Second assignment: Decision trees

Trabalho realizado por:

- Constança Fernandes (up202205398)
- Diogo Silva (up201605359)
- João Baptista (up202207629)

Código

Estruras de dados

As estruturas de dados utilizadas foram, uma **class Node**, para os nós intermediários e **LeafNode** para os nós folha e **DTree** para a árvore.

Foi criada uma **class Evaluation** com funções para analisar a precisão (accuracy) da árvore, e desenhar uma matriz de confusão (confusion_matrix).

```
class LeafNode:
    def __init__(self, classif, size, val_ori):
        self.classif = classif      # qual a classificacao a
atribuir pelo no folha
        self.counter = size         # o numero de exemplos que
originaram o no
        self.origin_value = val_ori # valor antecedente que originou
o no


class Node:
    def __init__(self, attrib):
        self.attribute = attrib # o attributo que selecionado para
saber qual split seguir
        self.splits = {}        # um dicionario onde as chaves sao
os valores possiveis do attribute, que identificam os nos criados a
partir dos mesmos
```

Class Evaluation

Na classe "Evaluation" temos 2 funções:

- **accuracy()** -> calcula a taxa de precisão 0 a 1.
- **confusion_matrix()** -> imprime uma matrix de confusão com coluna para cada valor único de classificação e 2 linhas, o número de corretamente previstos e o número de incorretos.

```
class Evaluation:
    def accuracy(df_pred, df_class):
        assert(len(df_pred) == len(df_class))
        well_classified = 0
```

```
        i = 0
        for classif in df_class:
            if df_pred[i] == classif:
                well_classified += 1
            i += 1
        print(f"Accuracy: {(well_classified/len(df_class)):.2f}")


    def confusion_matrix(df_pred, df_class):
        assert(len(df_pred) == len(df_class))
        well_classified = {}
        wrong_classified = {}
        #create dict for TP | FP
        for val in list(set(df_class)):
            assert(val in list(set(df_pred)))
            well_classified[val] = 0
            wrong_classified[val] = 0

        i = 0
        for classif in df_class:
            if df_pred[i] == classif:
                well_classified[classif] += 1
            else:
                wrong_classified[classif] += 1
            i += 1

        Evaluation.print_matrix(well_classified, wrong_classified)

    def print_matrix(well_classified:dict, wrong_classified:dict):
        spaces = []

        res = "Real classification was correctly predicted or
wrongly\n"
        #header
        res += f"class->"
        for key in well_classified.keys():
            res += f" | {key}"
            spaces.append(len(key))

        #correctly classified
        res += " |\ncorrect"
        i = 0
        for val in well_classified.values():
            space = " "*int((spaces[i]-len(str(val)))/2)
            res += f" | {space}{val}{space}" if ((spaces[i]-
len(str(val))) % 2 == 0) else f" | {space}{val}{space} "
            i += 1

        #wrongly classified
        res += " |\nwrong  "
        i = 0
        for val in wrong_classified.values():
            space = " "*int((spaces[i]-len(str(val)))/2)
            res += f" | {space}{val}{space}" if ((spaces[i]-
```

```
len(str(val))) % 2 == 0) else f" | {space}{val}{space} "
            i += 1
        res += " |\n"
        print(res)
```

Funções auxiliares e dependências

```
import math
import random
import pandas as pd
import numpy as np
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.model_selection import train_test_split
from copy import deepcopy


def best_attribute_split(df,dy):
    '''escolhe o melhor atributo para fazer split com base na
<information gain>
    returna o nome do atributo com maior valor de information
gain'''
    best = df.columns[0]
    size = dy.size
    entropy_class = entropy(dy)
    val_gain = float('-inf')

    for attri in df.columns:
        tmp_val = 0

        for val in df[attri].unique():
            subset_dy = dy.loc[df[df[attri] == val].index]
            tmp_val += ((len(subset_dy)/size) * entropy(subset_dy))

        tmp_val = entropy_class - tmp_val
        if val_gain < tmp_val:
            val_gain = tmp_val
            best = attri

    return best

def entropy(attribute):
    '''calcula a entropia de cada attributo'''
    size = attribute.size
    ent_val = 0
    for val in attribute.unique():
        count = 0
        for i in attribute.index:
            if attribute[i] == val:
                count += 1
        if(count == 0): # only true when val == nan
            count = attribute.isna().sum()
        p = count/size
        ent_val += (p * math.log2(p))
    return -ent_val
```

```
def all_classification_equal(setX, setY):
    '''True if all classification are equal, False otherwise'''
    classif = setY[setX.index[0]]
    for i in setX.index: # get the index of line i presente in sub
set (can be 1,3,..,13; not necessary linear 1..n)
        if classif != setY[i]:
            return False
    return True

def most_common_output(indices,setY):
    '''usado quando o subset for vazio ou nao for possivel fazer
outro split por nao haver mais attributos e a classificacao nao e
unanime
    em caso de empate entre 2 (ou mais) classificacoes e selecionada
uma aleatoriamente'''
    outputs = {}
    for i in indices:
        if setY[i] not in outputs.keys():
            outputs[setY[i]] = 1
        else: outputs[setY[i]] += 1

    max_value = max(outputs.values())
    keys_with_max_value = [key for key, value in outputs.items() if
value == max_value]

    return random.choice(keys_with_max_value)

def new_case(curr_node, best_node, max_counter):
    ''' usada quando queremos prever um exemplo e nao existe um
caminho direto ate a um no folha
    entao quando chegar ao limite da arvore é calculado o output
mais comum a partir da sub arvore com raiz no no atual'''
    if type(curr_node) == LeafNode:  # If the node is a leaf
        return curr_node

    for node in curr_node.splits.values():
        leaf_node = new_case(node, best_node, max_counter)
        if leaf_node.counter > max_counter:
            best_node = leaf_node
            max_counter = leaf_node.counter

    return best_node

def print_tree(node,depth):
    '''usada para imprimir a arvore como pedido'''
    if type(node) == LeafNode:
        space = "    "*(depth-1)
        return f"{space}{node.origin_value}: {node.classif}
({node.counter})\n"

    else:
        space = "    "*depth
        res = f"{space}<{node.attribute}>\n"
```

```
        for key in node.splits.keys():

            if type(node.splits[key]) == Node:
                res += f"{space}    {key}:\n"

            res += print_tree(node.splits[key],depth+2)

    return res
```

Class DTree

- create_DTree()

É a função principal para o desenvolvimento da árvore e usa as funções auxiliares:

- **best_attribute_split()** -> vai escolher qual a coluna que cria um melhor split com base na informação ganha (information gain), método referido no livro secção 19.3.3.
- **most_common_output()** -> retorna qual o valor que o nó folha vai classificar com base no mais comum que é atribuído ao sub set a considerar (em caso de empate escolhe aleatoriamente).
- **all_classification_equal()** -> retorna "True" caso todos os exemplos do sub set tenham a mesma classificação.

- predict()

Função usada para prever novos casos sem saber a sua classificação.

- Percorre a árvore para cada exemplo e classifica.
- Caso o exemplo percorra a árvore de uma forma que não tenha sido vista no treino, então usamos **new_case()** que retorna o nó folha com maior número de exemplos no "counter" a partir da sub árvore com raís em "curr_node".

- process_data()

Recebe o dataset completo e sem alterações.

- Separa a coluna com as classificações (a última) do resto do dataset.
- Analisa coluna a coluna do restante dataset para tratar cada tipo de dado, isto é:
-
  - Caso todos os valores da sejam diferentes vai ignorar o atributo pois não tem qualquer influência.
-
  - Para os atributos numéricos com mais de 10% de valores diferentes vamos discretizar com o *KBinsDiscretizer* do sklean.preprocessing agrupando os dados em 5 intervalos. (n_bins = 5)
-
  - Para atributos categoricos tranforma todos os caracteres em minúsculas.

Retorna os dados processados dfx com os atributos e dfy com os as respetivas previsoes caso ***treino*** seja True, caso contrário vai considerar que recebe apenas atributos e retornar apenas dfx sem separar a ultima coluna do dataset.

- start_algorithm()

Vai receber um dataframe com atributos e outro com as previsões.

- Caso ***split_test_train*** seja True vai dividir os dataframes em 70% treino e 30% teste e analisar automaticamente a accuracy.
- Se for False vai apenas fazer fit/criar a árvore com o dataframe todo.

```
class DTree:
    def __init__(self):
        self.root = None
        self.num_nodes = 0


    def process_data(dfx, new_examples = False):
        if not new_examples:
            dfy = dfx.iloc[:, -1]
            dfx.drop(columns=dfy.name ,inplace=True)

        '''process data based on it's category'''
        est = KBinsDiscretizer(n_bins=5, encode='ordinal',
strategy='uniform')

        for col in dfx.columns:
            #if all values are diffent we don't need that column
            if len(dfx) == len(dfx[col].unique()):
                dfx.drop(columns=col ,inplace=True)

            elif len(dfx[col].unique())/len(dfx) > 0.10 and
(np.issubdtype(dfx[col].dtype, np.integer) or
np.issubdtype(dfx[col].dtype, np.float64)):
                dfx[col] =
pd.DataFrame(est.fit_transform(dfx[col].to_frame()),columns=[col])

            elif type(dfx[col].dtype) == str:
                for i in dfx[col].index:
                    if not pd.isna(dfx[col][i]):
                        dfx[col][i] = dfx[col][i].lower()

        if new_examples: return dfx
        else: return dfx, dfy


    def start_algorithm(self,dfx:pd ,dfy ,split_test_train):
        #70/30 to train
        if split_test_train:
            X_train, X_test, y_train, y_test = train_test_split(dfx,
dfy, test_size=0.3)
            self.create_DTree(X_train,y_train)
            pred = self.predict(X_test)
            Evaluation.accuracy(pred,y_test)
```

```
            print()
            Evaluation.confusion_matrix(pred,y_test)

        else :
            self.create_DTree(dfx,dfy)


    def create_DTree(self, dx_train, dy_train, curr_node=None):

        # calculate best attribute to split based in information
gain and create Nodes
        best_attrib = best_attribute_split(dx_train,dy_train)
        #create a Node for the split
        if curr_node==None: #only true for first iteration when root
will be None
            node = Node(best_attrib)
            self.root = node
        else:
            curr_node.attribute = best_attrib
            node = curr_node

        dfa = deepcopy(dx_train)
        dfa.drop(columns=best_attrib,inplace=True)

        for val in dx_train[best_attrib].unique():
            sub_set = dfa[dx_train[best_attrib] == val]

            if len(sub_set) == 0:
                #create a leaf node with the most common output
                node.splits[val] =
LeafNode(most_common_output(dx_train.index,dy_train),len(dx_train.in
dex),val) #add to 'split' dictionary a leafnode for val
                self.num_nodes += 1

            elif all_classification_equal(sub_set,dy_train):
                #create a leaf node
                node.splits[val] =
LeafNode(dy_train[sub_set.index[0]],len(sub_set.index),val) #add to
'split' dictionary a leafnode for val
                self.num_nodes += 1

            elif len(sub_set.columns) == 0:
                #create a leaf node with the most common output
                node.splits[val] =
LeafNode(most_common_output(sub_set.index,dy_train),len(sub_set.inde
x),val) #add to 'split' dictionary a leafnode for val
                self.num_nodes += 1

            else:
                #need to split again
                node.splits[val] = Node(None)
                self.create_DTree(sub_set, dy_train,
curr_node=node.splits[val])
                self.num_nodes += 1
```

```
    def predict(self,df):
        pred = []
        for i in df.index: #each row in df to classify
            curr_node = self.root
            while type(curr_node) != LeafNode:
                try:
                    curr_node =
curr_node.splits[df[curr_node.attribute][i]]
                except:
                    #used when a new case is seen and there is no
specific branch for it
                    #in that case we will choose the classification
that occurs the most in subtree from the current node
                    curr_node = new_case(curr_node,None, float("-
inf"))
            pred.append(curr_node.classif)

        return pred


    def __str__(self):
        return print_tree(self.root,0)
```

Testes dos datasets

Restaurant dataset

```
df1 = pd.read_csv("datasets/restaurant.csv")
X1, y1 = DTree.process_data(df1)

arvore1 = DTree()
arvore1.start_algorithm(X1, y1, split_test_train = False)
print(f"Arvore gerada:\n{arvore1}\nPredicts:")
arvore1.predict(df1)
```

```
Arvore gerada:
<Pat>
    Some: Yes (4)
    Full:
        <Hun>
            Yes:
                <Type>
                    Thai:
                        <Fri>
                            No: No (1)
                            Yes: Yes (1)
                    Italian: No (1)
                    Burger: Yes (1)
            No: No (2)
    nan: Yes (12)
```

```
Predicts:

['Yes',
 'No',
 'Yes',
 'Yes',
 'No',
 'Yes',
 'Yes',
 'Yes',
 'No',
 'No',
 'Yes',
 'Yes']
```

Análise

- O algoritmo cria uma árvore equivalente ao exemplo do livro pelo que confirmamos que está correto.
- Apenas tivemos uma dificuldade ao considerar com os valores NaN (not a number) pois nesse caso o subset iria ficar vazio e em vez de calcular o mais comum com um subset, calculamos com o dataset dessa recursão e acabamos por ficamos com um valor "count" inflacionado.

Weather dataset

```
df2 = pd.read_csv("datasets/weather.csv")
X2, y2 = DTree.process_data(df2)

arvore2 = DTree()
arvore2.start_algorithm(X2, y2, split_test_train = False)
print(f"Arvore gerada:\n{arvore2}\nPredicts:")
arvore2.predict(df2)
```

```
Arvore gerada:
<Weather>
    sunny:
        <Humidity>
            3.0: no (1)
            4.0: no (2)
            0.0: yes (2)
    overcast: yes (4)
    rainy:
        <Windy>
            False: yes (3)
            True: no (2)

Predicts:

['no',
 'no',
 'yes',
```

```
 'yes',
 'yes',
 'no',
 'yes',
 'no',
 'yes',
 'yes',
 'yes',
 'yes',
 'yes',
 'no']
```

Análise

Pelo output do modelo podemos concluir que as melhores condições para jogar ténis são quando está:

- "overcast" (nublado)

ou

- "sunny" (ensolarado) e pouca humidade

ou

- "rainy" (chuvoso) e sem estar ventoso

Iris dataset

```
df3 = pd.read_csv("datasets/iris.csv")
X3, y3 = DTree.process_data(df3)


arvore3 = DTree()
arvore3.start_algorithm(X3, y3, split_test_train = True)
# divide_df = True significa que ao desenvolver o algoritmo vamos
dividir o dataset em 70% treino e 30% teste
# ao fazer divide_df=True o algoritmo vai fazer predict aos 30% de
teste
# e calcular a accuracy e criar uma matrix de confusao para a
classificacao real (se foi bem prevista ou nao)
print(f"Arvore gerada:\n{arvore3}")
```

```
Accuracy: 0.91

Real classification was correctly predicted or wrongly
class-> | Iris-setosa | Iris-versicolor | Iris-virginica |
correct |     16      |       14        |       11       |
wrong   |      1      |        3        |       0        |

Arvore gerada:
<petalwidth>
    2.0:
        <sepallength>
            1.0: Iris-versicolor (12)
```

```
            2.0:
                <petallength>
                    2.0: Iris-versicolor (4)
                    3.0:
                        <sepalwidth>
                            1.0: Iris-virginica (4)
                            0.0: Iris-virginica (1)
            3.0: Iris-versicolor (7)
    0.0: Iris-setosa (33)
    4.0: Iris-virginica (18)
    3.0:
        <sepallength>
            2.0:
                <petallength>
                    2.0: Iris-versicolor (1)
                    3.0:
                        <sepalwidth>
                            1.0: Iris-virginica (5)
                            2.0: Iris-virginica (4)
            0.0: Iris-virginica (1)
            1.0: Iris-virginica (2)
            4.0: Iris-virginica (6)
            3.0:
                <sepalwidth>
                    2.0:
                        <petallength>
                            3.0: Iris-virginica (3)
    1.0: Iris-versicolor (4)
```

Análise

O modelo apresenta uma precisão de 91% o que é uma taxa elevada, em 50 testes obtivemos uma média de 92% de acerto.

Concluímos que o nosso algoritmo pode ser bem utilizado para aprender padrões e classificar casos que nunca viu anteriormente (pelo menos com este tipo de dados).

Connect4 dataset

```
df4 = pd.read_csv("datasets/connect4.csv")
X4, y4 = DTree.process_data(df4)



arvore4 = DTree()
arvore4.start_algorithm(X4, y4, split_test_train = True)
# divide_df = True significa que ao desenvolver o algoritmo vamos
dividir o dataset em 70% treino e 30% teste
# ao fazer divide_df=True o algoritmo vai fazer predict aos 30% de
teste
# e calcular a accuracy e criar uma matrix de confusao para a
classificacao real (se foi bem prevista ou nao)
'''print(arvore4) a arvore é extensa pelo que nao vamos imprimir'''
```

```
Accuracy: 0.74

Real classification was correctly predicted or wrongly
class-> | loss | win | draw |
correct | 3205 | 11268 | 579  |
wrong   | 1706 | 2135  | 1374 |


'print(arvore4) a arvore é extensa pelo que nao vamos imprimir'
```

Análise

Apesar de o foco deste dataset não é dividir em teste/treino testamos na mesma para analisar o algoritmo e apesar da accuracy ser menor do que no dataset Iris, com 74%

Ao testar jogar 4connected A* com a heuristica do trabalho anterior vs A* com a predict da DT concluimos que usar a DT não é um bom método. Quando a board está praticamente vazia jogar em qualquer coluna era considerado vitória.

Ao analisar o dataset de treino reparamos que cada exemplo so tinha 8 peças, então criamos 5 boards aleatoriamente com 7 peças para analisar as previsões. Para dada board a árvore acaba por classificar as possiveis jogadas sempre com o mesmo output

```
from Game4InLine import Game4InLine as G4Line
import random

def gerar_board(game):
    for i in range(7):
        legal  = game.legal_moves()
        game.play(random.choice(legal))

#gerar 5 boards com 7 peças total e testar a predict da DT
for i in range(5):
    game=G4Line(6,7)
    gerar_board(game)
    print(game)
    game.resultado_DT()
    print()
```

```
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | o | - | - | - | - | - |
| o | o | x | x | x | x | - |

Colunas que originam "wins": []
Colunas que originam "draws": []
 Colunas que originam "loses": [1, 2, 3, 4, 5, 6, 7]

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
```

```
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | x |
| - | x | - | - | - | - | o |
| - | o | x | o | - | - | x |

Colunas que originam "wins": [1, 2, 3, 4, 5, 6, 7]
Colunas que originam "draws": []
 Colunas que originam "loses": []

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| x | - | - | - | - | - | x |
| o | o | - | - | o | x | x |

Colunas que originam "wins": [1, 2, 3, 4, 5, 6, 7]
Colunas que originam "draws": []
 Colunas que originam "loses": []

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | x | - | - | - | - |
| - | - | x | - | - | - | - |
| - | x | o | o | o | - | x |

Colunas que originam "wins": [1, 2, 3, 4, 5, 6, 7]
Colunas que originam "draws": []
 Colunas que originam "loses": []

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | x | - | - | - | - |
| - | - | o | - | - | - | - |
| x | o | x | - | x | o | - |

Colunas que originam "wins": [1, 2, 3, 4, 5, 6, 7]
Colunas que originam "draws": []
 Colunas que originam "loses": []
```