Sandrine Neang Fall 2023

Machine Learning & Data-Mining : Research Project

Implementation and experiments

Predictions of diseases using decision trees

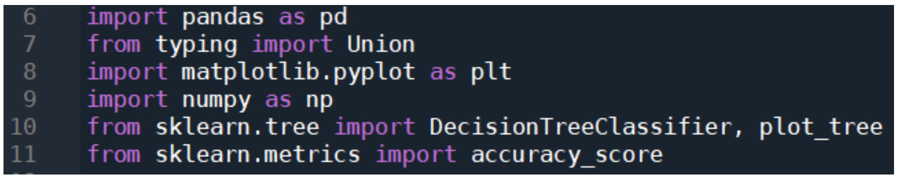
References for this project :
- Chronic kidney disease diagnosis usin decision trees algorithms (2021) (https://bmcnephrol.biomedcentral.com/articles/10.1186/s12882-021-02474-z)
- Simple Prediction of Type 2 Diabetes Mellitus via Decision Tree Modeling (2017) (https://brieflands.com/articles/ircrj-10657.pdf)
- Decision tree model in the diagnosis of breast cancer (2017) (https://ieeexplore.ieee.org/document/8789297)
- Early Prediction of Heart Disease Using Decision Tree Algorithm (2017) (https://www.researchgate.net/profile/Safish-Mary/publication/315023624_Early_Prediction_of_Heart_Disease_Using_Decision_Tree_Algorithm/links/58c84b57aca2723ab16eba60/Early-Prediction-of-Heart-Disease-Using-Decision-Tree-Algorithm.pdf)

I will get help with the code of the first assignment on decision trees for my implementation.

I am re-implementing what I coded with another dataset. I used the dataset « Breast cancer Wisconsin » : https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data

Let's highlight the key components of the code and their functionalities:

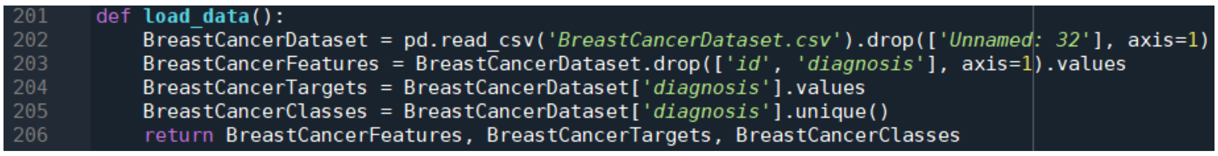
- Libraries :



```python
import pandas as pd
from typing import Union
import matplotlib.pyplot as plt
import numpy as np
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score
```

- Data Loading and Preprocessing:

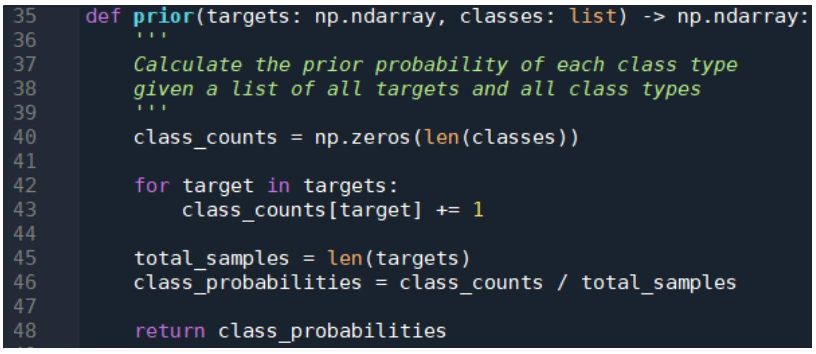
The load_data function loads the Breast Cancer dataset from a CSV file. This function drops unnecessary columns from the dataset and features and targets are extracted.



```python
def load_data():
    BreastCancerDataset = pd.read_csv('BreastCancerDataset.csv').drop(['Unnamed: 32'], axis=1)
    BreastCancerFeatures = BreastCancerDataset.drop(['id', 'diagnosis'], axis=1).values
    BreastCancerTargets = BreastCancerDataset['diagnosis'].values
    BreastCancerClasses = BreastCancerDataset['diagnosis'].unique()
    return BreastCancerFeatures, BreastCancerTargets, BreastCancerClasses
```

- Prior probability function

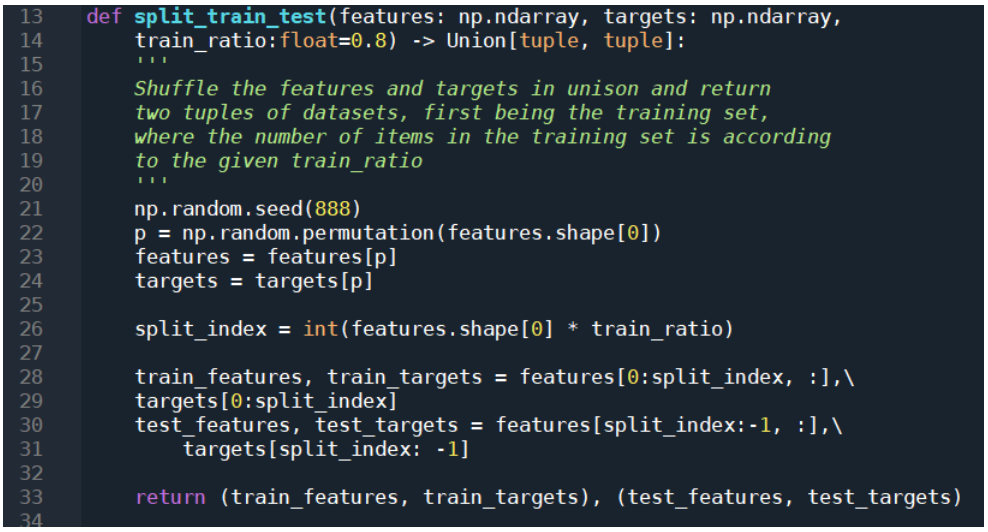
The prior function calculates the prior probability of each class type in a classification problem. It represents the distribution of classes in the dataset before considering any features.



```python
def prior(targets: np.ndarray, classes: list) -> np.ndarray:
    '''
    Calculate the prior probability of each class type
    given a list of all targets and all class types
    '''
    class_counts = np.zeros(len(classes))

    for target in targets:
        class_counts[target] += 1

    total_samples = len(targets)
    class_probabilities = class_counts / total_samples

    return class_probabilities
```

- Splitting Data:

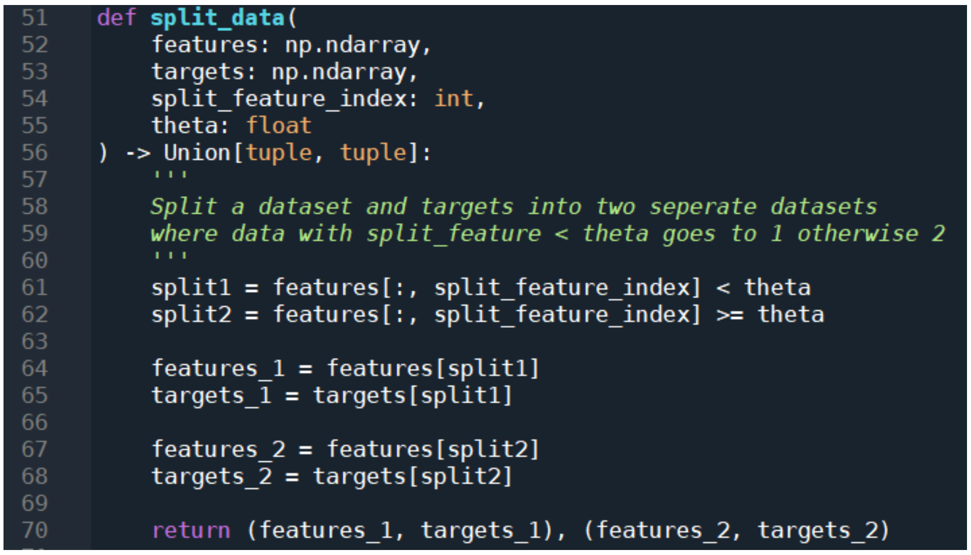
The split_train_test function shuffles and splits the dataset into training and testing sets based on a specified ratio. I am using the same split_train_test function from the tools of the assignment.



```python
def split_train_test(features: np.ndarray, targets: np.ndarray,
    train_ratio:float=0.8) -> Union[tuple, tuple]:
    '''
    Shuffle the features and targets in unison and return
    two tuples of datasets, first being the training set,
    where the number of items in the training set is according
    to the given train_ratio
    '''
    np.random.seed(888)
    p = np.random.permutation(features.shape[0])
    features = features[p]
    targets = targets[p]

    split_index = int(features.shape[0] * train_ratio)

    train_features, train_targets = features[0:split_index, :],\
    targets[0:split_index]
    test_features, test_targets = features[split_index:-1, :],\
        targets[split_index: -1]

    return (train_features, train_targets), (test_features, test_targets)
```

- Decision Tree Functions:
➢ split_data Function:

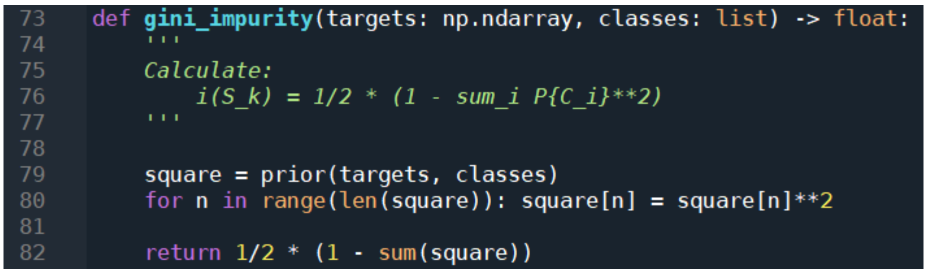
This function splits the dataset and targets into two separate datasets based on a given feature and threshold.



```python
def split_data(
    features: np.ndarray,
    targets: np.ndarray,
    split_feature_index: int,
    theta: float
) -> Union[tuple, tuple]:
    '''
    Split a dataset and targets into two seperate datasets
    where data with split_feature < theta goes to 1 otherwise 2
    '''
    split1 = features[:, split_feature_index] < theta
    split2 = features[:, split_feature_index] >= theta

    features_1 = features[split1]
    targets_1 = targets[split1]

    features_2 = features[split2]
    targets_2 = targets[split2]

    return (features_1, targets_1), (features_2, targets_2)
```

- gini_impurity Function:

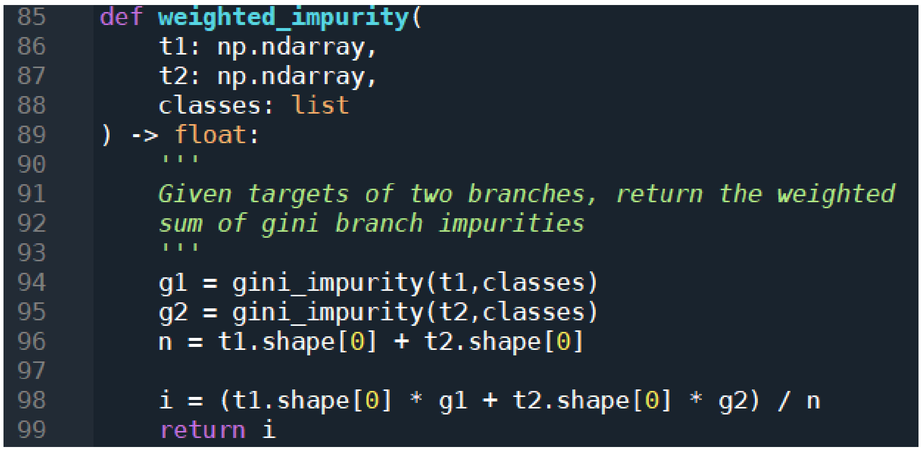
This function calculates the Gini impurity for a set of targets.



```python
def gini_impurity(targets: np.ndarray, classes: list) -> float:
    '''
    Calculate:
        i(S_k) = 1/2 * (1 - sum_i P{C_i}**2)
    '''

    square = prior(targets, classes)
    for n in range(len(square)): square[n] = square[n]**2

    return 1/2 * (1 - sum(square))
```

- weighted_impurity Function:

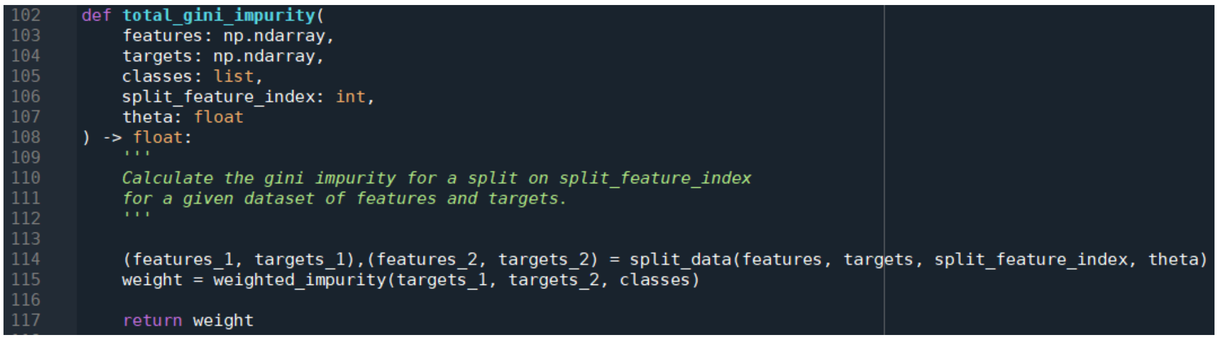
This function calculates the weighted sum of Gini impurities for two branches.



```python
def weighted_impurity(
    t1: np.ndarray,
    t2: np.ndarray,
    classes: list
) -> float:
    '''
    Given targets of two branches, return the weighted
    sum of gini branch impurities
    '''
    g1 = gini_impurity(t1,classes)
    g2 = gini_impurity(t2,classes)
    n = t1.shape[0] + t2.shape[0]

    i = (t1.shape[0] * g1 + t2.shape[0] * g2) / n
    return i
```

- total_gini_impurity Function:

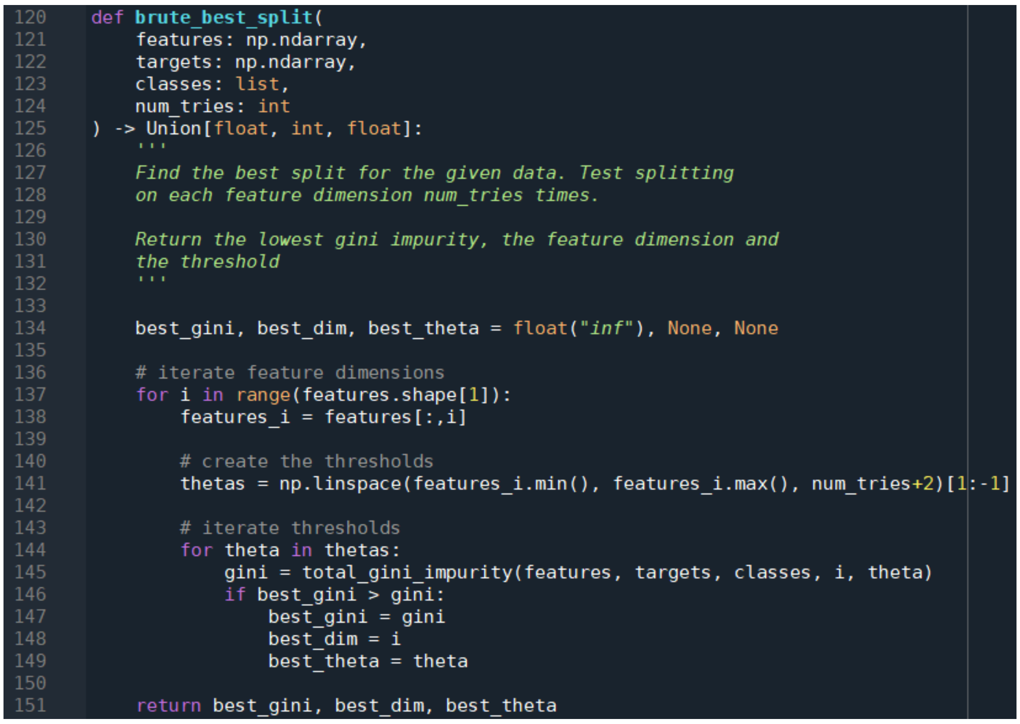
This function calculates the total Gini impurity which returns the weighted impurity given the dataset and threshold to split on.



```python
def total_gini_impurity(
    features: np.ndarray,
    targets: np.ndarray,
    classes: list,
    split_feature_index: int,
    theta: float
) -> float:
    '''
    Calculate the gini impurity for a split on split_feature_index
    for a given dataset of features and targets.
    '''

    (features_1, targets_1),(features_2, targets_2) = split_data(features, targets, split_feature_index, theta)
    weight = weighted_impurity(targets_1, targets_2, classes)

    return weight
```

- brute_best_split Function:

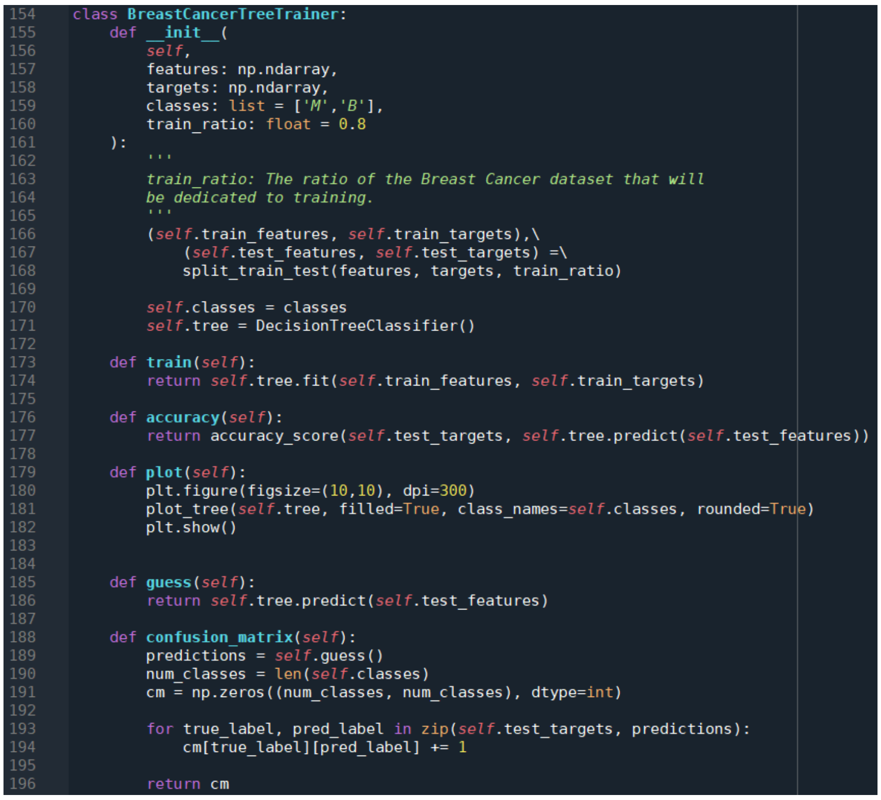
This function finds the best split for the given data by iterating over feature dimensions and thresholds.



```python
def brute_best_split(
    features: np.ndarray,
    targets: np.ndarray,
    classes: list,
    num_tries: int
) -> Union[float, int, float]:
    '''
    Find the best split for the given data. Test splitting
    on each feature dimension num_tries times.

    Return the lowest gini impurity, the feature dimension and
    the threshold
    '''

    best_gini, best_dim, best_theta = float("inf"), None, None

    # iterate feature dimensions
    for i in range(features.shape[1]):
        features_i = features[:,i]

        # create the thresholds
        thetas = np.linspace(features_i.min(), features_i.max(), num_tries+2)[1:-1]

        # iterate thresholds
        for theta in thetas:
            gini = total_gini_impurity(features, targets, classes, i, theta)
            if best_gini > gini:
                best_gini = gini
                best_dim = i
                best_theta = theta

    return best_gini, best_dim, best_theta
```

- Decision Tree Trainer Class:

This class handles the training and evaluation of the Decision Tree model of the breast cancer winsconsin dataset. It utilizes the DecisionTreeClassifier from scikit-learn and provides methods for training, accuracy calculation, plotting, making predictions, and generating a confusion matrix.

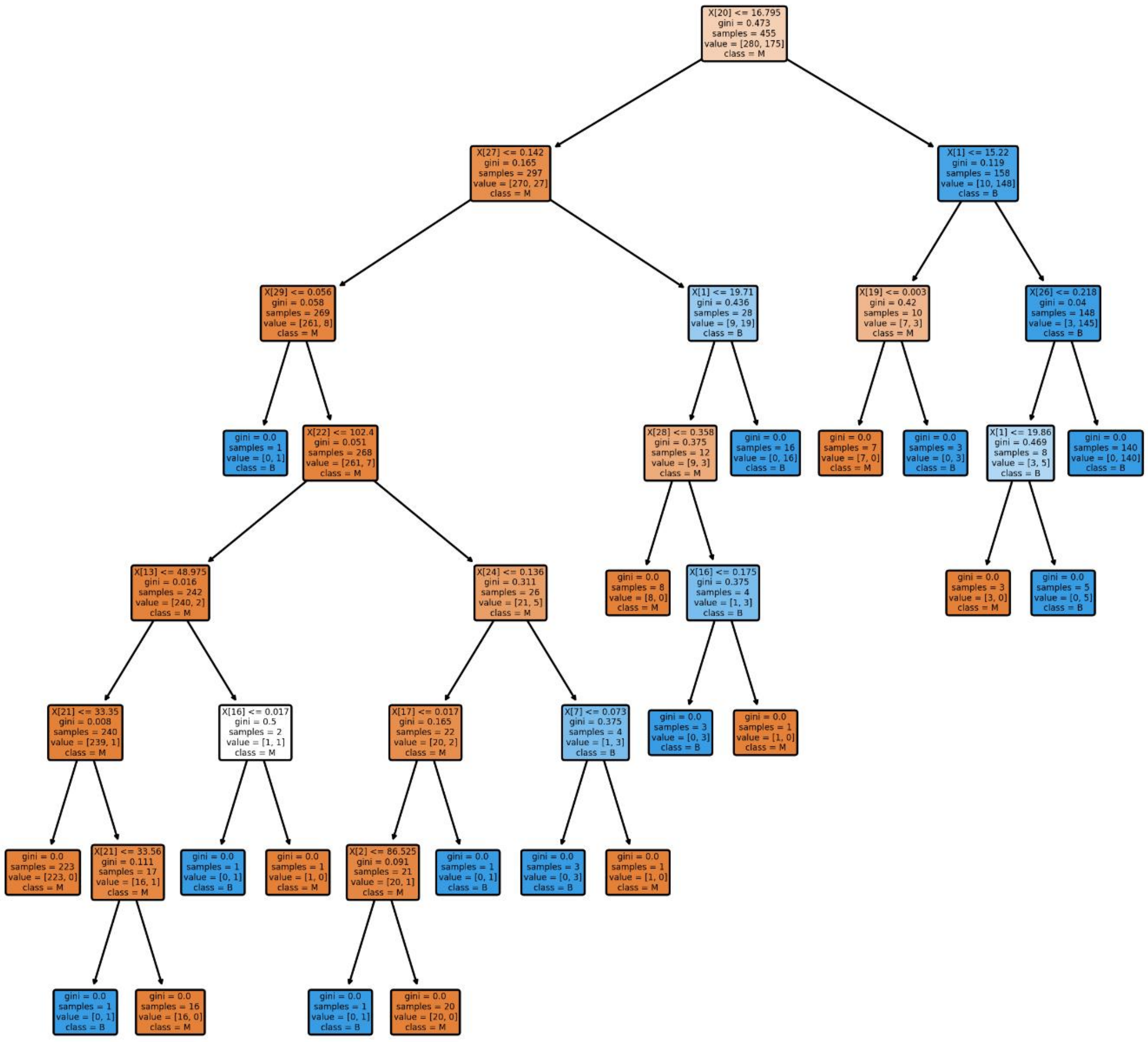


```python
class BreastCancerTreeTrainer:
    def __init__(
        self,
        features: np.ndarray,
        targets: np.ndarray,
        classes: list = ['M','B'],
        train_ratio: float = 0.8
    ):
        '''
        train_ratio: The ratio of the Breast Cancer dataset that will
        be dedicated to training.
        '''
        (self.train_features, self.train_targets),\
            (self.test_features, self.test_targets) =\
            split_train_test(features, targets, train_ratio)

        self.classes = classes
        self.tree = DecisionTreeClassifier()

    def train(self):
        return self.tree.fit(self.train_features, self.train_targets)

    def accuracy(self):
        return accuracy_score(self.test_targets, self.tree.predict(self.test_features))

    def plot(self):
        plt.figure(figsize=(10,10), dpi=300)
        plot_tree(self.tree, filled=True, class_names=self.classes, rounded=True)
        plt.show()


    def guess(self):
        return self.tree.predict(self.test_features)

    def confusion_matrix(self):
        predictions = self.guess()
        num_classes = len(self.classes)
        cm = np.zeros((num_classes, num_classes), dtype=int)

        for true_label, pred_label in zip(self.test_targets, predictions):
            cm[true_label][pred_label] += 1

        return cm
```

- Main Script:

In the main, I load data using the load_data function. I instantiate a BreastCancerTreeTrainer object and train a Decision Tree model in order to do prediction and classify if the patient's tumor is benigne or malignant. If it is malignant, it means that the patient has breast cancer.

For the moment, I have this decision tree while taking into consideration all the features of the Breast Cancer Dataset:


X[20] <= 16.795 gini = 0.473 samples = 455 value = [280, 175] class = M
X[27] <= 0.142 gini = 0.165 samples = 297 value = [270, 27] class = M
X[1] <= 15.22 gini = 0.119 samples = 158 value = [10, 148] class = B
X[29] <= 0.056 gini = 0.058 samples = 269 value = [261, 8] class = M
X[1] <= 19.71 gini = 0.436 samples = 28 value = [9, 19] class = B
X[19] <= 0.003 gini = 0.42 samples = 10 value = [7, 3] class = M
X[26] <= 0.218 gini = 0.04 samples = 148 value = [3, 145] class = B
gini = 0.0 samples = 1 value = [0, 1] class = B
X[22] <= 102.4 gini = 0.051 samples = 268 value = [261, 7] class = M
X[28] <= 0.358 gini = 0.375 samples = 12 value = [9, 3] class = M
gini = 0.0 samples = 16 value = [0, 16] class = B
gini = 0.0 samples = 7 value = [7, 0] class = M
gini = 0.0 samples = 3 value = [0, 3] class = B
X[1] <= 19.86 gini = 0.469 samples = 8 value = [3, 5] class = B
gini = 0.0 samples = 140 value = [0, 140] class = B
X[13] <= 48.975 gini = 0.016 samples = 242 value = [240, 2] class = M
X[24] <= 0.136 gini = 0.311 samples = 26 value = [21, 5] class = M
gini = 0.0 samples = 8 value = [8, 0] class = M
X[16] <= 0.175 gini = 0.375 samples = 4 value = [1, 3] class = B
gini = 0.0 samples = 3 value = [3, 0] class = M
gini = 0.0 samples = 5 value = [0, 5] class = B
X[21] <= 33.35 gini = 0.008 samples = 240 value = [239, 1] class = M
X[16] <= 0.017 gini = 0.5 samples = 2 value = [1, 1] class = M
X[17] <= 0.017 gini = 0.165 samples = 22 value = [20, 2] class = M
X[7] <= 0.073 gini = 0.375 samples = 4 value = [1, 3] class = B
gini = 0.0 samples = 3 value = [0, 3] class = B
gini = 0.0 samples = 1 value = [1, 0] class = M
gini = 0.0 samples = 223 value = [223, 0] class = M
X[21] <= 33.56 gini = 0.111 samples = 17 value = [16, 1] class = M
gini = 0.0 samples = 1 value = [0, 1] class = B
gini = 0.0 samples = 1 value = [1, 0] class = M
X[2] <= 86.525 gini = 0.091 samples = 21 value = [20, 1] class = M
gini = 0.0 samples = 1 value = [0, 1] class = B
gini = 0.0 samples = 3 value = [0, 3] class = B
gini = 0.0 samples = 1 value = [1, 0] class = M
gini = 0.0 samples = 1 value = [0, 1] class = B
gini = 0.0 samples = 16 value = [16, 0] class = M
gini = 0.0 samples = 1 value = [0, 1] class = B
gini = 0.0 samples = 20 value = [20, 0] class = M

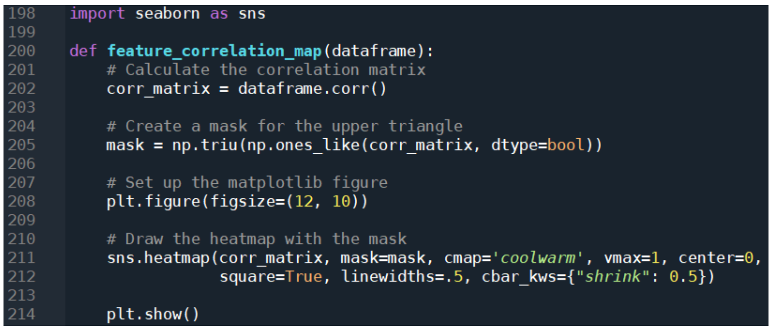
The accuracy achieved by my implementation is 95.58%, slightly surpassing the 94.3% reported in the referenced article (https://ieeexplore.ieee.org/document/9442043/). Despite this high accuracy, it's crucial to acknowledge the presence of bugs in my code, potentially leading to inaccurate results.

One identified issue lies in the prior function, which requires modification. The breast cancer dataset's classes, denoted as 'B' for benign and 'M' for malignant, are of string type. This contrasts with the integer type classes in the iris dataset. Consequently, the prior function needs adjustments to handle string-type classes.

Additionally, the confusion matrix is currently non-functional. Addressing this issue is essential for a comprehensive evaluation of the model's performance.

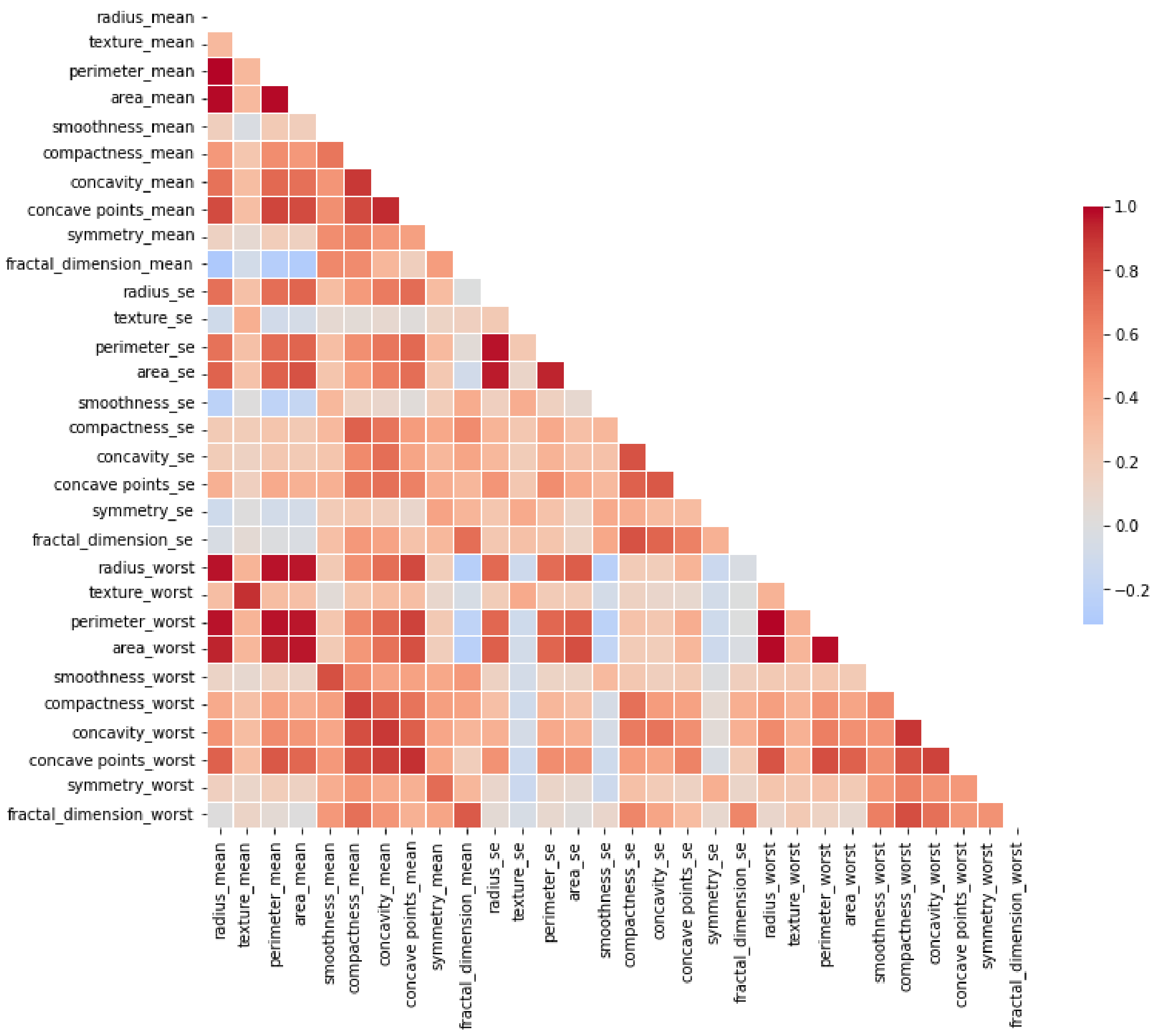
Aside of this first plot, I tried to see what features of the dataset have the biggest impact on results in order to reduce the number of features (more than 30). By retaining only features that are not correlated, the model can improve in interpretability.

This function helped me in obtaining a correlation map of the Breast Cancer dataset's features.

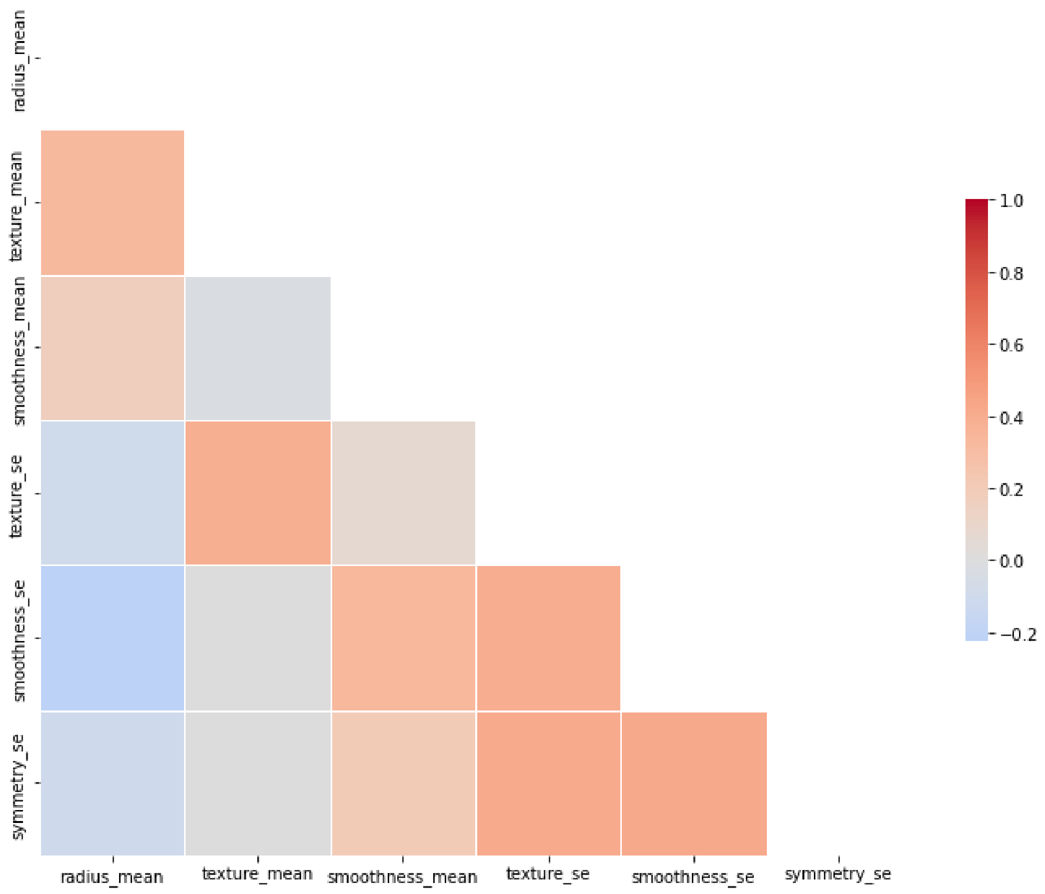


```python
import seaborn as sns

def feature_correlation_map(dataframe):
    # Calculate the correlation matrix
    corr_matrix = dataframe.corr()

    # Create a mask for the upper triangle
    mask = np.triu(np.ones_like(corr_matrix, dtype=bool))

    # Set up the matplotlib figure
    plt.figure(figsize=(12, 10))

    # Draw the heatmap with the mask
    sns.heatmap(corr_matrix, mask=mask, cmap='coolwarm', vmax=1, center=0,
                square=True, linewidths=.5, cbar_kws={"shrink": 0.5})

    plt.show()
```

I applied the feature_correlation_map to my Breast Cancer Dataset and I obtained this map :

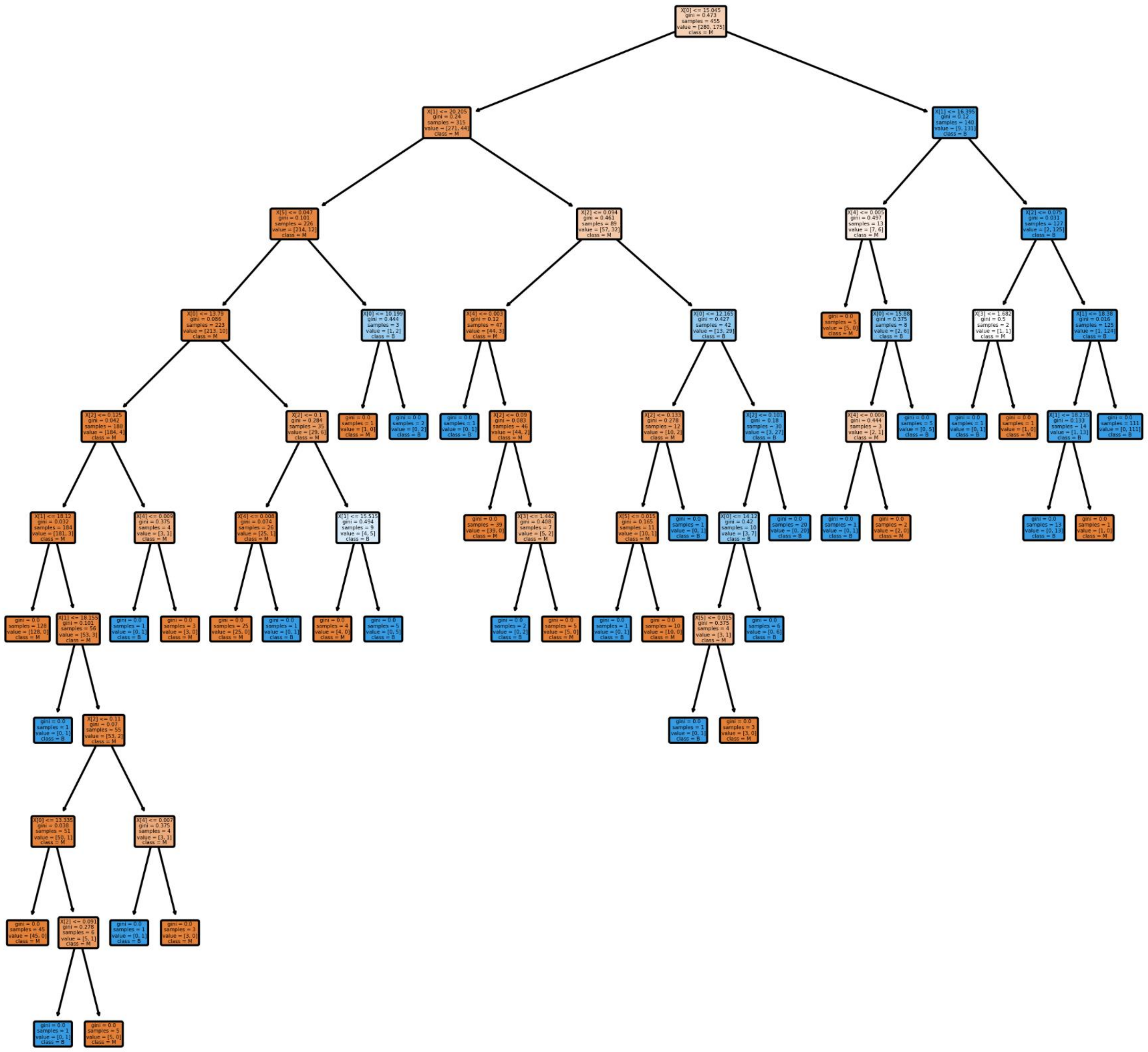


After retaining only the features that has a correlation higher than the threshold of 0.5, we obtain a new correlation map :



Radius_mean, texture_mean, smoothness_mean, texture_se, smoothness_se and symmetry_se are the features which are not correlated.

I tried to plot another decision tree but with the features « filtered » :



The accuracy of this decision tree is 88.4%

The next step of my implementation will be to correct bugs that I mention earlier and to understand more deeply how to interpret each decision tree’s plot.