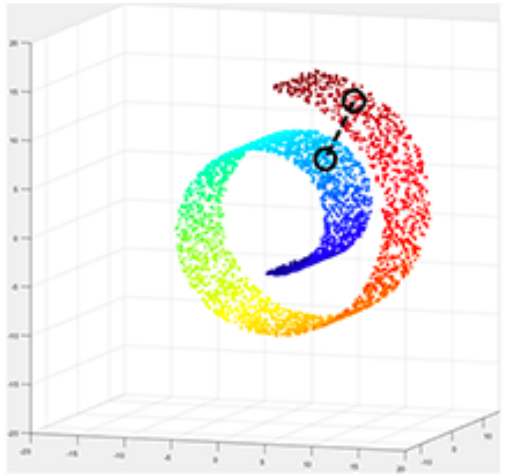
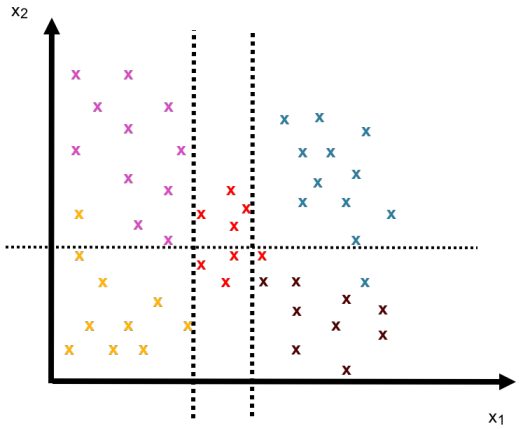
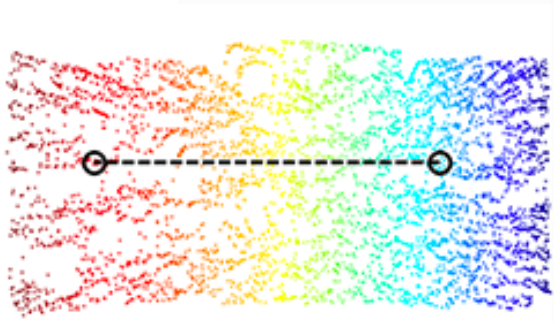
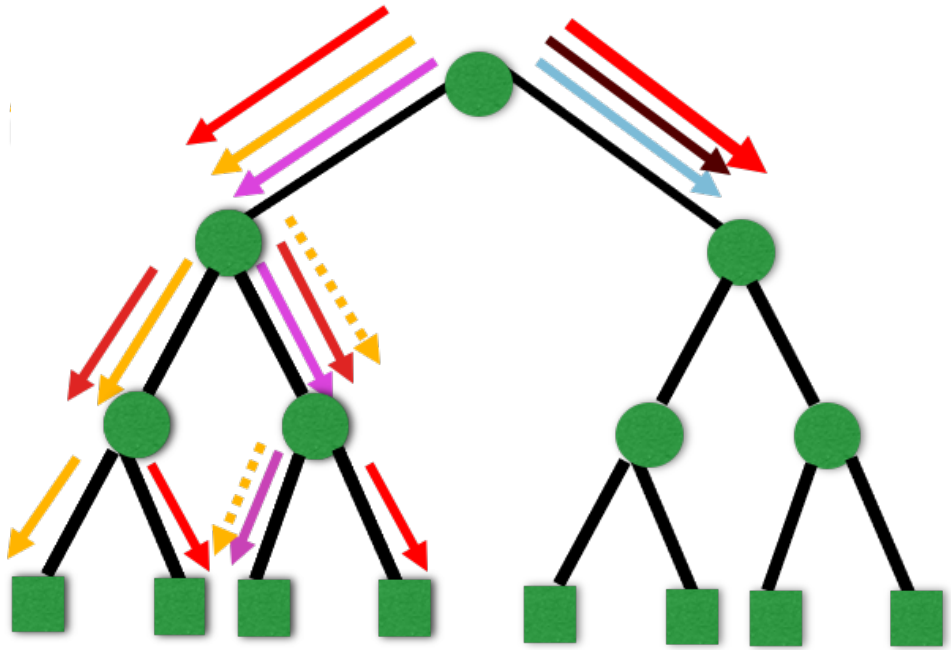
Machine Learning for Biomedical/Healthcare Applications









Laplacian Eigenmaps and Decision Trees

Dr Emma Robinson

Learning Objectives

1. Gain an **intuition for the reasoning behind manifold learning** and define the conditions, under which it is needed, over linear techniques such as PCA
2. Learn how to implement **Laplacian Eigenmaps** from scratch and scikit learn
3. Be able to define what is meant by a weak learner, a **decision stump** a **decision tree**
4. Be able to define and use **weak learning rules: Information Gain and Gini Index**,
   - to split a training set into two groups, on the values of one features
5. Learn step by step how to construct a **decision tree classifier from scratch** and using scikit-learn

New teams channel for Level 7!

I Laplacian Eigenmaps

Open 6.1.Laplacian_Eigenmaps.ipynb ; To do

1. Exercise 2 - Create a *symmetric* k-Nearest neighbour graph (30 mins)
2. Exercise 3 - implement the Laplacian Eigenmaps embedding of the swiss roll data set (10 mins)
3. Exercise 4 - Implementing Laplacian Eigenmaps with Scikit Learn (10 mins)

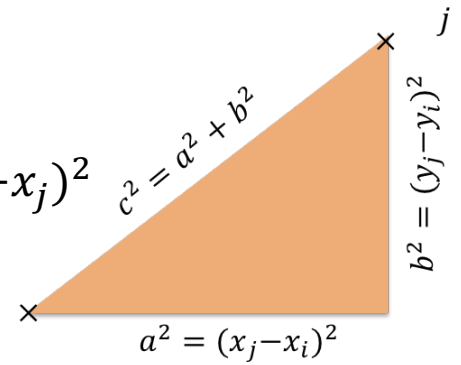
Laplacian Eigenmaps -Summary

- Non-linear manifold learning methods seek a low-dimensional embedding which preserves local manifolds
- The Laplacian embedding can be estimated through the following steps
  1. Estimate the squared distances between all points
  2. Use this to calculate the k-nearest neighbour adjacency graph $\mathbf{A}$
     - Setting k-nearest neighbours to 1, make all other values 0
     - Make this symmetric
  3. Calculate the degree matrix $\mathbf{D}$ a diagonal matrix with entries $\mathbf{D}_{\mathrm{ii}} = \sum_j \boldsymbol{A_{ij}}$
  4. Estimate graph Laplacian $\boldsymbol{L} = \mathbf{D} - \mathbf{A}$
  5. Estimate the embedding from the eigenvectors of $\boldsymbol{L}$ corresponding to the N smallest eigenvalues (above 0)

Tutorial: estimate a K-nearest Neighbour graph

- The first step of many manifold learning techniques is to define the k-nearest neighbour graph

- For some pre-defined k, estimate the *Adjacency* matrix $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{n}}$
  - Estimate the *Euclidean* (squared) distances ***between all points*** $SSD_i = \sum_j (x_i - x_j)^2$
  - Set **the k-nearest** distances to **1** (*excluding itself!*)
  - Set **all other distances** to **0**



- Typically we want $\boldsymbol{A}$ to be symmetric which means setting $\boldsymbol{A_{ij}} = \boldsymbol{A_{ji}}$ if either $\boldsymbol{A_{ij}} = \boldsymbol{1}$ or $\boldsymbol{A_{ji}} = \boldsymbol{1}$
  - E.g. using `np.maximum(A, A.T)`
  - `np.maximum` returns the element-wise maximum of its arguments

Tutorial: estimate a K-nearest Neighbour graph

- To go from distances to k-NN

Diagonals stay 0!

$$\sum(X[i]-X)^2 = \begin{bmatrix} 0 & 9 & 1 & 5 & 10 & 13 \\ 9 & 0 & 10 & 2 & 13 & 10 \\ 1 & 10 & 0 & 8 & 17 & 20 \\ 5 & 2 & 8 & 0 & 5 & 4 \\ 10 & 13 & 17 & 5 & 0 & 1 \\ 13 & 10 & 20 & 4 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 1 & 5 & 0 & 0 \\ 9 & 0 & 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 8 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 5 & 0 & 1 \\ 0 & 0 & 0 & 4 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

- Make symmetric

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Tutorial: Implemented in Scikit-Learn

- Scikit learn refers using Laplacian Eigenmaps as Spectral Embedding

class `sklearn.manifold.` **`SpectralEmbedding`**(*n_components=2, *, affinity='nearest_neighbors', gamma=None, random_state=None, eigen_solver=None, n_neighbors=None, n_jobs=None*) [source]

- By default it solves for the normalized graph Laplacian $\boldsymbol{L = \mathbf{D}^{-1/2}(\mathbf{D} - \mathbf{A})\,\mathbf{D}^{-1/2}}$
- which can lead to more stable solutions is degree is not distributed evenly across all points
- E.g.

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import SpectralEmbedding
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = SpectralEmbedding(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

https://scikit-learn.org/stable/modules/generated/sklearn.manifold.SpectralEmbedding.html#sklearn.manifold.SpectralEmbedding

Solutions – Ex 2 Laplacian Eigenmaps

- Exercise 2 - Create a *symmetric* k-Nearest neighbour graph (30 mins)

1. Estimate squared distances ($SSD_i = \sum_j (x_i - x_j)^2$) between each point X[i] and all others
   - Distances can be estimated with broadcasting - X-X[i]
   - Need square distance – so use power operator **2
   - Want sum of squares over features (columns) - achieved using np.sum over column axis (1)

```
1  def my_knn(X, k):
2      """ Finds k-nearest neighbours in X """
3      N, D = X.shape
4      A = np.zeros((N, N))
5      for i in range(N):
6          # 2.1 estimate the squared distances between point i and all neighbours
7          i_sq_distances =np.sum((X - X[i])**2, axis=1)
8          # 2.2 find the nearest points
9          nearest_points = np.argsort(i_sq_distances)
10         # [1:k+1] is because the nearest point to i is i itself - but we don't want it
11         k_nearest      = nearest_points[1:k+1]
12         for j in k_nearest:
13             A[i,j] = 1
14     return A
15
16 A = my_knn(X, 20)
```

Solutions – Ex 2 Laplacian Eigenmaps

- Exercise 2 - Create a *symmetric* k-Nearest neighbour graph (30 mins)

2. Find nearest distances
   - First get indices of nearest points using np.argsort(i_sq_distances)
   - Return the k nearest (ignoring the first one as that will be the distance between point and itself)
     - `nearest_points[1:k+1]`
   - A is initialised using `np.zeros((N,N))` so we just need to add ones for the ids of nearest neighbours

```
1  def my_knn(X, k):
2      """ Finds k-nearest neighbours in X """
3      N, D = X.shape
4      A = np.zeros((N, N))
5      for i in range(N):
6          # 2.1 estimate the squared distances between point i and all neighbours
7          i_sq_distances =np.sum((X - X[i])**2, axis=1)
8          # 2.2 find the nearest points
9          nearest_points = np.argsort(i_sq_distances)
10         # [1:k+1] is because the nearest point to i is i itself - but we don't want it
11         k_nearest      = nearest_points[1:k+1]
12         for j in k_nearest:
13             A[i,j] = 1
14     return A
15
16 A = my_knn(X, 20)
```

Solutions – Ex 2 Laplacian Eigenmaps

- Exercise 2 - Create a *symmetric* k-Nearest neighbour graph (30 mins)

  3. Make symmetric by running

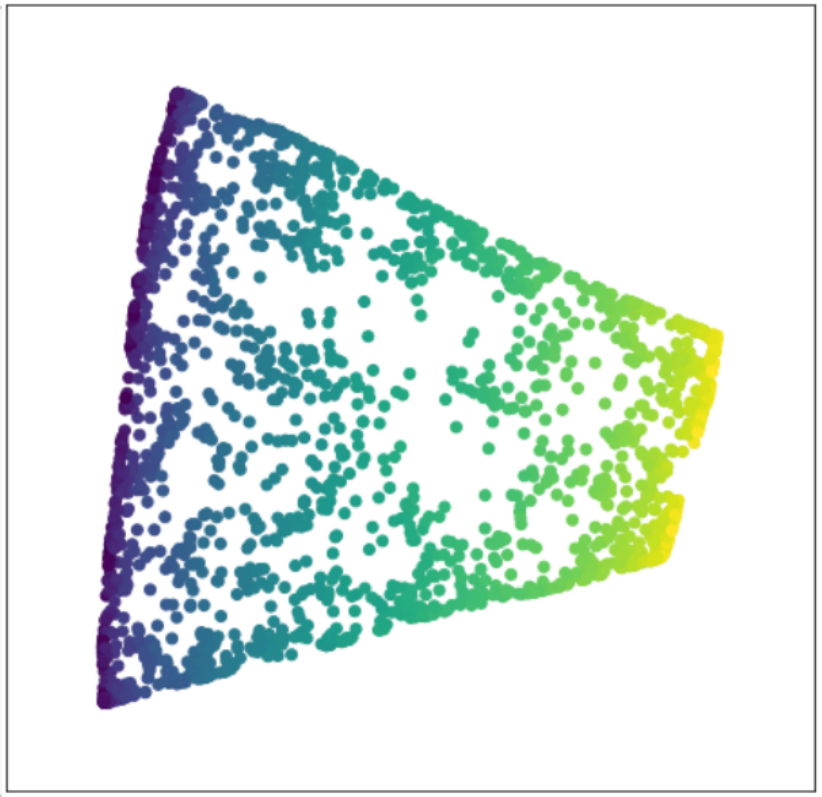
To do Run the cell below to return a symmetric your k-nearest neighbour *adjacency* matrix $\mathbf{A}$

```
:
1  def symmetrise(X):
2      """ Symmetrises the matrix X.
3
4      Notes
5      -----
6      np.maximum returns the element-wise maximum of its arguments."""
7      return np.maximum(X, X.T)
8  A = symmetrise(A)
```

Solutions – Ex 3 Laplacian Eigenmaps

- Exercise 3 - implement the Laplacian Eigenmaps embedding of the swiss roll data set

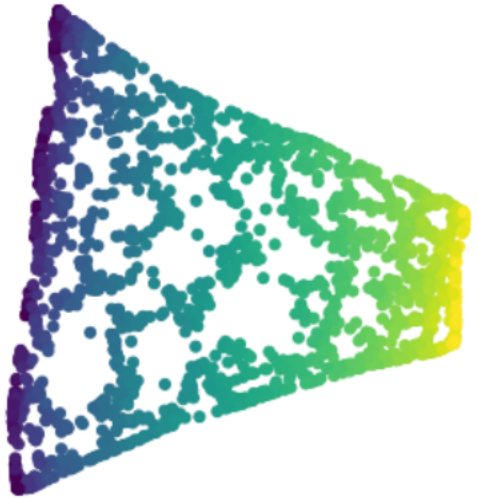
```python
def my_laplacian_eigenmap(X, k=20, d=2):
    # 3.1 a use function my_knn to return A
    A = my_knn(X, k)
    # 3.1 b use function symmetrise to make A symmetric
    A = symmetrise(A)
    # 3.2 create diagonal matrix D from A
    # the degree of each number is the sum the Adjacency for that row/column
    #(remember symmetric so these are the same)
    # to make a vector into a diagonal matrix using np.dig
    D = np.diag(np.sum(A, axis=0))
    # 3.3 create L
    L = D - A
    # 3.4 return eigenvectors of L using np.eigh
    # eigvals=(1,d) returns the d smallest (above 0 - in this case corresponding to indices 1 and 2)
    v, X = eigh(L, eigvals=(1,d))
    return X

Z = my_laplacian_eigenmap(X)

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
ax.set_xticks([])
ax.set_yticks([])
_ = ax.scatter(Z[:,0], Z[:,1], c=X_m[:,1], marker='o')
```



Solutions – Ex 4 Laplacian Eigenmaps

- Exercise 4 - Implementing Laplacian Eigenmaps with Scikit Learn

```
:
1  from sklearn.manifold import SpectralEmbedding
2
3  model=SpectralEmbedding(n_components=2,n_neighbors=10)
4
5  Z_2 = model.fit_transform(X)
6
7  fig = plt.figure(figsize=(8,8))
8  ax = fig.add_subplot(111)
9  ax.set_xticks([])
10 ax.set_yticks([])
11 _ = ax.scatter(Z_2[:,0], Z_2[:,1], c=X_m[:,1], marker='o')
```



May need to re-download – very slight typo wrt line numbers reported in Exercise 1.4

II Decision stumps and Trees

Open **6.2.Decision_Trees.ipynb** ; To do

1. Exercise 1 building a decision stump classifier from scratch (30-45 mins)
2. (optional) Exercise 2: building and testing a complete decision tree (15 mins)
3. Exercise 3 - Running Decision Trees with Scikit-Learn (15 mins)
    1. (optional) Compare against the decision tree built in Exercise 2

Summary

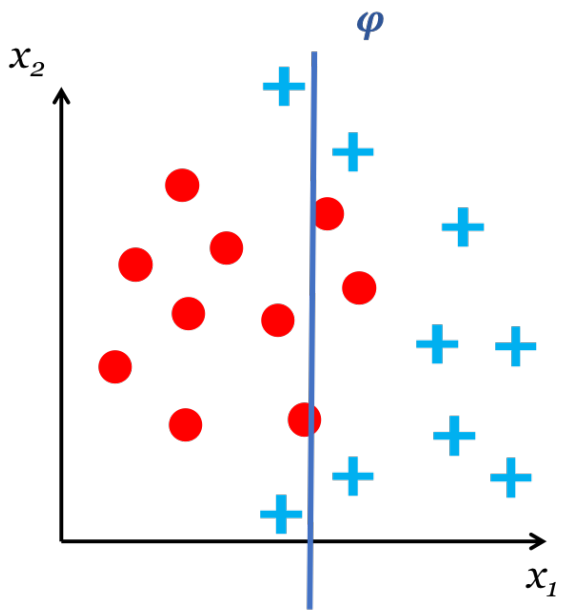
- Decision Trees are **hierarchies of decision stumps;** they are powerful because
  - **Easy to interpret**
  - Can be **used for classification or regression**
  - Don't require normalization of features
- Each decision stump must be fit by
  - Iterating over all features
  - Finding optimal threshold for each features
  - Choosing the feature which performs best (has optimal cost) relative to all others

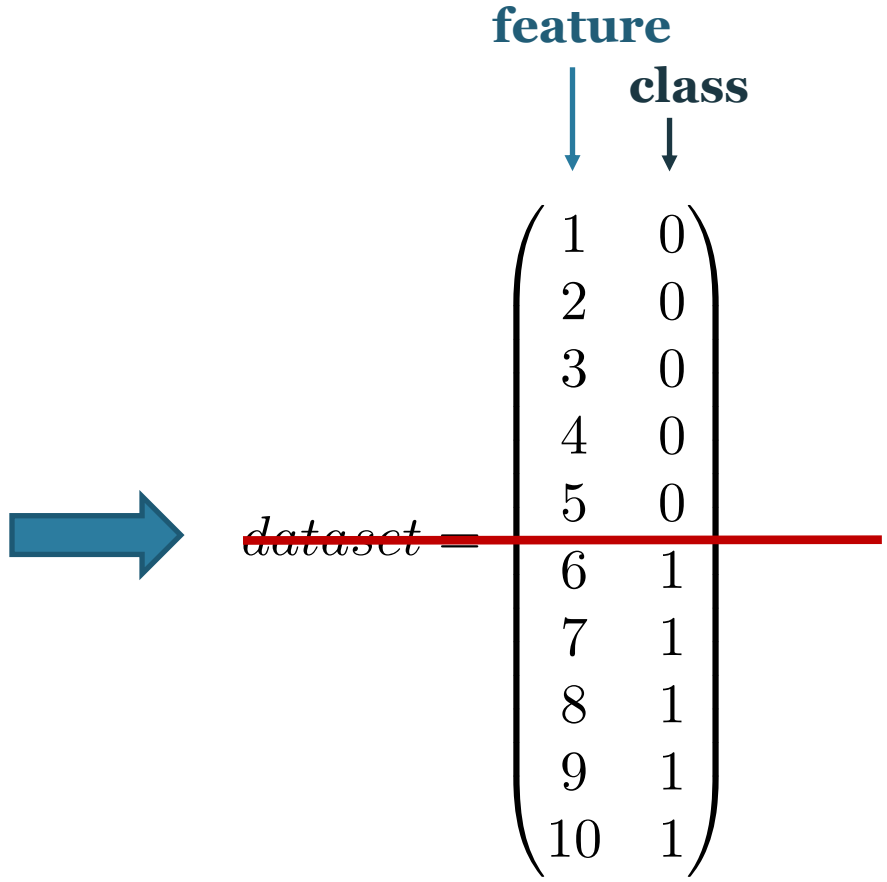
Todays Tutorial

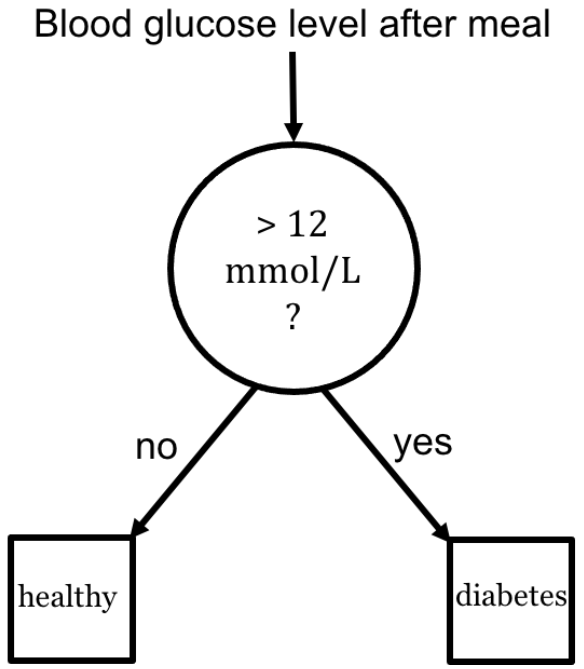
- **Labels must be assigned to each leaf** from distribution of the labels of the training examples which reach that node
- The robustness of decision trees can be increased through combining them in ensembles (next week)

Weak Learners/Decision stump

A **weak learner** is defined to be a classifier that is only slightly correlated with the true classification (it can label examples slightly better than random guessing) e.g **axis-aligned**





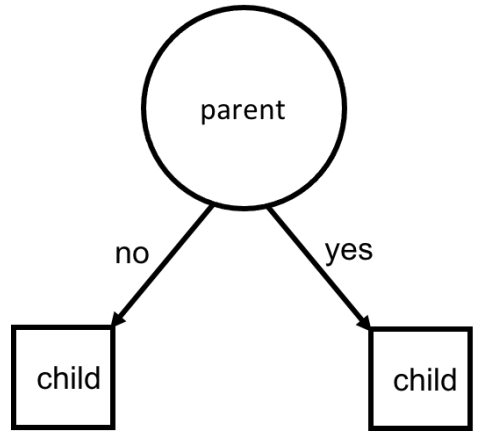


Decision Stump: Learning Rules

Two choices for deciding on threshold **for a classification problem**:

1. **Information gain:** Decrease in entropy after a dataset is split on an attribute.

$$I(S_j) = H(S_j) - \sum_i \frac{|S_j^i|}{|S_j|} H(S_j^i)$$



- Here $H(S_j)$ is entropy of root node and $H(S_j^i)$ is entropy of leaf node
- $|S_j|$= total number of data points reaching parent node j;
- $|S_j^i|$ = total number of data points reaching child i from parent j;

2. **Gini index:** Indicates how mixed the classes are following the split.

$$I(S_j) = \sum_i \frac{|S_j^i|}{|S_j|} Gini_i \qquad \text{where} \quad \text{Gini} = 1 - \sum_{y_k \in Y} p(y_k)^2$$

- $p(y_k)$ is proportion at given node that are of class k ;

Tutorial: Decision tree/stump - Optimisation:

At each tree node j:

For each possible feature k:

I.e. try thresholding on each feature value

$$dataset = \begin{pmatrix} 1 & 0 \\ 2 & 0 \\ 3 & 0 \\ 4 & 0 \\ 5 & 0 \\ 6 & 1 \\ 7 & 1 \\ 8 & 1 \\ 9 & 1 \\ 10 & 1 \end{pmatrix}$$

For each possible threshold τ on this feature:

Evaluate function $I(S_j, k)$

This estimates the cost of splitting training data into left/right* components (S_j^1, S_j^1)
*: leaf 1, leaf 2

Chose the best feature (k_{opt}) corresponding to optimum cost (I_{opt}) and threshold τ_{opt}

Split data according to this function

REPEAT*

until termination criterion is met

Assign leaf nodes majority label of training examples

Important points to remember! (see Tuesdays tutorial)

1. When using each splitting criteria:

Gini index must be **MINIMISED**
Information Gain must be **MAXIMISED**

2. **Try $I(S_j, k)$ on each value of the column X[:,k]** to find your optimum τ

3. The **amount of training examples will decrease** as you move down the tree
   - And different examples pass down different branches

4. When looping through features and threshold **you must retain variables memorising the current best feature/threshold/data split**

- In **Tuesday's Tutorial** we will go through these steps for **creating a single decision stump in Python from scratch**
- With **option to extend** to building a **whole decision tree**

Tutorial: Decision Trees in Python (optional)

- The decision tree can therefore be defined by a nested dictionary which defines, **for each tree node *j*** - keys:
  - **Index:** The **feature** k^j_{opt} that's split on
  - **Value:** The **threshold** τ^j_{opt}**needed** for an optimal split **on that feature** (for the training data that reaches it)
  - **Branches:** **the split of the data** (i.e. tuple of 2 data arrays) indicating the data rows (examples) sent down each branch
- Note that **for each subsequent node** of the tree **a new dictionary is created** for each split
- Thus the **branches** key gets replaced each time with new dictionaries representing the {'index': k^{j+1}_{opt}, 'value': τ^{j+1}_{opt}, 'branch': (X_{left}, X_{right})}
- **Leaf nodes are assigned a class label or a predicted fit $\hat{y}$**

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 7th 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
```

```
:
1 class_values=[0,1]
2 splitcost=split_cost(branches,class_values)
3
4 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'

```
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**

```
:
1  # first combine X_train and y_train togeher (and X_test, y_test) to put data into form expected by our tree
2  dataset=np.concatenate((X_train,y_train.reshape((y_train.shape[0],1))),axis=1)
3  test_dataset=np.concatenate((X_test,y_test.reshape((y_test.shape[0],1))),axis=1)
4
5  #3.2.1 train your tree - set max depth to 5 and min size to 1
6  tree = build_tree(dataset, 5,1)
7
8  #3.2.2 Get a prediction from your test data
9  prediction_DT1=predict(tree, test_dataset)
10
11 #3.2.3 Score the accuracy of your decision tree classifier
12 score_DT1=tree_score(y_test,prediction_DT1)
13 print('Our Decision Tree Score', score_DT1)
```

```
Our Decision Tree Score 0.875
```

Scikit learn applied pruning (see notebook)

Additional exercises

- Try constructing a regression tree from scratch; using the above classification tree as the basis but:
  - creating a new MSE cost, and
  - editing the prediction function accordingly (to fit constant function to mean);
- Try it out the following toy dataset (Taken from: http://scikit-learn.org/stable/auto_examples/tree/plot_tree_regression.html#sphx-glr-auto-examples-tree-plot-tree-regression-py)

```
# Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(16))
```

- Compare your result