UNIVERSIDADE DE ÉVORA

2020/2021 – 5º Semestre

RELATÓRIO - TRABALHO PRÁTICO

APRENDIZAGEM AUTOMÁTICA

Python 3 - Árvores de decisão

com "pruning"

Trabalho realizado por:

Diogo Castanho, 42496

Pedro Grilo, 43012

1.1 – Introdução e Objetivo

- Uma **árvore de decisão** é uma ferramenta de suporte à tomada de decisão que usa um gráfico no formato de árvore e demonstra visualmente as condições e as probabilidades para se chegar a resultados. É um método de aprendizado de máquina **supervisionado**.

- O método consiste na contínua subdivisão de um espaço amostral (**chamado de raiz**) em classes menores por meio de testes (**chamados de nós**) feitos para subdividir – em dois subespaços para manter uma maior homogeneidade na divisão - esse espaço amostral em classes até que se tenha um subconjunto homogêneo o suficiente para ser classificado como uma mesma classe, criando, assim, um nó terminal (**chamado de folha**).

- Quando árvores de decisão são construídas, muitos ramos ou subárvores podem conter ruídos ou erros. O aprendizado é muito específico ao conjunto de treinamento, não permitindo generalizar para o conjunto de teste (**overfitting**). Para melhorar o modelo, utilizam-se métodos de poda (**pruning**) na árvore, cujo objetivo é melhorar a taxa de acerto do modelo para novas amostras que não foram utilizadas no treinamento. Existem diversas formas de realizar uma poda, e todas elas são classificadas como **pré-poda** ou **pós-poda**.

- Dentre os métodos de poda existentes, destacam-se: **Cost Complexity Pruning, Reduced Error Pruning, Minimum Error Pruning (MEP), Pessimistic Pruning, Error-Based Pruning (EBP), Minimum Description Length (MDL) Pruning, Minimum Message Length (MML) Pruning, Critical Value Pruning (CVP), OPT e OPT-2**.

- Neste trabalho o objetivo é utilizar o método **Reduced Error Pruning** e, dentro dos algoritmos para construção de árvores de decisão será implementado o algoritmo **ID3**.

1.1 – Introdução e Objetivo

(Continuação)

Assim, considerando os temas e objetivos referidos na página anterior, pretende-se com implementar em Python 3 uma classe que:

- Seja parcialmente compatível com as classes dos classificadores do sklearn;
- Gere uma árvore de decisão de acordo com o algoritmo apresentado nos slides das aulas teóricas (algoritmo ID3) seguida de pruning com o método REP (Reduced Error Pruning);
- Ter uma parametrização para decidir a medida de pureza (opção entre gini, entropia e erro) e indicação se terá ou não pruning;
- Contenha o método fit(x,y) onde seja gerada uma árvore de decisão em função dos dados de treino x e y, em que X é um array bidimensional com conjunto de dados para o treino com a dimensão (n_samples, n_features) e y é um array unidimensional com as classes de cada exemplo, com a dimensão (n_samples);
- Na utilização do método fit, a geração da árvore seja feita com 75% dos dados de treino e o pruning (Reduced Error Pruning) feito com 25%.;
- Apresente o método score(x, y) onde este devolve o valor da exatidão (accuracy) com o conjunto de dados de teste x e y;

```
classifier = myDecisionTreeREPrune()
classifier.fit(x_train, y_train)
result = classifier.score(x_test, y_test)
print("Percentagem de casos corretamente classificados {:2.2%}".format(result))
```

Legenda: *Exemplo do teste que deverá ser feito à classe no programa;*

1.2 - Decisões tomadas na realização do trabalho

Inicialmente, tentou perceber-se o problema em causa (neste caso, o funcionamento das árvores de decisão e a sua construção seguindo os passos e respeitando as regras impostas pelo algoritmo) lendo por isso, mais que uma vez o enunciado e os slides disponibilizados.

Após se perceber bem os objetivos do trabalho, os valores gerados para **x_test, x_train**, **y_test** e **y_train** pela função utilizada **train_test_split()** do sklearn e a manutenção dos dados, começou-se por inicializar a classe principal e outras classes necessárias ao funcionamento do programa e as suas parametrizações;

1.2.1 – Ficheiros criados e utilizados

Para organização do trabalho foram criados e utilizados os seguintes ficheiros:

- ***DecisionTreeREPrune.py***
- ***main.py***
- ***treeNode.py***

Para testes, foram utilizados os seguintes ficheiros disponibilizados no moodle:

- **"weather.nominal.csv"**
- **"soybean.csv"**
- **"vote.csv"**
- **"contact-lenses.csv"**

1.3 – Explicação das funções utilizadas

- Dentro da função main.py apenas temos código para ler os dados, dividi-los usando o train_test_split do scikit-learn que nos foi aconselhado pelos professores.
- Dentro da treeNode.py temos:

```
class treeNode():
    '''
    A node class for a decision tree.
    '''
    def __init__(self):
        self.column = None  # index of feature to split on
        self.value = None   # value of the feature to split on
        self.name = None    #name of feature
        self.left = None    # (TreeNode) left child
        self.right = None   # (TreeNode) right child
        self.leaf = False   # (bool)   true if node is a leaf, false otherwise
        self.header = None  # array of headers from the data (just to use in the print function)
        self.classes = Counter()  # (Counter) only necessary for leaf node:
                                  #           key is class name and value is
                                  #           count of the count of data points
                                  #           that terminate at this leaf
```

- (cont) onde damos ao nosso Node todos os atributos que vão ser necessários durante a realização do trabalho. Para além disso, temos funções que vão ajudar a dar print na nossa árvore posteriormente, e uma função predict_one que retorna o valor da previsão da classificação de um atributo.
- Onde está a maior parte do código é no ficheiro DecisionTreeREPrune.py (o nosso classifier).
- Temos as funções gini e entropy, que calculam a pureza do conjunto de dados associados a estas

```
def _entropy_(self, y):    #returns the entropy impurity value from the y target values

    size= y.shape[0]   #amount of rows with y values
    impurity = 0
    for values in np.unique(y):  #np unique has ['no' 'yes'] when y is array with samples
        prob = sum(y == values) / float(size)
        impurity += prob * np.log2(prob)
    return -impurity


def _gini_(self, y):

    size = y.shape[0]  #amount of rows with y values
    impurity = 0
    for values in np.unique(y):    #np.unique is an array with all the unique values from the y input (wi
        prob = sum(y == values) / float(size)
        impurity += prob**2
    return 1 - impurity
```

- Depois temos uma função information gain para determinar o gain do atributo que vamos dar "split" (fazer a decisão na árvore):

```
def _information_gain(self, y, y1, y2):   #Return the information gain of making the given split.
    size = y.shape[0]
    child_inf = 0

    for index in (y1, y2):

        child_inf += self.criterion(index) * index.shape[0] / float(size)

    return self.criterion(y) - child_inf
```

- A função ***make_split*** recebe os parâmetros x e y, mas também o split_index que indica o um valor int da coluna da feature que queremos dar o split, mas também um split_value com o valor dessa feature. Retorna os novos valores de x,y e y1,y2, que serão usados depois noutras funções.

```
def _make_split(self, x, y, split_index, split_value):  #Return the s

    #Make split function receives 4 parameteres. X is a 2d array, and
    #split_index is the index from the column of the features, so if
    #split_valie represents the value from the column[i] of the featu

    idx = x[:, split_index] == split_value   #idx its an array equal
                                             #if split index is 0 = ['
                                             #when we do the == equali
                                             #for index 0, and split v

    #x[idx] = [['overcast' 'hot' 'high' 'FALSE']['overcast' 'hot' 'no
    #y[idx] = ['yes' 'yes'] reprensents the value from the target whe
    #x[idx == False] = [['rainy' 'mild' 'high' 'TRUE']
                       # ['rainy' 'mild' 'normal' 'FALSE']
                       #['sunny' 'hot' 'high' 'TRUE']
                       #['sunny' 'mild' 'high' 'FALSE']
                       # ['sunny' 'mild' 'normal' 'TRUE']
                       # ['rainy' 'mild' 'high' 'FALSE']
                       #['sunny' 'hot' 'high' 'FALSE']
                       # ['rainy' 'cool' 'normal' 'TRUE']]  represent
    #y[idx == False],['no' 'yes' 'no' 'no' 'yes' 'yes' 'no' 'no'], re

    return x[idx], y[idx], x[idx == False], y[idx == False]
```

- A função choose_split_index retorna os valores do split_index que vamos querer dividir, o valor desse mesmo index, e os novos valores dos arrays x,y,y1,y2 (usámos as função make_split para fazer essa divisão, esta função apenas vê dentro do valores que temos, qual o que tem maior gain e é esse mesmo que irá ser dividido com a função).

```
def _choose_split_index(self, x, y):  #Return the index and value of the feature to split on. Determine which feature 
                                      # value of the optimal split along with the split of the dataset.
                                      # Return None, None, None if there is no split which improves information gain
    split_index, split_value, splits = None, None, None      #split in the beginning dont have any values
    gain = 0

    for i in range(x.shape[1]):     #index from 0 to range(x.shape[1]) -> x.shape[1] represents the size of the second
        values = np.unique(x[:, i]) #values represent the unic values from feature 0 to range(x.shape[1])  example fro



        if len(values) < 1:  #if the array of the unic values is less thatn 1 we continue the for cicle
            continue

        for value in values:  #value = ['overcast' 'rainy' 'sunny'], so value[0] = overcast, value[1] = rainy, value[2

            x1, y1, x2, y2 = self._make_split(x, y, i, value)   #i represents the index of the feature to split on, va
                                                                #example i = 0 ['overcast' 'rainy' 'sunny']  and value
                                                                # in is target splits.  in weather.nominal.csv it will

            new_gain = self._information_gain(y, y1, y2)     #new_gain parameteres y1 = target values wherer rows from
                                                             #with y1 = ['yes' 'yes'] and y2 = ['no' 'yes' 'no' 'no' 'y

            if new_gain > gain:
                split_index = i             #split index equals i so we know the index of the column we pretend to sp
                split_value = value         #value of the split we are going to do -> value[0] = overcast, value[1] =
                splits = (x1, y1, x2, y2)   #splits saves all the new array with the values
                gain = new_gain             #gain is now the bigger gain that was calculated (maybe not the one from


    return split_index, split_value, splits
```

- De seguida temos a função build_tree que retorna um node, com os atributos column, value, array_header (apenas usado para dar print posteriormente), isto se não for uma leaf, se for, retorna um valor booleano node.leaf = True e a maior classe existente nessa leaf (ou yes ou no, por isso é que é uma leaf). Recursivamente chama-se novamente para que se continuem a subdividir em outras sub-árvores até que não hajam mais index para dividir.

```
def _build_tree(self, x, y): #Build the decision tree recursively

    node = treeNode()   #initialize a node

    index, value, splits = self._choose_split_index(x, y)    #uses the choose split function to obtain the index of the colum
                                                              # and  the splits that are the new array values from the make sp

    if index is None or len(np.unique(y)) == 1:    #len when only exists one value of target in the y array (so its a leaf an
        node.leaf = True                           # give the tree node leaf attribute the boolean value True so we know that
        node.classes = Counter(y)                  #Node classes  Counter({'no': 4})  Node classes  Counter({'yes': 3})
        node.name = node.classes.most_common(1)[0][0]  #node.name has the value of the most  common value in the node.classes

    else:
        X1, y1, X2, y2 = splits                    #arrays with the values calcualted in the choose split from above
        node.column = index                        #give the node.column the value from the index of the column we want to s
        node.value = value                         #give the node.value the value from the attribute we will split
        node.header = self.array_header[0][index]  #gives the node.header the value from the column (only use this in the pr
        node.left = self._build_tree(X1, y1)       #recursively do two new branches to do three (new trees)based on the new
        node.right = self._build_tree(X2, y2)

    return node
```

- Por último, temos as funções fit, que gera a árvore baseado nos valores de x e y que serão dados na main.py (deverão ser os valores x_train e y_train, os valores de treino), chamando a self.root que será a construção da nossa árvore.
- A função score calcula a exatidão para um conjunto de dados dado (são os x_test e y_test dados da divisão feita na main.py). Usa a função predict que retorna um array com os valores expectáveis.

```
def score(self, x, y):  #return accuracy of the test dates x and y from the original data

    self.x = np.array(x)
    self.y = np.array(y)
    N = self.x.shape[0]       #N = number of rows in the test file x
    y_pred = self.predict(x)     #ypred ->  ['yes' 'no' 'yes' 'yes' 'yes' 'yes' 'yes']
    accuracy = (np.sum(y == y_pred)) / N     #Array with comparison with the predict values and the train ones: y==y_pred -> [False
                                             #np.sum = 5, because there are 5 values that are 5 values in the predict that are the sa
                                             #accuracy = 5 / 7(x.shape[0] = number of rows on the data test)
    return accuracy

def fit(self, x, y):   #generates the decision tree based on the train data (75% in the normal, and 25% with prune)

   n_samples = np.size(x, 0)     #number of samples in the file (number of rows in the x array)
   n_features = np.size(x, 1)    #number of features in the file (number of columns in the x array)

   self.x = x[:n_samples, :n_features]
   self.y = y[:n_samples]

   self.root = self._build_tree(self.x, self.y)
```

1.4 – Análise de Dados e Desempenho do Programa

1.4.1 Análise do desempenho para:

- Weather.nominal.csv com random_state = 0 e usando “gini” como factor de impuridade

```
X train ->  [['sunny' 'hot' 'high' 'FALSE']
 ['rainy' 'cool' 'normal' 'TRUE']
 ['overcast' 'hot' 'normal' 'FALSE']]

Y train ->  ['no' 'no' 'yes']

---- Decision Tree ----

Is 0: outlook  equal to:
  |-> overcast:
  |    Predict ->  yes -> reached a leaf
  |-> no overcast:
  |    Predict ->  no -> reached a leaf

---------------------

X test ->  [['sunny' 'cool' 'normal' 'FALSE']
 ['overcast' 'cool' 'normal' 'TRUE']
 ['rainy' 'cool' 'normal' 'FALSE']
 ['overcast' 'mild' 'high' 'TRUE']
 ['overcast' 'hot' 'high' 'FALSE']
 ['rainy' 'mild' 'high' 'TRUE']
 ['rainy' 'mild' 'normal' 'FALSE']
 ['sunny' 'hot' 'high' 'TRUE']
 ['sunny' 'mild' 'high' 'FALSE']
 ['sunny' 'mild' 'normal' 'TRUE']
 ['rainy' 'mild' 'high' 'FALSE']]

Y test ->  ['yes' 'yes' 'yes' 'yes' 'yes' 'no' 'yes' 'no' 'no' 'yes' 'yes']

Percentagem de casos corretamente classificados 54.55%
```

- Weather.nominal.csv com random_state = 0 e usando “entropy” como factor de impuridade

```
X train ->  [['sunny' 'hot' 'high' 'FALSE']
 ['rainy' 'cool' 'normal' 'TRUE']
 ['overcast' 'hot' 'normal' 'FALSE']]

Y train ->  ['no' 'no' 'yes']

---- Decision Tree ----

Is 0: outlook  equal to:
  |-> overcast:
  |    Predict ->  yes -> reached a leaf
  |-> no overcast:
  |    Predict ->  no -> reached a leaf

---------------------

X test ->  [['sunny' 'cool' 'normal' 'FALSE']
 ['overcast' 'cool' 'normal' 'TRUE']
 ['rainy' 'cool' 'normal' 'FALSE']
 ['overcast' 'mild' 'high' 'TRUE']
 ['overcast' 'hot' 'high' 'FALSE']
 ['rainy' 'mild' 'high' 'TRUE']
 ['rainy' 'mild' 'normal' 'FALSE']
 ['sunny' 'hot' 'high' 'TRUE']
 ['sunny' 'mild' 'high' 'FALSE']
 ['sunny' 'mild' 'normal' 'TRUE']
 ['rainy' 'mild' 'high' 'FALSE']]

Y test ->  ['yes' 'yes' 'yes' 'yes' 'yes' 'no' 'yes' 'no' 'no' 'yes' 'yes']

Percentagem de casos corretamente classificados 54.55%
```

1.4 - Análise de Dados e Desempenho do Programa

(Continuação)

1.4.2 Análise do desempenho para:

- vote.csv com random_state = 0 e usando “gini” como factor de impuridade

```
-->  Vote data  <--

---- Decision Tree ----

Is 3: physician-fee-freeze  equal to:
   |-> n:
   |    Predict ->  democrat -> reached a leaf
   |-> no n:
   |    Is 2: adoption-of-the-budget-resolution  equal to:
   |     |-> n:
   |     |    Predict ->  republican -> reached a leaf
   |     |-> no n:
   |     |    Is 6: anti-satellite-test-ban  equal to:
   |     |     |-> n:
   |     |     |    Predict ->  democrat -> reached a leaf
   |     |     |-> no n:
   |     |     |    Predict ->  republican -> reached a leaf

----------------------

Percentagem de casos corretamente classificados 95.40%
```

- vote.csv com random_state = 0 e usando “entropy” como factor de impuridade

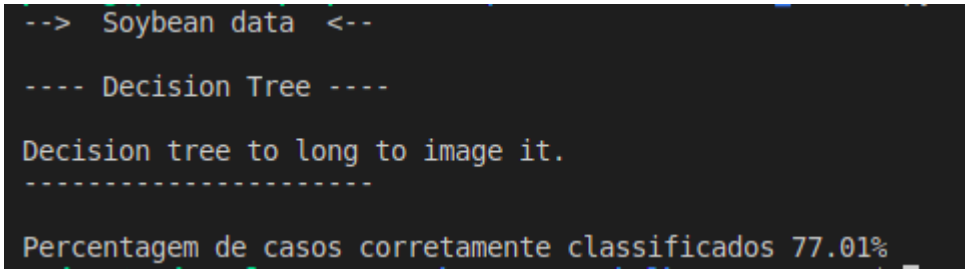
```
-->  Vote data  <--

---- Decision Tree ----

Is 3: physician-fee-freeze  equal to:
   |-> n:
   |    Predict ->  democrat -> reached a leaf
   |-> no n:
   |    Is 2: adoption-of-the-budget-resolution  equal to:
   |     |-> n:
   |     |    Predict ->  republican -> reached a leaf
   |     |-> no n:
   |     |    Is 6: anti-satellite-test-ban  equal to:
   |     |     |-> n:
   |     |     |    Predict ->  democrat -> reached a leaf
   |     |     |-> no n:
   |     |     |    Predict ->  republican -> reached a leaf

----------------------

Percentagem de casos corretamente classificados 95.40%
```

1.4 - Análise de Dados e Desempenho do Programa

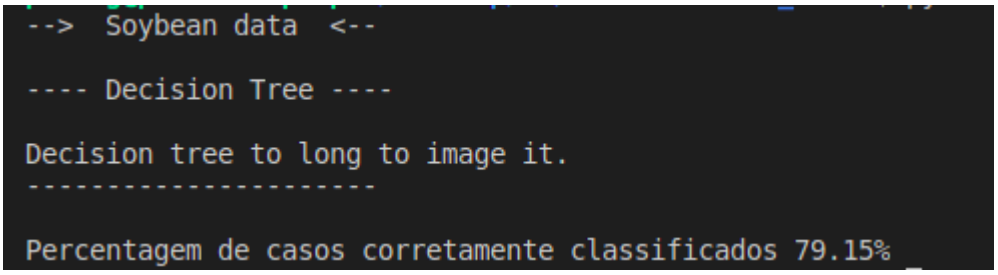
(Continuação)

1.4.3 Análise do desempenho para:

- soybean.csv com random_state = 0 e usando “gini” como factor de impuridade (não mostramos a decision tree e os valores de treino e de teste porque seriam demasiado grandes para caberem numa imagem)



- soybean.csv com random_state = 0 e usando “entropy” como factor de impuridade (não mostramos a decision tree e os valores de treino e de teste porque seriam demasiado grandes para caberem numa imagem)



1.4 - Análise de Dados e Desempenho do Programa

(Continuação)

1.4.4 Análise do desempenho para:

- contact-lenses.csv com random_state = 0 e usando "gini" como factor de impuridade

```
-->  Contact Lenses data  <--

X train ->  [['presbyopic' 'hypermetrope' 'yes' 'reduced']
 ['pre-presbyopic' 'myope' 'no' 'normal']
 ['pre-presbyopic' 'myope' 'no' 'reduced']
 ['pre-presbyopic' 'hypermetrope' 'no' 'reduced']
 ['pre-presbyopic' 'myope' 'yes' 'normal']
 ['young' 'hypermetrope' 'no' 'normal']]

Y train ->  ['none' 'soft' 'none' 'none' 'hard' 'soft']

---- Decision Tree ----

Is 3: tear-prod-rate  equal to:
  |-> normal:
  |    Is 2: astigmatism  equal to:
  |     |-> no:
  |     |    Predict ->  soft -> reached a leaf
  |     |-> no no:
  |     |    Predict ->  hard -> reached a leaf
  |-> no normal:
  |    Predict ->  none -> reached a leaf

----------------------

X test ->  [['pre-presbyopic' 'hypermetrope' 'no' 'normal']
 ['presbyopic' 'myope' 'yes' 'reduced']
 ['young' 'myope' 'yes' 'normal']
 ['pre-presbyopic' 'hypermetrope' 'yes' 'reduced']
 ['presbyopic' 'hypermetrope' 'no' 'reduced']
 ['presbyopic' 'myope' 'no' 'normal']
 ['pre-presbyopic' 'myope' 'yes' 'reduced']
 ['young' 'hypermetrope' 'no' 'reduced']
 ['young' 'myope' 'yes' 'reduced']
 ['presbyopic' 'myope' 'yes' 'normal']
 ['young' 'hypermetrope' 'yes' 'reduced']
 ['young' 'hypermetrope' 'yes' 'normal']
 ['presbyopic' 'hypermetrope' 'no' 'normal']
 ['young' 'myope' 'no' 'normal']
 ['presbyopic' 'myope' 'no' 'reduced']
 ['young' 'myope' 'no' 'reduced']
 ['pre-presbyopic' 'hypermetrope' 'yes' 'normal']
 ['presbyopic' 'hypermetrope' 'yes' 'normal']]

Y test ->  ['soft' 'none' 'hard' 'none' 'none' 'none' 'none' 'none' 'none' 'hard'
 'none' 'hard' 'soft' 'soft' 'none' 'none' 'none' 'none']

Percentagem de casos corretamente classificados 83.33%
```

1.4 - Análise de Dados e Desempenho do Programa

(Continuação)

1.4.4 Análise do desempenho para:

- contact-lenses.csv com random_state = 0 e usando “entropy” como factor de impuridade

```
-->  Contact Lenses data  <--

X train ->  [['presbyopic' 'hypermetrope' 'yes' 'reduced']
 ['pre-presbyopic' 'myope' 'no' 'normal']
 ['pre-presbyopic' 'myope' 'no' 'reduced']
 ['pre-presbyopic' 'hypermetrope' 'no' 'reduced']
 ['pre-presbyopic' 'myope' 'yes' 'normal']
 ['young' 'hypermetrope' 'no' 'normal']]

Y train ->  ['none' 'soft' 'none' 'none' 'hard' 'soft']

---- Decision Tree ----

Is 3: tear-prod-rate  equal to:
  |-> normal:
  |    Is 2: astigmatism  equal to:
  |     |-> no:
  |     |    Predict ->  soft -> reached a leaf
  |     |-> no no:
  |     |    Predict ->  hard -> reached a leaf
  |-> no normal:
  |    Predict ->  none -> reached a leaf

---------------------

X test ->  [['pre-presbyopic' 'hypermetrope' 'no' 'normal']
 ['presbyopic' 'myope' 'yes' 'reduced']
 ['young' 'myope' 'yes' 'normal']
 ['pre-presbyopic' 'hypermetrope' 'yes' 'reduced']
 ['presbyopic' 'hypermetrope' 'no' 'reduced']
 ['presbyopic' 'myope' 'no' 'normal']
 ['pre-presbyopic' 'myope' 'yes' 'reduced']
 ['young' 'hypermetrope' 'no' 'reduced']
 ['young' 'myope' 'yes' 'reduced']
 ['presbyopic' 'myope' 'yes' 'normal']
 ['young' 'hypermetrope' 'yes' 'reduced']
 ['young' 'hypermetrope' 'yes' 'normal']
 ['presbyopic' 'hypermetrope' 'no' 'normal']
 ['young' 'myope' 'no' 'normal']
 ['presbyopic' 'myope' 'no' 'reduced']
 ['young' 'myope' 'no' 'reduced']
 ['pre-presbyopic' 'hypermetrope' 'yes' 'normal']
 ['presbyopic' 'hypermetrope' 'yes' 'normal']]

Y test ->  ['soft' 'none' 'hard' 'none' 'none' 'none' 'none' 'none' 'none' 'hard'
 'none' 'hard' 'soft' 'soft' 'none' 'none' 'none' 'none']

Percentagem de casos corretamente classificados 83.33%
```

1.4 - Análise de Dados e Desempenho do Programa

(Continuação)

- Pelas imagens anteriores, conseguimos perceber que os valores da exatidão são iguais quer usando a impuridade gini quer a entropia. Apenas é diferente usando os dados do ficheiro soybean.csv derivado provavelmente de haver uma quantidade muito variada de exemplos.
- Para o ficheiro **weather.nominal.csv** podemos ver que a exatidão é de 54.55%, o que nos diz que provavelmente o modelo usado não está muito bem preparado para os dados de teste usados. É um valor intermediário quando usamos 75% dos dados de teste, o que nos diz que apenas metade destes estão corretamente classificados.
- Já pelo contrário, com o ficheiro **vote.csv**, temos um valor de exatidão de 95.40% para este conjunto de dados de teste, pelo que podemos dizer que o modelo é de facto apropriado a estes dados, pois irá classificar a maioria dos mesmos bem.
- Para o ficheiro **soybean.csv**, usando gini como fator de impuridade temos uma exatidão de 77.01%, o que é relativamente bom, e com entropia temos ainda um valor maior de 79.15%. Podemos dizer que o modelo é apropriado também aos valores de teste dados.
- Por último, para o ficheiro **contact-lenses.csv**, temos um valor de exatidão de 83.33%, conseguindo então prever que o irá também classificar corretamente a maioria dos dados de teste dados.
- Em geral, a árvore de decisão apresentada consegue classificar maioritariamente bem os valores de teste dos nossos dados, pelo que podemos dizer que é bastante apropriada para os ficheiros dados.

1.5 – Algumas considerações sobre o trabalho

- Após diversas tentativas de debug e erros no código, o grupo não conseguiu aplicar o "prunning" no algoritmo.
- Apesar de terem sido percebidas as funcionalidades e o objetivo de "prunnning" no algoritmo, não se conseguiu transportar na totalidade para o código o pedido, deixando este um pouco incompleto para uma análise total e correta dos resultados.
- O REP (Reduced error pruning) supostamente dividiria o conjunto de amostras em treinamento e validação. As amostras de treinamento são utilizadas para construir a árvore de decisão. As amostras de validação são utilizadas para verificar os erros de classificação cometidos ao utilizar sub-árvores da árvore gerada.
- Teoricamente, a poda de erro reduzido (REP) tem como objetivo resolver o problema de sobre-ajustamento, minimizando características existentes apenas no conjunto de treinamento.
- Deixou-se ainda assim o código feito e desenvolvido no trabalho para aplicação de "prunning" no ficheiro (mesmo sem este estar funcional, e usando a fórmula seguinte como inspiração para o mesmo).

```
function Prune:
    if either left or right is not a leaf:
        call Prune on that split
    if both left and right are leaf nodes:
        calculate error associated with merging two nodes
        calculate error associated without merging two nodes
        if merging results in lower error:
            merge the leaf nodes
```

1.4 – Breve Conclusão

- A realização deste trabalho permitiu-nos um maior contacto com a linguagem python, a qual era nova para nós, deixando-nos com uma melhor noção de todas as vantagens (e desvantagens) de usar uma linguagem como esta. Por esta razão, foi-nos um pouco difícil começar o trabalho derivado a isto, mas foi uma dificuldade ultrapassada.
- Conseguimos implementar a árvore de decisão pedida, não conseguindo fazer posteriormente o prune, por termos alguns erros quando tentávamos compilar.
- Posto isto, achamos que o trabalho foi bem conseguindo, e abriu-nos os horizontes para tudo aquilo que é a aprendizagem automática para grandes previsões, e como esta pode ser muito vantajosa em diversas áreas que possam existir.
- Também aprendemos a mexar com a interface Kaggle, que foi aconselhada pela professora.

1.5 – Bibliografia

- Slides no moodle
- https://en.wikipedia.org/wiki/Decision_tree
- https://scikit-learn.org/stable/modules/tree.html
- https://www.geeksforgeeks.org/decision-tree/
- Entre outros fóruns, vídeos, etc existentes