# CS 3430: SciComp with Py
# Assignment 11
# Experimenting with Decision Trees on IRIS Dataset

Vladimir Kulyukin
Department of Computer Science
Utah State University

April 05, 2020

## 1 Learning Objectives

1. IRIS Dataset
2. Splitting Data into Training and Testing Sets
3. Applying Binary ID3 to Multi-class Classification

## Introduction

In this assignment, we'll experiment with decision trees on the IRIS dataset from scikit-learn. We'll also apply binary ID3 to the multi-class classification problem posed by the IRIS dataset. In machine learning (ML), a multi-class classification problem is any problem where examples are classified into more than two classes. Thus, the IRIS dataset where flowers are classified into three classes (i.e., setosa, versicolor, and virginica) is one such problem. Recall that a problem with just two classes (`YES` and `NO` or `PLUS` and `MINUS`) is a binary classification problem. For example, Dr. Quinlan's PlayTennis problem from Assignment 10 and Lecture 20 is a binary classification problem.

This assignment doesn't require you to write a lot of code. It'll give you another chance to work on your solutions to Assignment 10 in `bin_id3.py`. You'll also experiment with scikit-learn decision trees and compare them with binary ID3.

## Problem 1 (0 points)

Review the materials for Lectures 21 and 22 on learning decision trees. I'd like to remind you that you **don't have to** watch the screencasts **and** read the pdfs. If the slides and/or code samples are sufficient for you, so be it! If you're an interactive learner and prefer the screencasts, by all means watch them.

Whatever materials you use, make sure you're comfortable with entropy, information gain, gain ratio, and gini. These concepts repeatedly surface in different areas of CS: in addition to ML, information theory and cryptography come to mind right away. Also, spend some time investigating the IRIS dataset by playing with `investigate_iris_dataset.py` from Lecture 22. I've also included it in this assignment. Once you understand the structure of the IRIS dataset, you'll be able to understand all other scikit-learn datasets.

## Problem 2 (4 points)

Let's start playing with decision trees on the IRIS dataset. There's a viewpoint in software engineering (shared by many, challenged by some) that on any serious project only 20% of your time is spent on coding whereas 80% of your time goes into debugging and maintenance. If we rephrase it for ML, we can say that on any serious ML project 80% of your time is spent on data curation and 20% – on developing ML algorithms. I'd say that in ML the time split feels more like 90-10 in favor of data curation.

### Discretization of Data

The first challenge that one confronts in datasets such as IRIS is how to handle continuous attribute values. In IRIS, the values of attributes are not nice discrete values (e.g., Sunny, Overcast, Rainy for the Outlook attribute in the PlayTennis dataset). Rather, they are reals.

What do you do in such cases? One way is to *discretize* them into distinct symbolic values. This is the approach that we'll take in this assignment. Of course, there are multiple ways of discretizing continues data. One straightforward way is to discover the lowest ($l$) and highest ($h$) values for each value interval and then split the interval $[l, h]$ into a finite set of non-overlapping intervals, each of which is mapped to a specific symbol. For example, we can write the function `disc_petlen()` (the definition is available in `disc_iris.py`) that maps a real value of the petal length attribute into five intervals, each of which is assigned a symbol (e.g., `PetLen2`). These symbol names are arbirary and can be replaced with integers.

```
def disc_petlen(petl):
    """
    Maps a petal length petlen to PetLen0, PetLen1, PetLen2, PetLen3, or PetLen4.
    """
    petlen = {'PetLen0': [1.0, 2.18],
              'PetLen1': [2.18, 3.3600000000000003],
              'PetLen2': [3.3600000000000003, 4.540000000000001],
              'PetLen3': [4.540000000000001, 5.720000000000001],
              'PetLen4': [5.720000000000001, 6.91]}
    for k, v in petlen.items():
        if v[0] <= petl < v[1]:
            return k
```

Once we know how to discretize individual IRIS attributes, we can discretize IRIS examples.

```
def disc_iris_example(example, named_target):
    """
    Maps an iris example example into a 5-tuple of discretized symbols.
    """
    assert named_target in set(['setosa', 'versicolor', 'virginica'])
    sepl, sepw, petl, petw = example
    return (disc_seplen(sepl), disc_sepwid(sepw), disc_petlen(petl), disc_petwid(petw), named_target)
```

We can now convert the entire IRIS dataset into a CSV file in such a way that each attribute will have a finite set of discrete values. The function `disc_iris(iris_data, csv_file)` does exactly that. Let's run it.

```
>>> disc_iris(iris_data, 'iris_data.csv')
```

This call will create `iris_data.csv` (included in this assignment) that looks as follows.

```
Number, SepLen, SepWid, PetLen, PetWid, Class
0, SepLen1, SepWid3, PetLen0, PetWid0, setosa
1, SepLen0, SepWid2, PetLen0, PetWid0, setosa
2, SepLen0, SepWid2, PetLen0, PetWid0, setosa
...
50, SepLen3, SepWid2, PetLen3, PetWid2, versicolor
51, SepLen2, SepWid2, PetLen2, PetWid2, versicolor
52, SepLen3, SepWid2, PetLen3, PetWid2, versicolor
...
100, SepLen2, SepWid2, PetLen4, PetWid4, virginica
101, SepLen2, SepWid1, PetLen3, PetWid3, virginica
102, SepLen3, SepWid2, PetLen4, PetWid4, virginica
```

The target attribute in this case is `Class` with three possible values: `setosa`, `versicolor`, and `virginica`. The other attributes have obvious interpretations. Note also that each attribute has a finite set of symbolic values.

## Splitting Data into Training and Testing Datasets

In real ML problem, we must split the available data into the non-overlapping training and testing datasets to avoid overfitting and bias (as much as we can!). The training datasets are used only for training ML models (e.g., decision trees). The testing datasets are used only for evaluating trained ML models. The function `train_test_split_iris(iris_data, train_size)` in `disc_iris.py` implements a simple version of the split for the IRIS dataset.

```
def train_test_split_iris(iris_data, train_size):
    assert 0.0 < train_size < 1.0
    setosa = get_iris_examples_classified_as(iris_data, 'setosa')
    versicolor = get_iris_examples_classified_as(iris_data, 'versicolor')
    virginica = get_iris_examples_classified_as(iris_data, 'virginica')
    random.shuffle(setosa)
    random.shuffle(versicolor)
    random.shuffle(virginica)
    train_index = int(50*train_size)
    train_setosa, test_setosa = setosa[:train_index], setosa[train_index:]
    train_versicolor, test_versicolor = versicolor[:train_index], versicolor[train_index:]
    train_virginica, test_virginica = virginica[:train_index], virginica[train_index:]
    return train_setosa, train_versicolor, train_virginica, \
           test_setosa, test_versicolor, test_virginica
```

Since we'd like to apply the binary ID3 algorithm to IRIS, we need to modify the discretized dataset in such a way the target attribute `Class` has only two values – `Yes` and `No`. Once we do that, we can use the binary ID3 algorithm to learn a ID3 tree for setosas so long as the values of the attribute `Class` are `Yes` and `No`, where `Yes` labels only setosa examples and `No` – all other examples. We can similarly learn binary ID3 trees for versicolors and virginicas. The function `disc_train_test_bin_split_for_flower(iris_data, flower_class, train_size)` generates the training and testing CSV files for the binary ID3 algorithm for a given `flower_class`.

```
>>> disc_train_test_bin_split_for_flower(iris_data, 'versicolor', 0.7)
```

The above call generates two CSV files, `versicolor_train_bin.csv` and `versicolor_test_bin.csv`, in the current directory that contain the training and testing CSV records.

The function `train_test_bin_id3_dt(iris_data, target_attrib, flower_class, n, train_size, dbg)` in `iris_bin_id3.py` uses these files to train and test a binary ID3 tree for a given flower class.

## Classification of IRIS Examples with Three Binary ID3 Trees

Write the function `bin_id3_iris_classify()` in `iris_bin_id3.py`. This function works as follows. It uses one of the decision trees (e.g., `versi_dt`) to classify an example. If the first tree returns `PLUS`, then the function returns the string `'versicolor'`. If the tree returns `MINUS`, then the function goes on to another decision tree (e.g., `virg_dt`) to classify the example. If this tree returns `PLUS`, then the function returns `'virginica'`. If the function returns `MINUS`, the function classifies the example with the third decision tree (e.g., `setosa_dt`). If the tree returns `PLUS`, the function returns `'setosa'`. If the tree returns `MINUS`, the function returns `'unknown'`. Of course, you may want to experiment with different choice sequences of the decision trees. For example, virginica – first, setosa – second, and versicolor – third. In the comments at the beginning of `iris_bin_id3.py` state which tree choice sequence gave you the best performance.

The function `train_test_bin_id3_multi_dt(iris_data, target_attrib, n, train_size, dbg)` in `iris_bin_id3.py` uses these files to train three binary ID3 trees for the three IRIS classes and uses `bin_id3_iris_classify()` to test the accuracy of classifying IRIS examples with three binary ID3 trees. Let's run the three function calls in the main of `iris_bin_id3.py`.

```
if __name__ == '__main__':
    train_test_bin_id3_multi_dt(iris_data, 'Class', 10, 0.8, False)
    train_test_bin_id3_multi_dt(iris_data, 'Class', 10, 0.7, False)
    train_test_bin_id3_multi_dt(iris_data, 'Class', 10, 0.6, False)
    pass
```

Here's what my output looks like. I'm deliberately not telling you which tree choice sequence I used in my implementation of `bin_id3_iris_classify()`, because I want you to play and experiment with different choice sequences.

```
Testing multi bin id3 tree classification for iris dataset
run 0) average acc = 0.8
run 1) average acc = 0.9
run 2) average acc = 0.9666666666666667
run 3) average acc = 0.9
run 4) average acc = 1.0
run 5) average acc = 0.9333333333333333
run 6) average acc = 0.9666666666666667
run 7) average acc = 0.9666666666666667
run 8) average acc = 0.9333333333333333
run 9) average acc = 0.9333333333333333
avrg acc = 0.93

Testing multi bin id3 tree classification for iris dataset
```

```
run 0) average acc = 0.9555555555555556
run 1) average acc = 0.9333333333333333
run 2) average acc = 0.9777777777777777
run 3) average acc = 0.9777777777777777
run 4) average acc = 0.9777777777777777
run 5) average acc = 0.9555555555555556
run 6) average acc = 0.9333333333333333
run 7) average acc = 0.8888888888888888
run 8) average acc = 0.9111111111111111
run 9) average acc = 0.9111111111111111
avrg acc = 0.9422222222222223

Testing multi bin id3 tree classification for iris dataset
run 0) average acc = 0.95
run 1) average acc = 0.9333333333333333
run 2) average acc = 0.9666666666666667
run 3) average acc = 0.9666666666666667
run 4) average acc = 0.9666666666666667
run 5) average acc = 0.9666666666666667
run 6) average acc = 0.8833333333333333
run 7) average acc = 0.9333333333333333
run 8) average acc = 0.9666666666666667
run 9) average acc = 0.9166666666666666
avrg acc = 0.945
```

For all three splits, the average classification accuracy varies from 93% to 95%. Of course, these accuracies vary from run to run, because we use random shuffling. I've included in the zip for this assignment the file `iris_dt.py` that uses the same data splits and fits the scikit-learn decision tree model to the IRIS data. Let's run its main.

```
if __name__ == '__main__':
    run_dt_train_test_split(examples, target, 10, 0.2)
    run_dt_train_test_split(examples, target, 10, 0.3)
    run_dt_train_test_split(examples, target, 10, 0.4)
```

Here's the output of a run on my laptop.

```
run 0) acc = 0.9
run 1) acc = 1.0
run 2) acc = 0.9
run 3) acc = 0.9
run 4) acc = 0.9666666666666667
run 5) acc = 1.0
run 6) acc = 1.0
run 7) acc = 0.9333333333333333
run 8) acc = 0.9666666666666667
run 9) acc = 0.9
test_size = 0.2; average acc = 0.9466666666666667

run 0) acc = 0.9555555555555556
run 1) acc = 0.9777777777777777
run 2) acc = 0.9777777777777777
run 3) acc = 0.9555555555555556
run 4) acc = 0.9111111111111111
run 5) acc = 0.9111111111111111
run 6) acc = 0.9777777777777777
run 7) acc = 0.8888888888888888
run 8) acc = 0.9333333333333333
run 9) acc = 1.0
test_size = 0.3; average acc = 0.9488888888888889

run 0) acc = 0.95
run 1) acc = 0.95
run 2) acc = 0.9666666666666667
run 3) acc = 0.9666666666666667
run 4) acc = 0.9833333333333333
run 5) acc = 0.9333333333333333
run 6) acc = 0.9
run 7) acc = 0.9666666666666667
run 8) acc = 0.9666666666666667
run 9) acc = 0.9333333333333333
test_size = 0.4; average acc = 0.9516666666666668
```

As you can see the accuracies of the native scikit-learn decision tree that uses gini (unlike our binary ID3 that uses information gain) are slightly higher, but in the same ballpark. Spend a few minutes to reflect on the fact how we managed to use the binary ID3 algorithm with information gain to solve a multi-class classification problem that works on par with scikit-learn that uses gini.

Two more things before we wrap up. First, there are no unit tests for this homework – just sample outputs. Why? Because as soon as you inject randomness into your experiments, the exact ground truths vanish. Only ballpark values remain. Second, you may have to modify your implementation of `bin_id3.py` from Assignment 10 to handle missing attribute values. Since we're injecting randomness in data splits, our training examples may not have specific values for specific attributes. A simple way to handle the missing attribute problem in a test example is to classify it as `MINUS`.

## What To Submit

Submit `iris_bin_id3.py`, `bin_id3.py`, `disc_iris.py`, and `investigate_iris_dataset.py` (and all other files needed to run your submission). Zip up your submission. Don't submit the CSV files!

Remember to write your comment at the beginning of `iris_bin_id3.py` to state which tree choice sequences gave you the best performance.

Stay healthy and have fun experimenting with decision trees on IRIS!