# MSDS 621 - Introduction to Machine Learning

**Homework 3**

Robert Clements

## 1 Implementing Classification and Regression Decision Trees

### 1.1 Turning it in

We will use Github classroom to submit assignments. For this assignment you must submit one file: - dtree.py file

### 1.2 Goal

In this homework you will implement both classification and regression decision trees with `fit` and `predict` methods. This will help you truly understand the inner workings of how a decision tree decides to segment the data, and hopefully will reveal how flexible and powerful tree-based models can be.

### 1.3 Getting Started

As always, there is template code in the repository. To make your life easier, I would use the template code as a guide, but if you are particularly ambitious, you can ignore it and create your own implementation, so long as the tests still pass.

### 1.4 Discussion

First you should consider creating two classes to represent the two types of nodes in our decision trees: `DecisionNode` and `LeafNode`. These two classes define the interior and leaf nodes of your decision trees. You can define your own objects, but make sure that your tree nodes respond to function `t.predict(x)` for some tree node t and 1D feature vector x. In other words, performing method call `t.predict(x)` will invoke function `DecisionNode.predict(t,x)` or

LeafNode.predict(t,x), depending on the type of t. That is what we call dynamic dispatch or method invocation (versus function call). They may look something like this:

```
class DecisionNode:
    def __init__(self, col, split, lchild, rchild):
        self.col = col
        self.split = split
        self.lchild = lchild
        self.rchild = rchild

    def predict(self, x_test):
        # Make decision based upon x_test[col] and split
        ...
```

```
class LeafNode:
    def __init__(self, y, prediction):
        """Create leaf node from y values and prediction; prediction is mean(y) or mode(y)
        Note you can use scipy.stats for the mode function"""
        self.n = len(y)
        self.prediction = prediction

    def predict(self, x_test):
        # return prediction
        ...
```

You will need to implement regression and classification trees, which means you will need two classes RegressionTree621 and ClassifierTree621 defined in order to ensure that you do the node splitting using the correct loss function (MSE for regression and Gini for classification), and that the prediction is stored using the correct approach (mean for regression and mode for classification). However, we will also define a third class, called DecisionTree621, which will have the **fit** and **predict** methods in it. This way you can *inherit* the methods from DecisionTree621 into the RegressionTree621 and ClassifierTree621 classes, meaning you only have to write them once, and can use them for both types of decision trees. Recall that the trees are fit the same way, the only thing that is different is the loss function that is used at each node, and how the final prediction is calculated in the leaf nodes. These classes might look like the following:

```
class DecisionTree621:
    def __init__(self, min_samples_leaf=1, loss=None):
        self.min_samples_leaf = min_samples_leaf
        self.loss = loss # loss function; either np.var or gini
    def fit(self, X, y):
        """
        Create a decision tree fit to (X,y) and save as self.root,
        the root of our decision tree, for either a classifier or
        regressor.
        Leaf nodes for classifiers predict the most common class
        (the mode) and regressors predict the average y
        for observations in that leaf.

        This function is a wrapper around fit_() that just stores
        the tree in self.root.
        """
        self.root = self.fit_(X, y)

    def fit_(self, X, y):
        """
        Recursively create and return a decision tree fit to (X,y)
        for either a classifier or regressor.  This function should
        call self.create_leaf(X,y) to create the appropriate leaf node,
        which will invoke either RegressionTree621.create_leaf() or
        ClassifierTree621. create_leaf() depending
        on the type of self.

        This function is not part of the class "interface" and is for
        internal use, but it embodies the decision tree fitting algorithm.

        (Make sure to call fit_() not fit() recursively.)
        """
        ...

    def predict(self, X_test):
        """
        Make a prediction for each record in X_test and return as array.
        This method is inherited by RegressionTree621 and ClassifierTree621 and
        works for both without modification!
        """
        ...
```

We can now use the above class as a sort of *parent* class. Then we can define our regression and classification classes like below:

```
class RegressionTree621(DecisionTree621):
    def __init__(self, min_samples_leaf=1):
        super().__init__(min_samples_leaf, loss=np.var)
    def score(self, X_test, y_test):
        "Return the R^2 of y_test vs predictions for each record in X_test"
        ...
    def create_leaf(self, y):
        """
        Return a new LeafNode for regression, passing y and mean(y) to
        the LeafNode constructor.
        """
        ...

class ClassifierTree621(DecisionTree621):
    def __init__(self, min_samples_leaf=1):
        super().__init__(min_samples_leaf, loss=gini)
    def score(self, X_test, y_test):
        "Return the accuracy_score() of y_test vs predictions for each
        record in X_test"
        ...
    def create_leaf(self, y):
        """
        Return a new LeafNode for classification, passing y and mode(y) to
        the LeafNode constructor.
        """
        ...
```

Notice that we are inheriting the `fit` and `predict` methods from the parent class `DecisionTree621` and defining `score` and `create_leaf`. You can think of inheriting as including or copying the functions from the superclass. Notice how we define the loss function here, and can also set a value for `min_samples_leaf` here. I've just chosen to keep the loss function for regression simple and use `var` from numpy. You should define your own `gini` loss function, though:

```
def gini(y):
    "Return the gini impurity score for values in y"
```

### 1.5 The `fit` and `fit_` methods

The training algorithm, embodied by function `fit_()`, exhaustively tries combinations of features and feature values, looking for an optimal split. The optimal split is one that splits a feature space for one feature into two sub-regions and the average variance (regression) or impurity (classification) is lower than that of the current node's observations and any other feature/value split.

The first decision node is created by looking at the entire set of training records in X, y. Once split into two regions, training recursively splits those two regions. In this way, different subsamples of the training data are examined to create the decision nodes of the tree. If every decision node split the current set of observations exactly in half, than the height of the tree would be roughly log(len(X)). Training returns a leaf node when there are less than or equal to min_samples_leaf observations in a subsample.

The algorithm looks like this:

dtreefit(X, y, loss, min_samples_leaf)

```
• if $|X| \leq$ min_samples_leaf or # of unique values in X == 1 then return Leaf(y)
• col,split = bestplit(X,y,loss)
• if col = -1 then return Leaf(y)
• lchild = dtreefit($X[X_{col} < split], y[X_{col} < split]$, loss, min_samples_leaf)
• rchild = dtreefit($X[X_{col} \geq split], y[X_{col} \geq split]$, loss, min_samples_leaf)
• return DecisionNode(col, split, lchild, rchild)
```

Finding the optimal split looks like the below. In our case, for speed reasons, we're going to pick a subset of all possible split values. Choose k=11 for our project.

```
• set best = (feature = -1, split = -1, loss = loss(y))
• for col i in 1...p do:
    – candidates = randomly pick k « n values from feature i
    – foreach split in candidates do:
        ∗ $yl = y[X < split]$
        ∗ $yr = y[X \geq split]$
        ∗ if |yl| < min_samples_leaf or |yr| < min_samples_leaf then continue
        ∗ l = $\frac{|yl| \times loss(yl) + |yr| \times loss(yr)}{|yl|+|yr|}$ weighted average of loss function
        ∗ if l = 0 then return col, split
```

```
            if l < best_loss then best = (feature, split, l)
        end
    end
    return best[feature], best[split]
```

We could also improve generality by picking splits midway between X values rather than at X values, but that means sorting or scanning values looking for the nearest value less than the split point.

### 1.6 Prediction

To make a prediction for a feature vector x, we start at the root node and descend through the decision nodes to the appropriate leaf. At each decision node, we test a specific variable's value, x[j], against the split value stored in the decision node. If x[j] is less than the split value, prediction descends down the left child. Otherwise, prediction descends down the right child. Upon reaching a leaf, we predict either the most common class or the average value among the y targets associated with that leaf. Here is the algorithm:

```
predict(node, x)
    if node is leaf then:
        if classification then return mode(node.y)
        return mean(node.y)
    end
    if x[node.col] < node.split then return predict(node.lchild, x)
    return predict(node.rchild, x)
```

In our project, we are breaking up this algorithm actually into two main parts, one for the decision nodes and one for the leaves. That is why you see the same method repeated in the class definitions associated with binary trees:

```
class DecisionNode
    def predict(self, x_test):
        # Make decision based upon x_test[col] and split
        # lines 6 and 7 from predict() algorithm above
```

```
class LeafNode:
    def predict(self, x_test):
        # return prediction passed to constructor of LeafNode
        # lines 3,4 from algorithm above
```

The `DecisionNode.predict()` method invokes `predict()` on the left or right child depending on x_test's values. The leaf node just returns the prediction (mean or mode) passed into the constructor for LeafNode, so it deviates a bit from the predict() algorithm.

### 1.7 Deliverables

To submit your project, ensure that your `dtree.py` file is submitted in the root of your repository. It should not have a main program; it should consist of a collection of classes and functions. You **must** implement the following classes:

- `DecisionTree621`: should contain `fit`, `fit_`, and `predict` methods
- `RegressionTree621`: should contain `score` and `create_leaf` methods, and *inherit* the `fit` and `predict` methods from `DecisionTree621`
- `ClassifierTree621`: should contain `score` and `create_leaf` methods, and *inherit* the `fit` and `predict` methods from `DecisionTree621`

You **can** create the two classes below that represent the two types of nodes, but you are free to implement your decision tree in any way that works, only the three classes above are absolutely required.

- `DecisionNode`
- `LeafNode`

And some helper functions you might want, but are not required to use, are:

- `find_best_split`: for finding the best split given the data and a loss function
- `gini`: a simple implementation of the gini index to use for classification

### 1.8 Evaluation

To evaluate your projects I will download your repository and run `test_dtree.py` from your root directory. As always, I encourage you to test your code both locally and using Github Actions by putting the `test.yml` file in a directory called `.github/workflows`. I will be using the unit tests **as a guide**, meaning that if you pass all unit tests, and the timing test, you will get 100%. If you do not pass all unit tests, then your code will be inspected and evaluated for errors. Note that if you *almost always* pass the unit tests (say at least 90% of the time) then your code is probably ok and you've run into a bit of bad luck on the training and testing splits. If you are failing multiple tests in a single run, there is likely something wrong with your code.

Your code should run on your local machine in less than one minute (or there will be a 10 point penalty).

Here is a sample test run:

```
platform darwin -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0 -- /opt/anaconda3/bin/python
cachedir: .pytest_cache
plugins: anyio-3.5.0
collected 7 items

test_dtree.py::test_boston PASSED   [ 14%]
test_dtree.py::test_boston_min_samples_leaf PASSED   [ 28%]
test_dtree.py::test_california_housing PASSED [ 42%]
test_dtree.py::test_iris PASSED  [ 57%]
test_dtree.py::test_wine PASSED  [ 71%]
test_dtree.py::test_wine_min_samples_leaf PASSED   [ 85%]
test_dtree.py::test_breast_cancer PASSED  [100%]


==================== 7 passed, 2 warnings in 21.69s ============
```