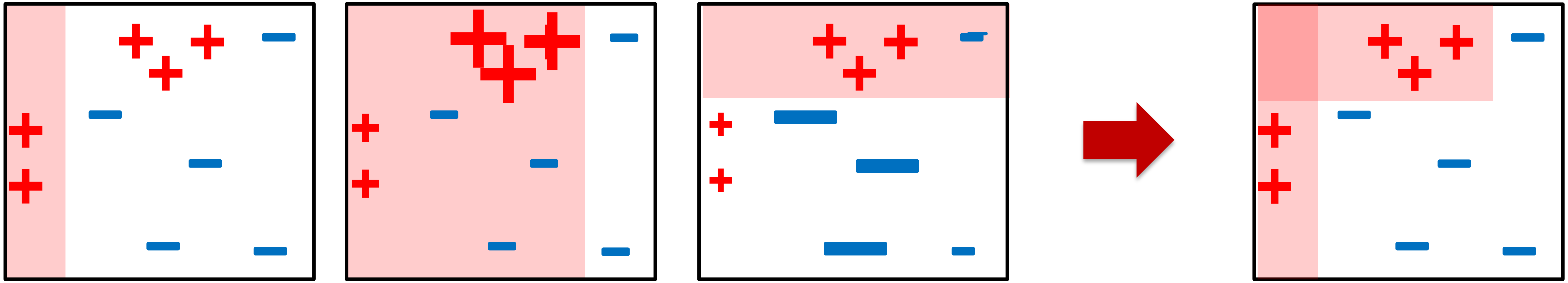
Machine Learning for Biomedical/Healthcare Applications

Combining weak classifiers....



....to make strong ones

Ensemble Learning :
Random Forests and Boosting

Dr Emma Robinson

Learning Outcomes from the tutorial

1. Understanding of how to implement Bagged ensembles of trees through
    1. Creating bootstrapped samples of a data set
    2. Training large numbers of trees
    3. Aggregating test predictions
2. Compare the performance of ensembles relative to decision Trees
3. Be able to deploy bagging, forests and boosting in Scikit-Learn
4. Be able to perform parameter optimization for ensemble methods

Solutions for decision trees

Solutions Ex 1 building a decision stump classifier from scratch

- **1.1 estimate gini index for a given split**
  1. Estimate the total number of samples by summing over the list branch
  2. Proportion in each branch equals (total number of samples in branch)/(total number of samples)
     - Total number of samples in branch is given by variable `class_total` (each item in list branch)
     - Total overall is given by output of step 1 – variable `split_size`
  3. Gini= $1 - \sum_{y_k \in Y} p(y_k)^2$

```python
def gini_coefficient(branch):

    """
        Estimates Gini Coefficient for a given class split
        input:
            split: list of length k (where k= number of classes).
                   The values at each index reflect the toal number of instances
                   of each class, for this proposed branch split

        output:
            gini: gini coefficient for this split
    """
    # 1.1.1 estimating total number of samples in branch split (by summing contents of split list)
    split_size=np.sum(branch)
    gini=1
    # iterating over all items in the array
    for class_total in branch:
        # 1.1.2. estimating p*p for this class label; subtracting from current gini total
        proportion_class_k=(class_total/split_size)
        # 1.1.3. subract the squaree of the propoertion per class from current estimate of the gininindex
        gini-=proportion_class_k*proportion_class_k

    return gini
```

Solutions Ex 1 building a decision stump classifier from scratch

- **1.2 Propose splits**
  1. check all the values of the features at that indexed position
     - Given by `row[index]` for each row (here `index` is the feature/attribute index)
  2. split the data into a left branch (if that data example's feature is below the threshold) and into a right branch (if that data example's feature is below the threshold).

```python
def test_split(index, value, dataset):
    """
        Split a dataset based on an attribute and an attribute value
        input:
            index = feature/attribute index (i.e. data column index) on which to split on
            value = threshold value (everything below this goes to left split,
                    everything above goes to right)
            dataset = array (n_samples,n_features+1)
                    rows are examples
                    last column indicates class membership
                    remaining columns reflect features/attributes of data

        output:
            left,right: data arrays reflecting data split into left and right branches, respectively
    """

    # create empty list that you will populate with rows of dataset
    left=[]
    right = []
    # the loop below will slice rows from data set
    for row in dataset:
        # if the value of this feature for this row is less than
        # the (threshold) value split into left branch, else split into right
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)

    return np.asarray(left), np.asarray(right)
```

Solutions Ex 1 building a decision stump classifier from scratch

- **1.2 Propose splits**
  - Then test for outcome of splitting on first feature (column index=6)
  - With threshold equal to 7^{th} value (or row)

```
index=0
rowindex=6
threshold=X[rowindex,index]
print('the value of the feature {} at row {} of the data set is {}'.format(index,rowindex,threshold))
```

```
the value of the feature 0 at row 6 of the data set is 9.00220326
```

```
branches=test_split(index, X[rowindex,index], X)

print('Our left branch is \n {}'.format(branches[0]))
print('Our right branch is \n {}'.format(branches[1]))
```

```
Our left branch is
 [[2.77124472 1.78478393 0.        ]
 [1.72857131 1.16976141 0.        ]
 [3.67831985 2.81281357 0.        ]
 [3.96104336 2.61995032 0.        ]
 [2.99920892 2.20901421 0.        ]
 [7.49754587 3.16295355 1.        ]
 [7.44454233 0.47668338 1.        ]
 [6.64228735 3.31998376 1.        ]]
Our right branch is
 [[ 9.00220326  3.33904719  1.        ]
 [10.12493903  3.23455098  1.        ]]
```

Solutions Ex 1 building a decision stump classifier from scratch

1.3: Estimate total cost of split

1. Iterate over class IDs (line 26)
2. Slices rows corresponding to that class (line 34)
3. Count number of rows for that slice (line 36)
4. Add these to list: `class_counts_for_branch`
   - input to `gini_coefficient` function
5. Estimate `gini_coefficient` for that branch
6. Weight by proportion of samples in branch
   - `branch.shape[0]/total_samples`

```
1  def split_cost(split,classes):
2  
3      """
4          Estimates the cost for a proposed split
5          input:
6              splits: tuple or form (L,R) where L reflects the data for the left split and
7                      R reflects data for left split
8              classes: list of class values i.e. [0,1]
9  
10         output:
11             cost: sum of gini coefficient for left and right sides of the split
12     """
13     cost=0
14     total_samples=0
15 
16     # estimate the relative size of each branch
17     for branch in split:
18         total_samples+=branch.shape[0]
19 
20     # for each (left/right) split on the proposed tree
21     for br_index,branch in enumerate(split):
22         # initialise list of class counts for this branch
23         class_counts_for_branch=[]
24         # for each class value, count total of data examples (rows)
25         # that have for this class, in this branch
26         for class_val in classes:
27 
28             if branch.shape[0] == 0: # don't continue if size of split is 0
29                 continue
30 
31             # 1.3.1 slice data to return only rows from branch which have this specific class value
32             # here branch[:,-1] returns the column containing the labels and we want to slice all rows
33             # for class=class_val
34             branch_per_class=branch[branch[:,-1]==class_val]
35             # 1.3.2 count the number of rows with this class in this branch and append
36             total_rows=branch_per_class.shape[0]
37             # this is generating list of class counts per branch which get fed to
38             # the gini_coefficient function
39             class_counts_for_branch.append(total_rows)
40 
41         # 1.3.3. estimate the gini coefficient for this split  (or branch)
42         gini_split=gini_coefficient(class_counts_for_branch)
43         # 1.3.4. estimated the weighted contribution for this split
44         weighted_by_sample_size=gini_split*(branch.shape[0]/total_samples)
45         # total cost is a weighted average of gini coefficients for both splits
46         cost+=weighted_by_sample_size
47 
48 
49     return cost
```

```
class_values=[0,1]
splitcost=split_cost(branches,class_values)

print('The cost of the proposed split is: ', splitcost)
```

```
The cost of the proposed split is:  0.375
```

Solutions

1.4 Choose optimal feature/threshold split

1. Line 28 - Iterate over all but last column:
   `np.arange(dataset.shape[1]-1)`

2. Iterate over all rows to propose threshold value (line 31)

3. Line 33 Create split accordingly:
   - Function – test split
   - Arguments: index (feature) row[index] (threshold) dataset

4. Estimate cost using split_cost (line 35)

5. If improved (`cost < best_cost`) as Gini must be minimized (l. 36)

6. If `cost<best_cost` then save current set of parameters as 'best'

```python
1
2  def get_best_split(dataset):
3      """
4          Search through all attributes and all possible thresholds to find the best split for the data
5          input:
6              dataset = array (n_samples,n_features+1)
7                      rows are examples
8                      last column indicates class membership
9                      remaining columns reflect features/attributes of data
10
11         output:
12             dict containing: 1) 'index' : index of feature used for splittling on
13                              2) 'value': value of threshold split on
14                              3) 'branches': tuple of data arrays reflecting the optimal split into left and right
15
16     """
17
18     # estimating the total number of classes by looking for the total number of different unique values
19     # in the final column of the data set (which represents class labels)
20     class_values=np.unique(dataset[:,-1])
21
22     # initalising optimal values prior to refinment
23     best_cost=sys.float_info.max # initialise to max float
24     best_value=sys.float_info.max # initialise to max float
25     best_index=dataset.shape[1]+1 # initialise as greater than total number of features
26     best_split=tuple()      # the best_split variable should contain the output of test_split that corresponds to t
27     #1.4.1 iterating over all features/attributes (columns of dataset)
28     for index in np.arange(dataset.shape[1]-1):
29
30         #Trialling splits defined by each row value for this attribute
31         for r_index,row in enumerate(dataset):
32             # 1.4.2. return branches corresponding to thresholding on feaure (index) and threshold value (for row i
33             branches=test_split(index, row[index], dataset)
34
35             cost=split_cost(branches,class_values) # 1.4.3 estimate cost for this split
36             if cost < best_cost: # 1.4.4. if this cost is an improvement on previous costs then save the
37                 best_cost=cost # cost
38                 best_split=branches #branches
39                 best_index=index # feature index
40                 best_value=row[index] # threshold value
41                 print('Best cost={}; Best feature={}; Best row={}'.format(best_cost,index,r_index) )
42
43     return {'index':best_index, 'value':best_value, 'branches':best_split}
44
```

Solutions

1.4 Choose optimal feature/threshold split

1. Line 28 - Iterate over all but last column: `np.arange(dataset.shape[1]-1)`

2. Iterate over all rows to propose threshold value (line 31)

3. Line 33 Create split accordingly:
   - Function – test split
   - Arguments: index (feature) row[index] (threshold) dataset

4. Estimate cost using split_cost (line 35)

5. If improved (`cost < best_cost`) as Gini must be minimized (l. 36)

6. If `cost<best_cost` then save current set of parameters as 'best'

```
1
2  def get_best_split(dataset):
3      """
4          Search through all attributes and all possible thresholds to find the best split for the data
5          input:
6              dataset = array (n_samples,n_features+1)
7                      rows are examples
8                      last column indicates class membership
9                      remaining columns reflect features/attributes of data
10
11         output:
12             dict containing: 1) 'index' : index of feature used for splittling on
13                              2) 'value': value of threshold split on
14                              3) 'branches': tuple of data arrays reflecting the optimal split into left and right
15
16     """
17
18     # estimating the total number of classes by looking for the total number of different unique values
19     # in the final column of the data set (which represents class labels)
20     class_values=np.unique(dataset[:,-1])
21
22     # initalising optimal values prior to refinment
23     best_cost=sys.float_info.max # initialise to max float
24     best_value=sys.float_info.max # initialise to max float
25     best_index=dataset.shape[1]+1 # initialise as greater than total number of features
26     best_split=tuple()      # the best_split variable should contain the output of test_split that corresponds to t
27     #1.4.1 iterating over all features/attributes (columns of dataset)
28     for index in np.arange(dataset.shape[1]-1):
29
30         #Trialling splits defined by each row value for this attribute
31         for r_index,row in enumerate(dataset):
32            # 1.4.2. return branches corresponding to thresholding on feaure (index) and threshold value (for row i
33             branches=test_split(index, row[index], dataset)
34
35             cost=split_cost(branches,class_values) # 1.4.3 estimate cost for this split
36             if cost < best_cost: # 1.4.4. if this cost is an improvement on previous costs then save the
37                 best_cost=cost # cost
38                 best_split=branches #branches
39                 best_index=index # feature index
40                 best_value=row[index] # threshold value
41                 print('Best cost={}; Best feature={}; Best row={}'.format(best_cost,index,r_index) )
42
43     return {'index':best_index, 'value':best_value, 'branches':best_split}
44
```

Solutions

- **3 - Running Decision Trees with Scikit-Learn**

1. instantiate a decision tree classifier model (line 29)

2. fit the model to training data (line 30)

3. Predict test labels (line 31)

4. Return performance score on test examples (line 32)

```
1  from sklearn.tree import DecisionTreeClassifier # import the scikit-Learn Decision Tree module
2  from sklearn.model_selection import train_test_split
3  from sklearn.preprocessing import StandardScaler
4  from sklearn.datasets import make_moons
5
6  # CREATE a Random Data set using the sklearn Make Moons dataset
7
8  DATA, LABELS =make_moons(noise=0.3, random_state=0)
9
10 # Plot the data
11
12 figure = plt.figure(figsize=(5, 5))
13 cm = plt.cm.RdBu
14 cm_bright = ListedColormap(['#FF0000', '#0000FF'])
15 ax = plt.subplot(1,1, 1)
16
17 ax.set_title("Input data")
18 ax.scatter(DATA[:, 0], DATA[:, 1], c=LABELS, cmap=cm_bright,
19            edgecolors='k')
20
21 # randomly split the data
22 X_train, X_test, y_train, y_test = train_test_split(DATA, LABELS, test_size=.4, random_state=42)
23
24 # **to DO** implement scikit learn decision tree classifier on this data
25 #**** complete the above steps for the scikit learn classifier ******
26 model = DecisionTreeClassifier(random_state=0)
27 model.fit(X_train,y_train)
28 pred=model.predict(X_test)
29 score = model.score(X_test, y_test)
30
31 print("Scikit-learn's decision tree Score", score)
32
33 # OPTIONALLY plot your results
34 # suggest plotting with  with different colours for each class
35 #and different markers for test and train data in order to aid visualisation
36
37 # just plot the dataset first
38 f, (ax1, ax2) = plt.subplots(2, 1, sharey=True, figsize=(5,10))
39 cm = plt.cm.RdBu
40 cm_bright = ListedColormap(['#FF0000', '#0000FF'])
41
42 ax1.set_title("True labels")
43 ax1.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
44            edgecolors='k')
45
46 ax2.set_title("Predicted labels")
47 ax2.scatter(X_test[:, 0], X_test[:, 1], c=pred, cmap=cm_bright,
48            edgecolors='k')
```

```
Scikit-learn's decision tree Score 0.95
```

Solutions

- **3.2 – optional compare own tree against Scikit-Learn**

```
# first combine X_train and y_train togeher (and X_test, y_test) to put data into form expected by our tree
dataset=np.concatenate((X_train,y_train.reshape((y_train.shape[0],1))),axis=1)
test_dataset=np.concatenate((X_test,y_test.reshape((y_test.shape[0],1))),axis=1)

#3.2.1 train your tree - set max depth to 5 and min size to 1
tree = build_tree(dataset, 5,1)

#3.2.2 Get a prediction from your test data
prediction_DT1=predict(tree, test_dataset)

#3.2.3 Score the accuracy of your decision tree classifier
score_DT1=tree_score(y_test,prediction_DT1)
print('Our Decision Tree Score', score_DT1)
```

```
Our Decision Tree Score 0.875
```

Scikit learn applied pruning (see notebook)

Ensembles exercises

Exercises

1. Exercise 2 - **Building a Bagging Classifier (30 mins)**
    1. Complete `create_bagged_ensemble` to bootstrap samples from data and train ensemble of decision trees
    2. Using decision tree code programmed last week (imported through DecisionTree.py)
    3. Complete `bagging_predict` to aggregate test predictions through majority voting
2. Exercise 3 - **Comparing our Bagged Predictor against our Decision Tree** (just run code – 5 mins)
3. Exercise 4 **- Comparing against Scikit learn (10-15 mins)**
    1. Build a decision tree and bagging classifier with scikit learn; fit model
    2. Return test prediction and accuracy (score) -> compare
4. Exercise 5: **Training Random Forests to Predict Gestational Age from Regional Brain Volumes¶ (20 mins)**
    - implement the training and testing of the DecisionTreeRegressor()
    - implement the training and testing of the RandomForestRegressor()
    - (optional) Return and plot feature importances see the scikit-learn tutorial for guidance

Exercises

5. Exercise 6 (optional): Perform Parameter Optimisation for Random Forests using GridSearchCV (10-15 mins)
6. Exercise 7 (optional/homework) Building a Random Forest Classifier from scratch (20-30 mins)
7. Exercise 8: **Training Adaboost to Predict Gestational Age from Regional Brain Volumes (20-30 mins)¶**
   - Implement adaboost regression using scikit learn
   - Apply GridCV optimisation of the ensemble and base learner paramters