Relatório-3ºTrabalho de IA

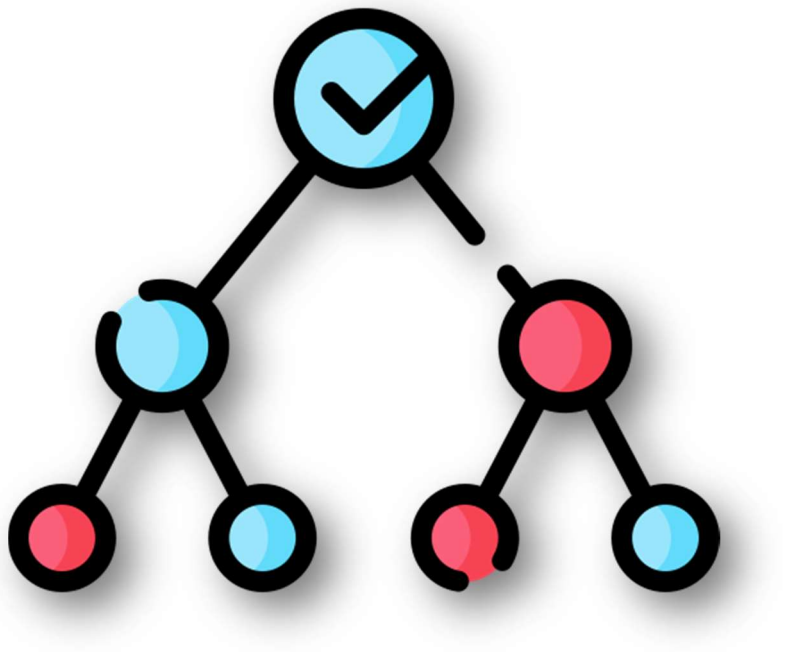

Imagem 1 - https://www.cliently.com/blog/decision-tree-models-explained_capa

Trabalho realizado por:

Anna Sellani (202107998);

Inês Cardoso (202107268);

Pedro Sousa (202108383);

Índice:

1.Introdução

O que é uma árvore de decisão?

As Árvores de Decisão (DT's) são um método de aprendizagem supervisionado não paramétrico utilizado para a classificação e regressão.
O objetivo deste método é criar um modelo que preveja o valor de uma variável de destino aprendendo regras de decisão simples inferidas com recurso aos dados.

Quando é que utilizamos árvores de decisão?

Árvores de decisão são utilizadas quando planeamos classificar um atributo de dados baseados em dados anteriores.

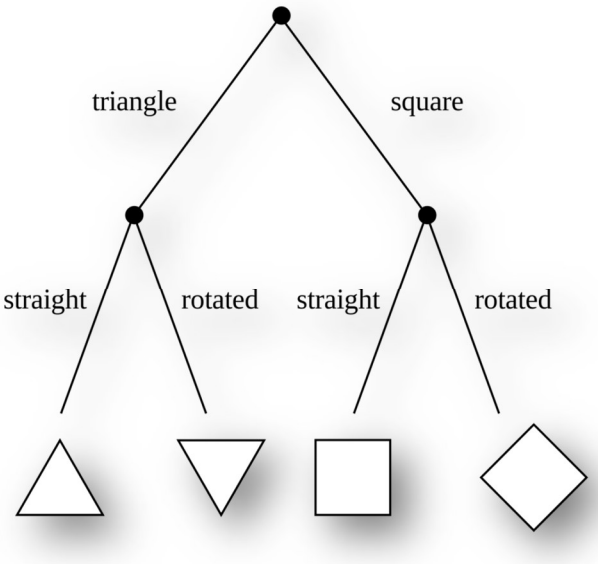

Imagem 2- Exemplo de uma árvore de decisão

Referência: https://corporatefinanceinstitute.com/resources/data-science/decision-tree/

2. Algoritmos para indução de árvores de decisão

Algoritmos mais populares

- **ID3**: é um algoritmo utilizado para gerar uma árvore de decisão a partir de um dataset.
  Este algoritmo utiliza uma abordagem *top-down greedy* para construir uma árvore de decisão, ou seja, começamos a construir a árvore por cima e, a cada iteração, selecionamos o melhor recurso no momento presente para criar um nó.

  Nota: Geralmente, o ID3 é usado apenas para problemas de classificação apenas com recursos nominais.

- **CART** (Classification and Regression Trees): tipo de algoritmo de classificação utilizado para construir uma árvore de decisão com base no índice de impurezas de Gini.
  Dentro deste algoritmo vamos ter os seguintes tipos de árvores:
  Árvores de classificação: quando a variável de destino é contínua, a árvore é utilizada para encontrar a "classe" na qual a variável de destino tem uma maior probabilidade de estar.
  Árvores de regressão: São usadas para prever o valor de uma variável contínua.

- **Qui-quadrado:** é um teste de hipótese estatística usado na análise de tabelas de contingência quando os tamanhos de amostra são grandes.
  Este teste é utilizado para examinar se duas variáveis categóricas são independentes da estatística do teste.

Algoritmo ID3, como funciona e as suas limitações

- Como referido anteriormente, o ID3 é um algoritmo de aprendizagem de árvores de decisão simples desenvolvido por Ross Quinlan.
- A ideia básica do algoritmo ID3 é construir a árvore de decisão com a ajuda da pesquisa greedy utilizando a abordagem *top-down greedy*, deste modo, testa cada atributo de cada nó da árvore.

  Para encontrar uma maneira ótima de classificar um conjunto de dados, vamos precisar de uma função que nos ajuda a perceber qual é a melhor maneira de dividir este nosso conjunto de dados.

- **Entropia:** conhecida como o controlador da árvore de decisão para decidir onde dividir os dados. O algoritmo ID3 usa entropia para calcular a homogeneidade de uma amostra.
  Se a amostra for completamente homogénea, a entropia é zero e, se a amostra for igualmente dividida, tem entropia de um.

$$H = -\sum_{i} p_i(\log_2 p_i)$$

Imagem 3 - Fórmula da entropia
Referência: https://computersciencesource.wordpress.com/2010/01/10/year-2-machine-learning-entropy/

- Limitações/Desvantagens:
  - Não garante uma solução ótima pois como utiliza uma estratégia greedy pode ficar preso em soluções locais ótimas (pode ser evitado se usarmos *backtracking*);
  - Pode dar *overfit* ao dataset de treino;
  - Este algoritmo é feito para ser utilizado dados nominais.

3.Descrição da Implementação

Linguagem utilizada: Python

Por que escolheu esta linguagem?

- Python foi escolhida como a linguagem de programação utilizada neste projeto devido ao facto de ser uma linguagem comumente entendida por todos os membros do grupo e devido ao facto de python ser muito utilizado na área da ciência de dados.

Há alguma vantagem em utilizar esta linguagem para resolver este tipo de problema?

- Como python é comum na área de data science, este possui uma vasta gama de librarias que nos suportaram no desenvolvimento do projeto (pandas, numpy).

- Também, devido à natureza simples de python, testar o programa e corrigir erros é relativamente intuitivo e rápido, facilitando o desenvolvimento do projeto.

Estruturas de dados utilizadas e justificação

- Durante o projeto, nós produzimos uma "libraria" para ter um funcionamento semelhante aos dataframes do Pandas. Esta tem como objetivo tornar a manipulação de dados mais legível, e a estrutura de dados bem-definida, de maneira a não cometer erros na manipulação destes.

- Como output da função ID3, temos um dicionário que guarda informações na seguinte estrutura:

$$\{\text{classe: [atributo, counter, } \{\} \text{]}\}$$

Imagem 4-Feita pelo grupo

Cada chave do dicionário representa uma classe que deu origem a um node na árvore, sendo que o seu valor é uma lista que contém o nome do seu atributo, a contagem de elementos que estão abrangidos nesse node, e outro dicionário que segue o mesmo formato. Caso seja uma folha, o dicionário dentro da lista será só uma string com o valor previsto.

Estrutura do código:

O nosso código é composto por 2 ficheiros python, e uma pasta *datasets*, que contém todos os datasets utilizados para o funcionamento da árvore de decisão:

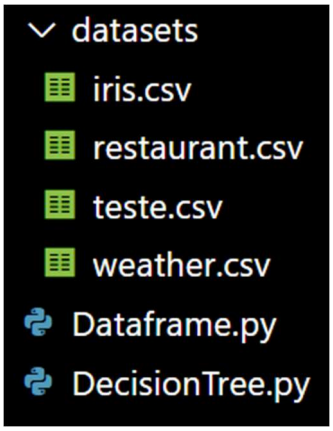

Imagem 5 - Hierarquia do código (imagem retirada do código)

Dataframe.py:

Cada objeto dataframe contém as seguintes variáveis:

1. Nome do ficheiro (ficheiro CSV);
2. Lista de atributos;
3. Matriz de dados;
4. Número de entradas;
5. Nome da variável alvo;
6. Coluna da variável alvo.

Dataframes podem ser criados chamando o inicializador com ficheiro CSV, com uma matriz, ou com outro Dataframe.

Cada dataframe também contém alguns métodos que fazem o seguinte:

1. read_csv() – lê o dataset dado pelo nome do ficheiro CSV;

2. Format_continuous() – Formata o dataframe para tornar as variáveis numéricas em categóricas (só é funcional para os datasets utilizados para teste);

3. Get_target_col() – retorna uma lista representante da coluna da variável alvo;

4. Get_data() – retorna uma matriz com as entradas (sem o header)
5.
6. If_contains(x) – retorna uma lista com as entradas que contenham o valor x;

7. If_contains_in_column(x, col) – semelhante à anterior, mas só adiciona à lista as entradas que contenham os valores x na coluna dada;

8. getColumn(x) – retorna o índice da coluna dada pelo nome x (usada para encontrar o índice das colunas de atributos);

9. get_unique_values(y) – retorna uma lista com todas os valores únicos da coluna y;

10. get_most_common_class() – retorna a classe mais comum do atributo alvo;

11. drop(col) – remove a coluna indicada do dataframe.

DecisionTree.py:

Este trata-se do nosso ficheiro principal aonde toda a parte da árvore de decisão é realizada. Neste ficheiro temos as seguintes funções:

1. calc_total_entropy() – calcula a entropia de todo o dataset;

2. calc_entropy() – calcula a entropia de um único atributo;

3. calc_info_gain() – calcula o ganho de informação de um atributo;

4. find_most_informative_feature() – utiliza as 3 funções anteriores para calcular e retornar o nome da função mais informativa;

5. make_tree() – função recursiva que gera a árvore de decisão;

6. print_dictionary() – função utilizada para dar print ao dicionário da árvore de modo mais legível;

7. Format_input() – recebe o input para prever na árvore e formata de modo a ficar dentro dos parâmetro da árvore;

8. Id3() – função chamada para começar o processo de gerar a árvore de decisão;

9. Predict_target() – função que prevê uma entrada no dicionário.

Estrutura dos Algoritmos:

O programa funciona de seguinte modo:

- O main é chamado, aonde é criado um dataframe utilizando o nome do dataset dado como argumento ao executar o programa. Este dataframe lê o ficheiro do dataset, remove a coluna do ID e formata os valores contínuos consoante o dataset usado.

```
def main():
    df = DataFrame("datasets/" + sys.argv[1] + ".csv") #importing the dataset from the disk

    df.read_csv() #reading the dataset
    df.drop(0) # drop the ID row
    df.format_continuous() # format continuous data (only works for iris and weather)

    tree = id3(df, df.targetCol)

    print_dictionary(tree)
```

Imagem 6 - Retirada do nosso código

- De seguida a função id3 é chamada.

```
def id3(train_data: DataFrame, TarCol: int) -> dict:
    tree = {} #tree which will be updated
    class_list = train_data.get_unique_values(TarCol) #getting unqiue classes of the label

    label = train_data.atributos[TarCol] #getting the label name
    return make_tree(tree, train_data, label, class_list) #start calling recursion
```

Imagem 7- Retirada do nosso código

- Esta cria um dicionário vazio que guardará a futura árvore de decisão, uma lista com todos os valores únicos da variável alvo e guarda numa variável *label* o nome da variável alvo, chamando assim a função recursiva make_tree.

```
def make_tree(root: dict, train_data: DataFrame, label: str, class_list: list):

    if len(train_data.get_data()) != 0: #if dataset becomes enpty after updating

        # if all the rows have same class, return the class
        if len(train_data.get_unique_values(train_data.getColumn(label))) == 1:
            root = train_data.get_unique_values(train_data.getColumn(label))[0]
            return root

        # if there are no more features, return the most common class
        if len(train_data.atributos) == 1:

            root = train_data.get_most_common_class()
            return root

        # if there are more features, find the most informative feature
```

Imagem 8- Retirada do nosso código

- Na função make_tree, começamos por testar se algum caso base foi atingido. Caso contrário, calculamos o atributo com o maior ganho de informação, chamando a função find_most_informative_feature()

```
def find_most_informative_feature(train_data: DataFrame, class_list: list):

    feature_list = train_data.atributos #finding the feature names in the dataset
    max_info_gain = -1
    max_info_feature = None

    for feature in feature_list[:-1]:  #for each feature in the dataset
        feature_info_gain = calc_info_gain(feature, train_data, class_list)

        if max_info_gain < feature_info_gain: #selecting feature name with highest information gain
            max_info_gain = feature_info_gain
            max_info_feature = feature

    return max_info_feature
```

Imagem 9- Retirada do nosso código

- Aqui, guardamos numa lista os atributos do dataframe atual, e, para cada atributo (exceto o target), calculamos o ganho. No final, retornamos o atributo com maior ganho (guardando sempre em memória o maior e o seu ganho por cada iteração, por este motivo, caso haja empates em ganho, o primeiro a ser calculado é o usado).

- O cálculo de ganho é feito nesta função:

```
def calc_info_gain(feature_name, train_data: DataFrame, class_list: list):

    feature_value_list = train_data.get_unique_values(train_data.getColumn(feature_name))
#unqiue values of the feature

    total_row = train_data.numEntradas
    feature_info = 0.0

    for feature_value in feature_value_list:
        feature_value_data = train_data.if_contains(feature_value) #filtering rows with that feature_value
        feature_value_data.insert(0, train_data.atributos)
        feature_value_data = DataFrame("nan", matrix= feature_value_data)

        feature_value_count = feature_value_data.numEntradas
        feature_value_entropy = calc_entropy(feature_value_data, class_list)
#calculcating entropy for the feature value
        feature_value_probability = feature_value_count/total_row
        feature_info += feature_value_probability * feature_value_entropy
#calculating information of the feature value


    return calc_total_entropy(train_data, class_list) - feature_info #calculating information gain by subtracting
```

Imagem 10 - Retirada do nosso código

- Começando por guardar os valores únicos do atributo a ser calculado, contamos o número de entradas totais, e, para cada valor único do atributo, criamos um "sub-dataframe" com todas as entradas que contêm esse valor.
  Usando esse "sub-dataframe", calculamos a entropia que será somada ao total. Essa parte é calculada em calc_entropy.

```
def calc_entropy(feature_value_data: DataFrame, class_list: list) -> float:

    class_count = feature_value_data.numEntradas
    entropy = 0

    for c in class_list:
        label_class_count = len(feature_value_data.if_contains(c)) #row count of class c

        entropy_class = 0

        if label_class_count != 0:
            probability_class = label_class_count/class_count #probability of the class
            entropy_class = - probability_class * np.log2(probability_class)  # entropy
        entropy += entropy_class # adding the entropy of the class to the total entropy of the feature value

    return entropy
```

Imagem 11 - Retirada do nosso código

- Com o cálculo da entropia feito para todos as classes, a função calc_info_gain() retorna a entropia total subtraída ao valor calculado de todas as entropias de cada classe.

- O cálculo da entropia total é realizado na seguinte função:

```
def calc_total_entropy(train_data: DataFrame, class_list: list) -> float:

    total_row = train_data.numEntradas #the total size of the dataset
    total_entr = 0 # total entropy of the dataset

    for c in class_list: #for each class in the label

        total_class_count = 0 # number of rows with that class
        total_class_count = len(train_data.if_contains(c))

        total_class_entr = 0
        if total_class_count != 0:
            total_class_entr = (float)(- (total_class_count/total_row)*np.log2(total_class_count/total_row))
#entropy of the class
        else:
            total_class_entr = 0 # if there are no rows with that class, entropy is 0 (log(0) is undefined)

        total_entr += total_class_entr #adding the class entropy to the total entropy of the dataset

    return total_entr
```

Imagem 12 - Retirada do nosso código

- Com o ganho de informação de todos os atributos, e o atributo escolhido pela find_most_informative_attribute(), a função make_tree() entra no processo recursivo de geração da árvore:

```
# separate the dataset based on the most informative feature and make subtrees based on the values of the feature
        feature_value_list = train_data.get_unique_values(train_data.getColumn(max_info_feature)) #unqiue values of the feature

        tree= {} #root of the tree

        col = train_data.getColumn(max_info_feature)

        for feature_value in feature_value_list:

            # the tree is actually built in this format:
            # {feature_name: [feature_value, subdata, class_count, subtree]}
            # where the subtree is a dictionary with the same format

            #this way, we can easily traverse the tree and find the class of a row, the entry count of a class, etc.
            node = []
            node.append(max_info_feature)

            # reset subdata
            sub_data = []

            # make deepcopies before passing to the function to avoid changing the original dataset (python am i right?)
            sub_data = copy.deepcopy(train_data.if_contains_in_column(feature_value, col)) #filtering rows with that feature_value
            atributos = copy.deepcopy(train_data.atributos)

            # add the attribute names to the subdata
            sub_data.insert(0, atributos)

            sub_data = DataFrame("nan", matrix= sub_data) # convert to dataframe

            sub_data.drop(col) #dropping the feature column

            node.append(sub_data.numEntradas)

            # (this is a base case inside the recursive function, how curious)
            # if there are no more entries, before we make a subtree, we find the most common class and return it
            if len(sub_data.get_data()) == 0:

                # define the leaf node as the most common class
                root[max_info_feature][feature_value] = train_data.get_most_common_class()
                return tree

            tree.update({feature_value: []}) #updating the tree with the feature name

            # recursive call to make_tree
            sub_dict = make_tree(tree[feature_value], sub_data, label, class_list)
            # add the subtree to the node
            node.append(sub_dict)
            tree[feature_value] = node #updating the tree with the subtree

    return tree
```

Imagem 13 - Retirada do nosso código

- Começando por guardar os valores únicos do atributo com maior ganho, esta cria um “nó” para a árvore de decisão, entrando num ciclo que, para cada classe do atributo, cria uma lista que guardará o nome do atributo, a contagem de elementos desse atributo, e os futuros sub-dicionários dos nós consequentes.

- Começando por dar append ao nome do atributo, este cria uma cópia dos dados e dos atributos, e cria um "sub-dataframe" com os dados que fazem parte da classe do atributo atual, eliminando a coluna desse atributo para as seguintes iterações. Se o "sub-dataframe" ficar vazio (um caso base) retornamos a função, caso contrário, adicionamos ao nó atual da árvore uma entrada sendo a chave o valor da classe do atributo e o valor uma lista.

- Neste momento, é chamada de maneira recursiva a função make_tree() para criar os restantes nós até chegarem a um caso base, e quando tal acontece, é colocado no valor da chave da classe do atributo a lista que contém um sub-dicionário caso não seja uma folha, ou uma string com a classificação caso seja uma folha.

- No final, é retornado o dicionário raiz, que contém a árvore na sua totalidade.

- Voltando à função main, com a árvore já gerada:

```
tree = id3(df, df.targetCol)

    print_dictionary(tree)

    print("\nWant to test prediction? (y/n)")
    if input() == "n":
        return

    print("Input filepath: ", end="")
    filepath = input()
    print("")
    predict_file(filepath, tree, df)
```

Imagem 14 - Retirada do nosso código

- É dado print da árvore de decisão segundo o modelo pedido pelo guião, através da função print_dictionary():

```
def print_dictionary(dictionary, indent=''):

    for key, value in dictionary.items():
        attribute, count, sub_dict = value

        print(indent + attribute + ":")
        print(indent + '    ' + key + ':' + " (" + str(count) + ")", end= '')

        if isinstance(sub_dict, str):
            print(" " + sub_dict)
        else:
            print("")
            print_dictionary(sub_dict, indent + "    ")
```

Imagem 15 - Retirada do nosso código

- E caso seja pedido para testar com inputs dados por um ficheiro, recebemos o caminho do CSV com as entradas prontas a testar, e chamamos a função predict_file():

```
def predict_file(filepath: str, tree: dict, df: DataFrame) -> None:

    file = open(filepath, "r")
    lines = file.readlines()
    lines.pop(0)
    file.close()

    f = copy.copy(df.atributos)

    for line in lines:
        print("Input: " + line, end="")
        df = format_input(line, df, f)
        print("Predicted class: " + predict_target(tree, df) + "\n")
```

Imagem 16 - Retirada do nosso código

- Lemos as entradas do CSV e, para cada linha, formatamos para seguirem o formato definido pelo dataset escolhido, e chamamos a função predict_target():

```
def predict_target(dictionary, feat: DataFrame) -> str:

    if isinstance(dictionary, str):
        return dictionary

    # get the first key of the dictionary
    key = list(dictionary.keys())

    attribute = dictionary[key[0]][0]

    # get the index of the feature in the row
    index = feat.getColumn(attribute)

    for key, value in dictionary.items():
        if isinstance(value[2], str) and key == feat.get_data()[0][index]:
            return value[2]

        if key == feat.get_data()[0][index]:
            sub_dict = value[2]
            return predict_target(sub_dict, feat)

    return "No class found"
```

Imagem 17 - Retirada do nosso código

- Esta função, também recursiva, navega pela estrutura do dicionário criada até encontrar um caso em que, ao invés de ter um sub-dicionário, tem a string que contém a classe atribuída pela árvore de decisão, retornando esse valor.

5. Resultados

Testando com os datasets de exemplo dados, podemos observar as seguintes árvores de decisão:

Dataset Weather:

```
Weather:
   sunny: (5)
   Humidity:
      81-90: (2) no
   Humidity:
      >90: (1) no
   Humidity:
      65-70: (2) yes
Weather:
   overcast: (4) yes
Weather:
   rainy: (5)
   Windy:
      FALSE: (3) yes
   Windy:
      TRUE: (2) no
```

Dataset Iris:

```
petallength:
   2-2.5: (50) Iris-setosa
petallength:
   >4: (84)
   petalwidth:
      1-2: (61)
      sepallength:
         6-7: (32)
         sepalwidth:
            >2: (32) Iris-versicolor
      sepallength:
         5-6: (22)
         sepalwidth:
            >2: (22) Iris-versicolor
      sepallength:
         4-5: (1) Iris-virginica
      sepallength:
         >7: (6) Iris-virginica
   petalwidth:
      2-3: (23) Iris-virginica
petallength:
   3.5-4: (11) Iris-versicolor
petallength:
   3-3.5: (4) Iris-versicolor
petallength:
   2.5-3: (1) Iris-versicolor
```

Dataset Restaurant:

```
Pat:
   Some: (4) Yes
Pat:
   Full: (6)
   Alt:
      Yes: (5)
      Bar:
         No: (3)
         Fri:
            No: (1) No
         Fri:
            Yes: (2)
            Hun:
               Yes: (1) Yes
            Hun:
               No: (1) No
      Bar:
         Yes: (2)
         Fri:
            Yes: (2)
            Hun:
               Yes: (2)
               Price:
                  $$$: (1) No
               Price:
                  $: (2)
                  Rain:
                     No: (2)
                     Res:
                        Yes: (1) No
                     Res:
                        No: (1) Yes
   Alt:
      No: (1) No
Pat:
   None: (2) No
```

6.Comentários Finais e Conclusões

Analisando estes dados, podemos observar que:

- As árvores de decisão foram geradas corretamente.

- O nosso programa consegue receber input de um ficheiro CSV e prever de acordo com a árvore dada.

Desta maneira, podemos dar como concluído o nosso projeto.
O nosso algoritmo ID3 foi implementado com sucesso e, apesar de algumas dificuldades, pode-se dizer que foi desenvolvido de acordo com o esperado. Conseguimos observar o funcionamento deste e de tirar informação relevante aos dados submetidos.

6.Referências Bibliográficas

- https://scikit-learn.org/stable/modules/tree.html
- https://www.seldon.io/decision-trees-in-machine-learning
- https://towardsdatascience.com/decision-trees-for-classification-id3-algorithm-explained-89df76e72df1
- https://en.wikipedia.org/wiki/ID3_algorithm
- https://www.analyticssteps.com/blogs/classification-and-regression-tree-cart-algorithm
- https://towardsdatascience.com/machine-learning-101-id3-decision-trees-and-entropy-calculation-1-a1d66ee9f728
- https://www.cse.unsw.edu.au/~cs9417ml/DT1/decisiontreealgorithm.html