Decision Trees

References

- https://en.wikipedia.org/wiki/Decision_tree_learning (https://en.wikipedia.org/wiki/Decision_tree_learning)
- ESL II, 8.7, 9.2, 15

Intuitive pictures

For typical regression problem, the final results split the x-axis into many regions, the value of each region is taken into the mean within the region. This is just like fitting a piecewise function (thus often we heard of spline fitting) to the data. Another intuitive example is the 2D case as in the book. Note this piecewise splitting is obtained by many times of minimization.

A brief description on how to obtain the above regression result by a greedy algorithm (called greedy because we first consider the whole region?). First split in position s for feature j, denoted by a split of (j, s). We can calculate the squared loss for this split by taking averages in two split regions (for regression). Then we move s along the j-axis to obtain different splits. That is different pairs of (j, s). Finally, we choose a pair of (j, s) which gives the minimum squared loss. This finished our first split on the whole region. Note in the above process, both j and s are varied to achieve a minimum loss. Later we need repeat this process in the sub regions until some conditions met. In summary, the above is a brief description of 'growing a regression tree' with an error metric of squared error. In each sub-region, we achieve the minimum error by taking the mean as the value for region.

To grow a classification tree, the idea is similar except we are going to use a different error metric. For example, we may use impurity metric such classification error, entropy or Gini index (see details later).

More intuitive pictures to understand decision trees and ensembling or aggregating can be found in the link https://www.youtube.com/channel/UCcAtD_VYwcYwVbTdvArsm7w (https://www.youtube.com/channel/UCcAtD_VYwcYwVbTdvArsm7w) M2.1 - 2.8. In around 2.7, it describes how aggregated estimator is better even we don't assume unbiased estimator, and don't assume the existence of deterministic true function.

The decision tree model is often used in the medical field. Imagine how the tree growing process introduced above can be used into a 2D data on tumor classification.

Metrics

Algorithms for constructing decision trees usually work top-down, by choosing a variable at each step that best splits the set of items. Different algorithms use different metrics for measuring "best". These metrics are applied to each candidate subset, and the resulting values are combined (e.g., averaged) to provide a measure of the quality of the split. We use squared error for regression problems. The following metrics are for classification problems.

The mis-classification error

In a sub-region, We calculate the misclassification rate by using majority vote. That is, calculate the ratio of misclassified cases to the total data in this region. We choosing the split line, we always try to minimize this rate.

Entropy

A more commonly used error metrics is entropy. When a sub-region is perfectly classificed, then the entropy is zero. The more we made mistakes in the classification, i.e., the more impure, then the more the entropy. See Sampling_Estimation_HypothesisTest_Entropy.pdf in the /statistics folder for details.

Gini impurity

Used by the CART (classification and regression tree) algorithm for classification trees, Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. Details see wikipedia. The key definition is $\sum_{i=1}^{J} p_i(1-p_i)$. From this definition, when a subset is perfectly classified (or completely ordered), then Gini impurity is zero. This is similar to the case of using entropy. This metric has non-zero value only when there is some disorder, or when the subset is not perfectly classified.

Information gain

Used by the ID3, C4.5 and C5.0 tree-generation algorithms. Information gain is based on the concept of entropy and information content from information theory. The key piece of definition is $\sum_{i=1}^{J} p_i \log_2 p_i$. This is actually similar to Gini impurity metric, which will have zero value when the sub is perfectly classified, or completely ordered. **Comments** The information refers to knowledge, ordering. This gain of ordering (more homogeneity in a subset), or the loss of cross entropy (metric for disordering) should always be pursued when doing splitting. See statistics notes and compare entropy with cross-entropy.

Variance reduction

Introduced in CART,variance reduction is often employed in cases where the target variable is continuous (regression tree). The variance reduction of a node N is defined as the total reduction of the variance of the target variable x due to the split at this node.

Decision-tree algorithms

- ID3 (Iterative Dichotomiser 3)
- C4.5 (successor of ID3)
- CART (Classification And Regression Tree)
- CHAID (CHi-squared Automatic Interaction Detector). Performs multi-level splits when computing classification trees.
- MARS: extends decision trees to handle numerical data better.
- Conditional Inference Trees. Statistics-based approach that uses non-parametric tests as splitting criteria, corrected for multiple testing to avoid overfitting. This approach results in unbiased predictor selection and does not require pruning.

Implementations

Many **data mining** software packages provide implementations of one or more decision tree algorithms. Examples include Salford Systems CART (which licensed the proprietary code of the original CART authors), IBM SPSS Modeler, RapidMiner, SAS Enterprise Miner, Matlab, R (an open-source software environment for statistical computing, which includes several CART implementations such as rpart, party and randomForest packages), Weka (a free and open-source data-mining suite, contains many decision tree algorithms), Orange, KNIME, Microsoft SQL Server, and scikit-learn (a free and open-source machine learning library for the Python programming language).

Time complexity for binary classification

Using the simple linear regression model above and a binary tree to understand the time complexity for the case of n samples, f feature, d depth,

- Test time: $O(d)$, or $O(logn)$ if balanced.
- Train time: each point in $O(d)$ nodes, costs $O(f)$ at each node. So the train time complexity is $O(nfd)$, where nf is the size of design matrix and d often $logn$.

Comments:

- Note the above complexity analysis is for binary classification.
- In a binary tree, if tree depth is d, then the number of nodes should be $2^d - 1$. How can we have $O(d)$ nodes?

- Using the picture above, each node, or a region, we have n data points...

Advantages

- Simple to understand and interpret.
- fast.
- **Can use categorical variable but without creating dummy variables.**
- Able to handle both numerical and categorical data.
- Requires little data preparation. **No need for data normalization. No need to create dummy variables.**
- Uses a white box model. If a given situation is observable in a model the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model, the explanation for the results is typically difficult to understand, for example with an artificial neural network.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Non-statistical approach that makes no assumptions of the training data or prediction residuals; e.g., no distributional, independence, or constant variance assumptions
- Performs well with large datasets. Large amounts of data can be analyzed using standard computing resources in reasonable time.
- Mirrors human decision making more closely than other approaches. This could be useful when modeling human decisions/behavior.
- **Robust against co-linearity, particularly boosting.**
- In built feature selection. Additional irrelevant feature will be less used so that they can be removed on subsequent runs.
- Decision trees can approximate any Boolean function eq. XOR.

Limitations

- Trees can be very non-robust. A small change in the training data can result in a large change in the tree and consequently the final predictions. Note however, these problems can be solved by ensembling.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristics such as the greedy algorithm where locally optimal decisions are made at each node. **Such algorithms cannot guarantee to return the globally optimal decision tree**. To reduce the greedy effect of local optimality, some methods such as the dual information distance (DID) tree were proposed.
- Single decision tree without ensembles give low bias and high variance.
- Decision-tree learners can create over-complex trees that do not generalize well from the training data. (This is known as overfitting.) Mechanisms such as pruning are necessary to avoid this problem (with the exception of some algorithms such as the Conditional Inference approach, that does not require pruning).

- For data including categorical variables with different numbers of levels, information gain in decision trees is **biased in favor of attributes with more levels**. However, the issue of biased predictor selection is avoided by the Conditional Inference approach, a two-stage approach, or adaptive leave-one-out feature selection.
- Bad at additive. Figure this out.

Question In cs229 notes, an example showing the North, south, hemisphere, equator....For 9 categories, 2^9 splits possible so does not scale well, except for binary classification case.

Regularization of decision trees

Decision trees, if fully grown, are with high variance and low bias. Use heuristics for regularization.

- Min leaf size.
- max depth.
- Max number of nodes.
- Min decrease in loss --dangerous since might be higher order interactions.
- Pruning -- use validation set, measure mis-classification or squared error. See section 9.2 of EML II.

Ensemble methods

General

Ensemble methods contains not just the popular bagging, but also boosting.

- Bagging
- Boosting Ensemble methods are actually a general approach which can be applied to other machine learning algorithms. Decision tree algorithms normally have a high variance but low bias and thus particularly suitable to apply ensemble methods such as bootstrap aggregating (bagging). **Bagging help suppress variance, while boosting reduce bias of an algorithm.**

Mathematical background for bagging

See details in Sampling_Estimation_HypothesisTest_Entropy.pdf in the mathematics/statistics/ folder.

When using iid bootstrapped samples to aggregate, the newly obtained random variable will have a smaller variance as it is inversely proportional to the number of samples.

$$Var(\bar{X}) = Var(\frac{\sum_{i=1}^{n} X_i}{n}) = \frac{1}{n^2} Var(\sum_{i=1}^{n} X_i) = \frac{n\sigma^2}{n^2} = \frac{\sigma^2}{n}$$

Even the samples drawn is not purely independent, but with an average correlation p, then the variance will still be reduced.

$$Var(\bar{X}) = p\sigma^2 + \frac{1-p}{n}\sigma^2$$

Because only unbiased estimator will have its sample average approach its mean, so it is desirable the estimator used is unbiased. However, even for biased estimators, bagging can help reduce the variance.

The reduction of variance means better generalization of machine learning algorithm after using bagging techniques.

Ways to ensemble

- Use different algorithms
- Use different training sets
- Bagging (bootstrap aggregating), random forest.
- Boosting (Ada boost, xgboost).

In ensemble methods, particularly when we do re-sampling, we need do this in the training set, **but not cross-validation set.** Typically we uniformly draw samples uniformly from the original data set.

Bagging of decision tree

- Bootstrap aggregated decision trees, an early ensemble method, builds multiple decision trees by repeatedly resampling training data with replacement, and voting the trees for a consensus prediction.

Benefits of bagging on decision tree

- Variance reduction (even more so for random forest due to the more independent samples).
- If a feature is missing simply don't use trees in the ensemble that contain that feature. Missing feature -> ignore the aggregated predictor's splitting on it. So deal with missing values nicely (even more so for Random forest).

- Out of bag estimation (OOB), free validation set. On average, Z (aggregated predictor) will contain 2/3 of samples. The remaining 1/3 is used to test error. In the limit of infinite number of samples, OOB gives equivalent results to LOOCV.

Downside of bagging

- Slight increase in bias (even more so for RF).
- Harder to interpret
- Still not additive (what does this mean?).
- More expensive.

Variable importance measure

Do lose some interpretability
For each feature, find each split that uses it in the ensemble, measure decrease in loss, average. Note does not measure degradation in perf it didn't have feature, since other features might be able to substitute.

Random forest

See notes later.

Boosting of decision tree

General

Bagging is variance reducing while boosting is bias reducing. Therefore, boosting is suitable for high bias and low variance models, weak learner. In terms of decision trees, just use decision stump.
A typical example is Ada (adaptive) Boosting and Gradient boosting.
Check these two web sites for the introduction. https://www.youtube.com/watch?v=sRktKszFmSk&t=314s (https://www.youtube.com/watch?v=sRktKszFmSk&t=314s) and https://www.youtube.com/watch?v=GM3CDQfQ4sw (https://www.youtube.com/watch?v=GM3CDQfQ4sw). The boosting can be thought as variation of bagging. Also XGBoost is a highly

optimized implementation of gradient boosted decision trees. It is popular because it is being used by some of the best data scientists in the world to win machine learning competitions. Check the link https://machinelearningmastery.com/start-here/#xgboost (https://machinelearningmastery.com/start-here/#xgboost) for some gentle introduction.

Comments

- The process of gradient boosting is just like the normal gradient descent approach applied in many other algorithms such as linear regression, logistic regression.

Adaptive boosting (Ada boost)

- Additive - combine together prediction of many weak models. Often times beats single strong model. **Comments: Compared to bagging, why the later is not additive?**.
- No longer independent of previous models in sequence, and thus can overfit.
- More general framework: forward stagewise Additive Modeling (FSAM). Though derived on its own, Ada boost can be thought of as special case of FSAM. Ada boost is FSAM with exponential loss and 2-class classification. Proof in EML.
- For squared loss for regression $L(y,f(x)) = (y - f(x))^2$, FSAM is same as fitting individual tree to residual $y - f_{m-1}(x)$.

Gradient boosting

- For more general losses, cannot always write out nice closed form solution for minimization problem. Turn to numerical optimization (basis of xgboost). Find an example on this in the future.
- One way is to take derivative, do gradient descent. But restricted to taking steps that are in model class.

Advantages

- Decreased bias.
- very good accuracy.

Downside

- Increased variance.
- Prone to overfitting.

Implementation of Random Forest

Introduction

- Decision trees can suffer from high variance -> Bootstrap aggregation (bagging) can reduce the variance but the trees are highly correlated -> Random Forest is an extension of bagging that in addition to building trees based on multiple samples of your training data, it also constrains the features that can be used to build the trees, forcing trees to be more independent. This in turn give a lift in performance. In other words, random Forest prevents overfitting most of the time, by creating random subsets of the features and building smaller trees using these subsets.
- **Comments:** Random forests suppress overfitting by using more independent sub-trees. This seems like the 'knock-out' regularization approach in deep neural network.

https://machinelearningmastery.com/implement-random-forest-scratch-python/ (https://machinelearningmastery.com/implement-random-forest-scratch-python/)

Calculating Splits

- In a decision tree, split points are chosen by finding the attribute and the value of that attribute that results in the lowest cost. For classification problems, this cost function is often the Gini index, that calculates the purity of the groups of data created by the split point.
- For bagging and random forest, this procedure is executed upon a sample of the training dataset, made with replacement. Sampling with replacement means that the same row may be chosen and added to the sample more than once.
- We can update this procedure for Random Forest. Instead of enumerating all values for input attributes in search of the split with the lowest cost, we can create a sample of the input attributes to consider. **Comments: in SGD or min-batch GD we calculate cost with random sample of observations (rows), here in random forest, we use random sample of features (columns)?**
- This sample of input attributes can be chosen randomly and **without replacement (different from the replacement of rows mentioned before)**, meaning that each input attribute needs **only be considered once** when looking for the split point with the lowest cost. **Comments. we have many features and only consider small portion, e.g. sqrt(N), we can obtain many such samples even without replacement**.

Below is a function name get_split() that implements this procedure. It takes a dataset and a fixed number of input features to evaluate as input arguments, where the dataset may be a sample of the actual training dataset.

The helper function test_split() is used to split the dataset by a candidate split point and gini_index() is used to evaluate the cost of a given split by the groups of rows created.

We can see that a list of features is created by randomly selecting feature indices and adding them to a list (called features), this list of features is then enumerated and specific values in the training dataset evaluated as split points.

In [2]:
```
def get_split(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1)
        if index not in features:
            features.append(index)
    for index in features:
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups
    return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

Now that we know how a decision tree algorithm can be modified for use with the Random Forest algorithm, we can piece this together with an implementation of bagging and apply it to a real-world dataset.

Sonar Dataset Case Study

- The dataset is first loaded, the string values converted to numeric and the output column is converted from strings to the integer values of 0 and 1. This is achieved with helper functions load_csv(), str_column_to_float() and str_column_to_int() to load and prepare the dataset.
- Use k-fold cross validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model error. Classification accuracy will be used to evaluate each model. These behaviors are provided in the cross_validation_split(), accuracy_metric() and evaluate_algorithm() helper functions.
- Use an implementation of the Classification and Regression Trees (CART) algorithm adapted for bagging including the helper functions test_split() to split a dataset into groups, gini_index() to evaluate a split point, our modified get_split() function discussed in the previous step, to_terminal(), split() and build_tree() used to create a single decision tree, predict() to make a prediction with a

decision tree, subsample() to make a subsample of the training dataset and bagging_predict() to make a prediction with a list of decision trees.

- A new function name random_forest() is developed that first creates a list of decision trees from subsamples of the training dataset and then uses them to make predictions.
- The key difference between Random Forest and bagged decision trees **is the one small change to the way that trees are created, here in the get_split() function**.

The complete example is listed below.

In [5]:
```
# extra code for exploring the data
import pandas as pd
df = pd.read_csv('sonar.all-data.csv', header = None)
print(len(df))
df.head()
```

```
208
```

Out[5]:

	0	1	2	3	4	5	6	7	8	9	...	51	52	53	54	55	56	5
0	0.0200	0.0371	0.0428	0.0207	0.0954	0.0986	0.1539	0.1601	0.3109	0.2111	...	0.0027	0.0065	0.0159	0.0072	0.0167	0.0180	0.008
1	0.0453	0.0523	0.0843	0.0689	0.1183	0.2583	0.2156	0.3481	0.3337	0.2872	...	0.0084	0.0089	0.0048	0.0094	0.0191	0.0140	0.004
2	0.0262	0.0582	0.1099	0.1083	0.0974	0.2280	0.2431	0.3771	0.5598	0.6194	...	0.0232	0.0166	0.0095	0.0180	0.0244	0.0316	0.016
3	0.0100	0.0171	0.0623	0.0205	0.0205	0.0368	0.1098	0.1276	0.0598	0.1264	...	0.0121	0.0036	0.0150	0.0085	0.0073	0.0050	0.004
4	0.0762	0.0666	0.0481	0.0394	0.0590	0.0649	0.1209	0.2467	0.3564	0.4459	...	0.0031	0.0054	0.0105	0.0110	0.0015	0.0072	0.004

5 rows × 61 columns

In [133]:
```
# Random Forest Algorithm on Sonar Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# Another way of reading CSV is using pandas.
# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# Convert string column to integer
# Comments: This is labeling. If there are many classes, then one-hot encoding is better.
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    for i, value in enumerate(unique):
        lookup[value] = i
    for row in dataset:
        row[column] = lookup[row[column]]
    return lookup


# The split below does  not consider the distribution of classes. If the class is skewed, then we need consider t
# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
```

```
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index)) #Note the pop() here.
        dataset_split.append(fold)

    return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
# scores = evaluate_algorithm(dataset, random_forest, n_folds, max_depth, min_size, sample_size, n_trees, n_featu
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, []) #Effectively, this reduces the nested list by one fold.

        # Use the following code to test the above function. Or use VS code to check in place.
        # train_set =[[[1,2,3],[4,5,6],[7,8,9]],[[1,2,3],[4,5,6],[7,8,9]]]
        # print(len(train_set))
        # print(train_set)
        # train_set = sum(train_set, [])
        # print(len(train_set))
        # print(train_set)
        # print(train_set[0])

        test_set = list()
        for row in fold:

            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
```

```
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores
```

In [138]:
```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right #this returns a tuple with two big elements, each of which is a nested list.

# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
    # count all samples at split point
    n_instances = float(sum([len(group) for group in groups]))
    # sum weighted Gini index for each group

    gini = 0.0
    for group in groups:
        size = float(len(group))
        # avoid divide by zero
        if size == 0:
            continue
        score = 0.0
        # score the group based on the score for each class
        for class_val in classes:
            p = [row[-1] for row in group].count(class_val) / size
            score += p * p
        # weight the group score by its relative size
        gini += (1.0 - score) * (size / n_instances)
    return gini

# Select the best split point for a dataset
def get_split(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1) #Note it is 'dataset[0]' but not 'dataset'
        if index not in features:
            features.append(index)

    for index in features:
```

```
    #feature index but not sample index. Note usually we only use index for samples and column names for features
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            #groups is a tupe, with two elements left and right.
            #(left[[e1,e2,.], [e3,e4,.],...],  right[[e1,e2,.], [e3,e4,.],...])
            #test_split() splits at a data value of a feature described by 'index' and in the row described by 'r

            gini = gini_index(groups, class_values)
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups
    return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

In [180]:

```
# Create a terminal node value
def to_terminal(group):
    outcomes = [row[-1] for row in group]
    #if max_depth is small, e.g. 2, then outcomes can be like [1,1,1,0,0,0,0,0]
    #Then we should return the element whose frequency in the list should be largest.
    #However, the original code to do this is"
    #return max(set(outcomes), key=outcomes.count).
    #This should be wrong because of the set(). set() will transform the list above to be {0,1}.
    #The correct code should be:
    return max(outcomes, key=outcomes.count)
    # When max_depth is small (e.g. 2), two versions of the 'return' will give different results.
    # When max_depth is large (e.g. 10), then two versions of 'return' will give same result. But this should co
    # See the test code in the end of this note.

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, n_features, depth):

    left, right = node['groups'] #node['groups] is a tuple.
    del(node['groups'])

    # check for a no split
    if not left or not right: #check if list 'left' or 'right' is emplty
        # Nomrally left and right will not be empty.
        node['left'] = node['right'] = to_terminal(left + right)
        return

    # check for max depth
    if depth >= max_depth:
        node['left'], node['right'] = to_terminal(left), to_terminal(right)
        return

    # process left child
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left, n_features)
        split(node['left'], max_depth, min_size, n_features, depth+1)


    # process right child
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
```

```
    else:
        node['right'] = get_split(right, n_features)
        split(node['right'], max_depth, min_size, n_features, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size, n_features):
    root = get_split(train, n_features)
    #root is a four-element dictionary. Its element root["groups"] has same format of 'groups' returned by get_sp

    split(root, max_depth, min_size, n_features, 1)
#     print(root) #This time root is different from that of running split() before. But still with four element
    return root

# Make a prediction with a decision tree
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

# Create a random subsample from the dataset with replacement
def subsample(dataset, ratio):
    sample = list()
    n_sample = round(len(dataset) * ratio)
    while len(sample) < n_sample:
        index = randrange(len(dataset))
        sample.append(dataset[index])
    return sample

# Make a prediction with a list of bagged trees
def bagging_predict(trees, row):
    predictions = [predict(tree, row) for tree in trees]
#     return max(set(predictions), key=predictions.count)
    return max(predictions, key=predictions.count)

# Random Forest Algorithm
def random_forest(train, test, max_depth, min_size, sample_size, n_trees, n_features):
```

```
    trees = list()
    for i in range(n_trees):
        sample = subsample(train, sample_size)
        #If sample_size = 1, then sample size is almost the size of train set size.

        tree = build_tree(sample, max_depth, min_size, n_features)
        #n_features here has already been taken as the square root of total number of features.
        trees.append(tree)
#         print(tree)
#         print(len(tree))

    predictions = [bagging_predict(trees, row) for row in test]
    return(predictions)
```

In [181]:
```
# Test the random forest algorithm
seed(2)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
# convert string attributes to integers
for i in range(0, len(dataset[0])-1):
    str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)

# evaluate algorithm
n_folds = 5
max_depth = 10 #10 Lower the value for clear visualization
min_size = 1
sample_size = 1.0
n_features = int(sqrt(len(dataset[0])-1)) # number of features, but not number of samples/observations.

for n_trees in [1,5,10,20,30, 40, 50, 60]:
    #when n_trees is very big, then the trees will be correlated, as the number of features is fixed.
    scores = evaluate_algorithm(dataset, random_forest, n_folds, max_depth, min_size, sample_size, n_trees, n_fea
    print('Trees: %d' % n_trees)
    print('Scores: %s' % scores)
    print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

```
Trees: 1
Scores: [56.09756097560976, 63.41463414634146, 60.97560975609756, 58.536585365853654, 73.17073170731707]
Mean Accuracy: 62.439%
Trees: 5
Scores: [70.73170731707317, 58.536585365853654, 85.36585365853658, 75.60975609756098, 63.41463414634146]
Mean Accuracy: 70.732%
Trees: 10
Scores: [80.48780487804879, 78.04878048780488, 92.6829268292683, 70.73170731707317, 68.29268292682927]
Mean Accuracy: 78.049%
Trees: 20
Scores: [75.60975609756098, 70.73170731707317, 80.48780487804879, 78.04878048780488, 80.48780487804879]
Mean Accuracy: 77.073%
Trees: 30
Scores: [82.92682926829268, 70.73170731707317, 80.48780487804879, 82.92682926829268, 85.36585365853658]
Mean Accuracy: 80.488%
Trees: 40
Scores: [65.85365853658537, 92.6829268292683, 75.60975609756098, 80.48780487804879, 85.36585365853658]
```

```
Mean Accuracy: 80.000%
Trees: 50
Scores: [78.04878048780488, 90.2439024390244, 78.04878048780488, 87.8048780487805, 73.17073170731707]
Mean Accuracy: 81.463%
Trees: 60
Scores: [73.17073170731707, 87.8048780487805, 87.8048780487805, 87.8048780487805, 80.48780487804879]
Mean Accuracy: 83.415%
```

Extensions

This section lists extensions to this tutorial that you may be interested in exploring.

Algorithm Tuning. The configuration used in the tutorial was found with a little trial and error but was not optimized. Experiment with larger numbers of trees, different numbers of features and even different tree configurations to improve performance. More Problems. Apply the technique to other classification problems and even adapt it for regression with a new cost function and a new method for combining the predictions from trees.

Miscellaneous function

In [177]:
```
def sumDigit(num):
    sum = 0
    while(num):
        sum += num % 10
        num = int(num / 10)
    return sum

# using max(arg1, arg2, *args, key)
print('Maximum is:', max(100, 321, 267, 59, 40, key=sumDigit))

# using max(iterable, key)
num = [15, 300, 2700, 821, 52, 10, 6]
print('Maximum is:', max(num, key=sumDigit))
```

```
Maximum is: 267
Maximum is: 821
```

In [179]:

```
outcomes1= [0,0,0,0,1,1]
outcomes2= [1,1,1,1,0,0]

a = max(outcomes1, key=outcomes1.count)
b = max(outcomes2, key=outcomes2.count)

print('a and b are respectively',a, b)
```

```
a and b are respectively 0 1
```