLflow permite que você monitore o desempenho dos modelos em produção, rastreando métricas em tempo real, detecção de derivas de dados e monitoramento de alertas.

Em resumo, o MLflow simplifica o gerenciamento do ciclo de vida de desenvolvimento de machine learning, desde a experimentação inicial até a implantação e monitoramento contínuo dos modelos em produção. Ele promove a reprodutibilidade, colaboração e eficiência no desenvolvimento de soluções de machine learning.

Explicação do código final

Pré-processamento de Dados:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
from deslib.dcs import OLA, LCA, MCB
from deslib.des import KNORAE, KNORAU
from sklearn.preprocessing import label_binarize
from sklearn.metrics import classification_report, roc_auc_score, accuracy_score
import mlflow
import mlflow.sklearn

# Funções de conversão, normalização e cross_validation
from conversao_normalizacao import converte_normaliza
from k_folds import cross_validation

# Caminho para o dataset
ds_path = 'Crop_Recommendation.csv'
folds = 10

# Carregar e pré-processar os dados
DS = converte_normaliza(ds_path)  # conversão e normalização
y = DS['Crop']  # extrai a última coluna, que é o label
```

Nesta seção, os dados são carregados do arquivo CSV usando as funções definidas nos arquivos conversao_normalizacao.py e k_folds.py. Os dados são então divididos em conjuntos de treinamento e teste usando validação cruzada com 10 folds.

Inicialização de Listas e Dicionários:

```
# Inicializa listas para armazenar os resultados de cada fold para seleções dinâmicas
resultKNU = []
resultKNE = []
resultLCA = []
resultOLA = []
resultMCB = []

# Inicializa dicionários para armazenar as métricas de cada classificador
metrics_knn = {"accuracy": [], "precision": [], "recall": [], "f1_score": [], "roc_auc": []}
metrics_svc = {"accuracy": [], "precision": [], "recall": [], "f1_score": [], "roc_auc": []}
metrics_rf = {"accuracy": [], "precision": [], "recall": [], "f1_score": [], "roc_auc": []}
```

Inicializa listas para armazenar os resultados de cada fold para seleções dinâmicas e inicializa dicionários para armazenar as métricas de cada classificador.

Criação e Definição do Experimento MLflow

```
# Criar o experimento
mlflow.create_experiment("exp_projeto_ciclo_2_V1")

# Definir o experimento atual
mlflow.set_experiment("exp_projeto_ciclo_2_V1")
```

Cria e define um experimento MLflow para rastrear os resultados do treinamento e avaliação dos modelos.

Loop de Treinamento e Avaliação dos Modelos

```python
for i in range(folds):
    # Divisão de treino e validação
    X_treino, X_validacao, y_treino, y_validacao = train_test_split( *arrays: X_train[i], y_train[i], test_size=0.5, random_state=None)

    with mlflow.start_run(run_name=f"Fold_{i + 1}"):
        # Criação e treinamento dos modelos KNN e RandomForest com Bagging e do SVM com AdaBoost
        knn = BaggingClassifier(estimator=KNeighborsClassifier(n_neighbors=5), n_estimators=50)
        svc = AdaBoostClassifier(estimator=SVC(kernel='linear', probability=True, C=1, gamma='auto'), n_estimators=50, learning_rate=1, algorithm='SAMME')
        rf = BaggingClassifier(estimator=RandomForestClassifier(n_estimators=10), n_estimators=50)

        # Treinamento dos classificadores
        knn.fit(X_treino, y_treino)
        svc.fit(X_treino, y_treino)
        rf.fit(X_treino, y_treino)

        # Avaliação dos modelos de seleção dinâmica
        # KNORA-U
        knu = KNORAU(pool_classifiers=[knn, svc, rf], k=5)
        knu.fit(X_treino, y_treino)
        resultKNU.append(knu.score(X_validacao, y_validacao))

        # KNORA-E
        kne = KNORAE(pool_classifiers=[knn, svc, rf], k=5)
        kne.fit(X_treino, y_treino)
        resultKNE.append(kne.score(X_validacao, y_validacao))

        # LCA
        lca = LCA(pool_classifiers=[knn, svc, rf], k=5)
        lca.fit(X_treino, y_treino)
        resultLCA.append(lca.score(X_validacao, y_validacao))

        # OLA
        ola = OLA(pool_classifiers=[knn, svc, rf], k=5)
        ola.fit(X_treino, y_treino)
        resultOLA.append(ola.score(X_validacao, y_validacao))

        # MCB
        mcb = MCB(pool_classifiers=[knn, svc, rf], k=5)
        mcb.fit(X_treino, y_treino)
        resultMCB.append(mcb.score(X_validacao, y_validacao))
```

Itera sobre cada fold dos dados de treino, divide os dados de treino em conjuntos de treino e validação. Dentro de cada fold, cria, treina e avalia modelos KNN, SVM e Random Forest, avalia os modelos usando diferentes métodos de seleção de modelos dinâmicos, avalia e registra as métricas de desempenho para cada modelo e registra os modelos treinados com MLflow.

Avaliação de Métricas

```python
def evaluate_and_log(model, X_validacao, y_validacao, metrics_dict, model_name):
    result = model.predict(X_validacao)
    acc = accuracy_score(y_validacao, result)
    metrics_dict["accuracy"].append(acc)

    report = classification_report(y_validacao, result, output_dict=True, zero_division=0)

    precision_values = [report[str(label)]['precision'] for label in np.unique(y) if str(label) in report]
    recall_values = [report[str(label)]['recall'] for label in np.unique(y) if str(label) in report]
    f1_score_values = [report[str(label)]['f1-score'] for label in np.unique(y) if str(label) in report]
    y_validacao_bin = label_binarize(y_validacao, classes=np.unique(y))
    result_bin = label_binarize(result, classes=np.unique(y))

    metrics_dict["precision"].append(np.mean(precision_values))
    metrics_dict["recall"].append(np.mean(recall_values))
    metrics_dict["f1_score"].append(np.mean(f1_score_values))

    if y_validacao_bin.shape == result_bin.shape:
        roc_auc = roc_auc_score(y_validacao_bin, result_bin, average="macro")
        metrics_dict["roc_auc"].append(roc_auc)
        mlflow.log_metric( key: f"{model_name} ROC AUC", roc_auc)

    mlflow.log_metric( key: f"{model_name} Accuracy", acc)
    mlflow.log_metric( key: f"{model_name} Precision", np.mean(precision_values))
    mlflow.log_metric( key: f"{model_name} Recall", np.mean(recall_values))
    mlflow.log_metric( key: f"{model_name} F1 Score", np.mean(f1_score_values))

    mlflow.log_params({f"{model_name} Parameters": model.get_params()})

# Avaliar e logar métricas para KNN, SVC e RF
evaluate_and_log(knn, X_validacao, y_validacao, metrics_knn,  model_name: "KNN")
evaluate_and_log(svc, X_validacao, y_validacao, metrics_svc,  model_name: "SVC")
evaluate_and_log(rf, X_validacao, y_validacao, metrics_rf,  model_name: "RF")

# Log models
mlflow.sklearn.log_model(knn, "knn_model")
mlflow.sklearn.log_model(svc, "svc_model")
mlflow.sklearn.log_model(rf, "rf_model")
```

Define uma função para avaliar o modelo e registrar suas métricas usando MLflow, avalia o modelo utilizando as previsões no conjunto de validação, calcula métricas como acurácia, precisão, recall, F1-score e AUC-ROC e Registra as métricas calculadas e os parâmetros do modelo com MLflow.

Resultados Gerais

```python
# Printar resultados médios das métricas
3 usages  new *
def print_metrics(metrics_dict, model_name):
    print(f"\nMétricas para {model_name}:")
    print(f"Acurácia: {np.mean(metrics_dict['accuracy']) * 100:.2f}%")
    print(f"Precisão: {np.mean(metrics_dict['precision']) * 100:.2f}%")
    print(f"Recall: {np.mean(metrics_dict['recall']) * 100:.2f}%")
    print(f"F1 Score: {np.mean(metrics_dict['f1_score']) * 100:.2f}%")
    if metrics_dict["roc_auc"]:
        print(f"ROC AUC: {np.mean(metrics_dict['roc_auc']) * 100:.2f}%")

print_metrics(metrics_knn, model_name: "KNN")
print_metrics(metrics_svc, model_name: "SVC")
print_metrics(metrics_rf, model_name: "RF")

with mlflow.start_run(run_name="Resultados gerais das métricas dos modelos de Machine Learning"):
    mlflow.log_metric( key: "KNN Mean Accuracy", np.mean(metrics_knn["accuracy"]) * 100)
    mlflow.log_metric( key: "KNN Mean Precision", np.mean(metrics_knn["precision"]) * 100)
    mlflow.log_metric( key: "KNN Mean Recall", np.mean(metrics_knn["recall"]) * 100)
    mlflow.log_metric( key: "KNN Mean F1 Score", np.mean(metrics_knn["f1_score"]) * 100)
    if metrics_knn["roc_auc"]:
        mlflow.log_metric( key: "KNN Mean ROC AUC", np.mean(metrics_knn["roc_auc"]) * 100)

    mlflow.log_metric( key: "SVC Mean Accuracy", np.mean(metrics_svc["accuracy"]) * 100)
    mlflow.log_metric( key: "SVC Mean Precision", np.mean(metrics_svc["precision"]) * 100)
    mlflow.log_metric( key: "SVC Mean Recall", np.mean(metrics_svc["recall"]) * 100)
    mlflow.log_metric( key: "SVC Mean F1 Score", np.mean(metrics_svc["f1_score"]) * 100)
    if metrics_svc["roc_auc"]:
        mlflow.log_metric( key: "SVC Mean ROC AUC", np.mean(metrics_svc["roc_auc"]) * 100)

    mlflow.log_metric( key: "RF Mean Accuracy", np.mean(metrics_rf["accuracy"]) * 100)
    mlflow.log_metric( key: "RF Mean Precision", np.mean(metrics_rf["precision"]) * 100)
    mlflow.log_metric( key: "RF Mean Recall", np.mean(metrics_rf["recall"]) * 100)
    mlflow.log_metric( key: "RF Mean F1 Score", np.mean(metrics_rf["f1_score"]) * 100)
    if metrics_rf["roc_auc"]:
        mlflow.log_metric( key: "RF Mean ROC AUC", np.mean(metrics_rf["roc_auc"]) * 100)
```

Define uma função para imprimir e registrar os resultados médios das métricas para cada modelo, calcula as médias das métricas de desempenho e imprime os resultados na tela e os registra com MLflow.

Resultados de Seleção Dinâmica

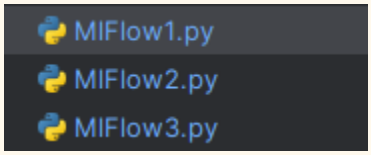
```
# Print and log dynamic selection results
print("\n","Resultado das seleções dinâmicas:", "\n")
print("KNORA-U: {:.2f}%".format(np.mean(resultKNU) * 100))
print("KNORA-E: {:.2f}%".format(np.mean(resultKNE) * 100))
print("LCA: {:.2f}%".format(np.mean(resultLCA) * 100))
print("OLA: {:.2f}%".format(np.mean(resultOLA) * 100))
print("MCB: {:.2f}%".format(np.mean(resultMCB) * 100))

with mlflow.start_run(run_name="Resultado dos algoritimos de seleção dinâmica"):
    mlflow.log_metric( key: "KNORA-U", np.mean(resultKNU) * 100)
    mlflow.log_metric( key: "KNORA-E", np.mean(resultKNE) * 100)
    mlflow.log_metric( key: "LCA", np.mean(resultLCA) * 100)
    mlflow.log_metric( key: "OLA", np.mean(resultOLA) * 100)
    mlflow.log_metric( key: "MCB", np.mean(resultMCB) * 100)
```

Imprime os resultados das seleções dinâmicas na tela, calcula a média dos resultados para cada método de seleção dinâmica e registra os resultados das seleções dinâmicas com MLflow.

Observações Finais

- Este código executa uma série de tarefas, desde a preparação e divisão dos dados até o treinamento, avaliação e registro dos resultados usando MLflow.
- Ele usa uma variedade de algoritmos de aprendizado de máquina e métodos de seleção de modelos dinâmicos para classificação de dados agrícolas.
- O uso de MLflow facilita o rastreamento e a comparação de diferentes configurações e experimentos.

Novos experimentos



Foram criados outros 3 arquivos praticamente iguais que mudar apenas os parâmetros dos modelos

MlFlow1:

Name	Value
KNN Parameters	{'bootstrap': True, 'bootstrap_features': False, 'estimator__algorithm': 'auto', 'estimator__leaf_size': 30, 'estimator__metric': 'minkowski', 'estimator__metric_params': None, 'estimator__n_jobs': None, 'estimator__n_neighbors': 5, 'estimator__p': 2, 'estimator__weights': 'uniform', 'estimator': KNeighborsClassifier(), 'max_features': 1.0, 'max_samples': 1.0, 'n_estimators': 50, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
RF Parameters	{'bootstrap': True, 'bootstrap_features': False, 'estimator__bootstrap': True, 'estimator__ccp_alpha': 0.0, 'estimator__class_weight': None, 'estimator__criterion': 'gini', 'estimator__max_depth': None, 'estimator__max_features': 'sqrt', 'estimator__max_leaf_nodes': None, 'estimator__max_samples': None, 'estimator__min_impurity_decrease': 0.0, 'estimator__min_samples_leaf': 1, 'estimator__min_samples_split': 2, 'estimator__min_weight_fraction_leaf': 0.0, 'estimator__monotonic_cst': None, 'estimator__n_estimators': 10, 'estimator__n_jobs': None, 'estimator__oob_score': False, 'estimator__random_state': None, 'estimator__verbose': 0, 'estimator__warm_start': False, 'estimator': RandomForestClassifier(n_estimators=10), 'max_features': 1.0, 'max_samples': 1.0, 'n_estimators': 50, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
SVC Parameters	{'algorithm': 'SAMME', 'estimator__C': 1, 'estimator__break_ties': False, 'estimator__cache_size': 200, 'estimator__class_weight': None, 'estimator__coef0': 0.0, 'estimator__decision_function_shape': 'ovr', 'estimator__degree': 3, 'estimator__gamma': 'auto', 'estimator__kernel': 'linear', 'estimator__max_iter': -1, 'estimator__probability': True, 'estimator__random_state': None, 'estimator__shrinking': True, 'estimator__tol': 0.001, 'estimator__verbose': False, 'estimator': SVC(C=1, gamma='auto', kernel='linear', probability=True), 'learning_rate': 1, 'n_estimators': 50, 'random_state': None}

```
knn =
BaggingClassifier(estimator=KNeighborsClassifier(n_neighbors=5),
n_estimators=50)

svc = AdaBoostClassifier(estimator=SVC(kernel='linear',
probability=True, C=1, gamma='auto'), n_estimators=50,
learning_rate=1, algorithm='SAMME')

rf =
BaggingClassifier(estimator=RandomForestClassifier(n_estimators=10),
n_estimators=50)
```

KNN com Bagging:

- `estimator=KNeighborsClassifier(n_neighbors=5)`: Este parâmetro especifica o tipo de estimador base a ser usado, neste caso, um classificador KNN com 5 vizinhos.
- `n_estimators=50`: Este parâmetro define o número de estimadores (ou classificadores) a serem criados no ensemble, ou seja, quantos classificadores KNN serão ajustados em subconjuntos diferentes dos dados.

SVM com AdaBoost:

- `estimator=SVC(kernel='linear', probability=True, C=1, gamma='auto')`: Este parâmetro especifica o tipo de estimador base a ser usado, neste caso, um classificador SVM com um kernel linear, probabilidade ativada, um parâmetro de regularização `C` de 1 e coeficiente de kernel `gamma` automático.
- `n_estimators=50`: Este parâmetro define o número máximo de estimadores na sequência do AdaBoost, ou seja, quantos classificadores SVM serão ajustados em sequência.

Random Forest com Bagging:

- `estimator=RandomForestClassifier(n_estimators=10)`: Este parâmetro especifica o tipo de estimador base a ser usado, neste caso, um classificador Random Forest com 10 árvores na floresta.
- `n_estimators=50`: Este parâmetro define o número de estimadores (ou classificadores) a serem criados no ensemble, ou seja, quantas florestas aleatórias serão ajustadas em subconjuntos diferentes dos dados.

MlFlow2:

Name	Value
KNN Parameters	{'bootstrap': True, 'bootstrap_features': False, 'estimator__algorithm': 'auto', 'estimator__leaf_size': 30, 'estimator__metric': 'minkowski', 'estimator__metric_params': None, 'estimator__n_jobs': None, 'estimator__n_neighbors': 10, 'estimator__p': 2, 'estimator__weights': 'distance', 'estimator': KNeighborsClassifier(n_neighbors=10, weights='distance'), 'max_features': 0.7, 'max_samples': 0.7, 'n_estimators': 100, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
RF Parameters	{'bootstrap': True, 'bootstrap_features': False, 'estimator__bootstrap': True, 'estimator__ccp_alpha': 0.0, 'estimator__class_weight': None, 'estimator__criterion': 'gini', 'estimator__max_depth': 20, 'estimator__max_features': 'sqrt', 'estimator__max_leaf_nodes': None, 'estimator__max_samples': None, 'estimator__min_impurity_decrease': 0.0, 'estimator__min_samples_leaf': 1, 'estimator__min_samples_split': 2, 'estimator__min_weight_fraction_leaf': 0.0, 'estimator__monotonic_cst': None, 'estimator__n_estimators': 50, 'estimator__n_jobs': None, 'estimator__oob_score': False, 'estimator__random_state': None, 'estimator__verbose': 0, 'estimator__warm_start': False, 'estimator': RandomForestClassifier(max_depth=20, n_estimators=50), 'max_features': 0.8, 'max_samples': 0.8, 'n_estimators': 100, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
SVC Parameters	{'algorithm': 'SAMME', 'estimator__C': 1.0, 'estimator__break_ties': False, 'estimator__cache_size': 200, 'estimator__class_weight': None, 'estimator__coef0': 0.0, 'estimator__decision_function_shape': 'ovr', 'estimator__degree': 3, 'estimator__gamma': 'scale', 'estimator__kernel': 'rbf', 'estimator__max_iter': -1, 'estimator__probability': True, 'estimator__random_state': None, 'estimator__shrinking': True, 'estimator__tol': 0.001, 'estimator__verbose': False, 'estimator': SVC(probability=True), 'learning_rate': 0.5, 'n_estimators': 100, 'random_state': None}

```
knn =
BaggingClassifier(estimator=KNeighborsClassifier(n_neighbors=10,
weights='distance'), n_estimators=100, max_samples=0.7,
max_features=0.7)

svc = AdaBoostClassifier(estimator=SVC(kernel='rbf',
probability=True, C=1.0, gamma='scale'),
n_estimators=100,learning_rate=0.5, algorithm='SAMME')

rf =
BaggingClassifier(estimator=RandomForestClassifier(n_estimators=50,
max_depth=20), n_estimators=100,max_samples=0.8, max_features=0.8)
```

KNN com Bagging:

- `estimator=KNeighborsClassifier(n_neighbors=10, weights='distance')`: Define o estimador base como um classificador KNN com 10 vizinhos e pesos de distância, o que significa que os vizinhos mais próximos têm mais influência.
- `n_estimators=100`: Especifica o número de estimadores no ensemble, neste caso, 100 classificadores KNN serão ajustados.
- `max_samples=0.7`: Define o tamanho máximo de cada subconjunto de treinamento, que é uma porcentagem do tamanho total do conjunto de treinamento. Aqui, 70% das amostras serão usadas para ajustar cada classificador KNN.
- `max_features=0.7`: Define o número máximo de características a serem consideradas ao procurar a melhor divisão em cada nó da árvore de decisão. Aqui, 70% das características serão consideradas em cada divisão.

SVM com AdaBoost:

- `estimator=SVC(kernel='rbf', probability=True, C=1.0, gamma='scale')`: Define o estimador base como um classificador SVM com kernel RBF (radial basis function), probabilidade ativada, um parâmetro de regularização `C` de 1.0 e coeficiente de kernel `gamma` configurado automaticamente pela escala dos dados.
- `n_estimators=100`: Especifica o número máximo de estimadores no ensemble, neste caso, 100 classificadores SVM serão ajustados.
- `learning_rate=0.5`: Define a taxa de aprendizado do AdaBoost, que controla a contribuição de cada classificador à previsão final. Um valor menor indica uma contribuição mais conservadora de cada classificador.
- `algorithm='SAMME'`: Especifica o algoritmo a ser usado no AdaBoost. 'SAMME' (Stagewise Additive Modeling using a Multiclass Exponential loss function) é uma versão geral do AdaBoost adaptada para problemas multiclasse.

Random Forest com Bagging:

- `estimator=RandomForestClassifier(n_estimators=50, max_depth=20)`: Define o estimador base como um classificador Random Forest com 50 árvores na floresta e uma profundidade máxima de cada árvore de decisão limitada a 20.
- `n_estimators=100`: Especifica o número de estimadores no ensemble, neste caso, 100 florestas aleatórias serão ajustadas.

- `max_samples=0.8`: Define o tamanho máximo de cada subconjunto de treinamento, que é uma porcentagem do tamanho total do conjunto de treinamento. Aqui, 80% das amostras serão usadas para ajustar cada floresta aleatória.
- `max_features=0.8`: Define o número máximo de características a serem consideradas ao procurar a melhor divisão em cada nó da árvore de decisão. Aqui, 80% das características serão consideradas em cada divisão.

Mlflow 3:

KNN Parameters	{'bootstrap': True, 'bootstrap_features': False, 'estimator__algorithm': 'auto', 'estimator__leaf_size': 30, 'estimator__metric': 'minkowski', 'estimator__metric_params': None, 'estimator__n_jobs': None, 'estimator__n_neighbors': 15, 'estimator__p': 2, 'estimator__weights': 'uniform', 'estimator': KNeighborsClassifier(n_neighbors=15), 'max_features': 0.6, 'max_samples': 0.6, 'n_estimators': 10, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
RF Parameters	{'bootstrap': True, 'bootstrap_features': False, 'estimator__bootstrap': True, 'estimator__ccp_alpha': 0.0, 'estimator__class_weight': None, 'estimator__criterion': 'gini', 'estimator__max_depth': 15, 'estimator__max_features': 'sqrt', 'estimator__max_leaf_nodes': None, 'estimator__max_samples': None, 'estimator__min_impurity_decrease': 0.0, 'estimator__min_samples_leaf': 1, 'estimator__min_samples_split': 5, 'estimator__min_weight_fraction_leaf': 0.0, 'estimator__monotonic_cst': None, 'estimator__n_estimators': 30, 'estimator__n_jobs': None, 'estimator__oob_score': False, 'estimator__random_state': None, 'estimator__verbose': 0, 'estimator__warm_start': False, 'estimator': RandomForestClassifier(max_depth=15, min_samples_split=5, n_estimators=30), 'max_features': 0.75, 'max_samples': 0.75, 'n_estimators': 10, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
SVC Parameters	{'algorithm': 'SAMME', 'estimator__C': 0.5, 'estimator__break_ties': False, 'estimator__cache_size': 200, 'estimator__class_weight': None, 'estimator__coef0': 0.0, 'estimator__decision_function_shape': 'ovr', 'estimator__degree': 3, 'estimator__gamma': 'auto', 'estimator__kernel': 'poly', 'estimator__max_iter': -1, 'estimator__probability': True, 'estimator__random_state': None, 'estimator__shrinking': True, 'estimator__tol': 0.001, 'estimator__verbose': False, 'estimator': SVC(C=0.5, gamma='auto', kernel='poly', probability=True), 'learning_rate': 0.8, 'n_estimators': 10, 'random_state': None}

```
knn =
BaggingClassifier(estimator=KNeighborsClassifier(n_neighbors=10,
weights='distance'), n_estimators=100, max_samples=0.7,
max_features=0.7)
```

```
svc = AdaBoostClassifier(estimator=SVC(kernel='rbf',
probability=True, C=1.0, gamma='scale'),
n_estimators=100,learning_rate=0.5, algorithm='SAMME')

rf =
BaggingClassifier(estimator=RandomForestClassifier(n_estimators=50,
max_depth=20), n_estimators=100,max_samples=0.8, max_features=0.8)
```

KNN com Bagging:

- `estimator=KNeighborsClassifier(n_neighbors=15, weights='uniform')`: Define o estimador base como um classificador KNN com 15 vizinhos e pesos uniformes, o que significa que todos os vizinhos têm o mesmo peso.
- `n_estimators=10`: Especifica o número de estimadores no ensemble, neste caso, 10 classificadores KNN serão ajustados.
- `max_samples=0.6`: Define o tamanho máximo de cada subconjunto de treinamento, que é uma porcentagem do tamanho total do conjunto de treinamento. Aqui, 60% das amostras serão usadas para ajustar cada classificador KNN.
- `max_features=0.6`: Define o número máximo de características a serem consideradas ao procurar a melhor divisão em cada nó da árvore de decisão. Aqui, 60% das características serão consideradas em cada divisão.

SVM com AdaBoost:

- `estimator=SVC(kernel='poly', probability=True, C=0.5, gamma='auto')`: Define o estimador base como um classificador SVM com kernel polinomial, probabilidade ativada, um parâmetro de regularização `C` de 0.5 e coeficiente de kernel `gamma` configurado automaticamente.
- `n_estimators=10`: Especifica o número máximo de estimadores no ensemble, neste caso, 10 classificadores SVM serão ajustados.
- `learning_rate=0.8`: Define a taxa de aprendizado do AdaBoost, que controla a contribuição de cada classificador à previsão final. Um valor menor indica uma contribuição mais conservadora de cada classificador.
- `algorithm='SAMME'`: Especifica o algoritmo a ser usado no AdaBoost, que é 'SAMME'.

Random Forest com Bagging:

- `estimator=RandomForestClassifier(n_estimators=30, max_depth=15, min_samples_split=5)`: Define o estimador base como um classificador Random Forest com 30 árvores na floresta, uma profundidade máxima de cada árvore de decisão limitada a 15 e o número mínimo de amostras necessárias para dividir um nó configurado como 5.
- `n_estimators=10`: Especifica o número de estimadores no ensemble, neste caso, 10 florestas aleatórias serão ajustadas.
- `max_samples=0.75`: Define o tamanho máximo de cada subconjunto de treinamento, que é uma porcentagem do tamanho total do conjunto de treinamento. Aqui, 75% das amostras serão usadas para ajustar cada floresta aleatória.
- `max_features=0.75`: Define o número máximo de características a serem consideradas ao procurar a melhor divisão em cada nó da árvore de decisão. Aqui, 75% das características serão consideradas em cada divisão.

Códigos para teste com algoritmos de boost

```python
# Lista dos modelos com parâmetros diferentes
models = {
    "XGBoost": XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100),
    "LightGBM": LGBMClassifier(max_depth=5, learning_rate=0.05, n_estimators=100),
    "GradientBoosting": GradientBoostingClassifier(max_depth=4, learning_rate=0.1, n_estimators=100)
}

# Criar o experimento
mlflow.create_experiment("exp_teste_modelos_de_boost")

# Definir o experimento atual
mlflow.set_experiment("exp_teste_modelos_de_boost")

# Loop sobre os modelos
for model_name, model in models.items():
    accuracies = []
    for i in range(folds):
        X_train_fold = np.array(X_train_folds[i])
        X_test_fold = np.array(X_test_folds[i])
        y_train_fold = np.array(y_train_folds[i])
        y_test_fold = np.array(y_test_folds[i])

        model.fit(X_train_fold, y_train_fold)
        y_pred = model.predict(X_test_fold)
        accuracy = accuracy_score(y_test_fold, y_pred)
        accuracies.append(accuracy)

    # Calcular a média das acurácias dos folds
    mean_accuracy = np.mean(accuracies)

    # Logar o resultado final no MLflow
    with mlflow.start_run(run_name=f"{model_name}_Final"):
        mlflow.log_metric( key: "Mean Accuracy", mean_accuracy)
        mlflow.sklearn.log_model(model, f"{model_name}_Final_Model")

print("Experimentos concluídos!")
```

Este experimento foi criado para avaliar especificamente algoritmos de boosting, como XGBoost, LightGBM e Gradient Boosting Classifier. Ao se concentrar em algoritmos de boosting, o objetivo provavelmente é explorar como esses métodos podem ser eficazes para o problema específico de classificação de culturas agrícolas.

O uso de algoritmos de boosting é interessante por várias razões. Esses algoritmos são conhecidos por sua capacidade de produzir modelos de alta qualidade, geralmente superando outros métodos em uma ampla variedade de conjuntos de dados. Eles funcionam construindo

uma sequência de modelos fracos e combinando suas previsões de uma maneira ponderada, enfatizando os exemplos mais difíceis de classificar. Isso pode ser especialmente útil para problemas complexos, nos quais as relações entre os recursos e os rótulos podem ser não lineares ou difíceis de capturar com métodos mais simples.

O experimento utiliza uma estratégia de validação cruzada com 10 folds, o que é uma prática comum para avaliar o desempenho dos modelos de forma robusta. Isso permite que cada modelo seja treinado e testado em diferentes subconjuntos dos dados, garantindo que os resultados não sejam influenciados pela aleatoriedade da divisão dos dados.

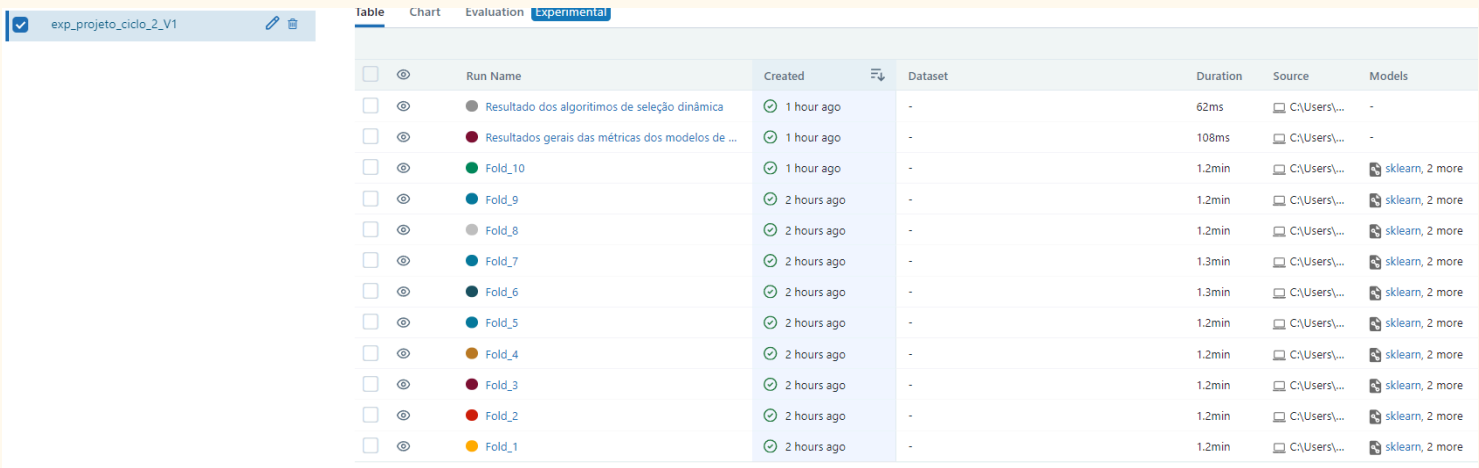
Além disso, ao usar o MLflow para registrar os resultados do experimento, o código facilita a organização e o rastreamento dos experimentos. O MLflow é uma ferramenta poderosa para gerenciar o ciclo de vida dos modelos de machine learning, desde o treinamento até a produção, e fornece recursos para monitorar e comparar o desempenho de diferentes modelos de forma eficiente.

Em resumo, este experimento representa uma abordagem sistemática e organizada para avaliar e comparar os algoritmos de boosting mais populares em um problema específico de classificação de culturas agrícolas, aproveitando as vantagens da validação cruzada e do rastreamento de experimentos com o MLflow para garantir resultados confiáveis e reprodutíveis.

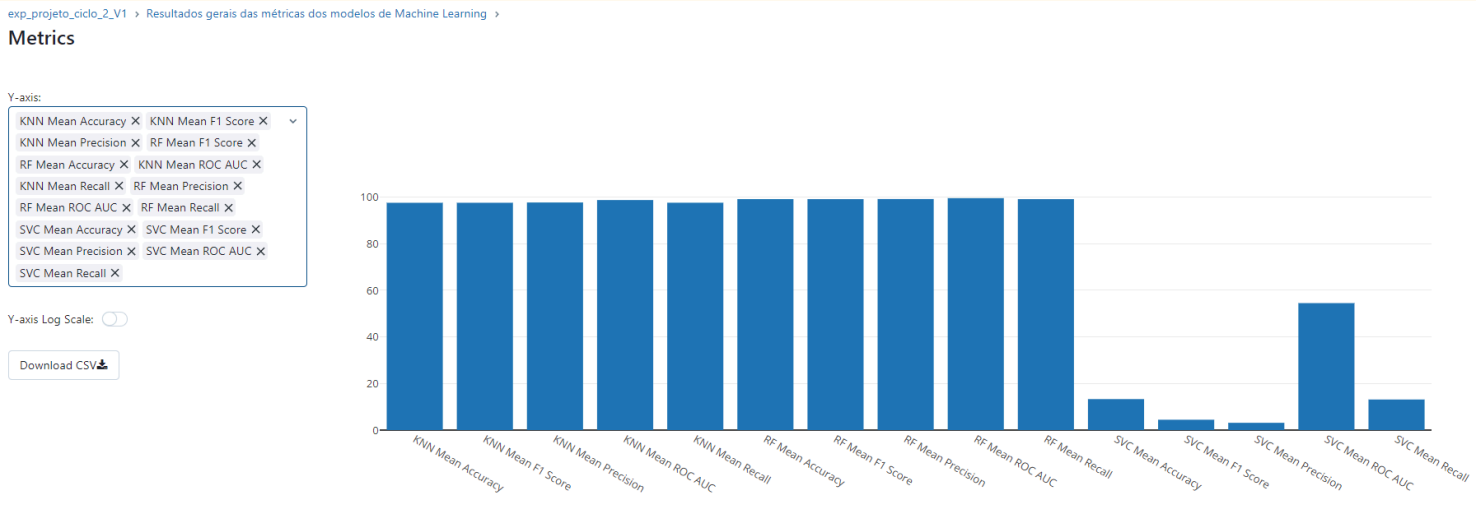
Resultados no MlFlow

Esse tópico abordará as prints dos resultados dos modelos na API do MLflow.

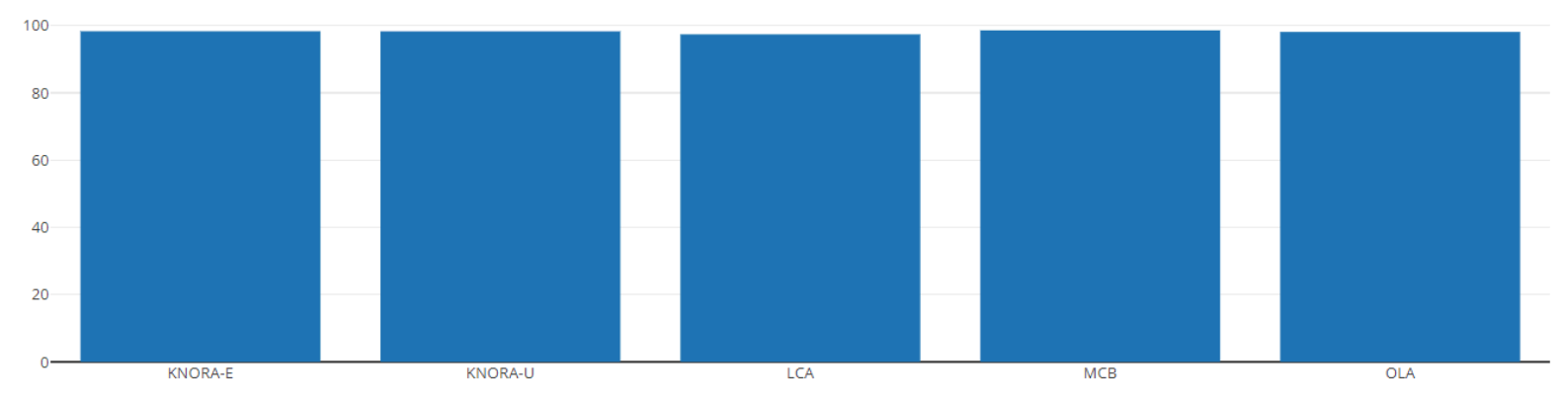
MlFlow1:



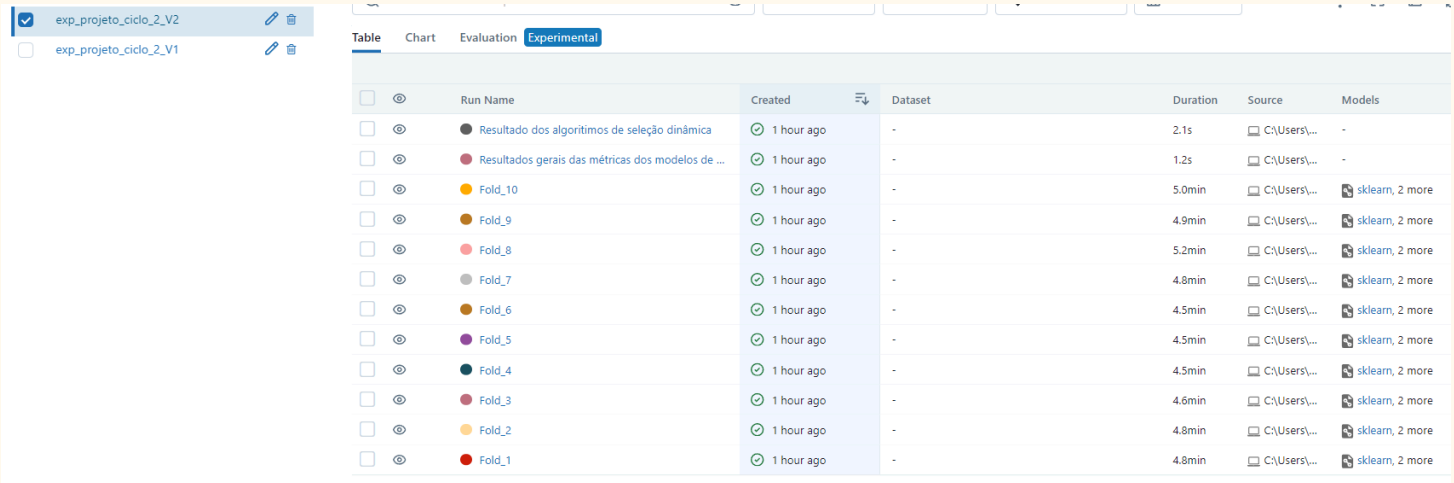
Resultados das métricas:



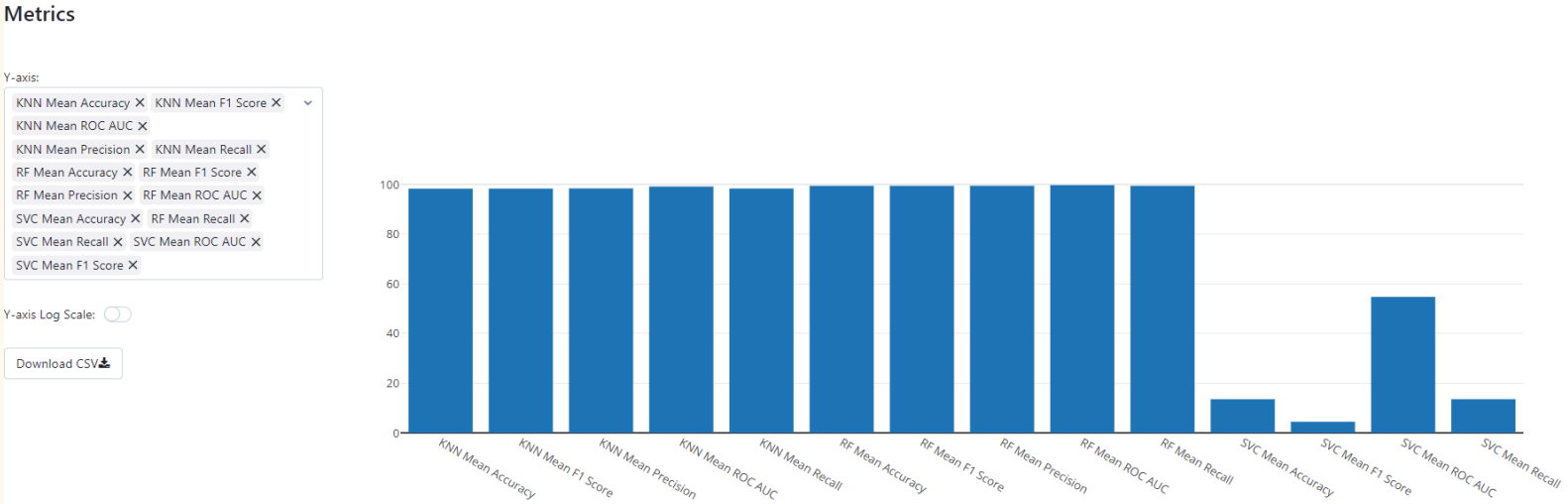
Resultados das seleções dinãmicas:



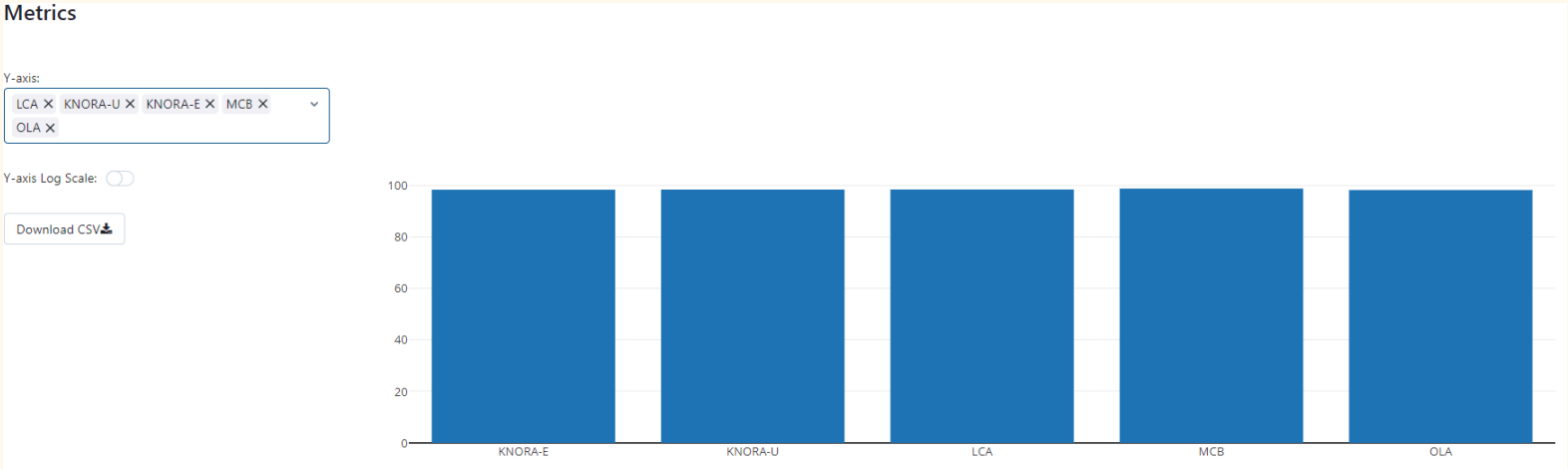
MlFlow2:



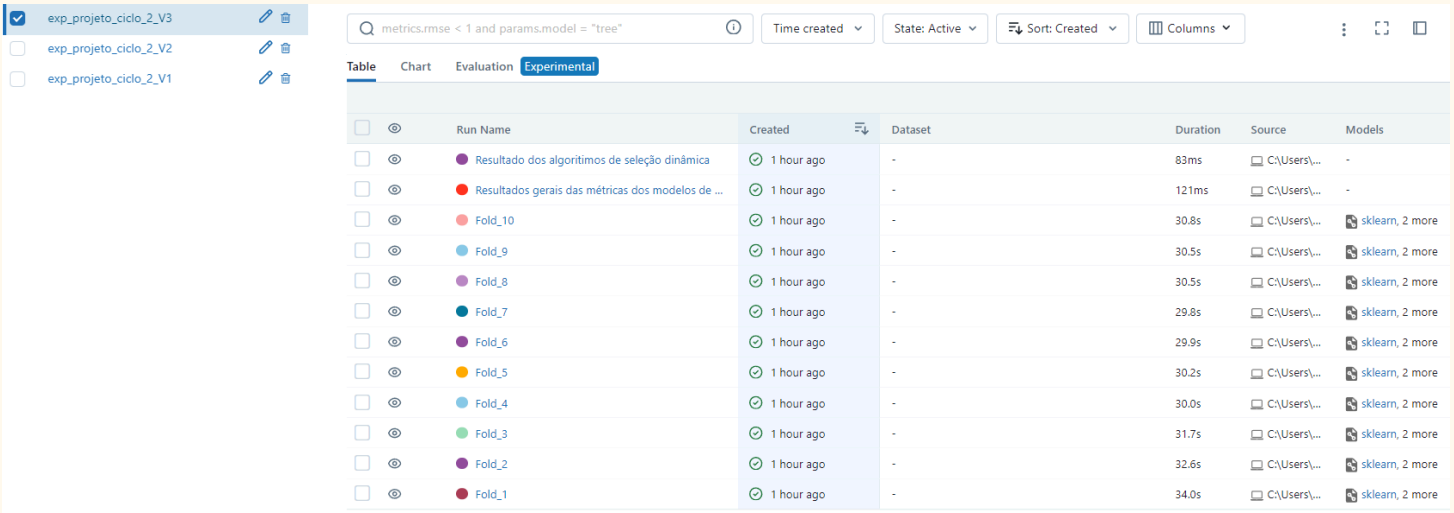
Resultados das métricas:



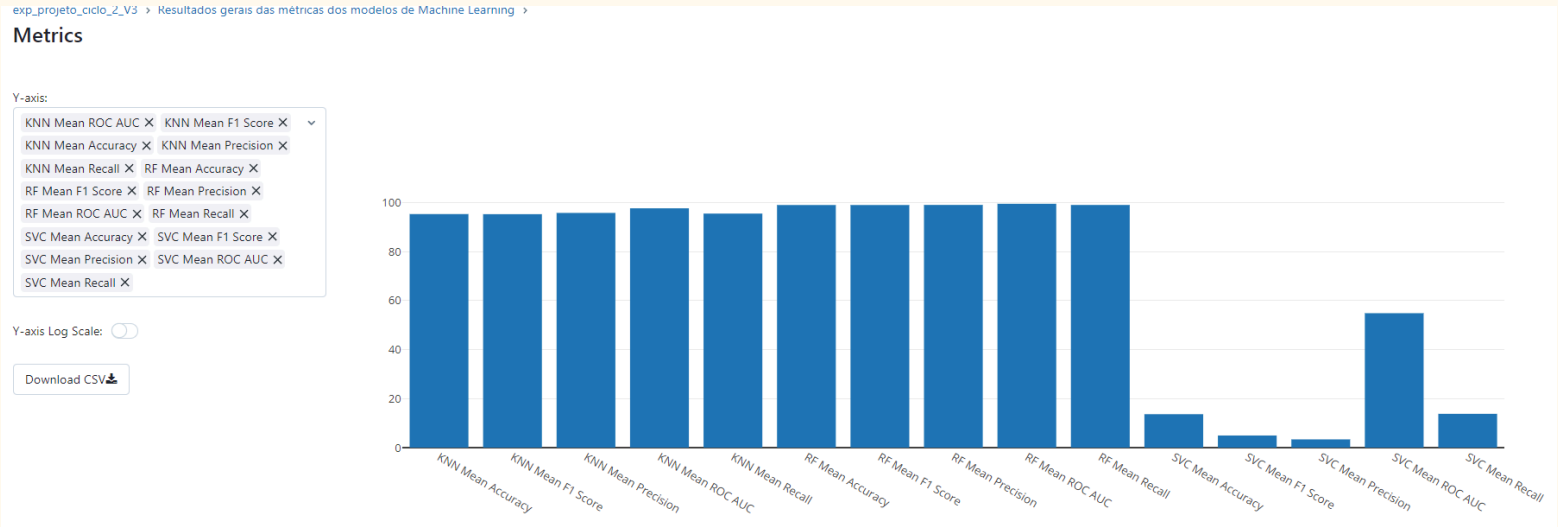
Resultados das seleções dinãmicas:



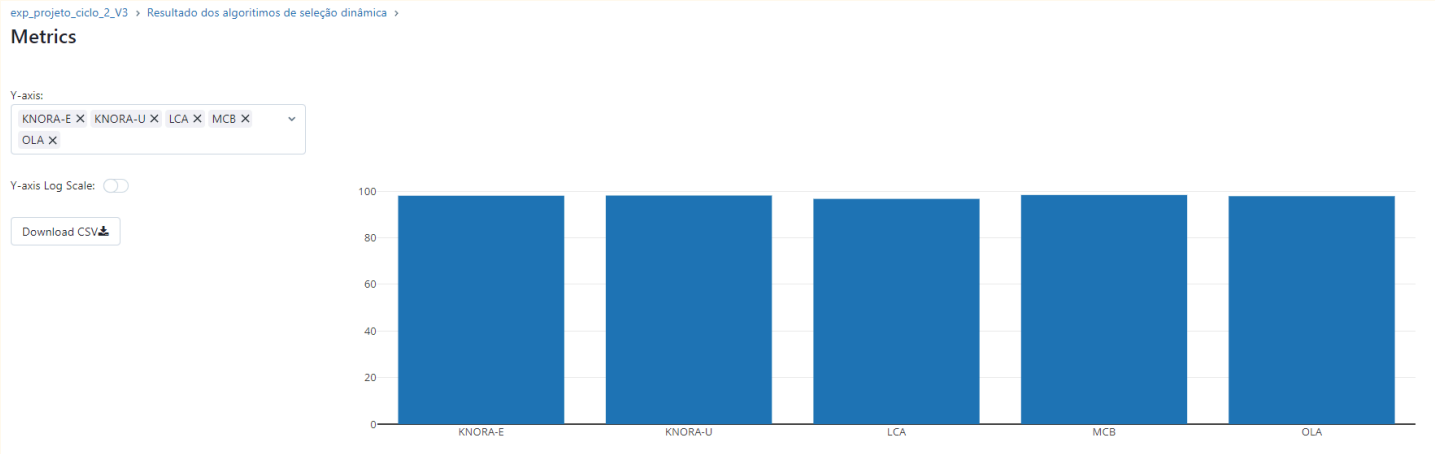
MlFlow3:



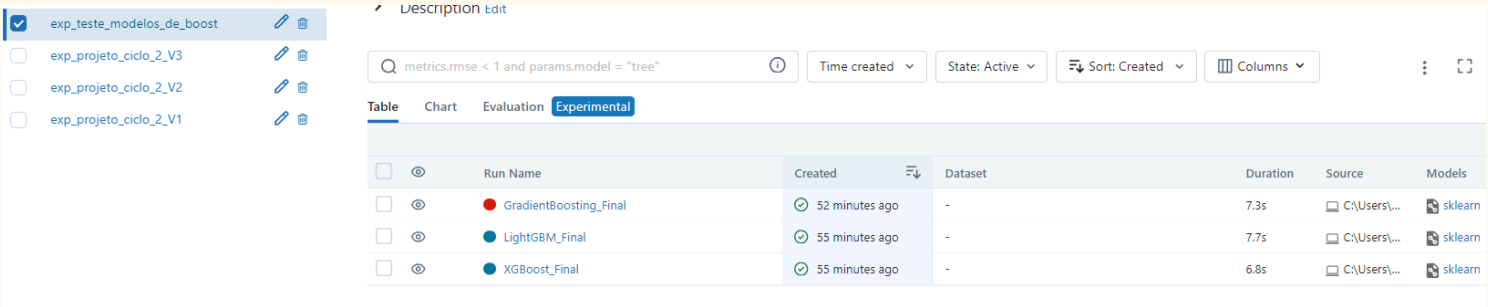
Resultado das métricas:



Resultado das seleções dinâmicas:



Algoritmos de Boost:



exp_teste_modelos_de_boost ›

GradientBoosting_Final

Run ID: 5eb3b9255d8946d2829d341d55111cef

User: Ronaldo Urquiza

› Description Edit

› Datasets

› Parameters

⌄ Metrics (1)

Name	Value
Mean Accuracy	0.9904545454545456

exp_teste_modelos_de_boost ›

LightGBM_Final

Run ID: bb56a606a93f4eeeaf7ce0524f8e1a74

User: Ronaldo Urquiza

› Description Edit

› Datasets

› Parameters

⌄ Metrics (1)

Name	Value
Mean Accuracy	0.9922727272727272

XGBoost_Final

Run ID: 01f007713b694d8e8bd377d3336059ed

User: Ronaldo Urquiza

› Description Edit

› Datasets

› Parameters

⌄ Metrics (1)

Name	Value
Mean Accuracy	0.9940909090909091