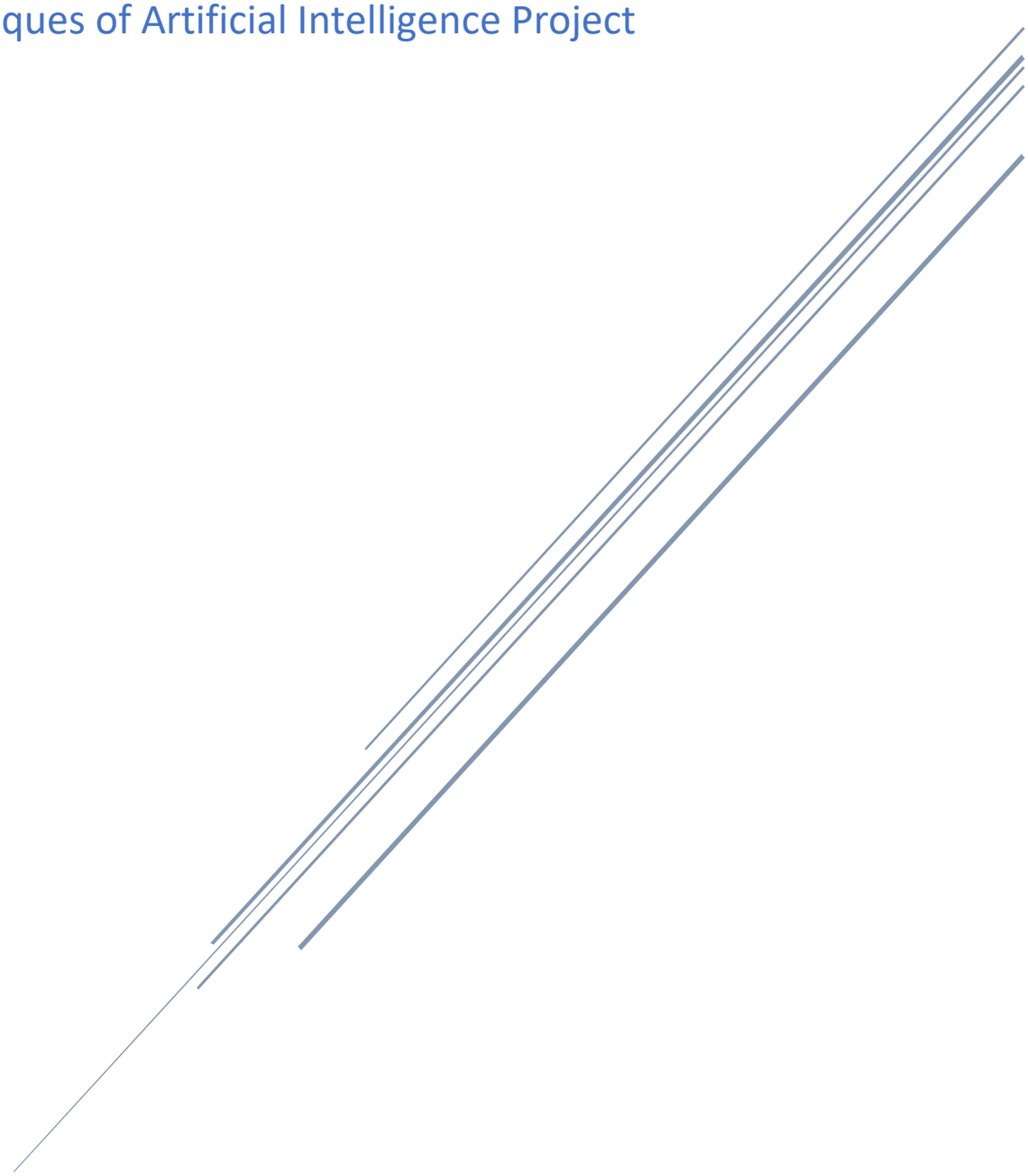
# CONSTRUCTING A DECISION TREE/BOOTSTRAP AGGREGATED DECISION TREE CLASSIFIER FROM SCRATCH IN PYTHON

Techniques of Artificial Intelligence Project



VUB
Samuel Oswald

# Contents

Introduction ..... 2
Methods ..... 3
Process ..... 3
Code ..... 4
Decision Tree ..... 4
Bootstrap Aggregated Trees ..... 5
Example Case: Classification of Hyperspectral Remotely Sensed Imagery ..... 5
Conclusion ..... 6
References ..... 6
Appendix ..... 7
DecisionTree.py ..... 7
Run.py ..... 14
DataManagement.py ..... 16

## Introduction

Decision Tree classifiers are a commonly used machine learning method, which utilise training data to develop a hierarchical set of rules which may then be used to classify new samples. Effectively, the classifier will, beginning from a root node which contains all training data, aim to most efficiently split this data into separable classifications. Each time a data split is performed, two children nodes are produced, which may either be further separated or classified into a leaf node, such that when further data is input to the model consistent with the rules which lead to that leaf, the new data will be classified as belonging to the same class. Leaf nodes will be created either when all samples are classified homogenously in a node, or when user-defined limits, such as the minimum available training samples; or maximum depth of the tree, is reached. An example decision tree is depicted in Figure 1. Decision trees present a number of advantages over other machine learning classifiers, with some of the key ones being summarised below:

1) The model output by the decision tree learner is more easily understandable by human readers than in the case of many other learning algorithms, as the decision tree consists of a series of conditional statements.
2) Can handle several data inputs (categorical, continuous) and problems with outputs which may include a binary classification, multiple classification, or regression.
3) Nonlinear relationships between variables will not affect tree performance. When continuous explanatory variables are used, the decision statements do not assume linearity in the data.
4) Decision trees are able to handle noisy training data, as well as data which may have missing attribute values.

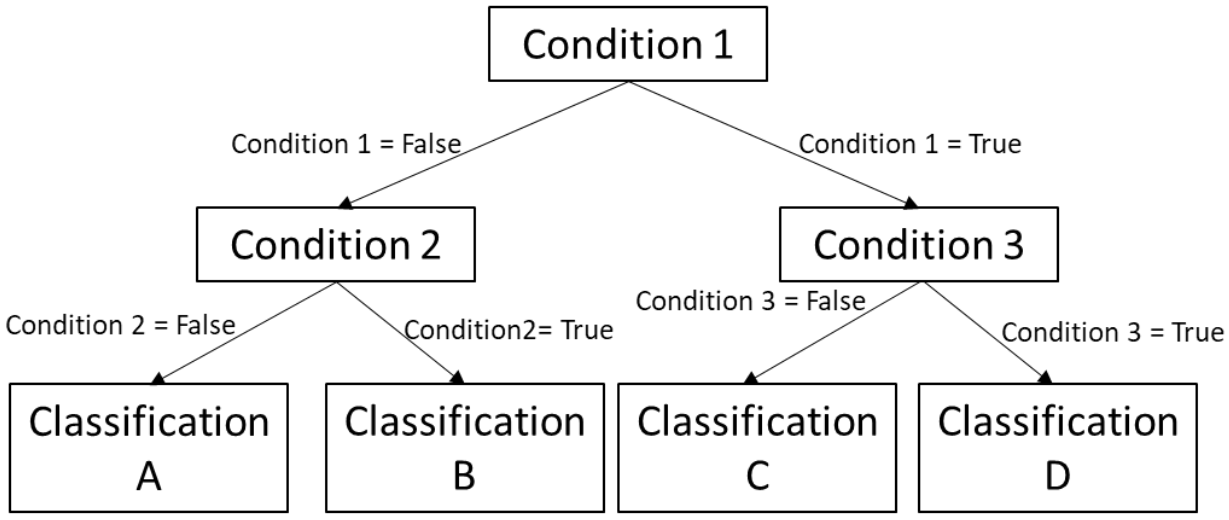



*Figure 1: Theoretical example of simple decision tree classifier.*

Despite the advantages of decision trees, there may be some issues with using it as a learning method. One key problem is that they are prone to overfitting the training data, whereby the decision tree perfectly explains the training data, however will have reduced accuracy when applied

to new testing data (Mitchell, 1997). They are additionally prone to high variance; whereby random noise is modelled into the learner. There are several methods which may prevent overfitting. During the implementation of the learning algorithm, limits to maximum depth of the decision tree, or a specification of the minimum number of samples needed to classify a node into a leaf, are two simple ways in which users may manually improve the accuracy of a decision tree, and which prevent nodes from being overly specified to certain data. Two more advanced methods are reduced-error pruning and rule post-pruning, which both involve fitting a full decision tree, before checking each node for possible locations where removal would improve accuracy and removing them.

Another method which has been found to improve decision tree accuracy, improving both the fit and variance, is using bootstrap aggregation. Bootstrap aggregation(or tree bagging) is an ensemble learner method, based on decision trees. It involves re-sampling training data with replacement, and then creating a decision tree with the re-sampled data, such that each decision tree approximates the original training set (Breiman, 1996). This is done multiple times, constructing many different trees(typically at least 500). From this collection of trees, new data are classified based on a consensus vote, where the mode of the various classifications (in the case of categorical output values) is selected.

For the class Techniques of Artificial Intelligence, a decision tree classifier was created in Python 3.6. This report summarises the script used to implement the decision tree and bootstrap aggregated classifier and applies both to a case study of a remote sensing learning problem.

# Methods

## Process

For the decision tree, the Classification and Regression Tree(CART) algorithm was implemented (Breiman, 1984). The steps of the CART algorithm are summarised below:

1) From current node(starting at root), find the best independent variables which splits the data according to the best splitting criteria. Divide node into two child nodes based on the split value.
2) If child node satisfies some stopping criteria, create a leaf node at this location.
3) Repeat until all branches of decision tree result in a leaf node.

For the bootstrap aggregated decision trees, the steps above are repeated for a user defined number of trees, with the difference being that a subsample of the input training samples is used, represented as a percentage of the total samples. Replacement of these samples is allowed, such that one sample may appear twice in a bagged training set.

The splitting criteria used in this case is gini impurity. Gini impurity is a measure of how impure a data set is and returns the probability that a classification would be wrong if randomly guessed. The best split is calculated as being the value which best reduces this impurity, homogenizing the data, otherwise known as information gain. Gini impurity can be calculated as:

$$I_G(p) = 1 - \sum_{i=1}^{J} {p_i}^2$$

Where $p_i$ represents the proportion of class i in the current set of data, and J is the total number of classes. To calculate the best information gain, the gini is calculated for the two candidate children nodes for each data split, and these, times the respective proportions of samples in each respective child node, are subtracted from the current gini. The equation is depicted as follows:

$$\text{Gain} = i(t) - p_L i(t_L) - p_R i(t_R),$$

Where *i(t)* represents the gini at the parent node, $i(t_L)$ and $i(t_R)$ represent gini at the children nodes, and *p* represents the proportion of total samples at each node.

For this project, the algorithm is coded to handle continuous variables, and will also function with boolean explanatory variables. In the future, it could be extended to additionally consider categorical variables.

## Code

Three python scripts were created to run the decision tree algorithm. The first is an importable module, "DecisionTree.py" which contains the functions used to build and analyse decision trees and bagged trees, which is discussed below. Building of the scripts is done through the "Run.py" script, which allows users to customise the data and decision tree parameters. A "DataManagement.py" module was also created, which converts classification classes to integer, if provided in other formats. No external libraries were included aside from numpy(in order to simplify some calculations and array management), as well as pandas(for import of data). All code is included in the appendix.

### Decision Tree

**dt_train:** Takes the user input from the run script, as well as parameters for max depth and minimum size, and is the initial function used to call other functions and build the decision tree. Creates the initial root node of the tree in the format which will be output, then passes this to the recursive "decision" function. Once the tree is fully built, it will be returned to the user.

**gini:** Calculates and returns gini impurity using the gini formula described above, using a node as input.

**gain:** Using the current node, current gini, and a splitting value as inputs, this function first divides the current node into two children. The gini impurity of each child node is calculated, and then the gain is calculated for this potential split using the gain function described above. It will return the gain, as well as the samples for the left side and right-side child nodes(should they be then found to have the best gain).

**split:** Iterates over every unique split value for each attribute in a node, to calculate where the best gain is. This is the most computationally cost-heavy of the functions, particularly when dealing with large datasets with many potential split points. Will return either the split where the best gain is found, or failing this, the parent node on the left side, and an empty node on the right, signalling that no improvement in classification can be found.

**leaf:** If a stopping criterion is identified, the node will be classified as being the most common occurring class in the training data at this point.

**decision:** Recursive function that progressively splits the decision tree using a number of if else statements, eventually calling stopping functions when conditions such as fully homogenous nodes, max depth, or minimum sample size are reached.

**classify:** Classification of a row of data, called from the predict function. Checks whether a sample satisfies the criteria of the decision tree, and ultimately attributes the sample to a class.

**dt_predict:** Iterates over a set of data, producing classifications based on a built decision tree. May be used for validation or prediction purposes. Returns a list of predictions.

**dt_confusion_matrix:** Compares predicted with actual values for validation purposes. Returns both a confusion matrix, as well as accuracy of the classifier as a whole.

### Bootstrap Aggregated Trees

**bt_train:** Trains a bagged tree classifier, using the same inputs as a decision tree, with the addition of the ratio of samples(as compared to total samples) to be included in each bagged tree, as well as the number of trees to be created. Calls the dt_build function n amount of times, where n is the specified number of trees to be created

**bt_predict:** Makes a prediction of the output value. This is done by calling the dt_predict function for each tree, creating an array of values for each new sample to be predicted. A single value is then selected by calculating the mode of the array of predictions, and a list of predictions returned.

**bt_confusion_matrix:** Simply calls the dt_confusion_matrix, using the same inputs and again returning a matrix with accuracy.

## Example Case: Classification of Hyperspectral Remotely Sensed Imagery

One field where machine learning algorithms are relevant is in the classification of remotely sensed imagery. To test the capability of the decision tree and bagged tree classifier, both were applied to a dimensionally-reduced APEX image, which consists of 14 continuous reflectance values, and is divided into 26 classes. The data includes 7912 training samples, and 3391 validation samples. The decision tree was calculated to a maximum depth of 20, with the minimum allowed number of samples before a node is forced to classify being 10. For the bagged tree classifier, 20 trees were built, using a bagged sample set 80% of the size of the total training data for each tree. Both classifiers performed well, suggesting that decision trees are an accurate solution to this problem. Even with only a relatively small number of bagged samples, the bagged tree classifier outperformed the single decision tree significantly. The former achieved an overall accuracy of 82.66%, whilst the latter achieved an overall accuracy of 89.20%. The raw image with training samples overlain, as well as the classification maps(visualized using matplotlib) are shown in Figure 2.

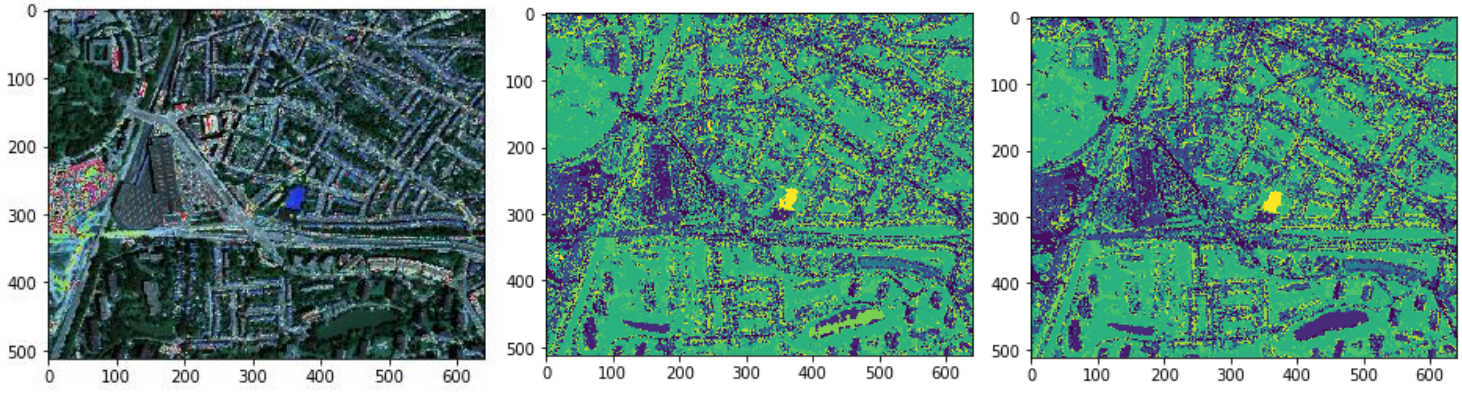



*Figure 2: (left) raw image with training samples overlain, (middle) classification output from decision tree, (right) classification output from bagged decision trees.*

## Conclusion

One issue with the current script, especially compared to more commonly used libraries, is that it is quite slow when building the model. For example, the classification described above took a total of 40 minutes. Further refinements to the script may improve this speed. Currently, it is likely most useful when classifying data with less unique values which may be split, which drastically increases computational time due to the algorithms exhaustive search through each possible split point.

Aside from computational time however, the script appears to successfully implement the CART algorithm, as well as a bootstrap aggregated version of it. Further improvements to the script could include adding an automated function for pruning, which may improve the accuracy of the single decision tree; or extending the bootstrap aggregated function to also be able to handle random forests, where subsets of the explanatory features are selected at each split point. When applied to a remote sensing learning problem, it was shown to perform strongly, correctly classifying validation data into a number of different classes, using numerous continuous explanatory variables.

## References

Breiman, L., 1996. Bagging predictors. *Machine Learning 24*, 123–140.

Breiman, L., 1984. *Classification and regression trees*. Chapman & Hall/CRC, New York, N.Y.

Mitchell, T.M., 1997. *Machine Learning*, McGraw-Hill series in computer science. McGraw-Hill, New York.

# Appendix

## DecisionTree.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed May  9 23:20:03 2018
Decision Tree classifier, used to classify datasets with any number of continuous attributes.


@author:Samuel Oswald
"""
##Import numpy for management of arrays.
import numpy as np

"""Build decision tree. data refers to the training dataset.
max_depth refers to how deep the tree can get. min_size is the minimum
amount of samples before a leaf node must be classified."""
def dt_train( data, max_depth, min_size = 1):
    max_depth = int(max_depth)
    min_size = int(min_size)
    attr, split_val, left, right = split(data)
    tree = {"attribute": attr, "split": split_val, "left": left, "right": right, "current_mode": leaf(data)}
    decision(tree,max_depth,min_size)
    return tree

def gini(node):
    """Calculate the gini impurity for a node. Aim is to minimize gini impurity(gain function)."""
    ##Find the number of classifications in current node.
    classifications = node[:,-1]
    samples = classifications.size
```

```
    unique, counts = np.unique(classifications, return_counts = True)
    ##calculate gini based on number of classes
    gini = 1
    for i in range (0, unique.size):
        proportion =  counts[i] / samples
        ##
        gini = gini - proportion * proportion
    return gini

def gain(values, cur_gini, attribute, split):
    """Calculate information gain for an attribute split at each level.
    Inputs are the current subset of data, initial gini at parent node,
    attribute to be split and split number."""
    i = attribute
    samples = values[:,-1].size
    left = values[values[:,i] < split, :]
    right = values[values[:,i] >= split, :]
    left_samples = left[:,-1].size
    right_samples = right[1:,-1].size

    ##Calculate left and right side gini
    left_gini = gini(left)
    right_gini = gini(right)

    ##Calculate information gain at this split value.
    gain = cur_gini - (left_samples/samples)*left_gini - (right_samples/samples)*right_gini
    return gain, left, right

def split(node):
```

```
    """Find the ideal split point by searching for the best information gain
    of all attributes and their potential split values.
    If no gain improves, node is split for leaf node creation as right side left at 0 samples."""
    cur_gini = gini(node)
    best_gain = 0
    best_attr = 0
    best_split = 0
    ##Implement greedy, exhaustive search for best information gain
    variables = len(node[0])
    best_left = node
    best_right = np.empty([0,variables])

    ##Seach through each unique value to find best division
    for v in range(0, variables-1):
        uniques = np.unique(node[:, v])
        for row in uniques:
            new_gain, left, right  = gain(node, cur_gini, v, row)

            ##Select the best gain, and associated attributes
            if new_gain > best_gain:
                best_gain = new_gain
                best_attr = v
                best_split = row
                best_left = left
                best_right = right
    #return {"attribute": best_attr, "split": best_split, "left": best_left, "right": best_right}
    return best_attr, best_split, best_left, best_right

def leaf(node):
```

```
    """Return classification value for leaf node,
    when either maximum depth of tree reached or node is suitably weighted to one class."""
    classes = node[:, -1].tolist()
    return max(set(node[:,-1]), key = classes.count)


def decision(tree, max_depth=10, min_size=0, depth=0):
    """Uses split and leaf functions to build a tree, using a root data set.
    Will assign leaf nodes if either maximum depth or minimum samples are reached.
    root node contains both current node data, as well as decision rules to that point.
    """
    left = tree["left"]
    right = tree["right"]

    ##If tree is at max depth, assign most common member.
    if depth >= max_depth:
        tree['left'] = leaf(left)
        tree['right'] = leaf(right)
        return
    ##If continuing sampling
    else:

        ##Left side child
        ##If minimum samples exist in current node, make it a leaf with max occuring value in samples.
        if left[:, -1].size <= min_size:
            tree['left'] = leaf(left)
        ##Else continue building tree.
        else:
            left_attr, left_split, left_left, left_right = split(left)
            ##Check if node is terminal. Make it a leaf node if so.
```

```
        if left_left.size == 0 or left_right.size == 0:
            tree['left'] = leaf(np.vstack([left_left,left_right]))
        ##Continue elsewise.
        else:
            tree['left'] = {"attribute": left_attr, "split": left_split, "left": left_left, "right": left_right,
"current_mode": leaf(left)}
            decision(tree['left'], max_depth, min_size, depth+1)

    ##right side child. Same process as above.
    if right[:, -1].size <= min_size:
        tree['right'] = leaf(right)
    else:
        right_attr, right_split, right_left, right_right = split(right)
        if right_left.size == 0 or right_right.size == 0:
            tree['right'] = leaf(np.vstack([right_left,right_right]))
        else:
            tree['right'] = {"attribute": right_attr, "split": right_split, "left": right_left, "right":
right_right, "current_mode": leaf(right)}
            decision(tree['right'], max_depth, min_size, depth+1)

def classify(tree,row):
    """classify new data based on current row.
    Involves searching through tree based on the attributes of validation data.
    Will return classification value once leaf of tree is reached."""
    ##Look at each sample to classify. append to list of output values.
    ##Recursively search through branches until an append can be made.
    if row[tree['attribute']] < tree['split']:
        if isinstance(tree['left'],dict):
            return classify(tree['left'], row)
        else:
```

```
            return tree['left']
        else:
            if isinstance(tree['right'],dict):
                return classify(tree['right'], row)
            else:
                return tree['right']


def dt_predict( tree, data):
    """For every row in the validation data,
    a call to the classify function is done,
    with results appended to prediction data."""
    predictions = []
    for row in data:
        pred = classify(tree, row)
        predictions.append(int(pred))
    return predictions


##functions for validation and pruning.
def dt_confusion_matrix( predicted, actual,classes):
    """Return a confusion matrix showing the difference between actual values,
    and model predicted values. Also returns total accuracy"""

    matrix = np.zeros((len(classes), len(classes)))
    for a, p in zip(actual, predicted):
        matrix[a][p] += 1
    accuracy = (actual == predicted).sum() / float(len(actual))*100
    return matrix, accuracy


"""Bagged decision trees contain a user-specified number of decision trees.
```

Classification of a sample is done by using the mode of each of these decision trees.

subsample is a fraction of the total dataset to be used.

trees refers to the number of trees to use in "forest" of trees.

By leaving default values for subsample and trees, a single decision tree classifier is created."""

```
def bt_train( data, max_depth, min_size = 1, subsample_ratio = 1,trees =1):


    ##Create a series of trees using sampling with replacement.
    size = data[:, -1].size
    division = int(size * subsample_ratio)
    forest = []
    for i in range (0,trees):
        samples = data[np.random.choice(data.shape[0], division, replace = True)]
        forest.append([])
        forest[i] = dt_train(samples, max_depth, min_size)
    return forest


def bt_predict( forest, data):
    """Classify validation data set based on built bagged trees.
    This is done by taking the mode of the classifications of each decision tree."""
    ##Use predict function from decision tree.
    ##Number of trees in forest, number of validation samples. Used to create empty array showing
classifications.
    forest_size = len(forest)
    samples = len(data)
    tree_classification = np.zeros((samples, forest_size))
    ##With each tree, find the classification of each validation sample.
    for i in range (0, forest_size):
        tree_classification[:, i] = dt_predict(forest[i], data)
    ##Create list of modes for each sample, using tree_classification matrix.
```

```
    predictions = []
    for i in range(0, samples):
        tree_pred = tree_classification[i,:].tolist()
        predictions.append(int(max(set(tree_pred), key = tree_pred.count)))
    return predictions


def bt_confusion_matrix( predicted, actual,classes):
    """Create confusion matrix for bagged trees. Makes call to DT method."""
    matrix, accuracy = dt_confusion_matrix(predicted, actual, classes)
    return matrix, accuracy
```

## Run.py

```
    # -*- coding: utf-8 -*-
"""
Created on Thu May 17 11:16:26 2018

##Implement the decision tree and bagged tree classifiers.

@author: Samuel Oswald
"""
##Import necessary classes. Decision Tree is custom class, numpy used for array processing,
##pandas used to manage overall data.
import DecisionTree as dt
import DataManagement as dm
import numpy as np

##Prep data. Index -1 assumed to be the dependent variable.
##Continuous data assumed for independent variables.
##Alter as needed for dataset.
```

```
data_folder = "Data/"
#image_folder = "Img/"

train_name = "APEX_training.csv"
test_name = "APEX_testing.csv"
#filename = "Social_Network_Ads_Cont.csv"
#image_file = "APEX.hdr"
##Prep data using functions at bottom
train, test, data = dm.data_prep_two_files(data_folder, train_name, test_name, ",")
##prep data using single file
#train, test, data = dm.data_prep_single_file(data_folder, filename, ";")

#Regularize APEX dataset as it starts 1 index 1, not 0.
train[:,-1] = train[:,-1] - 1
test[:,-1] = test[:,-1] - 1
data[:,-1] = data[:,-1] - 1

##Decision tree classes(used for confusion matrix, automated in other processes)
classes = np.unique(data[:,-1]-1).astype(int) ##Begin with class 0 for ease of indexing, if given classes start with 1.
#classes = np.unique(data[:,-1]).astype(int) ##Begin with class 0 for ease of indexing, if given classes start with 1.


##Decision Tree Building
tree = dt.dt_train(train, 20,10)
validation_dt = dt.dt_predict(tree, test)
confusion_dt,accuracy_dt = dt.dt_confusion_matrix(validation_dt, test[:, -1].astype(int), classes)
```

```
forest = dt.bt_train(train, 20, 10, 0.1, 150)
validation_rf = dt.bt_predict(forest, test)
confusion_rf, accuracy_rf = dt.bt_confusion_matrix(validation_rf, test[:, -1].astype(int),classes)


##Apply learned decision tree to classification of imagery.
##matplotlib to plot, spectral to manipulate hyperspectral imagery.
#import matplotlib.pyplot as plt
#from spectral import io
#
#image = io.envi.open(image_folder + image_file)[:,:,:]
## plt.imshow(image[:,:, 0:3])
#shp = image.shape
#image_array = np.reshape(image, (shp[0] * shp[1], shp[2]))
#
###Make predictions
#dt_predict = dt.dt_predict(tree, image_array)
#bt_predict = dt.bt_predict(forest, image_array)
#
###Reshape and show predicted values
#dt_classified = np.asarray(dt_predict).reshape(shp[0], shp[1])
#bt_classified = np.asarray(bt_predict).reshape(shp[0], shp[1])
#
#plt.imshow(dt_classified)
#plt.show()
#plt.imshow(bt_classified)
#plt.show()
```

## DataManagement.py

```
# -*- coding: utf-8 -*-
"""
```

```
Created on Tue May 22 16:25:05 2018

@author: SAM
"""

import pandas as pd
import numpy as np

def data_prep_single_file(data_folder,filename, delimiter = ",", data_split = 0.8):
    raw = pd.read_csv(data_folder + filename, delimiter = delimiter)
    if raw.iloc[:,-1].dtype != 'int64':
        raw_classes = raw.iloc[:,-1].unique()
        class_key = {}
        for i in range (0, len(raw_classes)):
            class_key[raw_classes[i]] = i
        ##Change classifications to ints
        raw_class = raw.iloc[:,-1]
        raw_name = raw_class.name##To use pandas replaces
        #raw_length = len(raw_class)
        raw_classified = raw.replace({raw_name:class_key})
        raw_train = raw_classified.sample(frac=0.8,random_state=40)
        raw_test = raw_classified.drop(raw_train.index)
        data = raw_classified.values
        train = raw_train.values
        test = raw_test.values
    else:
        raw_train = raw.sample(frac=data_split, random_state=40)
        raw_test = raw.drop(raw_train.index)
        data = raw.values
```

```
        train = raw_train.values
        test = raw_test.values
    return train, test, data


def data_prep_two_files(data_folder,train_name, test_name, delimiter = ","):
    raw_train = pd.read_csv(data_folder + train_name, delimiter = delimiter)
    raw_test = pd.read_csv(data_folder + test_name, delimiter = delimiter)
    if raw_train.iloc[:,-1].dtype != 'int64':
        raw_classes = raw_train.iloc[:,-1].unique()
        class_key = {}
        for i in range (0, len(raw_classes)):
            class_key[raw_classes[i]] = i
        ##Change classifications to ints
        raw_class = raw_train.iloc[:,-1]
        raw_name = raw_class.name##To use pandas replaces
        #raw_length = len(raw_class)
        raw_train_classified = raw_train.replace({raw_name:class_key})
        raw_test_classified = raw_train.replace({raw_name:class_key})
        train = raw_train_classified.values
        test = raw_test_classified.values
        data = np.append(train, test, axis = 0)
    else:
        train = raw_train.values
        test = raw_test.values
        data = np.append(train, test, axis = 0)
    return train, test, data
```