Lecture_1

March 27, 2023

1 Lecture 1

In this notebook, we first illustrate the basic machine learning workflow in Python (using the well-known Iris dataset), and then show an implementation of the decision tree learning algorithm for classification.

To run these examples, you need to have a Python installation that includes scikit-learn (for machine learning), matplotlib (for plotting), and pandas (for loading the dataset). If you use the Anaconda distribution, you'll have these libraries out of the box.

You'll also need to install graphviz to make the tree drawing work. If you use Anaconda, you need to install graphviz and python-graphviz.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

%config InlineBackend.figure_format = 'svg'
plt.style.use('seaborn')
%matplotlib inline
```

```
/tmp/ipykernel_12862/2960785217.py:6: MatplotlibDeprecationWarning: The seaborn
styles shipped by Matplotlib are deprecated since 3.6, as they no longer
correspond to the styles shipped by seaborn. However, they will remain available
as 'seaborn-v0_8-<style>'. Alternatively, directly use the seaborn API instead.
  plt.style.use('seaborn')
```

1.0.1 Loading the iris data

You can download the dataset as a CSV file here. We first use pandas to load the CSV file that stores the Iris data. Please note that you need to change the path to where you stored the csv file.

We shuffle the dataset and split it into an input part `X` and an output part `Y`. In this case, we want to predict the type of iris: *setosa*, *versicolor*, or *virginica*.

[2]:
```
import pandas as pd

data = pd.read_csv('iris.csv')
data_shuffled = data.sample(frac=1.0, random_state=0)
X = data_shuffled.drop('species', axis=1)
```

```
Y = data_shuffled['species']
```

Let's take a peek at the data. As you can see, there are four numerical column that we use as input, and then the discrete output column representing the species, which we'll use as the output.

[3]:
```
data_shuffled.head()
```

[3]:

	sepal_length	sepal_width	petal_length	petal_width	species
114	5.8	2.8	5.1	2.4	virginica
62	6.0	2.2	4.0	1.0	versicolor
33	5.5	4.2	1.4	0.2	setosa
107	7.3	2.9	6.3	1.8	virginica
7	5.0	3.4	1.5	0.2	setosa

The example above uses a pandas `DataFrame` to store the data. If you want to convert into a raw NumPy matrix, it can be done easily by calling `to_numpy`.

[4]:
```
X.to_numpy()[:5]
```

[4]:
```
array([[5.8, 2.8, 5.1, 2.4],
       [6. , 2.2, 4. , 1. ],
       [5.5, 4.2, 1.4, 0.2],
       [7.3, 2.9, 6.3, 1.8],
       [5. , 3.4, 1.5, 0.2]])
```

1.0.2 Basic workflow

Let's have some basic machine learning examples.

We start by splitting the data into a training and test part. 40% of the data will be used for testing and the rest for training. We use the utility function `train_test_split` from the scikit-learn library.

The train/test split is done randomly. The parameter `random_state` is used to set the random seed to a constant value, so that our results are reproducible.

[5]:
```
from sklearn.model_selection import train_test_split

Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.4,␣
 ↪random_state=0)
```

We import a scikit-learn class representing decision tree classifiers.

We create a decision tree classifier and train it on the training set (`fit`).

[6]:
```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier()
clf.fit(Xtrain, Ytrain);
```

Let's see what the classifier says about one particular instance. We take a look at the first instance in the test set:

[7]:
```
Xtest.head(1)
```

[7]:
```
     sepal_length  sepal_width  petal_length  petal_width
118           7.7          2.6           6.9          2.3
```

We comput the classifier's prediction for this instance, and in this case the classifier thinks that this is a *virginica*.

[8]:
```
one_instance = Xtest.head(1)

clf.predict(one_instance)
```

[8]:
```
array(['virginica'], dtype=object)
```

By looking at the output part of the test set, we can verify that the classifier was correct in this case.

[9]:
```
Ytest.head(1)
```

[9]:
```
118    virginica
Name: species, dtype: object
```

We can compute predictions for all instances in the test set. Note that the `predict` method for scikit-learn predictors normally expects a *collection* of instances, not a single instance.

[10]:
```
all_predictions = clf.predict(Xtest)

all_predictions
```

[10]:
```
array(['virginica', 'virginica', 'versicolor', 'setosa', 'versicolor',
       'setosa', 'virginica', 'virginica', 'virginica', 'versicolor',
       'setosa', 'virginica', 'setosa', 'virginica', 'versicolor',
       'virginica', 'setosa', 'virginica', 'versicolor', 'versicolor',
       'virginica', 'versicolor', 'virginica', 'virginica', 'virginica',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'virginica',
       'virginica', 'versicolor', 'versicolor', 'virginica', 'virginica',
       'setosa', 'virginica', 'setosa', 'virginica', 'versicolor',
       'setosa', 'versicolor', 'setosa', 'virginica', 'versicolor',
       'virginica', 'setosa', 'versicolor', 'versicolor', 'setosa',
       'versicolor', 'versicolor', 'virginica', 'setosa', 'virginica',
       'setosa', 'virginica', 'versicolor', 'versicolor'], dtype=object)
```

How good are our predictions? We compute the *accuracy* of our classifier for this test set. The accuracy is defined as the proportion of right answers. (See here for some other ways to evaluate classifiers.)

The iris dataset is quite easy, and the accuracy is quite high for our classifier.

[11]:
```
from sklearn.metrics import accuracy_score

accuracy_score(Ytest, all_predictions)
```

[11]:
```
0.8833333333333333
```

Alternatively, we can call the method `score` for our classifier. This method will predict on the test set and then evaluate the predictions using the accuracy.

[12]:
```
clf.score(Xtest, Ytest)
```

[12]:
```
0.8833333333333333
```

Normally, we'll carry out several evaluations while we are selecting the best model. This will be done using a separate *validation set*, like a second test set, or using *cross-validation*.

In scikit-learn, there are a couple of ways that we can do cross-validation. The simplest way is to call `cross_val_score`, which for each cross-validation fold will call the method `score` mentioned above.

[13]:
```
from sklearn.model_selection import cross_val_score

cross_val_score(clf, Xtrain, Ytrain, cv=5)
```

[13]:
```
array([0.94444444, 0.94444444, 0.94444444, 1.        , 1.        ])
```

Alternatively, you can call `cross_validate`, where you can specify what kind of metric to use. The output of this function is also a bit more detailed.

[14]:
```
from sklearn.model_selection import cross_validate

cross_validate(clf, Xtrain, Ytrain, cv=5, scoring='accuracy')
```

[14]:
```
{'fit_time': array([0.00537682, 0.00477219, 0.00388432, 0.00552106,
0.00517249]),
 'score_time': array([0.0035615 , 0.00277257, 0.00374126, 0.00408792,
0.00389504]),
 'test_score': array([1.        , 0.94444444, 0.94444444, 1.        , 1.
])}
```

Finally, let's evaluate some other classifiers in addition to the decision trees. We first consider linear support vector classifiers.

[15]:
```
from sklearn.svm import LinearSVC

clf2 = LinearSVC(max_iter=10000)
cross_validate(clf2, Xtrain, Ytrain, cv=5, scoring='accuracy')
```

[15]:
```
{'fit_time': array([0.01838899, 0.01777387, 0.01408386, 0.01211381,
0.01553512]),
 'score_time': array([0.00314093, 0.00347018, 0.00208282, 0.00190234,
0.00232649]),
 'test_score': array([1.        , 0.94444444, 0.94444444, 0.94444444, 1.
])}
```

It is useful to compare classifiers to a *baseline*, such as a trivial majority-class classifier.

[16]:
```
from sklearn.dummy import DummyClassifier

majority_baseline = DummyClassifier(strategy='most_frequent')
cross_validate(majority_baseline, Xtrain, Ytrain, cv=5, scoring='accuracy')
```

[16]:
```
{'fit_time': array([0.00246549, 0.00184083, 0.00177288, 0.00206447, 0.0021174
]),
 'score_time': array([0.00144744, 0.00185871, 0.0011332 , 0.00169063,
0.00134254]),
 'test_score': array([0.33333333, 0.33333333, 0.38888889, 0.38888889,
0.33333333])}
```

1.0.3 Illustrating the iris example

We'll provide some illustrations of how the various types of classifiers work by drawing the *decision boundaries* for a two-dimensional dataset.

Since the iris dataset has four dimensions, we select two columns (petal length and width) so that we can plot the data in a two-dimensional scatterplot.

[17]:
```
X2 = X[['petal_length', 'petal_width']]
Y_encoded = Y.replace({'setosa':0, 'versicolor':1, 'virginica':2})

plt.figure(figsize=(5,5))
plt.scatter(X2.petal_length, X2.petal_width, c=Y_encoded, cmap='tab10');
```

We define a utility function that makes a scatterplot of a dataset and also draws the decision boundary. Here, we're using a bit of NumPy and matplotlib tricks. Don't worry about the details.

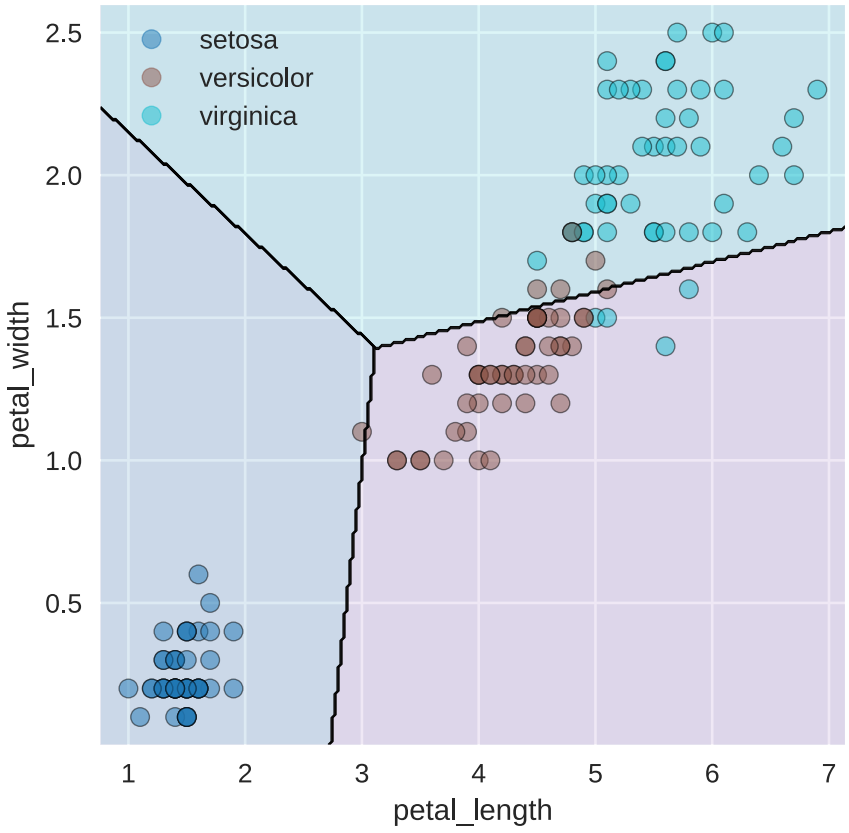
[18]:
```
def plot_boundary(clf, X, Y, cmap='tab10', names=None):

    if isinstance(X, pd.DataFrame):
        if not names:
            names = list(X.columns)
        X = X.to_numpy()

    x_min, x_max = X[:,0].min(), X[:,0].max()
    y_min, y_max = X[:,1].min(), X[:,1].max()

    x_off = (x_max-x_min)/25
    y_off = (y_max-y_min)/25
    x_min -= x_off
    x_max += x_off
    y_min -= y_off
    y_max += y_off

    xs = np.linspace(x_min, x_max, 250)
```

```
    ys = np.linspace(y_min, y_max, 250)

    xx, yy = np.meshgrid(xs, ys)

    lenc = {c:i for i, c in enumerate(clf.classes_)}
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = np.array([lenc[z] for z in Z])
    Z = Z.reshape(xx.shape)
    Yenc = [lenc[y] for y in Y]
    plt.figure(figsize=(5,5))
    plt.contourf(xx, yy, Z, cmap=cmap, alpha=0.15)
    plt.contour(xx, yy, Z, colors='k', linewidths=0.2)

    sc = plt.scatter(X[:,0], X[:,1], c=Yenc, cmap=cmap, alpha=0.5,␣
↪edgecolors='k', linewidths=0.5);

    plt.legend(handles=sc.legend_elements()[0], labels=list(clf.classes_))

    if names:
        plt.xlabel(names[0])
        plt.ylabel(names[1])
```

Now we can plot some decision boundaries. Here, we see the how it looks for a linear classifier:

[19]:
```
from sklearn.svm import LinearSVC

cls = LinearSVC()
cls.fit(X2, Y)
plot_boundary(cls, X2, Y)
```

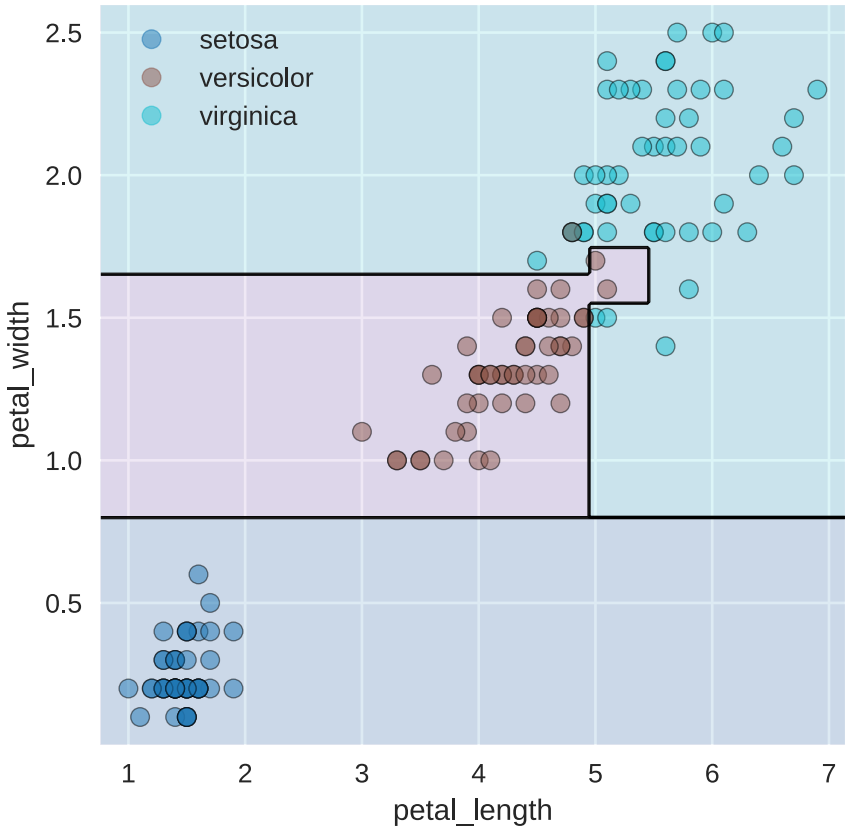
```
/home/david/.local/lib/python3.10/site-packages/sklearn/base.py:420:
UserWarning: X does not have valid feature names, but LinearSVC was fitted with
feature names
  warnings.warn(
```



And for a decision tree, the decision boundary has the characteristic shape with right angles.

[20]:
```
cls = DecisionTreeClassifier()
cls.fit(X2, Y)
plot_boundary(cls, X2, Y)
```

```
/home/david/.local/lib/python3.10/site-packages/sklearn/base.py:420:
UserWarning: X does not have valid feature names, but DecisionTreeClassifier was
fitted with feature names
  warnings.warn(
```



1.0.4 Decision tree implementation

In the following section, we'll implement a learning algorithm for decision tree classifiers.

To represent a decision tree, we define the two components of decision trees: leaves and branches. A *leaf* corresponds to the base case of a recursion. It is a "dummy" tree that always returns a constant value.

The code that is used to draw the trees can be ignored if you want.

[21]:
```
class DecisionTreeLeaf:

    def __init__(self, value):
        self.value = value

    # This method computes the prediction for this leaf node. This will just␣
↪return a constant value.
    def predict(self, x):
        return self.value

    # Utility function to draw a tree visually using graphviz.
```

```
    def draw_tree(self, graph, node_counter, names):
        node_id = str(node_counter)
        val_str = f'{self.value:.4g}' if isinstance(self.value, float) else␣
↪str(self.value)
        graph.node(node_id, val_str, style='filled')
        return node_counter+1, node_id

    def __eq__(self, other):
        if isinstance(other, DecisionTreeLeaf):
            return self.value == other.value
        else:
            return False
```

A *branch* will look at one feature, and will select a subtree depending on the value of the feature. That subtree will then be called recursively to compute the prediction.

This implementation assumes that the feature is numerical. Depending on whether the feature is or isn't greater than a threshold, the "high" or "low" subtree will be selected.

[22]:
```
class DecisionTreeBranch:

    def __init__(self, feature, threshold, low_subtree, high_subtree):
        self.feature = feature
        self.threshold = threshold
        self.low_subtree = low_subtree
        self.high_subtree = high_subtree

    # For a branch node, we compute the prediction by first considering the␣
↪feature, and then
    # calling the upper or lower subtree, depending on whether the feature is␣
↪or isn't greater
    # than the threshold.
    def predict(self, x):
        if x[self.feature] <= self.threshold:
            return self.low_subtree.predict(x)
        else:
            return self.high_subtree.predict(x)

    # Utility function to draw a tree visually using graphviz.
    def draw_tree(self, graph, node_counter, names):
        node_counter, low_id = self.low_subtree.draw_tree(graph, node_counter,␣
↪names)
        node_counter, high_id = self.high_subtree.draw_tree(graph,␣
↪node_counter, names)
        node_id = str(node_counter)
        fname = f'F{self.feature}' if names is None else names[self.feature]
        lbl = f'{fname} > {self.threshold:.4g}?'
```

```
        graph.node(node_id, lbl, shape='box', fillcolor='yellow',␣
↪style='filled, rounded')
        graph.edge(node_id, low_id, 'False')
        graph.edge(node_id, high_id, 'True')
        return node_counter+1, node_id
```

Now, we have the components needed to implement decision tree classifiers and regression models.

Following standard practice in scikit-learn, we inherit from the class `BaseEstimator`, which is the base class of all classifiers and regression models.

We write the `DecisionTree` class as an abstract class that contains the functionality common to all types of decision trees. Classification and regression models will be implemented as subclasses. The classification subclass is given below, while the regression subclass will be implemented as a part of the first assignment. Following scikit-learn naming conventions, we'll call the training and prediction methods `fit` and `predict`, respectively.

The functions that the subclasses will need to implement, and which are done differently for classification and regression, are the following:

- `get_default_value`: what output value to use if we decide to return a leaf node. For classification, this will be the most common output value, while for regression it will be the mean.
- `is_homogeneous`: tests whether a set of output values is homogeneous. For classification, this means that all outputs are identical; for regression, we'll probably test whether the variance is smaller than some threshold.
- `best_split`: finds the best splitting point for that feature. For classification, this will be based on one of the classification criteria (information gain, gini impurity, majority sum); for regression, it will be based on variances in the subsets.

[23]:
```
from graphviz import Digraph
from sklearn.base import BaseEstimator, ClassifierMixin
from abc import ABC, abstractmethod

class DecisionTree(ABC, BaseEstimator):

    def __init__(self, max_depth):
        super().__init__()
        self.max_depth = max_depth

    # As usual in scikit-learn, the training method is called *fit*. We first␣
↪process the dataset so that
    # we're sure that it's represented as a NumPy matrix. Then we call the␣
↪recursive tree-building method
    # called make_tree (see below).
    def fit(self, X, Y):
        if isinstance(X, pd.DataFrame):
            self.names = X.columns
```

```
            X = X.to_numpy()
        elif isinstance(X, list):
            self.names = None
            X = np.array(X)
        else:
            self.names = None
        Y = np.array(Y)
        self.root = self.make_tree(X, Y, self.max_depth)

    def draw_tree(self):
        graph = Digraph()
        self.root.draw_tree(graph, 0, self.names)
        return graph

    # By scikit-learn convention, the method *predict* computes the classification or regression output
    # for a set of instances.
    # To implement it, we call a separate method that carries out the prediction for one instance.
    def predict(self, X):
        if isinstance(X, pd.DataFrame):
            X = X.to_numpy()
        return [self.predict_one(x) for x in X]

    # Predicting the output for one instance.
    def predict_one(self, x):
        return self.root.predict(x)

    # This is the recursive training
    def make_tree(self, X, Y, max_depth):

        # We start by computing the default value that will be used if we'll return a leaf node.
        # For classifiers, this will be the most common value in Y.
        default_value = self.get_default_value(Y)

        # First the two base cases in the recursion: is the training set completely
        # homogeneous, or have we reached the maximum depth? Then we need to return a leaf.

        # If we have reached the maximum depth, return a leaf with the majority value.
        if max_depth == 0:
            return DecisionTreeLeaf(default_value)
```

```
        # If all the instances in the remaining training set have the same␣
↪output value,
        # return a leaf with this value.
        if self.is_homogeneous(Y):
            return DecisionTreeLeaf(default_value)

        # Select the "most useful" feature and split threshold. To rank the␣
↪"usefulness" of features,
        # we use one of the classification or regression criteria.
        # For each feature, we call best_split (defined in a subclass). We then␣
↪maximize over the features.
        n_features = X.shape[1]
        _, best_feature, best_threshold = max(self.best_split(X, Y, feature)␣
↪for feature in range(n_features))

        if best_feature is None:
            return DecisionTreeLeaf(default_value)

        # Split the training set into subgroups, based on whether the selected␣
↪feature is greater than
        # the threshold or not
        X_low, X_high, Y_low, Y_high = self.split_by_feature(X, Y,␣
↪best_feature, best_threshold)

        # Build the subtrees using a recursive call. Each subtree is associated
        # with a value of the feature.
        low_subtree = self.make_tree(X_low, Y_low, max_depth-1)
        high_subtree = self.make_tree(X_high, Y_high, max_depth-1)

        if low_subtree == high_subtree:
            return low_subtree

        # Return a decision tree branch containing the result.
        return DecisionTreeBranch(best_feature, best_threshold, low_subtree,␣
↪high_subtree)

    # Utility method that splits the data into the "upper" and "lower" part,␣
↪based on a feature
    # and a threshold.
    def split_by_feature(self, X, Y, feature, threshold):
        low = X[:,feature] <= threshold
        high = ~low
        return X[low], X[high], Y[low], Y[high]

    # The following three methods need to be implemented by the classification␣
↪and regression subclasses.
```

```
    @abstractmethod
    def get_default_value(self, Y):
        pass

    @abstractmethod
    def is_homogeneous(self, Y):
        pass

    @abstractmethod
    def best_split(self, X, Y, feature):
        pass
```

This is the subclass that implements decision tree classification. This implementation makes heavy use of the `Counter` class, which is a standard Python data structure for frequency counting.

[24]:
```
from collections import Counter

class TreeClassifier(DecisionTree, ClassifierMixin):

    def __init__(self, max_depth=10, criterion='maj_sum'):
        super().__init__(max_depth)
        self.criterion = criterion

    def fit(self, X, Y):
        # For decision tree classifiers, there are some different ways to␣
 ↪measure
        # the homogeneity of subsets.
        if self.criterion == 'maj_sum':
            self.criterion_function = majority_sum_scorer
        elif self.criterion == 'info_gain':
            self.criterion_function = info_gain_scorer
        elif self.criterion == 'gini':
            self.criterion_function = gini_scorer
        else:
            raise Exception(f'Unknown criterion: {self.criterion}')
        super().fit(X, Y)
        self.classes_ = sorted(set(Y))

    # Select a default value that is going to be used if we decide to make a␣
 ↪leaf.
    # We will select the most common value.
    def get_default_value(self, Y):
        self.class_distribution = Counter(Y)
        return self.class_distribution.most_common(1)[0][0]
```

```
    # Checks whether a set of output values is homogeneous. In the␣
↪classification case,
    # this means that all output values are identical.
    # We assume that we called get_default_value just before, so that we can␣
↪access
    # the class_distribution attribute. If the class distribution contains just␣
↪one item,
    # this means that the set is homogeneous.
    def is_homogeneous(self, Y):
        return len(self.class_distribution) == 1

    # Finds the best splitting point for a given feature. We'll keep frequency␣
↪tables (Counters)
    # for the upper and lower parts, and then compute the impurity criterion␣
↪using these tables.
    # In the end, we return a triple consisting of
    # - the best score we found, according to the criterion we're using
    # - the id of the feature
    # - the threshold for the best split
    def best_split(self, X, Y, feature):

        # Create a list of input-output pairs, where we have sorted
        # in ascending order by the input feature we're considering.
        sorted_indices = np.argsort(X[:, feature])
        X_sorted = list(X[sorted_indices, feature])
        Y_sorted = list(Y[sorted_indices])

        n = len(Y)

        # The frequency tables corresponding to the parts *before and including*
        # and *after* the current element.
        low_distr = Counter()
        high_distr = Counter(Y)

        # Keep track of the best result we've seen so far.
        max_score = -np.inf
        max_i = None

        # Go through all the positions (excluding the last position).
        for i in range(0, n-1):

            # Input and output at the current position.
            x_i = X_sorted[i]
            y_i = Y_sorted[i]

            # Update the frequency tables.
            low_distr[y_i] += 1
```

```
            high_distr[y_i] -= 1

            # If the input is equal to the input at the next position, we will
            # not consider a split here.
            #x_next = XY[i+1][0]
            x_next = X_sorted[i+1]
            if x_i == x_next:
                continue

            # Compute the homogeneity criterion for a split at this position.
            score = self.criterion_function(i+1, low_distr, n-i-1, high_distr)

            # If this is the best split, remember it.
            if score > max_score:
                max_score = score
                max_i = i

        # If we didn't find any split (meaning that all inputs are identical), return
        # a dummy value.
        if max_i is None:
            return -np.inf, None, None

        # Otherwise, return the best split we found and its score.
        split_point = 0.5*(X_sorted[max_i] + X_sorted[max_i+1])
        return max_score, feature, split_point
```

Here, we define the various criteria we can use to find the "quality" of a split in terms of how homogeneous the subsets are. See the reading material for the mathematical definitions of these criteria.

[25]:
```
def majority_sum_scorer(n_low, low_distr, n_high, high_distr):
    maj_sum_low = low_distr.most_common(1)[0][1]
    maj_sum_high = high_distr.most_common(1)[0][1]
    return maj_sum_low + maj_sum_high

def entropy(distr):
    n = sum(distr.values())
    ps = [n_i/n for n_i in distr.values()]
    return -sum(p*np.log2(p) if p > 0 else 0 for p in ps)

def info_gain_scorer(n_low, low_distr, n_high, high_distr):
    return -(n_low*entropy(low_distr)+n_high*entropy(high_distr))/(n_low+n_high)

def gini_impurity(distr):
    n = sum(distr.values())
    ps = [n_i/n for n_i in distr.values()]
```

```
    return 1-sum(p**2 for p in ps)

def gini_scorer(n_low, low_distr, n_high, high_distr):
    return -(n_low*gini_impurity(low_distr)+n_high*gini_impurity(high_distr))/
 ↪(n_low+n_high)
```

OUR CODE FOR TASK 2

We create a list containing depth values from 1 to 16, and from there we pick and print the one with the highest accuracy.

It seems like 12 is the best value for this scenario, and after a depth of 12 the acurracy wstarts to fall of for this data-set.

We also print and show a desicion tree with depth 3

[46]:
```
cls = TreeClassifier(max_depth=3)

data = pd.read_csv('data.csv', skiprows=1)
# Select the relevant numerical columns.
selected_cols = ['LB', 'AC', 'FM', 'UC', 'DL', 'DS', 'DP', 'ASTV', 'MSTV',␣
 ↪'ALTV',
                 'MLTV', 'Width', 'Min', 'Max', 'Nmax', 'Nzeros', 'Mode',␣
 ↪'Mean',
                 'Median', 'Variance', 'Tendency', 'NSP']
data = data[selected_cols].dropna()

# Shuffle the dataset.
data_shuffled = data.sample(frac=1.0, random_state=0)

# Split into input part X and output part Y.
XNEW = data_shuffled.drop('NSP', axis=1)

# Map the diagnosis code to a human-readable label.
def to_label(y):
    return [None, 'normal', 'suspect', 'pathologic'][(int(y))]

YNEW = data_shuffled['NSP'].apply(to_label)
cls.fit(XNEW, YNEW)

best_classifier = _, 0
for max_depth in range(1, 16):
    print('Classifying tree at', max_depth+1, 'of', 16, end="\r")
    clfTree = TreeClassifier(max_depth)
    clfTree.fit(XNEW, YNEW)
    score = cross_val_score(clfTree, XNEW, YNEW, cv=5, scoring='accuracy').
 ↪mean()
    if score > best_classifier[1]:
        best_classifier = clfTree, score
```

```
print(best_classifier)
cls.draw_tree()
```

```
(TreeClassifier(max_depth=12), 0.912039768019884)
```

[46]: