Introduction of Random forest algorithm

Overview:

One of the most popular methods or frameworks used by data scientists at the Rose Data Science Professional Practice Group is Random Forests. The Random Forests algorithm is one of the best among classification algorithms - able to classify large amounts of data with accuracy.

Random Forests are an ensemble learning method (also thought of as a form of nearest neighbor predictor) for classification and regression that construct a number of decision trees at training time and outputting the class that is the mode of the classes output by individual trees (Random Forests is a trademark of Leo Breiman and Adele Cutler for an ensemble of decision trees).

Random Forests are a combination of tree predictors where each tree depends on the values of a random vector sampled independently with the same distribution for all trees in the forest. The basic principle is that a group of "weak learners" can come together to form a "strong learner". Random Forests are a wonderful tool for making predictions considering they do not overfit because of the law of large numbers. Introducing the right kind of randomness makes them accurate classifiers and regressors.

Single decision trees often have high variance or high bias. Random Forests attempts to mitigate the problems of high variance and high bias by averaging to find a natural balance between the two extremes. Considering that Random Forests have few parameters to tune and can be used simply with default parameter settings, they are a simple tool to use without having a model or to produce a reasonable model fast and efficiently.

Random Forests are easy to learn and use for both professionals and lay people

- with little research and programming required and may be used by folks without a strong statistical background. Simply put, you can safely make more accurate predictions without most basic mistakes common to other methods.

The Random Forests algorithm was developed by Leo Breiman and Adele Cutler. Random Forests grows many classification trees. Each tree is grown as follows:

1. If the number of cases in the training set is N, sample N cases at random - but with replacement, from the original data. This sample will be the training set for growing the tree.

2. If there are M input variables, a number mM is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing.

3. Each tree is grown to the largest extent possible. There is no pruning.

How random forest works:

https://www.youtube.com/watch?v=loNcrMjYh64

The algorithm works as follows: for each tree in the forest, we select a bootstrap sample from S where S (i) denotes the i-th bootstrap. We then learn a decision-tree using a modified decision-tree learning algorithm. The algorithm is modified as follows: at each node of the tree, instead of examining all possible feature-splits, we randomly select some subset of the features $f \subseteq F$. where F is the set of features. The node then splits on the best feature in f rather than F. In practice f is much, much smaller than F. Deciding on which feature to split is oftentimes the most computationally expensive aspect of decision

tree learning. By narrowing the set of features, we drastically speed up the learning of the tree.

pseudocode:

Precondition: A training set S := (x1, y1), . . . ,(xn, yn), features F, and number of trees in forest B.

```
Function RandomForest(S , F)
    H ← ∅
    for i ∈ 1, . . . , B do
        S (i) ← A bootstrap sample from S
        hi ← RandomizedTreeLearn(S (i) , F)
        H ← H ∪ {hi}
end for
return H
end function:
    At each node:
        f ← very small subset of F
        Split on best feature in f
    return The learned tree
end function
```

example:

dataset from: UCI Machine Learning repository：
https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)
name as: sonar.all-data.csv

```python
# Random Forest Algorithm on Sonar Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# 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
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

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = 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))
        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
```

```
54
55  # Evaluate an algorithm using a cross validation split
56  def evaluate_algorithm(dataset, algorithm, n_folds, *args):
57      folds = cross_validation_split(dataset, n_folds)
58      scores = list()
59      for fold in folds:
60          train_set = list(folds)
61          train_set.remove(fold)
62          train_set = sum(train_set, [])
63          test_set = list()
64          for row in fold:
65              row_copy = list(row)
66              test_set.append(row_copy)
67              row_copy[-1] = None
68          predicted = algorithm(train_set, test_set, *args)
69          actual = [row[-1] for row in fold]
70          accuracy = accuracy_metric(actual, predicted)
71          scores.append(accuracy)
72      return scores
73
74  # Split a dataset based on an attribute and an attribute value
75  def test_split(index, value, dataset):
76      left, right = list(), list()
77      for row in dataset:
78          if row[index] < value:
79              left.append(row)
80          else:
81              right.append(row)
82      return left, right
83
84  # Calculate the Gini index for a split dataset
85  def gini_index(groups, class_values):
86      gini = 0.0
87      for class_value in class_values:
88          for group in groups:
89              size = len(group)
90              if size == 0:
91                  continue
92              proportion = [row[-1] for row in group].count(class_value) / float(size)
93              gini += (proportion * (1.0 - proportion))
94      return gini
95
```

```
# 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)
        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}

# Create a terminal node value
def to_terminal(group):
    outcomes = [row[-1] for row in group]
    return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, n_features, depth):
    left, right = node['groups']
    del(node['groups'])
    # check for a no split
    if not left or not right:
        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(dataset, n_features)
    split(root, max_depth, min_size, n_features, 1)
    return root
```

```
str_column_to_int(dataset, len(dataset[0])-1)
# evaluate algorithm
n_folds = 5
max_depth = 10
min_size = 1
sample_size = 1.0
n_features = int(sqrt(len(dataset[0])-1))
for n_trees in [1, 5, 10]:
    scores = evaluate_algorithm(dataset, random_forest, n_folds, max_depth, min_size, sample_size, n_trees, n_features)
    print('Trees: %d' % n_trees)
    print('Scores: %s' % scores)
        print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))

```

Final result:

Trees: 1

Scores: [68.29268292682927, 75.60975609756098,

70.73170731707317, 63.41463414634146, 65.85365853658537]

Mean Accuracy: 68.780%

Trees: 5

Scores: [68.29268292682927, 68.29268292682927,

78.04878048780488, 65.85365853658537, 68.29268292682927]

Mean Accuracy: 69.756%

Trees: 10

Scores: [68.29268292682927, 78.04878048780488,

75.60975609756098, 70.73170731707317, 70.73170731707317]

Mean Accuracy: 72.683%

Advantage And Disadvantage:

Advantage*:*

- Accuracy
- Runs efficiently on large data bases
- Handles thousands of input variables without variable deletion
- Gives estimates of what variables are important in the classification
- Generates an internal unbiased estimate of the generalization error as the forest building progresses
- Provides effective methods for estimating missing data
- Maintains accuracy when a large proportion of the data are missing
- Provides methods for balancing error in class population unbalanced data sets
- Generated forests can be saved for future use on other data
- Prototypes are computed that give information about the relation between the variables and the classification.
- Computes proximities between pairs of cases that can be used in clustering, locating outliers, or (by scaling) give interesting views of the data
- Capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection
- Offers an experimental method for detecting variable interactions

Disadvantage:

- Random forests have been observed to overfit for some datasets with noisy classification/regression tasks.
- Unlike decision trees, the classifications made by random forests are difficult for humans to interpret.

- For data including categorical variables with different number of levels, random forests are biased in favor of those attributes with more levels. Therefore, the variable importance scores from random forest are not reliable for this type of data. Methods such as partial permutations were used to solve the problem.,
- If the data contain groups of correlated features of similar relevance for the output, then smaller groups are favored over larger groups
- The main limitation of the Random Forests algorithm is that a plenty of trees may make the algorithm slow for real-time prediction.
-

Application Area:

1. Internet traffic interception - certain governments (possibly from the middle east) would like to restrict certain categories of web pages. For example, due to religious restrictions, certain movie pages may be restricted/censored. This is a clear application of classification. But a hard problem, given that this must be done in real-time and based on unstructured text content. We have developed a solution for one of our clients.
2. Video classification - as and when you upload a video on youtube, the video has to be classified into appropriate categories and meta-

data added to it (annotations). Again a hard problem, given the state-of-the-art video processing and real-time nature of things.
3. Image classification - classic problem in image processing. Is this an image of a dog or a cat? Google recently has come up with a large scale training mechanism for neural networks: Large Scale Distributed Deep Networks. The same could also be used for face recognition.
4. Voice classification: classify and if possible, identify the voice. Very useful in Siri type applications.