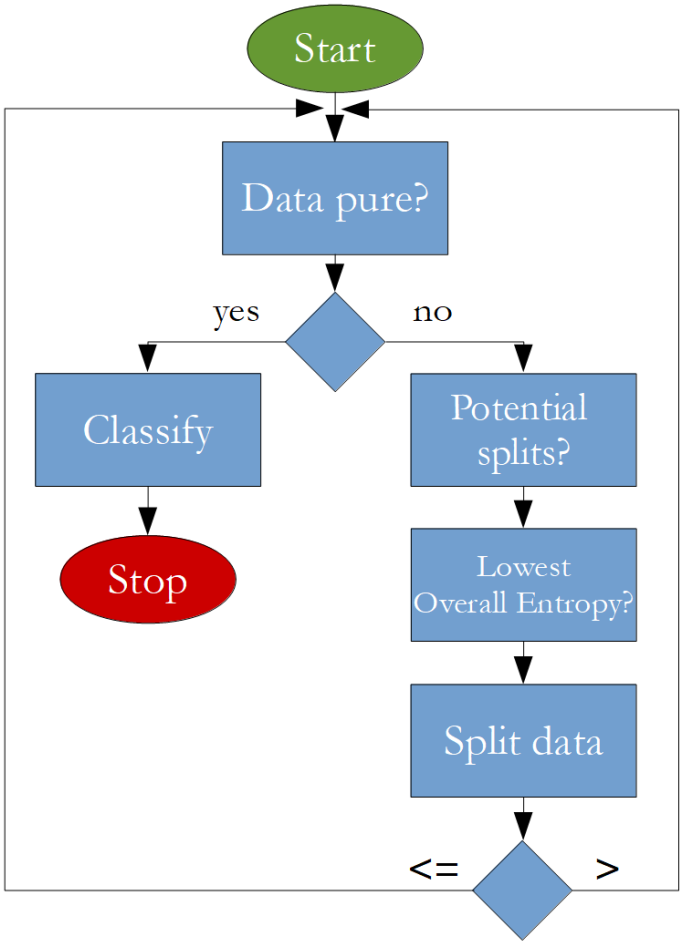
Implementação

O objetivo deste projeto é programar uma Árvore de Decisão de Classificação que pode ser utilizada mediante a seguinte API:

```
df = pd.read_csv("data.csv")

train_df, test_df = train_test_split(df, test_size=0.2) #Step 1
tree = decision_tree_algorithm(train_df, ml_task="classification")
#Step 2
accuracy = calculate_accuracy(test_df, tree) #Step 3
```

O algoritmo que será implementado seguirá o seguinte modelo:



Importação de bibliotecas

In [1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

import random
from pprint import pprint
```

In [2]:
```
%matplotlib inline
sns.set_style("darkgrid")
```

Carregamento e Preparo dos Dados

Formato do data set

- A última coluna do data frame deve conter a coluna classificatória e deve se chamar "label".
- Não pode haver valores faltantes no data frame.

In [3]:
```
df = pd.read_csv("creditcard.csv") #Carregando o dataset
df = df.rename(columns={df.columns[-1]: "label"}) #Renomeando última coluna
df = df.drop(["Time"], axis=1) #Excluindo colunas desnecessárias
```

In [5]:
```
print(df.shape)
df.head()
```

```
(284807, 30)
```

Out[5]:

	V1	V2	V3	V4	V5	V6	V7	V8	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0

5 rows × 30 columns

In [6]:
```
class_names = {0:'Not Fraud', 1:'Fraud'}
print(df.label.value_counts().rename(index = class_names))
```

```
Not Fraud    284315
Fraud           492
Name: label, dtype: int64
```

Separação de Dados para Teste

Para o *#Step 1* da API, é chamada uma função, *train_test_split* que recebe 2 parâmetros: 1) *parametro 1*: dataframe que se deseja obter a classificação. Nesse caso, *df*. 2) *parametro 2*: *test_size* é o tamanho desejado da amostra que será utilizada do dataframe para testar o algoritmo de classificação. Pode ser o número de linhas ou uma proporção. Nesse caso foi determinado *0,2* - que representa *20%* do dataframe.

A função *train_test_size* executa essa separação e retorna dois dataframes: 1) *train_df*: dataframe que será utilizado no treinamento do algoritmo. 2) *test_df*: dataframe que será utilizado para testar a acurácia do modelo de classificação gerado.

In [7]:
```
def train_test_split(df, test_size):

    if isinstance(test_size, float):
        test_size = round(test_size * len(df)) #Caso o valor de test_size pas

    indices = df.index.tolist() #Passando os indices para uma lista
    test_indices = random.sample(population=indices, k=test_size) #Pegando um
```

```
    test_df = df.loc[test_indices] #Selecionando as linhas
    train_df = df.drop(test_indices) #Retirando as linhas selecionadas do dat

    return train_df, test_df
```

In [8]:
```
random.seed(0) #Definindo uma semente para o gerador random a fim de poder-se
train_df, test_df = train_test_split(df, test_size=0.2)
```

Funções Auxiliares

No decorrer da criação desta API, houve a necessidade de modificar a biblioteca utilizada para a manipulação do dataset. Notou-se que, utilizando um array bidimensional NumPy, o algoritmo ficou cerca de 10 vezes mais rápido. Sendo assim, criou-se uma variável chamada *data* contendo os valores do dataframe *train_df*.

In [9]:
```
data = train_df.values
data[:5]
```

Out[9]:
```
array([[-1.35980713, -0.07278117,  2.53634674,  1.37815522, -0.33832077,
         0.46238778,  0.23959855,  0.0986979 ,  0.36378697,  0.09079417,
        -0.55159953, -0.61780086, -0.99138985, -0.31116935,  1.46817697,
        -0.47040053,  0.20797124,  0.02579058,  0.40399296,  0.2514121 ,
        -0.01830678,  0.27783758, -0.11047391,  0.06692807,  0.12853936,
        -0.18911484,  0.13355838, -0.02105305,  0.24496426,  0.        ],
       [ 1.19185711,  0.26615071,  0.16648011,  0.44815408,  0.06001765,
        -0.08236081, -0.07880298,  0.08510165, -0.25542513, -0.16697441,
         1.61272666,  1.06523531,  0.48909502, -0.1437723 ,  0.63555809,
         0.46391704, -0.11480466, -0.18336127, -0.14578304, -0.06908314,
        -0.22577525, -0.63867195,  0.10128802, -0.33984648,  0.1671704 ,
         0.12589453, -0.0089831 ,  0.01472417, -0.34247454,  0.        ],
       [-1.35835406, -1.34016307,  1.77320934,  0.37977959, -0.50319813,
         1.80049938,  0.79146096,  0.24767579, -1.51465432,  0.20764287,
         0.62450146,  0.06608369,  0.71729273, -0.16594592,  2.34586495,
        -2.89008319,  1.10996938, -0.12135931, -2.2618571 ,  0.52497973,
         0.24799815,  0.7716794 ,  0.90941226, -0.68928096, -0.32764183,
        -0.13909657, -0.05535279, -0.05975184,  1.16068593,  0.        ],
       [-0.96627171, -0.18522601,  1.79299334, -0.86329128, -0.01030888,
         1.24720317,  0.23760894,  0.37743587, -1.38702406, -0.05495192,
        -0.22648726,  0.17822823,  0.50775687, -0.28792375, -0.63141812,
        -1.05964725, -0.68409279,  1.965775  , -1.23262197, -0.20803778,
        -0.10830045,  0.0052736 , -0.19032052, -1.17557533,  0.64737603,
        -0.22192884,  0.06272285,  0.06145763,  0.14053425,  0.        ],
       [-1.15823309,  0.87773675,  1.54871785,  0.40303393, -0.40719338,
         0.09592146,  0.59294075, -0.27053268,  0.81773931,  0.75307443,
        -0.82284288,  0.53819555,  1.34585159, -1.11966983,  0.17512113,
        -0.45144918, -0.23703324, -0.03819479,  0.80348692,  0.40854236,
        -0.0094307 ,  0.79827849, -0.13745808,  0.14126698, -0.20600959,
         0.50229222,  0.21942223,  0.21515315, -0.07340334,  0.        ]])
```

Dados Puros?

Aqui é testada a pureza dos dados em relação a coluna de classificação.

- Caso haja apenas uma classe no dataframe passado à função, ela retorna *True*.
- Caso contrário, retorna *False*.

Essa função é importante para que se possa gerar a classificação dos dados. É aqui que é identificado quantas classes existem no dataframe e quais são elas.

In [10]:

```
def check_purity(data):

    label_column = data[:, -1] #Selecionando a coluna de classificação
    unique_classes = np.unique(label_column) #Selecionando as classes que exi

    if len(unique_classes) == 1:
        return True
    else:
        return False
```

Classificação

Essa função é responsável por identificar a classe que mais aparece no dataframe.

In [11]:

```
def classify_data(data):

    label_column = data[:, -1] #Selecionando a coluna de classificação
    unique_classes, counts_unique_classes = np.unique(label_column, return_co

    index = counts_unique_classes.argmax() #Retornando o index do elemento ma
    classification = unique_classes[index] #Retornando o valor da classe.

    return classification
```

Possíveis clusters

Essa função permite que o algoritmo seja implementado em dataframes em que não se sabe, de antemão, quantas classes existem. Isso permite que este algoritmo seja implementado para outras classificações além da binária. Ela retorna um *array* com os possíveis clusters.

In [12]:

```
def get_potential_splits(data):

    potential_splits = {}
    _, n_columns = data.shape
    for column_index in range(n_columns - 1):           #Excluindo a coluna de
        values = data[:, column_index]
        unique_values = np.unique(values)

        potential_splits[column_index] = unique_values

    return potential_splits
```

Split Data

Esta função separa o data frame em dois: um com valores que estão abaixo de um valor previamente estabelecido e outro com os valores acima.

In [13]:

```
def split_data(data, split_column, split_value):

    split_column_values = data[:, split_column]

    type_of_feature = FEATURE_TYPES[split_column]
    if type_of_feature == "continuous":
        data_below = data[split_column_values <= split_value]
        data_above = data[split_column_values >  split_value]
```

```
    # feature is categorical
    else:
        data_below = data[split_column_values == split_value]
        data_above = data[split_column_values != split_value]

    return data_below, data_above
```

Menor Entropia Absoluta?

In [14]:
```
def calculate_entropy(data):

    label_column = data[:, -1]
    _, counts = np.unique(label_column, return_counts=True)

    probabilities = counts / counts.sum()
    entropy = sum(probabilities * -np.log2(probabilities))

    return entropy
```

In [15]:
```
def calculate_overall_entropy(data_below, data_above):

    n = len(data_below) + len(data_above)
    p_data_below = len(data_below) / n
    p_data_above = len(data_above) / n

    overall_entropy =  (p_data_below * calculate_entropy(data_below)
                      + p_data_above * calculate_entropy(data_above))

    return overall_entropy
```

In [16]:
```
def determine_best_split(data, potential_splits):

    overall_entropy = 9999
    for column_index in potential_splits:
        for value in potential_splits[column_index]:
            data_below, data_above = split_data(data, split_column=column_ind
            current_overall_entropy = calculate_overall_entropy(data_below, d

            if current_overall_entropy <= overall_entropy:
                overall_entropy = current_overall_entropy
                best_split_column = column_index
                best_split_value = value

    return best_split_column, best_split_value
```

Decision Tree Algorithm

Representação da Árvore de Decisão

In [17]:
```
sub_tree = {"question": ["yes_answer",
                         "no_answer"]}
```

Determinação do Tipo de Métrica

In [18]:
```
def determine_type_of_feature(df):

    feature_types = []
    n_unique_values_treshold = 15
    for feature in df.columns:
        if feature != "label":
            unique_values = df[feature].unique()
            example_value = unique_values[0]

            if (isinstance(example_value, str)) or (len(unique_values) <= n_u
                feature_types.append("categorical")
            else:
                feature_types.append("continuous")

    return feature_types
```

Algoritmo

In [19]:
```
def decision_tree_algorithm(df, counter=0, min_samples=2, max_depth=5):

    # data preparations
    if counter == 0:
        global COLUMN_HEADERS, FEATURE_TYPES
        COLUMN_HEADERS = df.columns
        FEATURE_TYPES = determine_type_of_feature(df)
        data = df.values
    else:
        data = df


    # base cases
    if (check_purity(data)) or (len(data) < min_samples) or (counter == max_d
        classification = classify_data(data)

        return classification


    # recursive part
    else:
        counter += 1

        # helper functions
        potential_splits = get_potential_splits(data)
        split_column, split_value = determine_best_split(data, potential_spli
        data_below, data_above = split_data(data, split_column, split_value)

        # check for empty data
        if len(data_below) == 0 or len(data_above) == 0:
            classification = classify_data(data)
            return classification

        # determine question
        feature_name = COLUMN_HEADERS[split_column]
        type_of_feature = FEATURE_TYPES[split_column]
        if type_of_feature == "continuous":
            question = "{} <= {}".format(feature_name, split_value)

        # feature is categorical
        else:
            question = "{} = {}".format(feature_name, split_value)

        # instantiate sub-tree
```

```
        sub_tree = {question: []}

        # find answers (recursion)
        yes_answer = decision_tree_algorithm(data_below, counter, min_samples
        no_answer = decision_tree_algorithm(data_above, counter, min_samples,

        # If the answers are the same, then there is no point in asking the q
        # This could happen when the data is classified even though it is not
        # yet (min_samples or max_depth base case).
        if yes_answer == no_answer:
            sub_tree = yes_answer
        else:
            sub_tree[question].append(yes_answer)
            sub_tree[question].append(no_answer)

        return sub_tree
```

In [31]:

```
tree = decision_tree_algorithm(train_df, max_depth=3)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/var/folders/np/sdvcmn9166l6z2_x7dl6z9m40000gn/T/ipykernel_88758/2824870231.py
in <module>
----> 1 tree = decision_tree_algorithm(train_df, max_depth=3)

/var/folders/np/sdvcmn9166l6z2_x7dl6z9m40000gn/T/ipykernel_88758/1509369994.py
in decision_tree_algorithm(df, counter, min_samples, max_depth)
     24         # helper functions
     25         potential_splits = get_potential_splits(data)
---> 26         split_column, split_value = determine_best_split(data, potenti
al_splits)
     27         data_below, data_above = split_data(data, split_column, split_
value)
     28

/var/folders/np/sdvcmn9166l6z2_x7dl6z9m40000gn/T/ipykernel_88758/2364581489.py
in determine_best_split(data, potential_splits)
      4     for column_index in potential_splits:
      5         for value in potential_splits[column_index]:
----> 6             data_below, data_above = split_data(data, split_column=col
umn_index, split_value=value)
      7             current_overall_entropy = calculate_overall_entropy(data_b
elow, data_above)
      8

/var/folders/np/sdvcmn9166l6z2_x7dl6z9m40000gn/T/ipykernel_88758/120946100.py
 in split_data(data, split_column, split_value)
      5     type_of_feature = FEATURE_TYPES[split_column]
      6     if type_of_feature == "continuous":
----> 7         data_below = data[split_column_values <= split_value]
      8         data_above = data[split_column_values >  split_value]
      9

KeyboardInterrupt:
```

Matriz de Confusão

In [22]:

```
from sklearn.metrics import confusion_matrix
class_names = ['Não fraude', 'Fraude']
matrix = confusion_matrix(y_test, prediction)
# Create pandas dataframe
```

```
dataframe = pd.DataFrame(matrix, index=class_names, columns=class_names)
# Create heatmap
sns.heatmap(dataframe, annot=True, cbar=None, cmap="Blues", fmt = 'g')
plt.title("Matriz de Confusão"), plt.tight_layout()
plt.ylabel("Classe Real"), plt.xlabel("Classe Predita")
plt.show()
```

Calculo de Acurácia

In [23]:
```
def calculate_accuracy(df, tree):

    df["classification"] = df.apply(classify_example, axis=1, args=(tree,))
    df["classification_correct"] = df["classification"] == df["label"]

    accuracy = df["classification_correct"].mean()

    return accuracy
```

In [30]:
```
accuracy = calculate_accuracy(test_df, tree)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/np/sdvcmn9166l6z2_x7dl6z9m40000gn/T/ipykernel_88758/3156084455.py
in <module>
----> 1 accuracy = calculate_accuracy(test_df, tree)
      2 print(cl('Valor de acurácia do modelo de Árvore de Decisão: {}'.format
(accuracy), attrs = ['bold'], color='green'))

NameError: name 'tree' is not defined
```

In [29]:
```
from termcolor import colored as cl # text customization

print(cl('Valor de acurácia do modelo de Árvore de Decisão: {}'.format(accura
```

Valor de acurácia do modelo de Árvore de Decisão: 0.9992275552122467