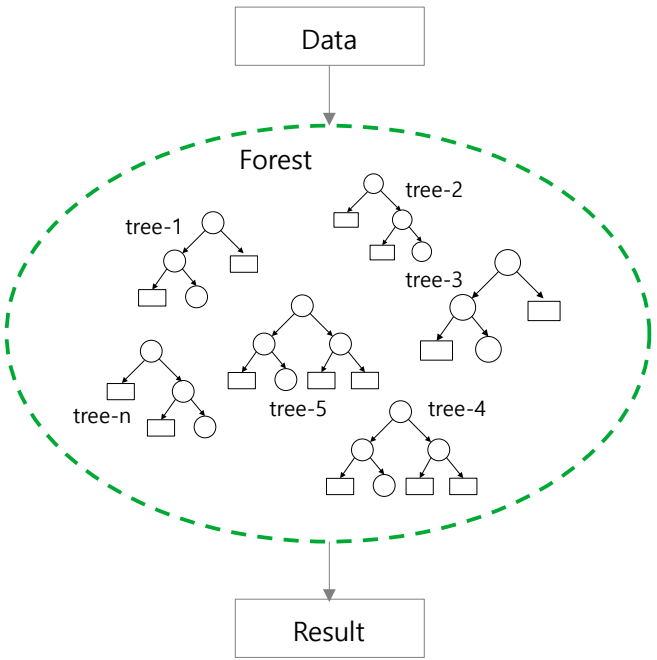


8. Random Forest

- Boostrap Aggregation (Bagging)
- Out-Of-Bag (OOB) score (error rate)
- Missing value, Outlier Detection using Proximity Matrix

www.youtube.com/@meanxai

MX-AI

MX-AI

▪ Random Forest: Overview

- The first algorithm for random decision forests was created in 1995 by Tin Kam Ho. An extension of the algorithm was developed by Leo Breiman and Adele Cutler [Wikipedia]. Random Forest is a type of bagging ensemble that has many advantages, including less variance, less overfitting, and better overall performance, etc. I referred to the following two documents [1], [2], and a video [3].

*** Reference [1]**

RANDOM FORESTS

Leo Breiman
Statistics Department University of California Berkeley, CA 94720, January 2001

Abstract

Random forests are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. The generalization error for forests converges a.s. to a limit as the number of trees in the forest becomes large. The generalization error of a forest of tree classifiers depends on the strength of the individual trees in the forest and the correlation between them. Using a random selection of features to split each node yields error rates that compare favorably to Adaboost (Freund and Schapire[1996]), but are more robust with respect to noise. Internal estimates monitor error, strength, and correlation and these are used to show the response to increasing the number of features used in the splitting. Internal estimates are also used to measure variable importance. These ideas are also applicable to regression

*** Reference [2]**

Random Forests
Leo Breiman and Adele Cutler

Random Forests(tm) is a trademark of Leo Breiman and Adele Cutler and is licensed exclusively to Salford Systems for the commercial release of the software.
Our trademarks also include RF(tm), RandomForests(tm), RandomForest(tm) and Random Forest(tm).

classification/clustering | regression | survival analysis
description | manual | code | papers | graphics | philosophy | copyright | contact us

Contents

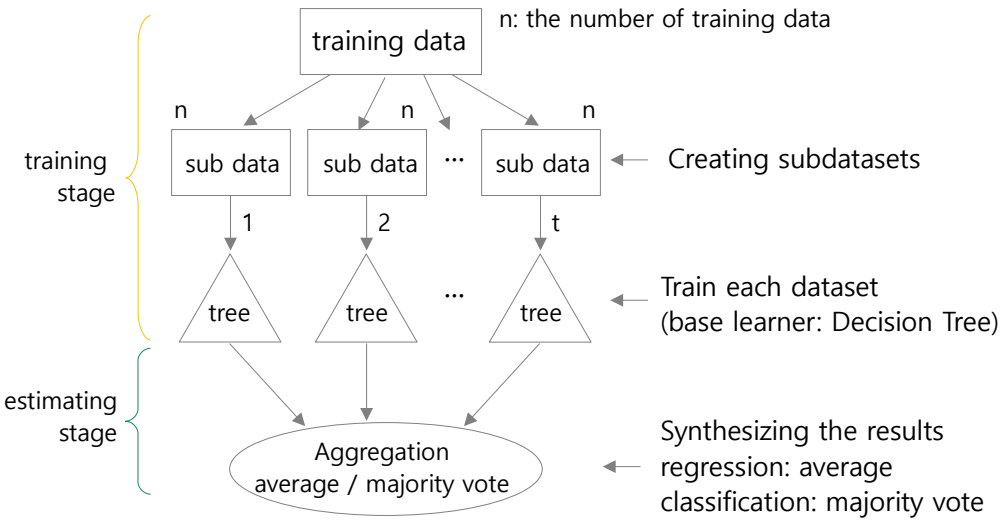
*** Reference [3]**

- Youtube:
- StatQuest: Random Forests Part 1: Building, Using and Evaluating
- StatQuest: Random Forests Part 2: Missing data and clustering

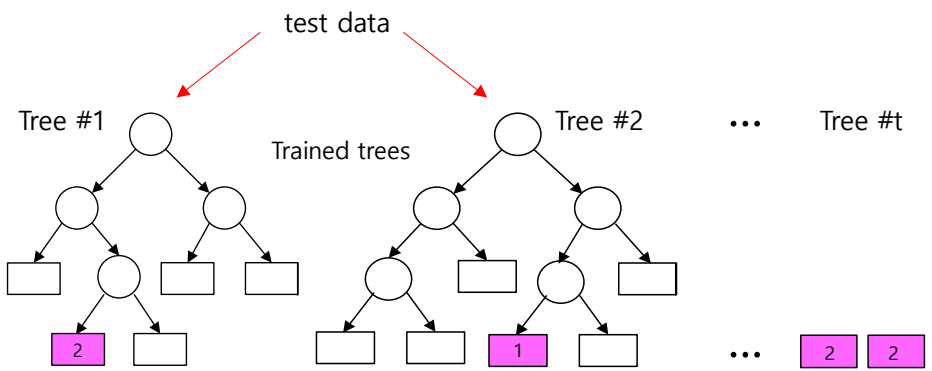
MX-AI

▪ Random Forest : Bootstrap Aggregation (Bagging)

- Random Forest uses multiple Decision Trees. Each tree is then built based on randomly and uniformly drawn samples with replacement for the training data.
- Subdatasets are created from samples extracted from the training data. And the size of each subdataset is equal to the size of the training data.
- In the figure below, the number of subdatasets and the number of trees is t, and the number of the training data is n.
- Samples are drawn from rows and columns of the training data. Column sampling is the random selection of features. This results in lower correlation between trees.
- Each tree is grown to the largest extent possible. There is no pruning. (Reference [2]). Random Forest uses multiple deep decision trees, but it is less prone to overfitting because it uses sample data, and average the results of the trees. This is why pruning is not necessary.
- After training, test data is inserted into each tree and the results are synthesized. For regression, it is estimated as the average of each tree's results, and for classification, it is estimated as the most frequent class of each tree's results. (majority voting).



* General structure of bagging (parallel structure)

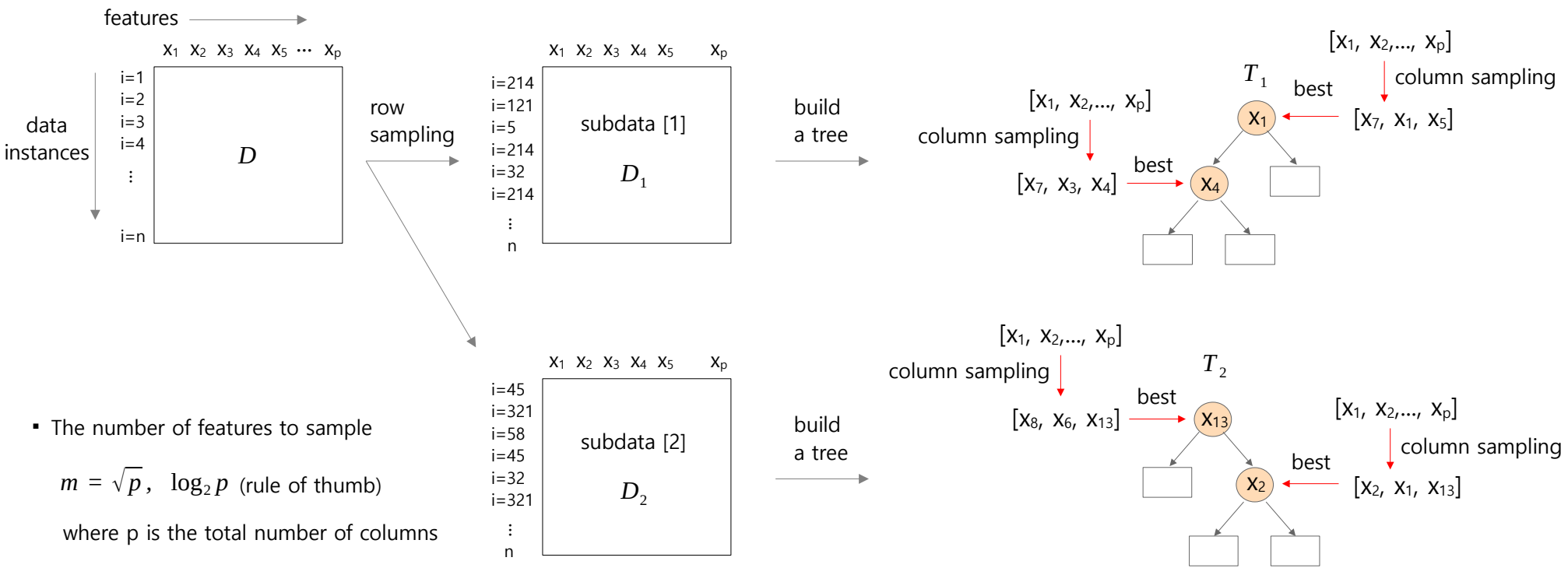


- If you insert one test data point into T trained trees, you can get t results.
- In classification, if the results are [2, 1, 2, 2, ...] as shown above, the class of the test data point is assumed to be 2 because 2 is the majority.
- In regression, it is estimated as the average value of the results.

MX-AI

▪ Data Sampling : row (data instances) and column (features) sampling

- The training data (D) is sampled row-wise and column-wise. The reason for sampling is to reduce the correlation between each tree, thus reducing the estimation variance.
- Row sampling is done with replacement, and column sampling is done without replacement, but after sampling, all are replaced for the next sampling. That is, column sampling for node splitting is done without replacement, but with replacement within an individual tree.
- The number of columns (features) to sample is calculated as m=sqrt(p) or m=log2(p) by a rule of thumb, where p is the total number of columns.



- The number of features to sample

$m = \sqrt{p}$, $\log_2 p$ (rule of thumb)

where p is the total number of columns

MX-AI

▪ Data Sampling : row (data instance) and column (feature) sampling

▪ **Row sampling** with replacement (bootstrap samples) reduces the correlation between the decision trees. Without row sampling, the results of many trees can be similar, reducing ensemble effects. **Column sampling** (sampling features) can further reduce correlation between trees. Without column sampling, if there are a few key features, these features are selected from many trees, making the trees similar (correlated) again.

$$\hat{y} = \frac{1}{t}\sum_{i=1}^{t} T_i(x)$$

$$Var(\hat{y}) = Var\left(\frac{1}{t}\sum_{i=1}^{t} T_i(x)\right)$$

$$= \frac{1}{t^2}\sum_{i=1}^{t}\sum_{j=1}^{t} Cov\left(T_i(x),\ T_j(x)\right)$$

$$= \frac{1}{t^2}\sum_{i=1}^{t}\left(\sum_{j\neq i} Cov\left(T_i(x), T_j(x)\right) + Var\left(T_i(x)\right)\right)$$

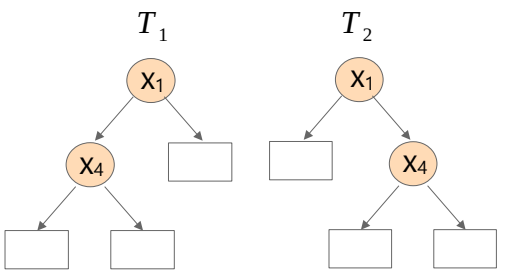
$$= \frac{1}{t^2}\sum_{i=1}^{t}\left((t-1)\sigma^2\rho + \sigma^2\right) \quad \leftarrow \rho = \frac{Cov(x,y)}{\sigma_x \sigma_y}$$

$$= \frac{t(t-1)\rho\sigma^2 + t\sigma^2}{t^2}$$

$$Var(\hat{y}) = \rho\sigma^2 + \sigma^2\frac{1-\rho}{t}$$

- Row/column sampling reduces ρ between trees, making Var(y) smaller. Also, the larger the number of trees, t, the smaller Var(y) becomes.
- The smaller m is, the smaller ρ is.

- **Without column sampling:**
- If x1 is the most important feature, it is likely to be used as the first split point for many trees, even if the data varies depending on row sampling. Then the trees will be similar.



Reference: https://stat.ethz.ch/education/semesters/ss2012/ams/slides/v10.2.pdf
(Applied Multivariate Statistics – Spring 2012)

MX-AI

- Coding practice: Implement Random Forest from scratch

```python
# MyDTreeClassifierRF.py: Create a tree for Random Forest.
# This code is an upgraded version of MyDTreeClassifier
# shown in [MXML-2-07] video.
# Features: subsampling by rows and columns,
#           predict Out-Of-Bag (OOB) data points
import numpy as np
from collections import Counter
import copy

# Implement a simplified Random Forest using binary tree.
class MyDTreeClassifierRF:
    def __init__(self, max_depth, max_samples, max_features):
        self.max_depth = max_depth
        self.max_samples = max_samples
        self.max_features = max_features
        self.u_class = None       # unique class (target y value)
        self.estimator1 = dict() # tree result-1
        self.estimator2 = dict() # tree result-2
        self.feature = None       # feature data
        self.target = None        # target class data
        self.iob_pred=None  # predicted classes of train data points
        self.oob_pred=None  # predicted classes of OOB data points

    # Calculate Gini index of a leaf node
    def gini_index(self, leaf):
        n = leaf.shape[0]

        if n > 1:
            gini = 1.0
            for c in self.u_class:
                cnt = (self.target[leaf] == c).sum()
                gini -= (cnt / n) ** 2
            return gini
        else:
            return 0.0
```

```python
    # Split a node into left and right.
    # Find the best split point with highest information gain,
    # and split node with it.
    # did: data index on the leaf node.
    def node_split(self, did):
        n = did.shape[0]

        # Gini index of parent node before splitting.
        p_gini = self.gini_index(did)

        # perform column subsampling without replacement
        p = self.feature.shape[1]
        m = self.max_features
        f_list = np.random.choice(np.arange(0, p), m,
                                  replace=False)

        # Split the node into all candidates for all features
        # and find the best feature and the best split point with
        # the highest information gain.
        # fid: feature_id
        max_ig = -999999
        for fid in f_list:
            # feature data to be split
            x_feat = self.feature[did, fid].copy()

            # split x_feat using the best feature and the best
            # split point. Note: The code below is inefficient
            # because it sorts x_feat every time it is split.
            # Future improvements are needed.
```

MX-AI

- Coding practice: Implement Random Forest from scratch

```
        # remove duplicates of x_feat and sort in ascending order
        x_uniq = np.unique(x_feat)

        # list up all the candidates, which are the midpoints of
        # adjacent data points.
        s_point = [np.mean([x_uniq[i-1], x_uniq[i]]) \
                   for i in range(1, len(x_uniq))]

        # len(s_point) > 1: Calculate the information gain for all
        # candidates, and find the candidate with the largest
        # information gain.
        # len(s_point) < 1: skip the for-loop. x_feat either has
        # only one data point or has all the same values.
        # No need to split.
        for p in s_point:
            # split x_feat into the left and the right node.
            left = did[np.where(x_feat <= p)[0]]
            right = did[np.where(x_feat > p)[0]]

            # calculate Gini index after splitting.
            l_gini = self.gini_index(left)
            r_gini = self.gini_index(right)

            # calculate information gain
            ig = p_gini - (l_gini * left.shape[0] / n)
                        - (r_gini * right.shape[0] / n)

            # find where the information gain is greatest.
            if ig > max_ig:
                max_ig = ig
                b_fid = fid       # best feature id
                b_point = p       # best split point
                b_left = left     # data index on the left node.
                b_right = right   # data index on the right node.

        if max_ig > 0.:     # split
            return {'fid':b_fid, 'split_point':b_point,
                    'left':b_left, 'right':b_right}
        else:
            return  None    # No split

    # Create a binary tree using recursion
    def recursive_split(self, node, curr_depth):
        left = node['left']
        right = node['right']

        # exit recursion
        if curr_depth >= self.max_depth:
            return

        # recursion
        s = self.node_split(left)
        if isinstance(s, dict):   # splitting to the left done.
            node['left'] = s
            self.recursive_split(node['left'], curr_depth+1)

        s = self.node_split(right)
        if isinstance(s, dict):   # splitting to the right done.
            node['right'] = s
            self.recursive_split(node['right'], curr_depth+1)

    # majority vote
    def majority_vote(self, did):
        c = Counter(self.target[did])
        return c.most_common(1)[0][0]
```

MX-AI

- Coding practice: Implement Random Forest from scratch

```
    # Change the data in the leaf node to majority class.
    def update_leaf(self, d):
        if isinstance(d, dict):
            for key, value in d.items():
                if key == 'left' or key == 'right':
                    rtn = self.update_leaf(value)
                    if rtn[0] == 1:      # leaf node
                        d[key] = rtn[1]
            return 0, 0  # the first 0 means this is not a leaf node.
        else:            # leaf node
            # the first 1 means this is a leaf node.
            return 1, self.majority_vote(d)

    # create a tree using training data
    # x : feature data, y: target data
    def fit(self, x, y):
        # perform row subsampling with replacement
        n = x.shape[0]
        i_rows = np.random.choice(np.arange(0, n), self.max_samples,
                                  replace=True)

        self.feature = x[i_rows, :]
        self.target = y[i_rows]
        self.u_class = np.unique(y)

        # Initially, the root node holds all data points IDs.
        root = self.node_split(np.arange(x.shape[0]))
        if isinstance(root, dict):
            self.recursive_split(root, curr_depth=1)

        # tree result-1. Every leaf node has data point IDs.
        self.estimator1 = root

        # tree result-2. Every leaf node has the majority class.
        self.estimator2 = copy.deepcopy(self.estimator1)
        self.update_leaf(self.estimator2)         # tree result-2
```

```
        # predict Out-Of-Bag (OOB) data points
        # initialize the predicted classes of OOB
        # and train data points
        self.iob_pred = np.ones(shape=(x.shape[0],), dtype=int) * -1
        self.oob_pred = np.ones(shape=(x.shape[0],), dtype=int) * -1

        # predict training dataset
        y_pred = self.predict(x)

        # predict training and OOB dataset
        i_train = set(np.arange(0, x.shape[0]))
        i_oobs = list(i_train - set(i_rows))   # OOB data point IDs

        self.iob_pred[i_rows] = y_pred[i_rows] # for training data
        self.oob_pred[i_oobs] = y_pred[i_oobs] # for OOB data

        return self.iob_pred, self.oob_pred

    # Estimate the target class of a test data.
    def x_predict(self, p, x):
        if x[p['fid']] <= p['split_point']:
            if isinstance(p['left'], dict): # recursion if not leaf.
                return self.x_predict(p['left'], x)   # recursion
            else:           # return the value in the leaf, if leaf.
                return p['left']
        else:
            if isinstance(p['right'], dict) # recursion if not leaf.
                return self.x_predict(p['right'], x)  # recursion
            else:           # return the value in the leaf, if leaf.
                return p['right']

    # Estimate the target class of a x_test.
    def predict(self, x_test):
        p = self.estimator2    # predictor
        y_pred = [self.x_predict(p, x) for x in x_test]
        return np.array(y_pred)
```

MX-AI

▪ Coding practice: Implement Random Forest from scratch

```
# [MXML-8-02]: 1.RF(titanic).py
# Implement Random Forest using MyDTreeClassifierRF
import numpy as np
import pandas as pd
from MyDTreeClassifierRF import MyDTreeClassifierRF
from sklearn.model_selection import train_test_split

# Read preprocessed Titanic data.
df = pd.read_csv('data/titanic_clean.csv')

# Survived  Pclass  Sex   Age  SibSp  Parch   Fare  Embarked  Title
#        0       3    1  22.0      1      0   3.62         3      2
#        1       1    0  38.0      1      0  35.64         0      3
#        1       3    0  26.0      0      0   7.92         3      1
#        1       1    0  35.0      1      0  26.55         3      3
#        0       3    1  35.0      0      0   8.05         3      2

y = np.array(df['Survived'])
x = np.array(df.drop('Survived', axis=1))
x_train, x_test, y_train, y_test = train_test_split(x, y)

n_estimators = 100
n_features = round(np.sqrt(x.shape[1])) # the number of features
                                        # for column sampling
n_depth = 3                             # max_depth of tree

models = []  # base model list
for i in range(n_estimators):
    # Create a tree for Random Forest
    model = MyDTreeClassifierRF(max_depth=n_depth,
                                max_samples = x_train.shape[0],
                                max_features=n_features)
    # train the tree
    # subsampling by rows and columns is performed within the
    # model
    model.fit(x_train, y_train)

    # save trained tree
    models.append(model)

# prediction
y_estimates = np.zeros(shape=(x_test.shape[0], n_estimators))
for i, model in enumerate(models):
    y_estimates[:, i] = model.predict(x_test)

# synthesizing the estimation results
y_prob = y_estimates.mean(axis=1)
y_pred = (y_prob >= 0.5) * 1
print('\nAccuracy = {:.4f}'.format((y_pred == y_test).mean()))
```

```
Result:
Accuracy = 0.8027

Accuracy = 0.8161
Accuracy = 0.7982
Accuracy = 0.8296
```

Result of running multiple times

MX-AI

- Coding practice: Implement Random Forest from scratch

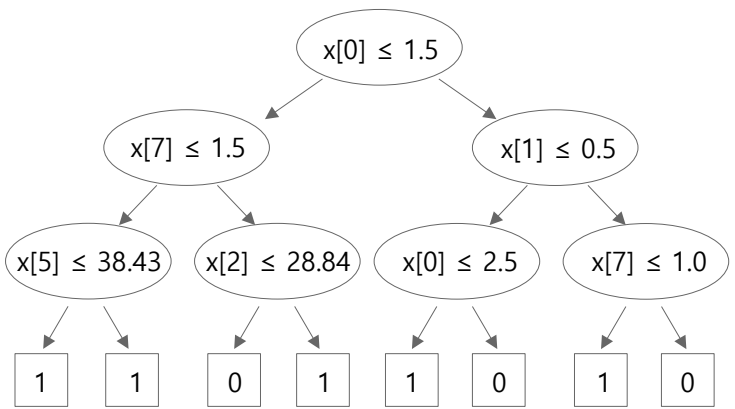
```
models
[<MyDTreeClassifierRF.MyDTreeClassifierRF at 0x7f7c89047bb0>,
 <MyDTreeClassifierRF.MyDTreeClassifierRF at 0x7f7c89047a30>,
 <MyDTreeClassifierRF.MyDTreeClassifierRF at 0x7f7c89044ee0>,
 <MyDTreeClassifierRF.MyDTreeClassifierRF at 0x7f7c89046a40>,
 ...

y_estimates.shape
(223, 100)

y_estimates
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 1., 1., ..., 1., 0., 0.],
       [1., 1., 1., ..., 1., 0., 0.],
       ...,

y_estimates[0, :]
array([0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 1., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 1., 1., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 1., 0., 0.,
       1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 1., 0., 0., 0.,
       0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 1., 1., 1., 0.,
       0., 1., 1., 0., 0., 0., 0., 0., 0., 0.])
```

```
(y_estimates[0, :] == 0.0).sum()
78

(y_estimates[0, :] == 1.0).sum()
22

y_prob[0]  <-- P(y = 1)
0.22

y_pred[0]  <-- The class of the first test data point is
0              estimated as 0.
```

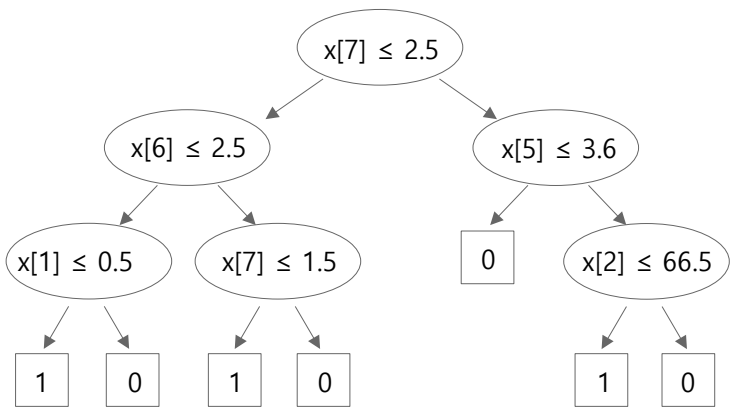
MX-AI

- Coding practice: Implement Random Forest from scratch

```
models[1].estimator2
Out[4]:
{'fid': 0,
 'split_point': 1.5,
 'left': {'fid': 7,
  'split_point': 1.5,
  'left': {'fid': 5, 'split_point': 38.43, 'left': 1, 'right': 1},
  'right': {'fid': 2,
   'split_point': 28.84,
   'left': 0,
   'right': 1}},
 'right': {'fid': 1,
  'split_point': 0.5,
  'left': {'fid': 0, 'split_point': 2.5, 'left': 1, 'right': 0},
  'right': {'fid': 7, 'split_point': 1.0, 'left': 1, 'right': 0}}}
```




```
models[3].estimator2
Out[6]:
{'fid': 7,
 'split_point': 2.5,
 'left': {'fid': 6,
  'split_point': 2.5,
  'left': {'fid': 1, 'split_point': 0.5, 'left': 1, 'right': 0},
  'right': {'fid': 7, 'split_point': 1.5, 'left': 1, 'right': 0}},
 'right': {'fid': 5,
  'split_point': 3.60,
  'left': 0,
  'right': {'fid': 2, 'split_point': 66.5, 'left': 1, 'right': 0}}}
```



MX-AI

▪ Coding practice: Implement Random Forest using DecisionTreeClassifier & RandomForestClassifier

```
# [MXML-8-02]: 2.RF(sklearn).py
# Implement Random Forest using scikit-learn.
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Read preprocessed Titanic data.
df = pd.read_csv('data/titanic_clean.csv')
y = np.array(df['Survived'])
x = np.array(df.drop('Survived', axis=1))
x_train, x_test, y_train, y_test = train_test_split(x, y)

n_estimators = 100
n_depth = 3  # max_depth of tree

# Implement Random Forest using DecisionTreeClassifier
models = []  # base model list
n = x_train.shape[0]  # the number of train data points
for i in range(n_estimators):
    # row subsampling
    i_row = np.random.choice(np.arange(0, n), n, replace=True)
    x_sample = x_train[i_row, :]
    y_sample = y_train[i_row]

    # Create a tree for Random Forest
    # Column subsampling for each split is performed within the
    # model.
    model = DecisionTreeClassifier(max_depth=n_depth,
                                   max_features="sqrt")
    # train the tree.
    model.fit(x_sample, y_sample)

    # save trained tree
    models.append(model)

# prediction
y_estimates = np.zeros(shape=(x_test.shape[0], n_estimators))
for i, model in enumerate(models):
    y_estimates[:, i] = model.predict(x_test)

# synthesizing the estimation results
y_prob = y_estimates.mean(axis=1)
y_pred = (y_prob >= 0.5) * 1
print('\nAccuracy1 = {:.4f}'.format((y_pred == y_test).mean()))

# Implement Random Forest using RandomForestClassifier
model = RandomForestClassifier(n_estimators=n_estimators,
                               max_depth=n_depth,
                               max_samples=n,       # default
                               max_features="sqrt") # default
model.fit(x_train, y_train)
y_pred = model.predict(x_test)
print('\nAccuracy2 = {:.4f}'.format((y_pred == y_test).mean()))

model.estimators_
# [DecisionTreeClassifier(max_depth=3, max_features='sqrt',
#                         random_state=1090277217),
#  DecisionTreeClassifier(max_depth=3, max_features='sqrt',
#                         random_state=1758239483),
#  DecisionTreeClassifier(max_depth=3, max_features='sqrt',
#                         random_state=1420256802) ...
```

MX-AI

▪ Out-Of-Bag (OOB) score (or error rate)

- Data points that are not selected by row subsampling are called Out-of-Bag (OOB) data. OOB data can be used to evaluate the performance of the model.
- Using OOB score eliminates the need for cross-validation. No need to use a validation dataset.
- If you have a lot of data available, you can use a separate validation dataset to evaluate your model's performance, but if you have less data, you can use OOB data to evaluate its performance. OOB increases the efficiency of data use.

Training data

	x_1	x_2	x_3	x_4
i=1				
i=2				
i=3				
i=4				
i=5				

Data points selected for each tree

T_1	T_2	T_3	T_4	T_5	T_6
4	1	3	1	3	5
2	5	3	5	2	5
3	5	2	1	1	1
2	1	2	2	1	2
4	1	4	2	5	2

OOB data points

T_1	T_2	T_3	T_4	T_5	T_6
1	2	1	3	4	3
5	3	5	4		4
	4				

OOB tree list

	Tree list			
i=1	T_1	T_3		
i=2	T_2			
i=3	T_2	T_4	T_6	
i=4	T_2	T_4	T_5	T_6
i=5	T_1	T_3		

i=1 data point is used to evaluate T_1 and T_3
i=2 data point is used to evaluate T_2
i=3 data point is used to evaluate T_2, T_4, T_6
i=4 data point is used for T_2, T_4, T_5, T_6
i=5 data point is used for T_1, T_3

- The probability that the subset does not contain the original data. (Out-Of-Bag probability)
- the number of subsets is n, and the number of data points in a subset is also n.
- The probability that
  data instance i will be selected when selecting a sample: $\frac{1}{n}$
  data instance i will not be selected when selecting a sample: $1-\frac{1}{n}$
  data instance i will not be selected at all while selecting n samples: $\left(1-\frac{1}{n}\right)^n$
- If n is large enough: $\lim_{n\to\infty}\left(1-\frac{1}{n}\right)^n = \frac{1}{e} = 0.3679$ ← OOB probability

OOB tree map

	T_1	T_2	T_3	T_4	T_5	T_6
i=1	1		1			
i=2		1				
i=3		1		1		1
i=4		1		1	1	1
i=5	1		1			

OOB prediction map

	T_1	T_2	T_3	T_4	T_5	T_6
i=1	$T_1(x_1)$		$T_3(x_1)$			
i=2		$T_2(x_2)$	$T_3(x_2)$			
i=3		$T_2(x_3)$		$T_4(x_3)$		$T_6(x_3)$
i=4		$T_2(x_4)$		$T_4(x_4)$	$T_5(x_4)$	$T_6(x_4)$
i=5	$T_1(x_5)$		$T_3(x_5)$			

prediction

$\frac{1}{2}\sum_{i=[1,3]} T_i(x_1)$

$\frac{1}{3}\sum_{i=[2,4,6]} T_i(x_3)$

MX-AI

- Implement Random Forest from scratch and measure OOB score

```
# This code is a part of MyDTreeClassifierRF class.
# You can find the full code at [MXML-8-02] video.
# In this video, we'll just look at the fit() function.
```

self.iob_pred

i	0	1	2	3	4	5	...
y_pred	-1	0	-1	1	0	-1	

self.oob_pred

i	0	1	2	3	4	5	...
y_pred	0	-1	1	-1	-1	0	

```
# create a tree using training data
# x : feature data, y: target data
def fit(self, x, y):
    # perform row subsampling with replacement
    n = x.shape[0]
    i_rows = np.random.choice(np.arange(0, n), self.max_samples,
                              replace=True)

    self.feature = x[i_rows, :]
    self.target = y[i_rows]
    self.u_class = np.unique(y)

    # Initially, the root node holds all data points IDs.
    root = self.node_split(np.arange(x.shape[0]))
    if isinstance(root, dict):
        self.recursive_split(root, curr_depth=1)

    # tree result-1. Every leaf node has data point IDs.
    self.estimator1 = root

    # tree result-2. Every leaf node has the majority class.
    self.estimator2 = copy.deepcopy(self.estimator1)
    self.update_leaf(self.estimator2)          # tree result-2

    # predict In-Of-Bag (IOB) and Out-Of-Bag (OOB) data points
    # initialize the predicted classes of IOB and OOB data points
    self.iob_pred = np.ones(shape=(x.shape[0],), dtype=int) * -1
    self.oob_pred = np.ones(shape=(x.shape[0],), dtype=int) * -1

    # predict training dataset
    y_pred = self.predict(x)

    # predict IOB and OOB data points
    i_train = set(np.arange(0, x.shape[0]))
    i_oobs = list(i_train - set(i_rows))   # OOB data point IDs

    self.iob_pred[i_rows] = y_pred[i_rows] # for IOB data
    self.oob_pred[i_oobs] = y_pred[i_oobs] # for OOB data

    return self.iob_pred, self.oob_pred

# Estimate the target class of a test data.
def x_predict(self, p, x):
    if x[p['fid']] <= p['split_point']:
        if isinstance(p['left'], dict): # recursion if not leaf.
            return self.x_predict(p['left'], x)   # recursion
        else:           # return the value in the leaf, if leaf.
            return p['left']
    else:
        if isinstance(p['right'], dict) # recursion if not leaf.
            return self.x_predict(p['right'], x)  # recursion
        else:           # return the value in the leaf, if leaf.
            return p['right']

# Estimate the target class of a x_test.
def predict(self, x_test):
    p = self.estimator2    # predictor
    y_pred = [self.x_predict(p, x) for x in x_test]
    return np.array(y_pred)
```

MX-AI

- Implement Random Forest from scratch and measure OOB score

```
# [MXML-8-03] 3.RF_OOB.py
# Add Out-Of-Bag (OOB) score feature to 2.RF(titanic).py.
import numpy as np
import pandas as pd
from MyDTreeClassifierRF import MyDTreeClassifierRF
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Read preprocessed Titanic data.
df = pd.read_csv('data/titanic_clean.csv')

# Survived  Pclass  Sex   Age  SibSp  Parch   Fare  Embarked  Title
#        0       3    1  22.0      1      0   3.62         3      2
#        1       1    0  38.0      1      0  35.64         0      3
#        1       3    0  26.0      0      0   7.92         3      1
#        1       1    0  35.0      1      0  26.55         3      3

N = x_train.shape[0]  # the number of train data points
y = np.array(df['Survived'])
x = np.array(df.drop('Survived', axis=1))
x_train, x_test, y_train, y_test = train_test_split(x, y)

n_estimators = 50
n_depth = 5        # max_depth of tree
max_features = round(np.sqrt(x_train.shape[1])) # for column sampling

# majority vote for iob_pred, or oob_pred (p = iob_pred or oob_pred)
def majority_vote(p):
    cnt_0 = (p == 0).sum(axis=1)
    cnt_1 = (p == 1).sum(axis=1)
    cnts = np.array([cnt_0, cnt_1])   # shape = (2, 668)
    return np.argmax(cnts, axis=0)
```

```
models = []     # base model list
iob_score = []  # Error rate measured with IOB
oob_score = []  # Error rate measured with OOB

# initialize IOB and OOB prediction map
iob_pred = np.ones(shape=(N, n_estimators)) * -1
oob_pred = np.ones(shape=(N, n_estimators)) * -1

# Create n_estimators models
for i in range(n_estimators):
    # Create a tree for Random Forest
    model = MyDTreeClassifierRF(
                max_depth=n_depth,
                max_samples = N,
                max_features = max_features)

    p1, p2 = model.fit(x_train, y_train)

    # save trained tree
    models.append(model)

    # Create IOB and OOB prediction map
    iob_pred[:, i] = p1
    oob_pred[:, i] = p2

    # Calculate IOB and OOB score
    y_trn = majority_vote(iob_pred)
    y_oob = majority_vote(oob_pred)

    iob_score.append((y_trn != y_train).mean())
    oob_score.append((y_oob != y_train).mean())
```

iob_pred

	T_1	T_2	T_3	T_4	T_5	T_6
i=1	.	0	.	1	0	.
i=2	1	.	1	0	.	1
i=3	.	0	.	0	.	0
i=4	1	1	.	.	1	0
i=5	1	.	1	0	.	1

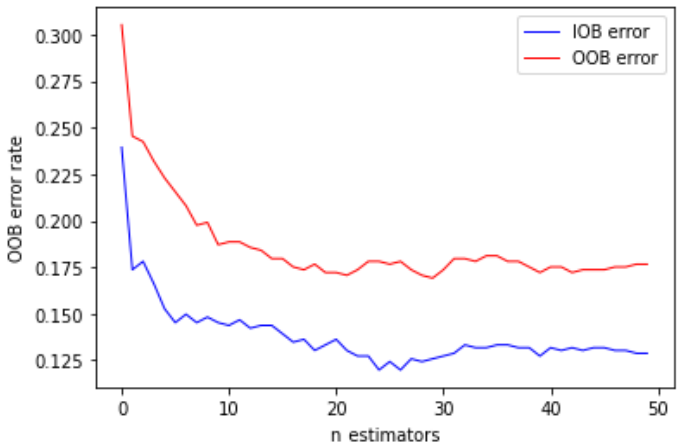
oob_pred

	T_1	T_2	T_3	T_4	T_5	T_6
i=1	0	.	0	.	.	1
i=2	.	1	.	.	1	.
i=3	0	.	0	.	1	.
i=4	.	.	1	0	.	.
i=5	.	1	.	.	1	.

MX-AI

▪ Implement Random Forest from scratch and measure OOB score

```
# Visualize IOB and OOB score
plt.figure(figsize=(6, 4))
plt.plot(iob_score, color='blue', lw=1.0, label='IOB error')
plt.plot(oob_score, color='red', lw=1.0, label='OOB error')
plt.legend()
plt.xlabel('n_estimators')
plt.ylabel('OOB error rate')
plt.show()

# prediction
y_estimates = np.zeros(shape=(x_test.shape[0], n_estimators))
for i, model in enumerate(models):
    y_estimates[:, i] = model.predict(x_test)

# synthesizing the estimation results
y_prob = y_estimates.mean(axis=1)
y_pred = (y_prob >= 0.5) * 1
accuracy = (y_pred == y_test).mean()
print('\nAccuracy of test data = {:.4f}'.format(accuracy))
print('final OOB error rate = {:.4f}'.format(oob_score[-1]))

# OOB probability
# In theory, it would be 0.3679.
# This means that x_train is selected with probability 0.6321
# by row subsampling. (1.0 - 0.3679 = 0.6321)
oob_percent = ((oob_pred >= 0).sum(axis=0) / N).mean()
print('OOB probability = {:.4f}'.format(oob_percent))
```




```
Accuracy of test data = 0.8251
Final OOB error rate = 0.1811
OOB probability = 0.3644

oob_pred[:8, :10]

array([[-1., -1.,  0., -1.,  0., -1., -1.,  1.,  0.,  0.],
       [-1., -1., -1., -1.,  1.,  1., -1., -1., -1., -1.],
       [ 0.,  1.,  1., -1., -1.,  1., -1.,  0.,  1., -1.],
       [ 0., -1., -1., -1., -1., -1.,  0., -1., -1., -1.],
       [-1., -1., -1., -1.,  0., -1.,  0., -1.,  0.,  0.],
       [-1., -1.,  0., -1.,  0., -1., -1., -1., -1., -1.],
       [ 1.,  1.,  1.,  1., -1.,  1.,  1., -1., -1., -1.],
       [-1., -1., -1., -1., -1., -1., -1.,  1.,  1., -1.])
```

MX-AI

- Implement Random Forest using DecisionTreeClassifier and RandomForestClassifier and measure OOB score

```
# [MXML-8-03] 4.RF_OOB(sklearn).py
# Add Out-Of-Bag (OOB) score feature to 2.RF(sklearn).py
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Read preprocessed Titanic data.
df = pd.read_csv('data/titanic_clean.csv')
y = np.array(df['Survived'])
x = np.array(df.drop('Survived', axis=1))
x_train, x_test, y_train, y_test = train_test_split(x, y)

N = x_train.shape[0]  # the number of train data points
n_estimators = 50
n_depth = 5                                        # tree의 max_depth
max_features= round(np.sqrt(x_train.shape[1])) # for column sampling

# Implement Random Forest using DecisionTreeClassifier
# ----------------------------------------------------
# majority vote for iob_pred, or oob_pred
# p = iob_pred or oob_pred
def majority_vote(p):
    cnt_0 = (p == 0).sum(axis=1)
    cnt_1 = (p == 1).sum(axis=1)
    cnts = np.array([cnt_0, cnt_1])   # shape = (2, 668)
    return np.argmax(cnts, axis=0)
```

```
sk_models = []  # base model list
iob_score = []  # Error rate measured with IOB
oob_score = []  # Error rate measured with OOB

# initialize IOB and OOB prediction map
iob_pred = np.ones(shape=(N, n_estimators)) * -1
oob_pred = np.ones(shape=(N, n_estimators)) * -1
i_train = set(np.arange(0, N))

# Create n_estimators models
for i in range(n_estimators):
    # row subsampling
    i_row = np.random.choice(np.arange(0, N), N, replace=True)
    x_sample = x_train[i_row, :] # bootstrapped data
    y_sample = y_train[i_row]

    # Create a Decision Tree for Random Forest
    # Column sampling for each split is performed within the model.
    model = DecisionTreeClassifier(max_depth=n_depth,
                                   max_features="sqrt")
    # Training
    model.fit(x_sample, y_sample)
    sk_models.append(model)

    # Create IOB and OOB prediction map
    i_oob = list(i_train - set(i_row))   # OOB index
    iob_pred[i_row, i] = model.predict(x_train[i_row])
    oob_pred[i_oob, i] = model.predict(x_train[i_oob])
```

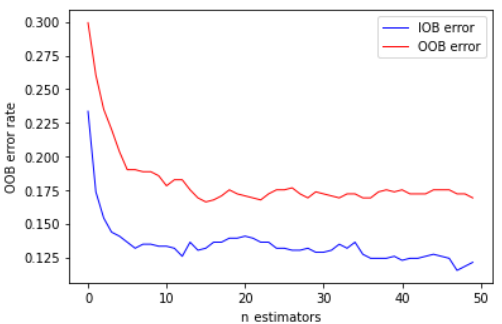
MX-AI

- Implement Random Forest using DecisionTreeClassifier and RandomForestClassifier and measure OOB score

```
    # Calculate IOB and OOB score
    y_trn = majority_vote(iob_pred)
    y_oob = majority_vote(oob_pred)

    iob_score.append((y_trn != y_train).mean())
    oob_score.append((y_oob != y_train).mean())

# Visualize IOB and OOB score
plt.figure(figsize=(6, 4))
plt.plot(iob_score, color='blue', lw=1.0, label='IOB error')
plt.plot(oob_score, color='red', lw=1.0, label='OOB error')
plt.legend()
plt.xlabel('n_estimators')
plt.ylabel('OOB error rate')
plt.show()

# prediction
y_estimates = np.zeros(shape=(x_test.shape[0], n_estimators))
for i, model in enumerate(sk_models):
    y_estimates[:, i] = model.predict(x_test)

# synthesizing the estimation results
y_prob = y_estimates.mean(axis=1)
y_pred = (y_prob >= 0.5) * 1
accuracy = (y_pred == y_test).mean()
print('\nThe result from DecisionTreeClassifier:')
print('Accuracy of test data = {:.4f}'.format(accuracy))
print('Final OOB error rate = {:.4f}'.format(oob_score[-1]))
```

```
# OOB probability: In theory, it would be 0.3679.
# This means that x_train is selected with probability 0.6321
# by row subsampling. (1.0 - 0.3679 = 0.6321)
oob_percent = ((oob_pred >= 0).sum(axis=0) / N).mean()
print('OOB probability = {:.4f}'.format(oob_percent))

# Implement Random Forest using RandomForestClassifier
# ----------------------------------------------------
rf_model = RandomForestClassifier(n_estimators=n_estimators,
                                  max_depth=n_depth,
                                  max_features="sqrt",  # default
                                  max_samples=N,        # default
                                  oob_score=True)
rf_model.fit(x_train, y_train)
y_pred1 = rf_model.predict(x_test)

print('\nThe result from RandomForestClassifier:')
print('Accuracy of test data = {:.4f}'\
      .format((y_pred1 == y_test).mean()))
print('Final OOB error rate = {:.4f}'\
      .format(1 - rf_model.oob_score_))
```

```
The result from DecisionTreeClassifier:
Accuracy of test data = 0.7937
Final OOB error rate = 0.1692
OOB probability = 0.3685

The result from RandomForestClassifier:
Accuracy of test data = 0.8117
Final OOB error rate = 0.1677
```



MX-AI

▪ Missing value : Proximity Matrix

- If there are missing values in the data, they can be estimated through Random Forest. The idea is to estimate the missing values in data points by referring to the values in similar data points. Proximity matrix (PM) is used to measure the similarity. (1 – PM) is distance matrix.
- Proximity matrix (PM) is generated by the following procedure. When normalizing PM, it is divided by the number of trees. Here, it is normalized so that the sum of the columns is 1, as shown in the bottom right. This is to create weights for a weighted average to be used later.

x y

	Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
i=1	No	No	No	125	No
i=2	Yes	Yes	Yes	180	Yes
i=3	Yes	Yes	No	210	No
i=4	Yes	Yes	No	167.5	No

The first round of training, leaf indices in each tree

tree 1	tree 2	tree 3	...	tree 10
3	3	1		
0	0	0		
1	0	2		
1	0	2		

missing values — most common value — median or average value

initial guess

Only data points where y=No are considered.

```
model=RandomForestClassifier(...)
model.fit(x, y)
leaf_id = model.apply(x)
```

* apply trees in the forest to x, return leaf indices.

proximity matrix of tree-1

	i=1	i=2	i=3	i=4
i=1				
i=2				
i=3				1
i=4			1	

```
t = np.array([3, 0, 1, 1])
pm = np.equal.outer(t, t) * 1
np.fill_diagonal(pm, 0)
```

proximity matrix of tree-2

	i=1	i=2	i=3	i=4
i=1				
i=2			1	1
i=3		1		1
i=4		1	1	

```
t = np.array([3, 0, 0, 0])
pm = np.equal.outer(t, t) * 1
np.fill_diagonal(pm, 0)
```

proximity matrix of tree-3

	i=1	i=2	i=3	i=4
i=1				
i=2				
i=3				1
i=4			1	

```
t = np.array([1, 0, 2, 2])
pm = np.equal.outer(t, t) * 1
np.fill_diagonal(pm, 0)
```

∑

	i=1	i=2	i=3	i=4
i=1				
i=2			1	1
i=3		1		3
i=4		1	3	

element-wise sum of the proximity matrices from tree-1 to tree-3.

	i=1	i=2	i=3	i=4
i=1		2	1	1
i=2	2		1	1
i=3	1	1		8
i=4	1	1	8	

If element-wise sum of the proximity matrices from tree-1 to tree-10 is:

	i=1	i=2	i=3	i=4
i=1	0	0.5	0.1	0.1
i=2	0.5	0	0.1	0.1
i=3	0.25	0.25	0	0.8
i=4	0.25	0.25	0.8	0
∑	1	1	1	1

normalization of column values

Proximity Matrix

The data point number 3 and 4 are in the same leaf node in 3 trees. This means that the 2 data points are similar.

* Reference: [StatQuest] Random Forests Part 2: Missing data and clustering

MX-AI

▪ Missing value imputation – training data

- Missing values are estimated using the proximity matrix (PM).
- Continuous variable is estimated as a weighted average using PM, and categorical variable is estimated as a category with a large probability times weight.
- Update the proximity matrix using the estimated missing values and repeat this process until there are no changes.

x y

	Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
i=1	No	No	No	125	No
i=2	Yes	Yes	Yes	180	Yes
i=3	Yes	Yes	No	210	No
i=4	Yes	Yes	No	167.5	No

missing values

initial guess

Proximity Matrix (pm)

	i=1	i=2	i=3	i=4
i=1	0	0.5	0.1	0.1
i=2	0.5	0	0.1	0.1
i=3	0.25	0.25	0	0.8
i=4	0.25	0.25	0.8	0
∑	1	1	1	1

▪ **Continuous variable**: Weight

np.dot(x[:, 3].reshape(1, -1), pm)

$$\begin{bmatrix}125 & 180 & 210 & 167.5\end{bmatrix} \cdot \begin{array}{c} \begin{matrix} i=1 & i=2 & i=3 & i=4 \end{matrix} \\ \begin{bmatrix} 0.00 & 0.50 & 0.10 & 0.10 \\ 0.50 & 0.00 & 0.10 & 0.10 \\ 0.25 & 0.25 & 0.00 & 0.80 \\ 0.25 & 0.25 & 0.80 & 0.00 \end{bmatrix} \end{array} = \begin{bmatrix}184.38 & 156.88 & 164.5 & 198.5\end{bmatrix}$$

The "weight" value for data point number 4 is assumed to be this value.

125 * 0.1 + 180 * 0.1 + 210 * 0.8 + 167.5 * 0

▪ **Categorical variable**: Blocked Arteries

For each category, calculate the probability times the weight.
The probability is calculated by excluding missing values.

Yes: $\frac{1}{3} \times \frac{0.1}{0.1+0.1+0.8} = 0.033$

No: $\frac{2}{3} \times \frac{0.1+0.8}{0.1+0.1+0.8} = 0.6$ ← Since it is larger, the missing value is assumed to be "No".

Because of normalization, the denominator is always 1.

* Reference: [StatQuest] Random Forests Part 2: Missing data and clustering

▪ Missing value imputation – test data

* Reference [2] in [MXML-8-01]

Proximities

<skipped>

When a test set is present, the proximities of each case in the test set with each case in the training set can also be computed. The amount of additional computing is moderate.

Missing value replacement for the test set

When there is a test set, there are two different methods of replacement depending on whether labels exist for the test set.

If they do, then the fills derived from the training set are used as replacements. If labels no not exist, then each case in the test set is replicated nclass times (nclass= number of classes). The first replicate of a case is assumed to be class 1 and the class one fills used to replace missing values. The 2nd replicate is assumed class 2 and the class 2 fills used on it.

This augmented test set is run down the tree. In each set of replicates, the one receiving the most votes determines the class of the original case.

▪ **Continuous variable:** (No detailed explanation in the documentation. This is my guess.)

	Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
i=1	No	No	No	125	No
i=2	Yes	Yes	Yes	180	Yes
i=3	Yes	Yes	No	210	No
i=4	Yes	Yes	No	198.5	No
test	Yes	No	No	178.4	?

missing value

tree 1	tree 2	tree 3	…
3	3	1	
0	0	0	
1	0	2	
1	0	2	
1	1	2	

tree의 leaf-id matrix

	i=1	i=2	i=3	i=4	test
i=1	0	0.4	0.1	0.1	
i=2	0.4	0	0.1	0.1	
i=3	0.25	0.2	0	0.5	0.3
i=4	0.25	0.2	0.5	0	0.3
test	0.1	0.2	0.3	0.3	

Proximity Matrix

1. Initial guess. There is no y in the test data, so all data points, including the training data, are used.
2. Create a proximity matrix using all data points, including the training data.
3. Estimate the missing value using proximity matrix

* Please see the code in the following video for more details.

▪ **Categorical variable:**

label = [Yes, No]

	Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
test	Yes	No	?	125	?

Example from above reference

replicate

(A)

	Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
test	Yes	No	**Yes**	125	**Yes**

Since we have y, we estimate it in the same way as the train data.

y_pred = [1, 1, 0, 0, 1] → Yes = 3/5

(B)

	Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
test	Yes	No	**No**	125	**No**

y_pred = [1, 1, 0, 1, 1] → No = 1/5

consider all cases

Use the model to predict replicated data, A and B, then select the one with higher accuracy in y_pred. In this example, the missing value is assumed to be "yes".

MX-AI

▪ Implement of Missing value imputation using Proximity Matrix

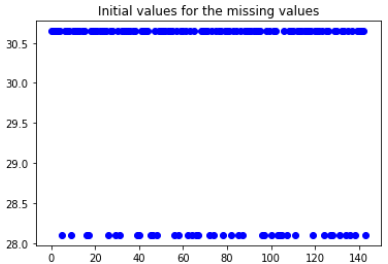
```
# [MXML-8-05] 5.RF_proximity.py
# Add missing value estimation functionality to
# RandomForestClassifier.
# To understand this code, you need to learn about the
# proximity matrix from the previous video, [MXML-8-04].

import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Read Titanic dataset
df = pd.read_csv('data/titanic.csv')
df['Embarked'].fillna('N', inplace = True) # Replace with 'N'
df['Sex'] = df['Sex'].factorize()[0]           # encoding
df['Embarked'] = df['Embarked'].factorize()[0] # encoding
df.drop(['PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1,
        inplace=True)
df.info()

# RangeIndex: 891 entries, 0 to 890
# Data columns (total 8 columns):
#  #   Column    Non-Null Count  Dtype
# ---  ------    --------------  -----
#  0   Survived  891 non-null    int64
#  1   Pclass    891 non-null    int64
#  2   Sex       891 non-null    int64
#  3   Age       714 non-null    float64
#  4   SibSp     891 non-null    int64
#  5   Parch     891 non-null    int64
#  6   Fare      891 non-null    float64
#  7   Embarked  891 non-null    int64
```

```
#     Survived  Pclass  Sex   Age  SibSp  Parch     Fare  Embarked
# 0         0       3    0  22.0      1      0   7.2500         0
# 3         1       1    1  35.0      1      0  53.1000         0
# 4         0       3    0  35.0      0      0   8.0500         0
# 5         0       3    0   NaN      0      0   8.4583         2
# 6         0       1    0  54.0      0      0  51.8625         0
# 8         1       3    1  27.0      0      2  11.1333         0

# create training and test data
y = np.array(df['Survived'])
x = np.array(df.drop('Survived', axis=1))
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

# Initially, missing values in 'Age' are replaced with the average value.
AGE = 2  # column number of 'Age' feature

# the position of missing values. It is for later use.
i_train = np.where(np.isnan(x_train[:, AGE]))[0]  # training data
i_test = np.where(np.isnan(x_test[:, AGE]))[0]    # test data

# indices where y_train=0, and y_train=1
i_y0 = np.where(y_train == 0)[0]
i_y1 = np.where(y_train == 1)[0]

# the mean value of 'Age' where y_train=0, and the same where y_train=1.
y0_mean = np.nanmean(x_train[i_y0, AGE]) # where y_train=0
y1_mean = np.nanmean(x_train[i_y1, AGE]) # where y_train=1

# replace nan in 'Age' where y_train = 0 to y0_mean
x_train[i_y0, AGE] = np.nan_to_num(x_train[i_y0, AGE], nan=y0_mean)

# replace nan in 'Age' where y_train = 1 to y1_mean
x_train[i_y1, AGE] = np.nan_to_num(x_train[i_y1, AGE], nan=y1_mean)
```

MX-AI

▪ Implement of Missing value imputation using Proximity Matrix

```
print('Before:\n', x_train.round(2))
print('y0_mean={:.2f}, y1_mean={:.2f}'.format(y0_mean, y1_mean))
plt.plot(x_train[i_train, AGE], 'bo')
plt.title('Initial values for the missing values')
plt.show()
#i_train = np.where(np.isnan(x_train[:, AGE]))[0]

# Before: [[ 3.    0.   36.   ...  0.    7.5   0.  ]
#          [ 3.    1.   27.78 ...  0.   16.1   0.  ]
#          [ 2.    0.   25.   ...  0.   13.    0.  ]
# y0_mean = 30.82, y1_mean = 27.78
```



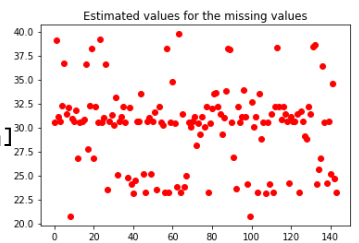
leaf node ID

T1	T2	T3	...
3	3	1	
0	0	0	
1	0	2	
1	0	2	

pm

	i=1	i=2	i=3	i=4
i=1	0	0.5	0.1	0.1
i=2	0.5	0	0.1	0.1
i=3	0.25	0.25	0	0.8
i=4	0.25	0.25	0.8	0

```
# Create Proximity matrix
# normalize = 0: pm / n_tree
# normalize ≠ 0: Normalize columns to sum to 1
def proximity_matrix(model, x, normalize=0):
    n_tree = len(model.estimators_)

    # Apply trees in the forest to X, return leaf indices.
    leaf = model.apply(x)  # shape = (x.shape[0], n_tree)

    pm = np.zeros(shape=(x.shape[0], x.shape[0]))
    for i in range(n_tree):
        t = leaf[:, i]
        pm += np.equal.outer(t, t) * 1.
    np.fill_diagonal(pm, 0)
    if normalize == 0:
        return pm / n_tree
    else:
        return pm / pm.sum(axis=0, keepdims=True)

n_estimators = 50
n_depth = 5

# Missing value imputation using the proximity matrix
for i in range(5):   # 5 iterations
    model = RandomForestClassifier(n_estimators=n_estimators,
                                   max_depth=n_depth,
                                   oob_score=True)
    model.fit(x_train, y_train)

    # Create proximity matrix
    pm = proximity_matrix(model, x_train, normalize=1)

    # estimate the missing values of 'Age' using the proximity matrix
    x_age = x_train[:, AGE].copy()
    u_age = np.dot(x_age, pm) # updated 'Age'
    x_train[i_train, AGE] = u_age[i_train]

print('\nAfter:\n', x_train.round(2))
plt.plot(x_train[i_train, AGE], 'ro')
plt.title('Estimated values for the missing val
plt.show()

# After:
# [[ 3.    0.   36.   ...  0.    7.5   0.  ]
#  [ 3.    1.   26.45 ...  0.   16.1   0.  ]
#  [ 2.    0.   25.   ...  0.   13.    0.  ]
```



MX-AI

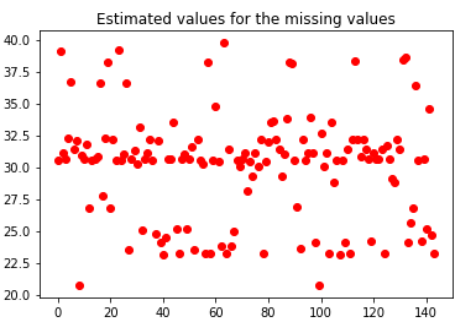
▪ Implement of Missing value imputation using Proximity Matrix

```
# Train a new model after imputing missing values of training data
model = RandomForestClassifier(n_estimators=n_estimators,
                               max_depth=n_depth,
                               oob_score=True)
model.fit(x_train, y_train)

# Predict the test data. There are also missing values in 'Age' in the
# test data.
#
# [2] Proximities
# When a test set is present, the proximities of each case in the test
# set with each case in the training set can also be computed.

# Initially, there is no target value y in the test data, so the
# missing values are replaced with the mean value of the training data.
x_test[i_test, AGE] = x_train[:, AGE].mean()

x_data = np.vstack([x_train, x_test]) # combine training and test data
pm = proximity_matrix(model, x_data, normalize=1)
x_age = x_data[:, AGE].copy()         # 'Age' feature data
u_age = np.dot(x_age, pm)             # updated 'Age' feature

u_age = u_age[-x_test.shape[0]:]      # 'Age' of test data
u_test = x_data[-x_test.shape[0]:]    # test data
u_test[i_test, AGE] = u_age[i_test]   # update the missing values in
                                      # test data


# predict
y_pred = model.predict(u_test)

print('\nResults:')
print('Accuracy = {:.4f}'.format((y_pred == y_test).mean()))
print('Final OOB error rate = {:.4f}'.format(1 - model.oob_score_))
```

```
Before:
 [[ 3.    0.   36.   ...  0.    7.5   0.  ]
  [ 3.    1.   27.78 ...  0.   16.1   0.  ]
  [ 2.    0.   25.   ...  0.   13.    0.  ]
  ...
  [ 3.    0.   30.   ...  0.    8.05  0.  ]
  [ 3.    1.   16.   ...  0.    7.73  2.  ]
  [ 2.    1.   24.   ...  1.   27.    0.  ]]

After:
 [[ 3.    0.   36.   ...  0.    7.5   0.  ]
  [ 3.    1.   26.45 ...  0.   16.1   0.  ]
  [ 2.    0.   25.   ...  0.   13.    0.  ]
  ...
  [ 3.    0.   30.   ...  0.    8.05  0.  ]
  [ 3.    1.   16.   ...  0.    7.73  2.  ]
  [ 2.    1.   24.   ...  1.   27.    0.  ]]

Results:
Accuracy = 0.8324
Final OOB error rate = 0.1812
```

PM (example in [MXML-8-04])

	i=1	i=2	i=3	i=4	test
i=1	0	0.4	0.1	0.1	
i=2	0.4	0	0.1	0.1	
i=3	0.25	0.2	0	0.5	0.3
i=4	0.25	0.2	0.5	0	0.3
test	0.1	0.2	0.3	0.3	



MX-AI

▪ Outlier Detection: Algorithm

- Random Forest's proximity matrix can be used to detect outliers.
- It is an outlier detection algorithm in the form of supervised learning.

Reference [2] : **Outliers**

Outliers are generally defined as cases that are removed from the main body of the data. Translate this as: outliers are cases whose proximities to all other cases in the data are generally small. A useful revision is to define outliers relative to their class. Thus, an outlier in class j is a case whose proximities to all other class j cases are small.

(1)

Define the **average proximity** from case n in class j to the rest of the training data class j as:

$$\bar{P}(n) = \sum_{class(k)=j} prox^2(n,k)$$

(2)

The **raw outlier measure** for case n is defined as

$$r(n) = \frac{nsample}{\bar{P}(n)}$$

This will be large if the average proximity is small. Within each class find the median of these raw measures, and their absolute deviation from the median. Subtract the median from each raw measure, and divide by the absolute deviation to arrive at the **final outlier measure**. **(3)**

Proximity matrix (normalized by the number of trees)

case →

	n=1	n=2	n=3	n=4
n=1		0.2	0.1	0.1
n=2	0.2		0.1	0.1
n=3	0.1	0.1		0.8
n=4	0.1	0.1	0.8	

class j

y
0
1
0
0

1) Average proximity

$$\bar{P}(n=1) = \sum_{class(k)=0} prox^2(1,k)$$

$$= \sum_{k=[3,4]} prox^2(1,k) = 0.1^2+0.1^2=0.02$$

$$\bar{P}(n=2) = \sum_{class(k)=1} prox^2(2,k) = 0$$

$$\bar{P}(n=3) = 0.1^2+0.8^2=0.65$$

$$\bar{P}(n=4) = 0.1^2+0.8^2=0.65$$

2) Raw outlier measure

$$r(n=1) = \frac{4}{0.02} = 200$$

$$r(n=2) = \frac{4}{0} = \infty$$

$$r(n=3) = \frac{4}{0.65} = 6.15$$

$$r(n=4) = \frac{4}{0.65} = 6.15$$

3) Final outlier measure

- Data with an excessively large r_f are considered outliers.

y=0인 r(n) = [200, 6.15, 6.15] → median = m_0, absolute deviation = s_0

y=1인 r(n) = [∞, ...] → median = m_1, absolute deviation = s_1

$$r_f(n=1) = \frac{r(n=1)-m_0}{s_0} \qquad r_f(n=3) = \frac{r(n=3)-m_0}{s_0}$$

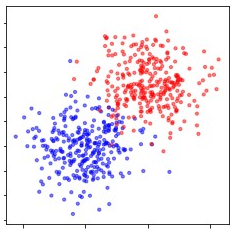
$$r_f(n=2) = \frac{r(n=2)-m_1}{s_1} \qquad r_f(n=4) = \frac{r(n=4)-m_0}{s_0}$$

* Evaluate how far each data point is from the center of the normalized distribution.

MX-AI

▪ Outlier Detection: Implemention

```
# [MXML-8-06] 6.RF_outlier.py
# Outlier detection using Random Forest's proximity matrix
# Reference [2]:
# https://www.stat.berkeley.edu/~breiman/RandomForests/
#         cc_home.htm#outliers
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt

# Generate training data
x, y = make_blobs(n_samples=600, n_features=2,
                  centers=[[0., 0.], [0.5, 0.5]],
                  cluster_std=0.2, center_box=(-1., 1.))
```



```
# Create Proximity matrix
# normalize = 0: pm / n_tree
# normalize ≠ 0: Normalize columns to sum to 1
def proximity_matrix(model, x, normalize=0):
    n_tree = len(model.estimators_)

    # Apply trees in the forest to X, return leaf indices.
    leaf = model.apply(x)  # shape = (x.shape[0], n_tree)

    pm = np.zeros(shape=(x.shape[0], x.shape[0]))
    for i in range(n_tree):
        t = leaf[:, i]
        p = np.equal.outer(t, t) * 1.
        pm += p

    np.fill_diagonal(pm, 0)
    if normalize == 0:
        return pm / n_tree
    else:
        return pm / pm.sum(axis=0, keepdims=True)

n_estimators = 50
n_depth = 5

# Detect outliers using a proximity matrix
model = RandomForestClassifier(n_estimators=n_estimators,
                               max_depth=n_depth,
                               max_features="sqrt",  # default
                               bootstrap=True,       # default
                               oob_score=True)
model.fit(x, y)

# Create a proximity matrix
pm = proximity_matrix(model, x)

i_y0 = np.where(y == 0)[0]
i_y1 = np.where(y == 1)[0]
i_y = [i_y0, i_y1]

# 1) average proximity
pi_bar = []
for i in range(pm.shape[0]):
    j_class = y[i]        # the class of data instance i
    j_same = i_y[j_class] # Data point IDs with the same class as
                          # data point i
    pi_bar.append(np.sum(pm[i, j_same] ** 2))
```

PM

	n=1	n=2	n=3	n=4
n=1		0.2	0.1	0.1
n=2	0.2		0.1	0.1
n=3	0.1	0.1		0.8
n=4	0.1	0.1	0.8	

class j

y
0
1
0
0

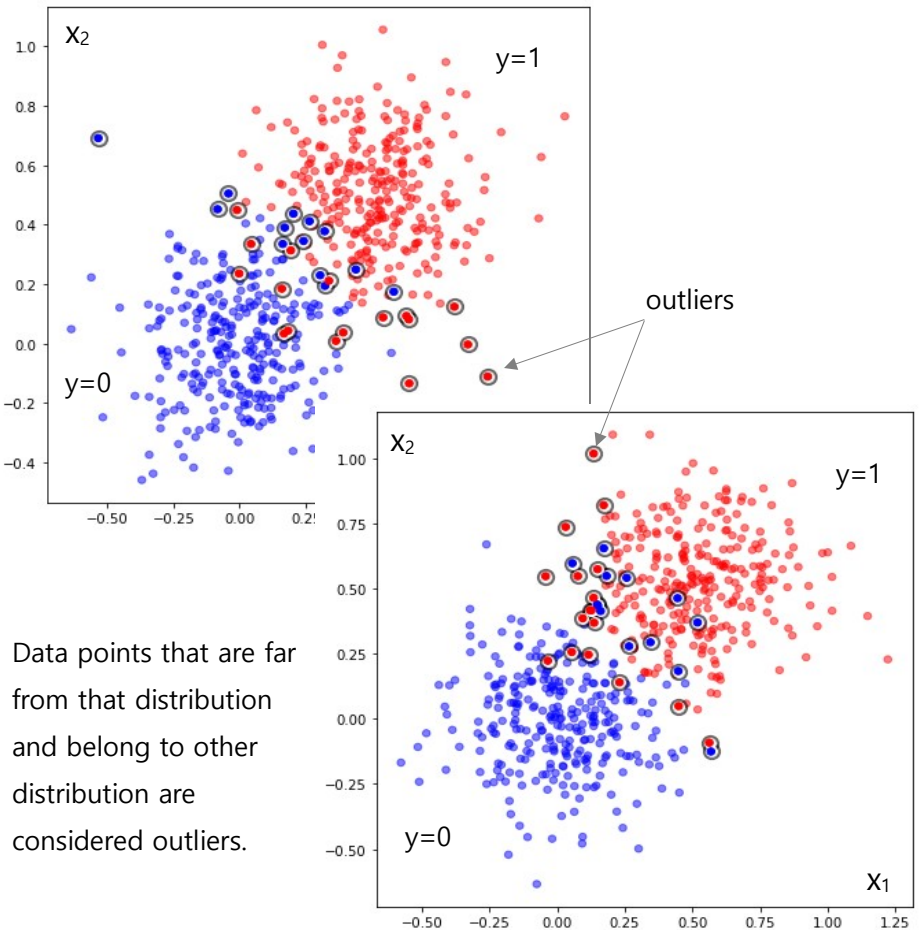
$$\bar{P}(n) = \sum_{class(k)=j} prox^2(n,k)$$

MX-AI

▪ Outlier Detection: Implementation

```
# 2) raw outlier measure
o_raw = x.shape[0] / np.array(pi_bar)

# 3) final outlier measure
# For convenience of coding, the mean value was used instead of the
# median, and the standard deviation was used instead of the absolute
# deviation.
f_measure = []
for i in range(o_raw.shape[0]):
    j_class = y[i]         # the class of the data instance i
    j_same = i_y[j_class]  # Data point IDs with the same class as
                           # data point i
    f_measure.append((o_raw[i] - o_raw[j_same].mean()) /\
                     o_raw[j_same].std())

# Data in the upper top_rate percentage of f_measure are considered
# outliers.
top_rate = 0.05  # top 5%
top_idx = np.argsort(f_measure)[::-1][:int(top_rate * x.shape[0])]

# Visualize normal data and outliers by color.
plt.figure(figsize=(7, 7))
color = [['blue', 'red'][i] for i in y]
color_out = [['blue', 'red'][i] for i in y[top_idx]]
plt.scatter(x[:, 0], x[:, 1], s=30, c=color, alpha=0.5)
plt.scatter(x[top_idx, 0], x[top_idx, 1], s=150, c='black', alpha=0.5)
plt.scatter(x[top_idx, 0], x[top_idx, 1], s=60, c='white')
plt.scatter(x[top_idx, 0], x[top_idx, 1], s=30, c=color_out)
plt.show()
```

$$r(n) = \frac{nsample}{\bar{P}(n)}$$



Data points that are far from that distribution and belong to other distribution are considered outliers.

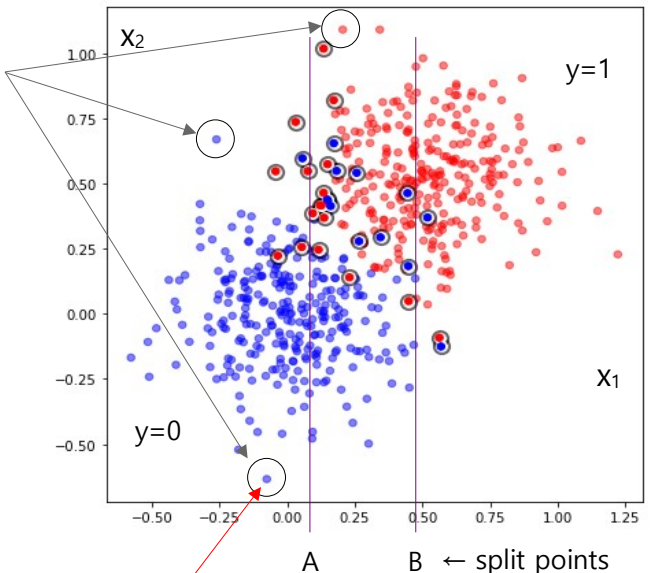
MX-AI

Outlier Detection: Interpretation of the result

- Let's consider why outliers are near the decision boundary.

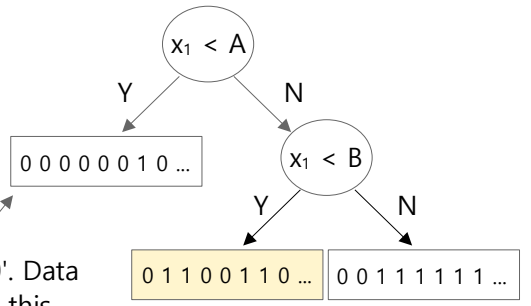
Outlier detection algorithms in the form of unsupervised learning consider these data to be outliers. This is because they are far from the entire distribution.
(ex: Isolation Forest, Auto Encoder, etc)

Random forest is a supervised learning method. Supervised learning type outlier detection algorithms take classes into account to determine outliers.



This data point is far from the y=0 cluster centroid, but belongs to the same leaf as the centroid, so it has high similarity to the centroid data. So it is not considered an outlier.

Within the decision boundary, no matter how far away from the center it is, it is not an error and therefore does not need to be treated as an outlier (in the "hinge loss" concept).



Most of them are '0'. Data points belonging to this leaf node have high similarity to each other. It doesn't matter how far they are from the cluster center.

Most of them are '1'

Since there is a mix of '0' and '1', this node will be split again. Then these data points belong to different leaf nodes and their similarity (proximity) will be lowered. If the proximity becomes smaller, the outlier measure becomes larger, so they will be considered as outliers.

MX-AI

▪ Isolation Forest (iForest): Outlier Detection - Overview

- Isolation Forest is not part of Random Forest. The purpose of this video is to compare the outlier detection result of Random Forest and Isolation Forest.
- iForest is an outlier detection algorithm in the form of unsupervised learning proposed by Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou in 2008.
- Like Random Forest, Isolation Forest uses multiple trees, performs row and column subsampling. However, row subsampling is performed with small sizes and without replacement. Column subsampling randomly selects only one column and randomly selects the split point.

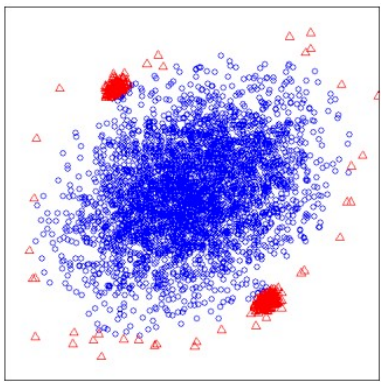
Isolation Forest

Fei Tony Liu, Kai Ming Ting
Gippsland School of Information Technology
Monash University, Victoria, Australia
{tony.liu},{kaiming.ting}@infotech.monash.edu.au

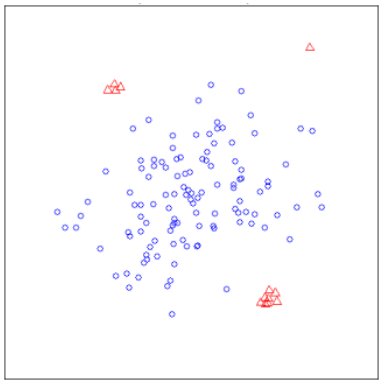
Zhi-Hua Zhou
National Key Laboratory
for Novel Software Technology
Nanjing University, Nanjing 210093, China
zhouzh@lamda.nju.edu.cn

Abstract

Most existing model-based approaches to anomaly detection construct a profile of normal instances, then identify instances that do not conform to the normal profile as anomalies. This paper proposes a fundamentally different model-based method that explicitly isolates anomalies instead of profiles normal points. To our best knowledge, the concept of isolation has not been explored in current literature. The use of isolation enables the proposed method, iForest, to exploit sub-sampling to an extent that is not feasible in existing methods, creating an algorithm which has a linear time complexity with a low constant and a low memory requirement. Our empirical evaluation shows that iForest performs favourably to ORCA, a near-linear time complexity distance-based method, LOF and Random Forests in terms of AUC and processing time, and especially in large data sets. iForest also works well in high dimensional problems which have a large number of irrelevant attributes, and in situations where training set does not contain any anomalies.



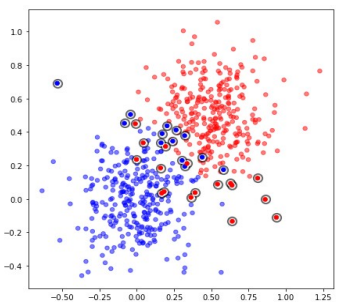
(a) Original sample
(4096 instances)



(b) Sub-sample
(128 instances)

Figure 4: Blue circles denote normal instances and red triangles denote anomalies.

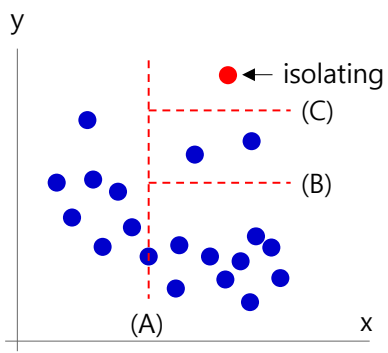
Outliers in Random Forest →

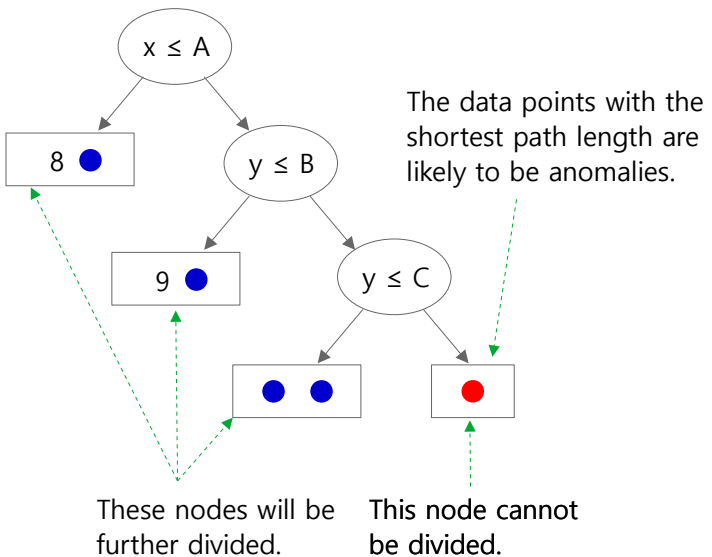


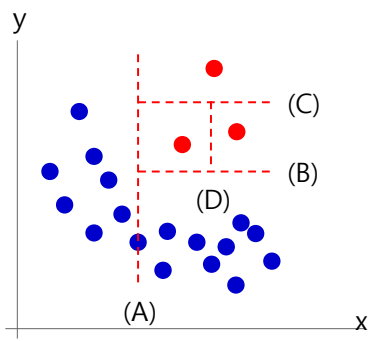
MX-AI

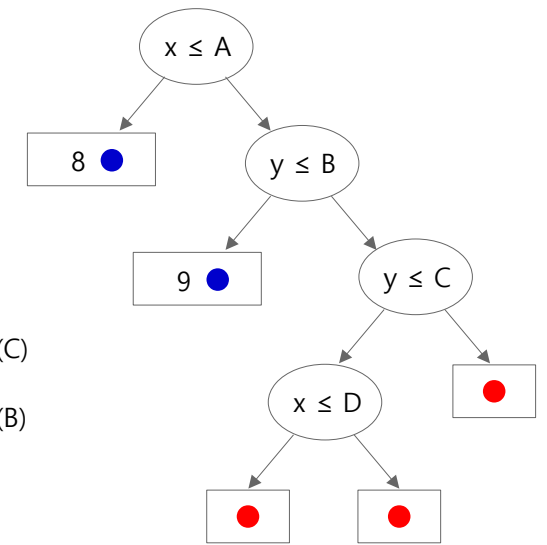
Isolation Forest (iForest): Outlier Detection – Basic idea

- Creating a binary tree from training data isolates outliers much faster than normal samples. Outliers have short decision depth, path length.
- Splitting the node with normal samples until one sample remains in a leaf node makes the tree deeper. That is because the normal data points are clustered in the distribution. Therefore, the path length for normal data points becomes large.
- In the picture on the left, the data point is left alone in the leaf node by the split point (C). Its path length is 3. Other data points require more splits and therefore have greater depth. In the picture on the right, dividing by (D) separates two additional data points, which are also likely outliers.
- In practice, we create multiple trees with sample data, calculate the anomaly score for each isolated leaf node, and calculate the average to determine the data points with larger scores as outliers.







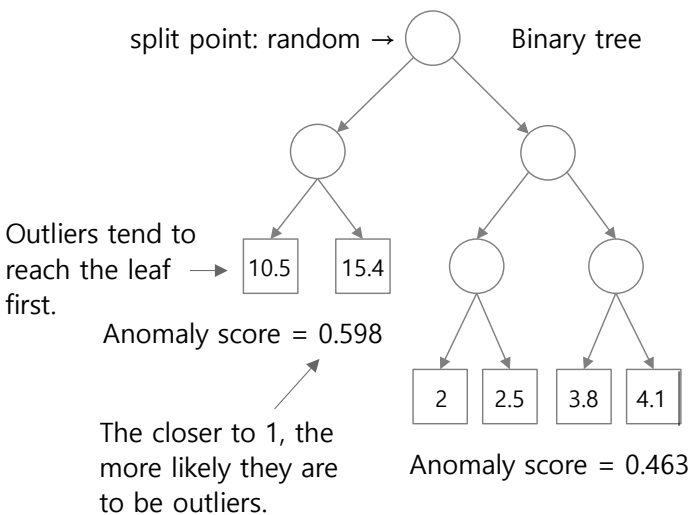


MX-AI

▪ Isolation Forest (iForest): Outlier Detection – Algorithm

- Isolation Forest uses anomaly scores to detect outliers. The trees in Isolation Forest, iTrees, have the same structure as binary search trees (BST).
- The anomaly score is calculated as the average path length taken from the BST.
- The closer the Anomaly score is to '1', the more likely it is to be an outlier, and the closer it is to '0', the more likely it is to be normal data. The closer it is to 0.5, the less clear the distinction between normal and abnormal data is. For the data below, 10.5 and 15.4 are likely outliers.

Data (x) = [2, 2.5, 3.8, 4.1, 10.5, 15.4]



◯ : internal node (n-1 = 5)

□ : external node (n = 6)

1) average path length of unsuccessful search in BST
- average depth of leaf nodes

$$c(n)=2H(n-1)-\frac{2(n-1)}{n}$$

$$H(i)=\log(i)\ +\ 0.5772156649 \text{ (Euler's constant)}$$

actual average depth: $\frac{2+2+3+3+3+3}{6}=2.67$

$$c(n)=2(\log(5)+0.577)-\frac{2\times5}{6}=2.70$$

similar

2) Anomaly score

$$s(x,n)=2^{-\frac{E[h(x)]}{c(n)}}$$

h(x) – path length of data point x
c(n) – average of h(x) given n.
E[h(x)] - average of h(x) from all trees.

if the number of trees is just one, the left tree.

$$s(2,n=6)=2^{-\frac{3}{2.7}}=0.463$$

$$s(10.5,n=6)=2^{-\frac{2}{2.7}}=0.598$$

$$s(15.4,n=6)=2^{-\frac{2}{2.7}}=0.598$$

$E[h(x)] \rightarrow c(n),\ \ s \rightarrow 0.5$

$E[h(x)] \rightarrow 0,\ \ s \rightarrow 1$

$E[h(x)] \rightarrow n-1,\ \ s \rightarrow 0$

MX-AI

▪ Coding practice: Implement Isolation Forest

```python
# [MXML-8-07] 7.iForest_test.py
# Implementation of Isolation Forest using ExtraTreeRegressor
# sklearn's IsolationForest library makes it easy to implement
# Isolation Forest, but I used ExtraTreeRegressor to better
# understand how it works.
from sklearn.tree import ExtraTreeRegressor
import numpy as np

# simple dataset
x = np.array([2, 2.5, 3.8, 4.1, 10.5, 15.4],
             dtype=np.float32).reshape(-1, 1)
n = x.shape[0]   # the number of data points
n_trees = 10     # the number of trees in Isolation Forest

# H(i) is the harmonic number and it can be estimated
# by ln(i) + 0.5772156649 (Euler's constant).
def H(n):
    return np.log(n) + 0.5772156649

# average path length of unsuccessful search in BST
def C(n):
    return 2 * H(n-1) - (2 * (n-1) / n)

hx = np.zeros(n)
for t in range(n_trees):
    # Create a tree using random split points
    model = ExtraTreeRegressor(max_depth=3, max_features=1)

    # Fit the model to training data.
    # Since it is unsupervised learning and there is no target
    # value, a binary tree is created by randomly generating
    # target values.
    model.fit(x, np.random.uniform(size=n))
    leaf_id = model.apply(x)  # indices of leaf nodes

    # depth of each node, internal and external nodes.
    node_depth = model.tree_.compute_node_depths()

    # h(x): accumulated path length of data points
    hx += node_depth[leaf_id] - 1.0
    print('Tree',t,':', (hx / (t+1)).round(1))

Ehx = hx / n_trees           # Average of h(x)
S = 2 ** (-Ehx / C(n)))      # Anomaly scores for each data point
i_out = np.argsort(S)[-2:]   # Top 2 anomaly scores
outliers = x[i_out]          # outliers

print('\nAnomaly scores:'); print(S.round(3))
print('\nOutliers:'); print(outliers)
```

```
Tree 0 : [3. 3. 3. 3. 2. 1.]
Tree 1 : [3. 3. 3. 3. 2. 1.]
Tree 2 : [3.  3.  3.  3.  2.  1.3]
Tree 3 : [2.8 3.  3.  3.  2.  1.5]
Tree 4 : [2.8 3.  3.  3.  2.  1.6]
Tree 5 : [2.7 3.  3.  3.  2.2 1.7]
Tree 6 : [2.7 3.  3.  3.  2.3 1.6]
Tree 7 : [2.8 3.  3.  3.  2.2 1.5]
Tree 8 : [2.7 2.9 3.  3.  2.3 1.7]
Tree 9 : [2.7 2.9 3.  3.  2.3 1.7]

Anomaly scores:
[0.501 0.476 0.464 0.464 0.555 0.647]

Outliers:
[[10.5]
 [15.4]]
```

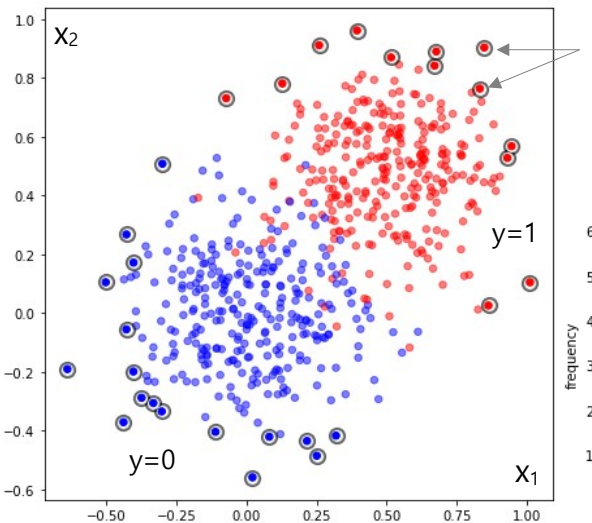
MX-AI

▪ Isolation Forest (iForest): Outlier Detection

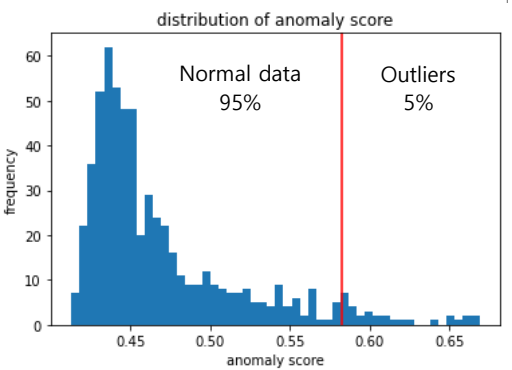
```
# [MXML-8-07] 8.iForest_outlier.py
# Outlier detection using Isolation Forest (iForest)
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.ensemble import IsolationForest
import matplotlib.pyplot as plt

# Create training dataset
x, y = make_blobs(n_samples=600, n_features=2,
                  centers=[[0., 0.], [0.5, 0.5]],
                  cluster_std=0.2, center_box=(-1., 1.))

model = IsolationForest(n_estimators=50, contamination=0.05)
model.fit(x)
outlier = model.predict(x)   # Normal = 1, Outlier = -1

# Extract outliers
i_outlier = np.where(outlier == -1)[0]
x_outlier = x[i_outlier, :]

# Visualize normal data points and outliers by color.
plt.figure(figsize=(7, 7))
color = [['blue', 'red'][i] for i in y]
color_out = [['blue', 'red'][i] for i in y[i_outlier]]
plt.scatter(x[:, 0], x[:, 1], s=30, c=color, alpha=0.5)
plt.scatter(x_outlier[:, 0], x_outlier[:, 1], s=150, c='black',
            alpha=0.5)  # outlier scatter
plt.scatter(x_outlier[:, 0], x_outlier[:, 1], s=60, c='white')
plt.scatter(x_outlier[:, 0], x_outlier[:, 1], s=30, c=color_out)
plt.show()
```

Anomaly score top 5%

```
# Check out the distribution of Anomaly score
score = abs(model.score_samples(x))
score[i_outlier].min()
plt.hist(score, bins = 50)
plt.title('distribution of anomaly score')
plt.xlabel('anomaly score')
plt.ylabel('frequency')
plt.axvline(x=score[i_outlier].min(), c='red')
plt.show()
```



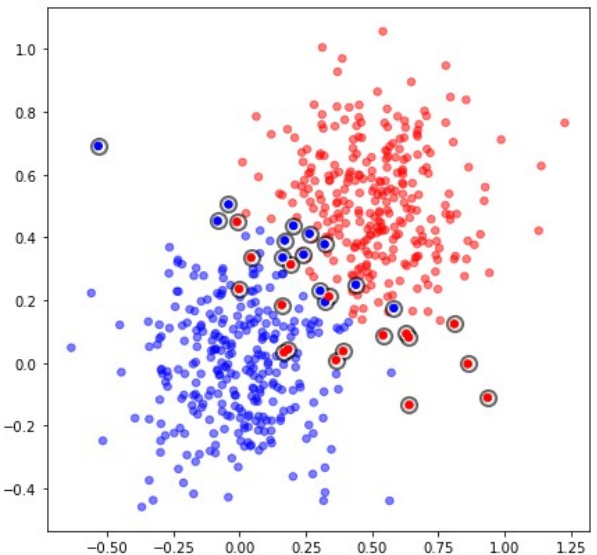
Outlier detection algorithms in the form of unsupervised learning consider these data to be outliers. (far from the overall distribution)



MX-AI

- Comparison of outlier detection results between Random Forest and Isolation Forest

Random Forest [MXML-8-06]
Supervised learning



Isolation Forest
Unsupervised learning