Ensemble Learning

Dr. Tran Anh Tuan
Department of Maths & Computer Sciences
University of Sciences, HCMUS, VietNam

Outlier

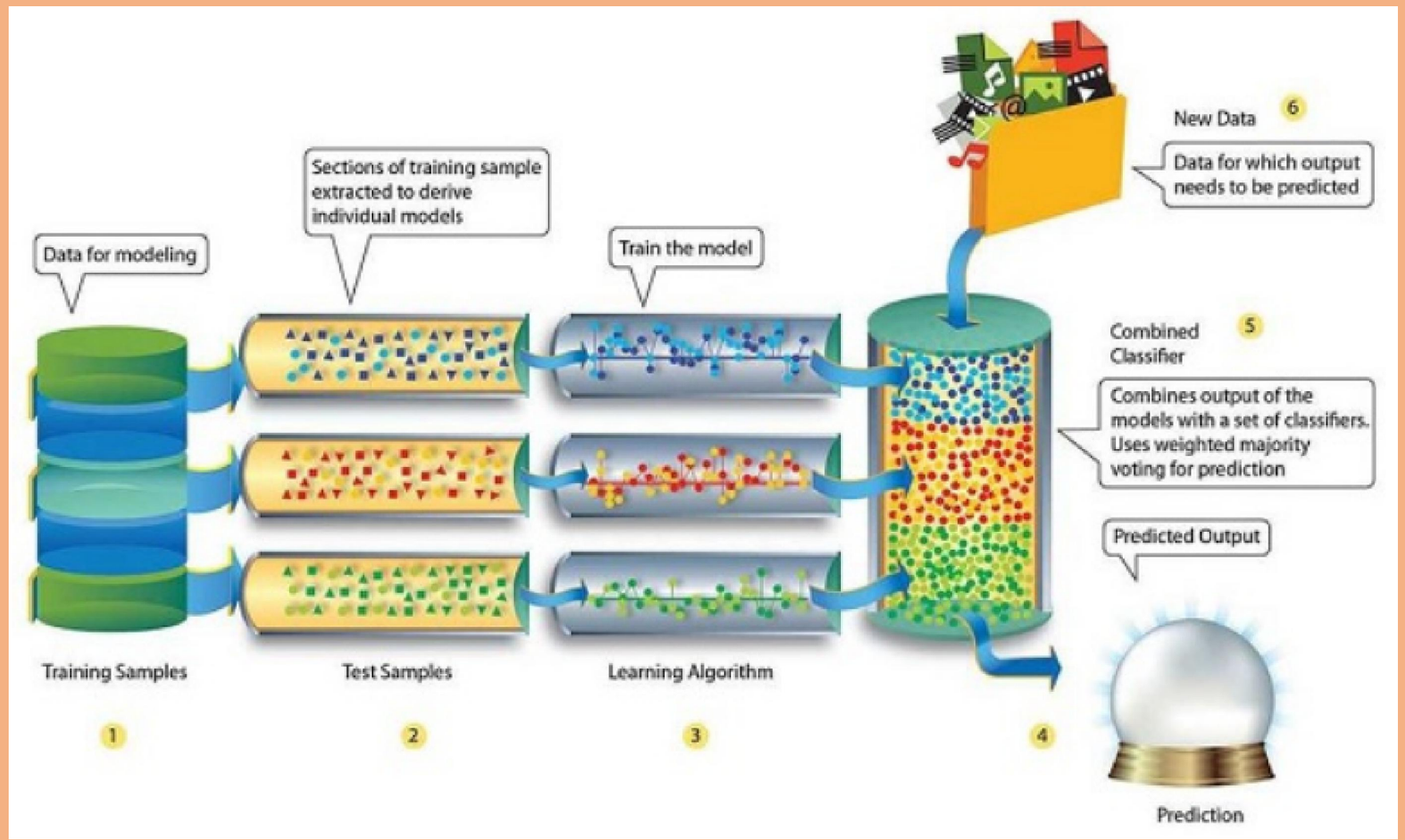
Ensemble Vote Classifier

- Hard Voting
- Soft Voting
- Weighted Majority Vote

Ensemble combination method

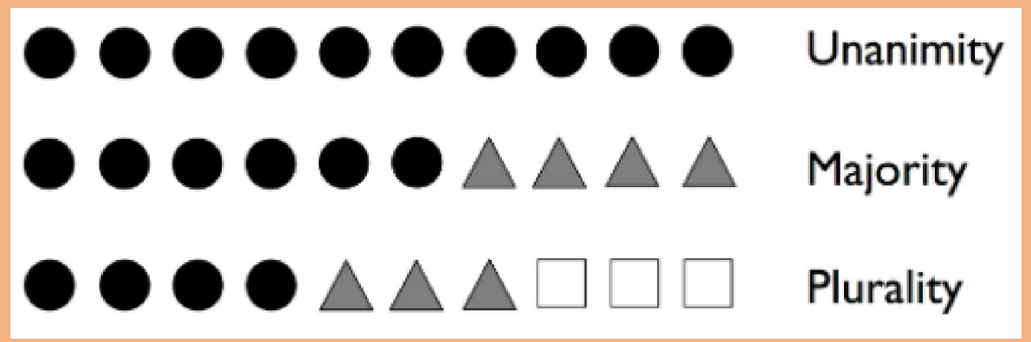
- Bagging
- Boosting
- Stacked generalization

Bagging vs Boosting



Ensemble Vote Classifier

The Ensemble Vote Classifier is a meta-classifier for combining similar or conceptually different machine learning classifiers for classification via majority or plurality voting. (For simplicity, we will refer to both majority and plurality voting as majority voting.

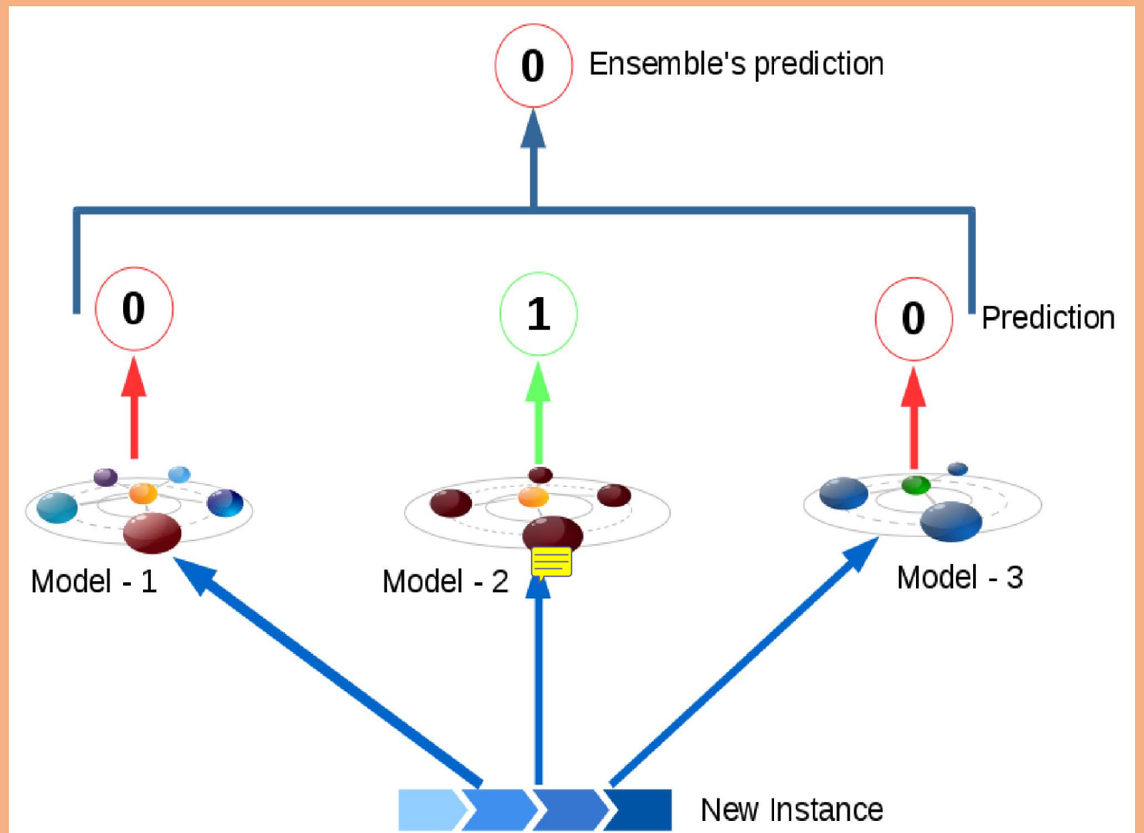


The Ensemble Vote Classifier implements "hard" and "soft" voting.

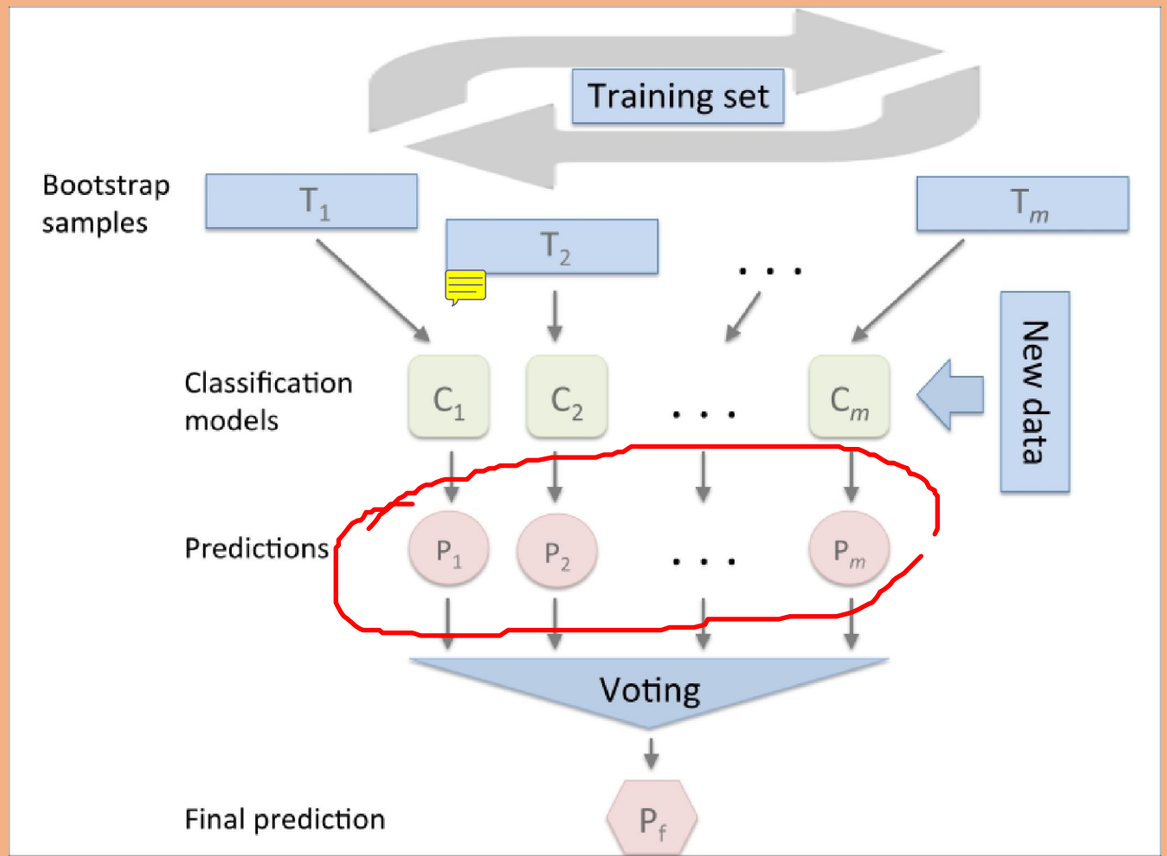
- In hard voting, we predict the final class label as the class label that has been predicted most frequently by the classification models.
- In soft voting, we predict the class labels by averaging the class-probabilities (only recommended if the classifiers are well-calibrated)

Ensemble Vote Classifier

Hard Voting



Soft Voting



Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

Ensemble Vote Classifier

Majority Voting / Hard Voting

Hard voting is the simplest case of majority voting. Here, we predict the class label y^ via majority (plurality) voting of each classifier Cj:

$$\hat{y} = mode\{C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_m(\mathbf{x})\}$$

Assuming that we combine three classifiers that classify a training sample as follows:

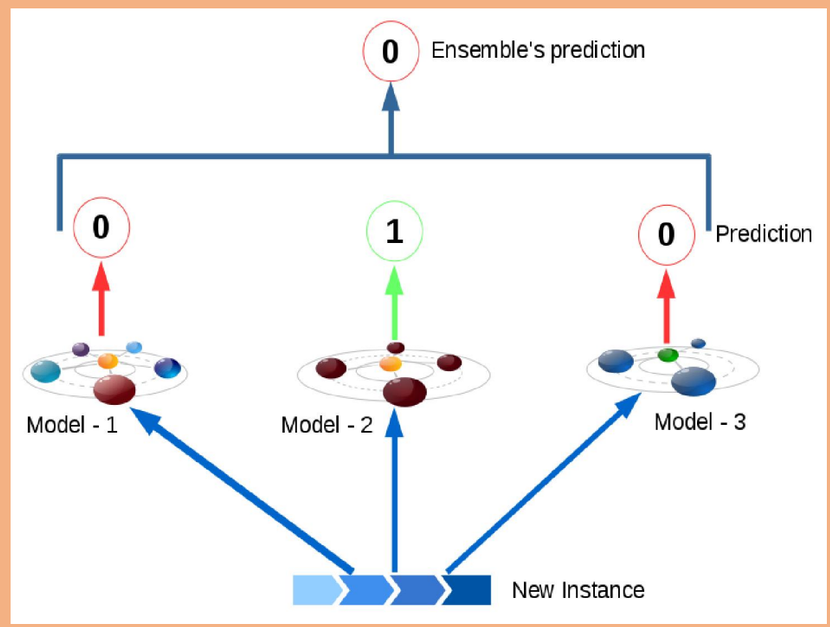
classifier 1 -> class 0

classifier 2 -> class 0

classifier 3 -> class 1

y^=mode{0,0,1}=0

Via majority vote, we would we would classify the sample as "class 0.“



Ensemble Vote Classifier

Weighted Majority Vote

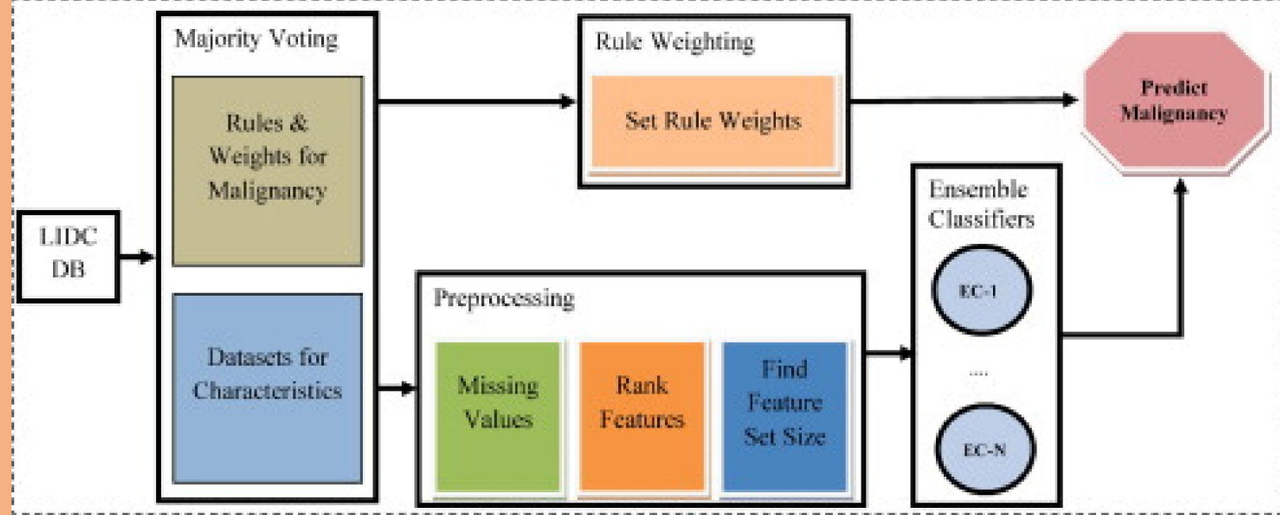
In addition to the simple majority vote (hard voting) as described in the previous section, we can compute a weighted majority vote by associating a weight wj with classifier Cj:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A\big(C_j(\mathbf{x}) = i\big),$$

where χA is the characteristic function [Cj(x)=i∈A], and A is the set of unique class labels.

Continuing with the example from the previous section
classifier 1 -> class 0
classifier 2 -> class 0
classifier 3 -> class 1
assigning the weights {0.2, 0.2, 0.6} would yield a prediction y^=1:

$$\arg\max_i [0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1] = 1$$



Ensemble Vote Classifier

Soft Voting

In soft voting, we predict the class labels based on the predicted probabilities p for classifier -- this approach is only recommended if the classifiers are well-calibrated.

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij},$$

where wj is the weight that can be assigned to the jth classifier.

- Assuming the example in the previous section was a binary classification task with class labels i∈{0,1}, our ensemble could make the following prediction
- Using uniform weights, we compute the average probabilities:

- $C_1(\mathbf{x}) \rightarrow [0.9, 0.1]$
- $C_2(\mathbf{x}) \rightarrow [0.8, 0.2]$
- $C_3(\mathbf{x}) \rightarrow [0.4, 0.6]$

$$p(i_0 \mid \mathbf{x}) = \frac{0.9 + 0.8 + 0.4}{3} = 0.7$$

$$p(i_1 \mid \mathbf{x}) = \frac{0.1 + 0.2 + 0.6}{3} = 0.3$$

$$\hat{y} = \arg\max_i \left[p(i_0 \mid \mathbf{x}), p(i_1 \mid \mathbf{x})\right] = 0$$

$$p(i_0 \mid \mathbf{x}) = 0.1 \times 0.9 + 0.1 \times 0.8 + 0.8 \times 0.4 = 0.49$$

$$p(i_1 \mid \mathbf{x}) = 0.1 \times 0.1 + 0.2 \times 0.1 + 0.8 \times 0.6 = 0.51$$

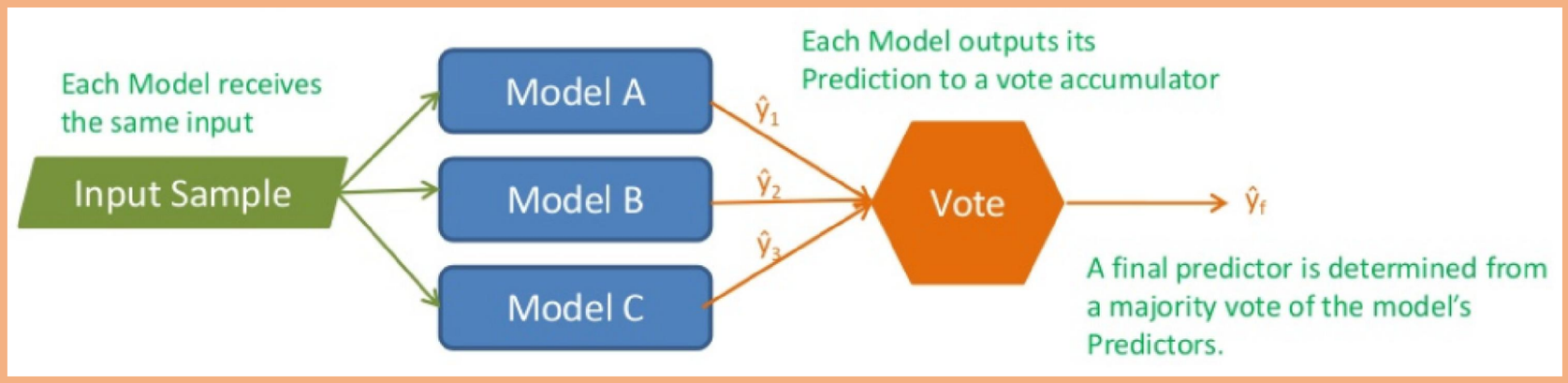
$$\hat{y} = \arg\max_i \left[p(i_0 \mid \mathbf{x}), p(i_1 \mid \mathbf{x})\right] = 1$$

However, assigning the weights {0.1, 0.1, 0.8} would yield a prediction y^=1:

Ensemble Vote Classifier

Ensemble combination method

An ensemble of classifiers can be trained simply on different subsets of the training data, different parameters of the classifiers, or even with different subsets of features as in random subspace models. The classifiers can then be combined using one of the combination rules. Some of these combination rules operate on class labels only, whereas others need continuous outputs that can be interpreted as support given by the classifier to each of the classes.



(a)*Abstract-level* output, where each classifier outputs a unique class label for each input pattern.
(b)*Rank-level* output, where each classifier outputs a list of ranked class labels for each input pattern.
(c)*Measurement-level* output, where each classifier outputs a vector of continuous-valued measures that can represent estimates of class posterior probabilities or class-related confidence values that represent the support for the possible classification hypotheses.

Ensemble Vote Classifier

Summary of combination rule

Approach	Method	Formula	Effectiveness
Class label combination	Majority voting	$\sum_{t=1}^{T} d_{t,j}(x) = \max_{j=1}^{C} \sum_{t=1}^{T} d_{t,j}$	Gives average performance when majority does not give accurate prediction
	Weighted majority voting	$\sum_{t=1}^{T} w_t d_{t,J}(x) = \max_{j=1}^{C} \sum_{t=1}^{T} w_t d_{t,j}$	Performs well only if the weights of the classifiers are assigned precisely
Continuous output combination	Sum rule	$\mu_j(x) = \sum_{t=1}^{T} d_{t,j}(x)$	Gives average performance when majority does not give accurate prediction
	Weighted sum rule	$\mu_j(x) = \sum_{t=1}^{T} w_t d_{t,j}(x)$	Performs well only if the weights of the classifiers are assigned precisely

Ensemble Vote Classifier

Summary of combination rule

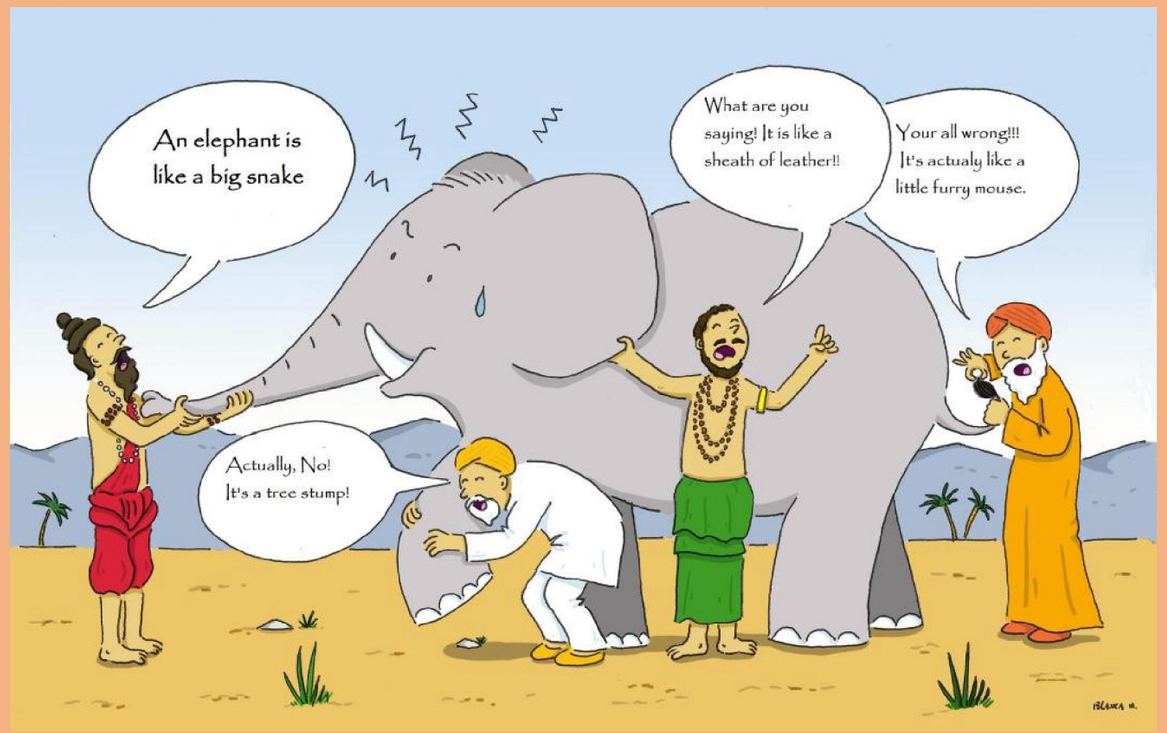
Approach	Method	Formula	Effectiveness
Continuous output combination	Mean rule	$\mu_j(x) = \frac{1}{T}\sum_{t=1}^{T} d_{t,j}(x)$	Performance is significantly affected by outliers
	Product rule	$\mu_j(x) = \prod_{t=1}^{T} d_{t,j}(x)$	Sensitive to low probability value
	Maximum rule	$\mu_j(x) = \max_{t=1}^{T}\{d_{t,j}(x)\}$	Chooses the most optimistic value
	Minimum rule	$\mu_j(x) = \min_{t=1}^{T}\{d_{t,j}(x)\}$	Performance is significantly affected by outliers
	Median rule	$\mu_j(x) = \text{median}_{t=1}^{T}\{d_{t,j}(x)\}$	Performance is significantly affected by outliers
	Generalized rule	$\mu_{j,\propto}(x) = \left[\frac{1}{T}\sum_{t=1}^{T} d_{t,j}(x)^{\propto}\right]^{\frac{1}{\propto}}$	Affected by outliers

Ensemble Vote Classifier

ENSEMBLE LEARNING

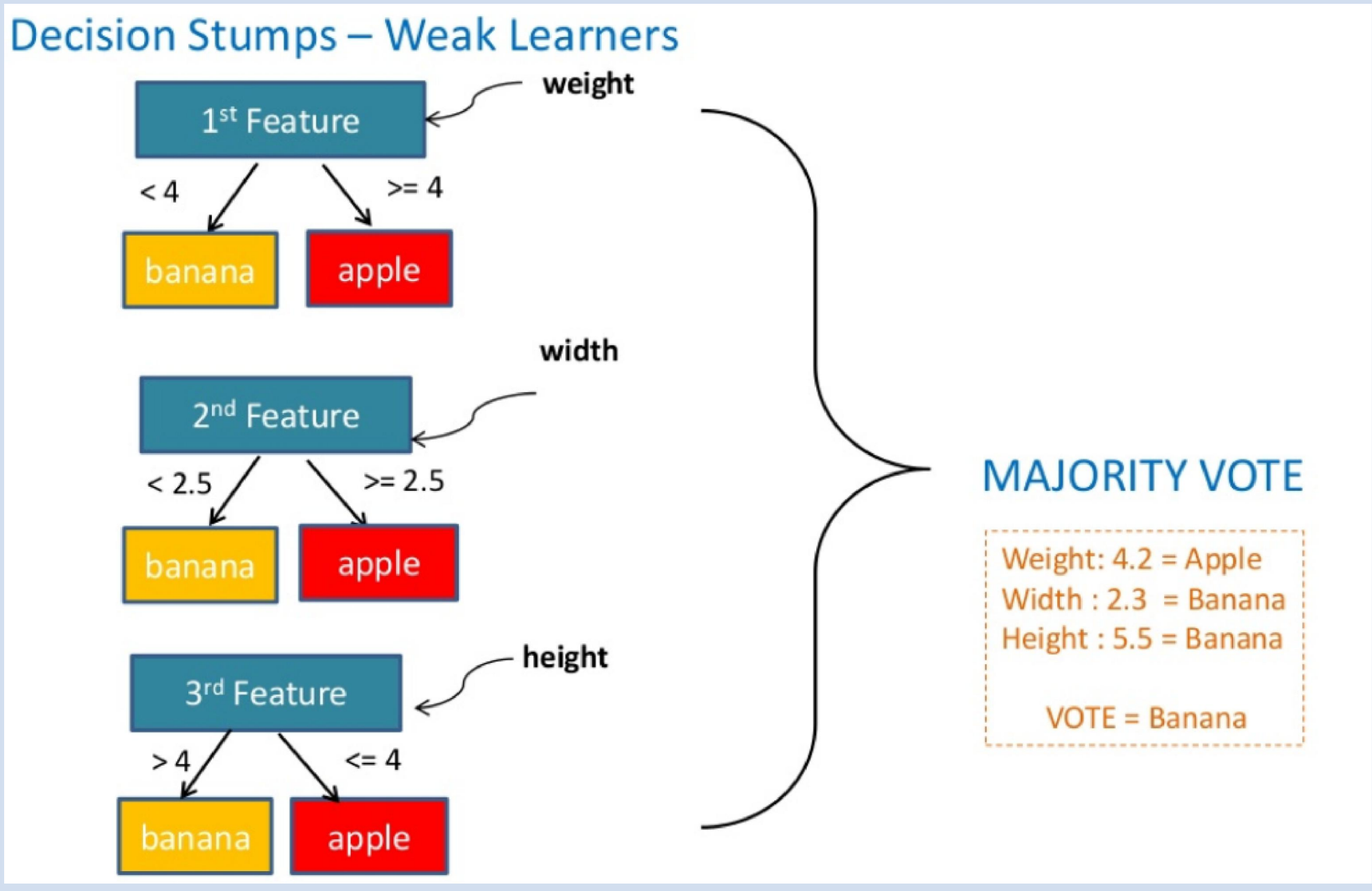
Ensemble methods combine several classifiers to produce better predictive performance than a single classifier. The main principle behind the ensemble model is that a group of weak learners come together to form a strong learner, thus increasing the accuracy of the model. When we try to predict the target variable using any machine learning technique, the main causes of difference in actual and predicted values are noise, variance, and bias. Ensemble helps to reduce these factors (except noise, which is irreducible error).

Another way to think about Ensemble learning is Fable of blind men and elephant. All of the blind men had their own description of the elephant. Even though each of the description was true, it would have been better to come together and discuss their understanding before coming to final conclusion. This story perfectly describes the Ensemble learning method.



Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

Ensemble Vote Classifier

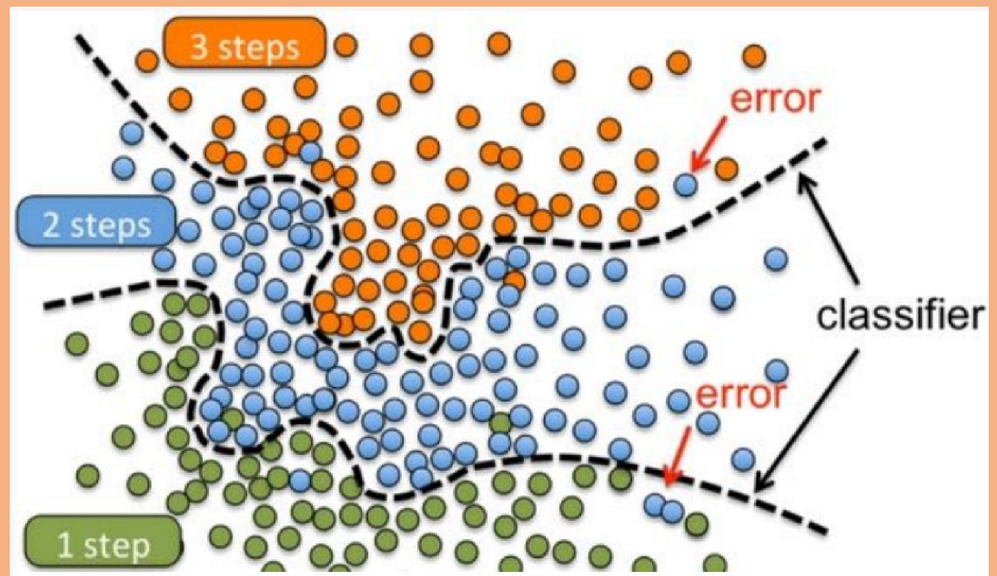


Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

Ensemble Vote Classifier

Motivation for a new ensemble method

- Algorithms from different classification families can be used with appropriate combination method to form an effective ensemble.
- To reduce error rates, it is a necessary condition to combine the relatively uncorrelated output predictions. With highly correlated output predictions, there is little scope for the reduction in error, as the *committee of experts* has no diversity to draw from.
- A strong reason for combining models across different algorithm families can be stated as—different algorithms will provide uncorrelated output estimates because of their varied classification functions.
- Abbott (1994) showed considerable differences in classifier performance class by class—information that is clear, once classifier is obscure to another. Since it is difficult to know a priori which algorithm(s) will produce the lowest error for each domain (on unseen data), combining models across algorithm families mitigates that risk by including contributions from all the families.



Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

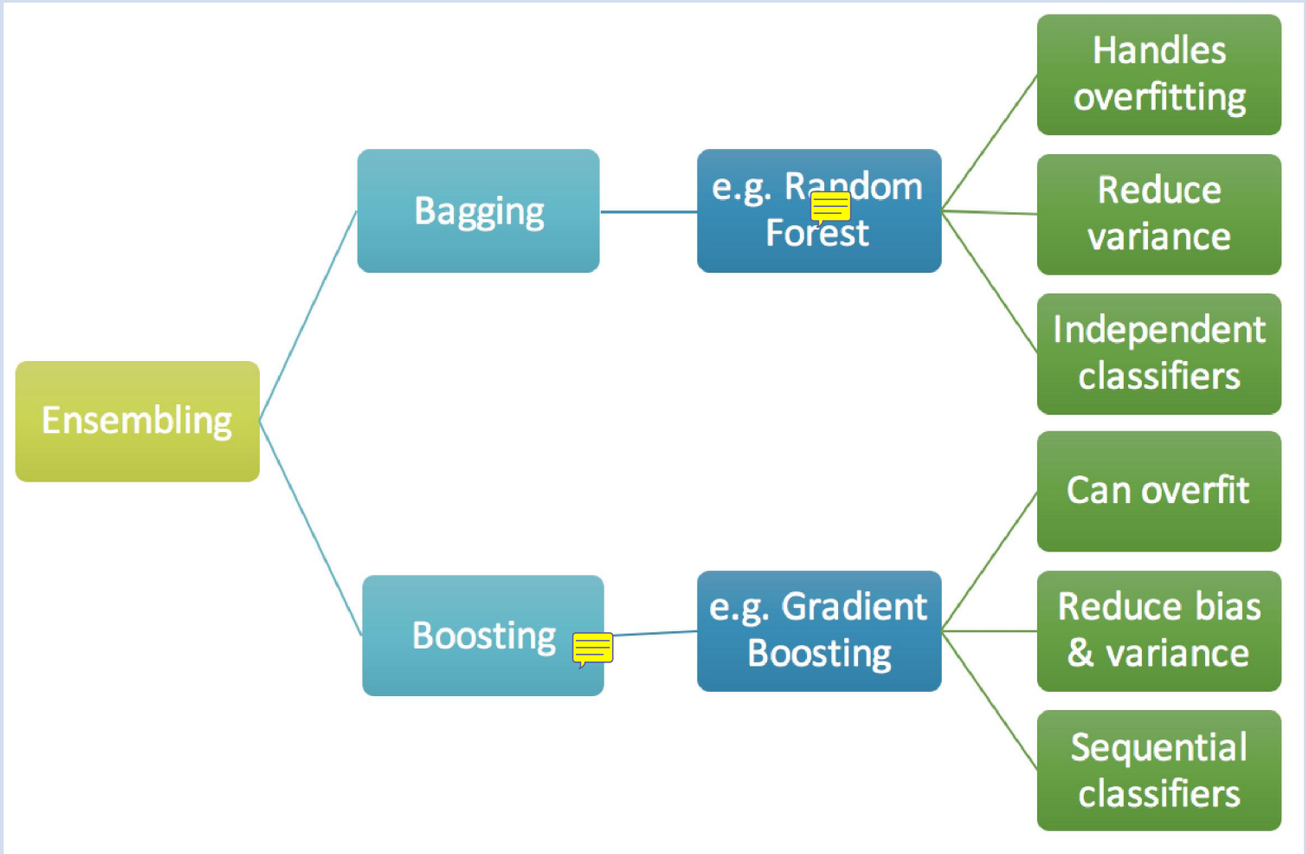
Ensemble Vote Classifier

Summary of the existing ensemble approaches

Method	Training approach	Classifiers	Decision fusion	Classifier priority	Input parameter	Pros	Cons
Bagging	Re-sampling	Unstable learner trained over re-sampled sets outputs different models	Majority voting	No	Training data, no. of classes, no. of dimensions, no. of iterations	Simple and easy to understand and implement	Accuracy value lower than other ensemble approaches
Boosting	Re-sampling	Weak learner re-weighted in every iteration	Weighted majority voting	No	Training data, no. of classes, no. of dimensions, no. of iterations	Performance of the weak learner boosted manifold	Degrades with noise
Stack Generalization	Re-sampling and *k*-folding	Diverse base classifiers	Meta-classifier	No	Training data, no. of classes, no. of dimensions, no. of iterations	Good performance	Storage and time complexity

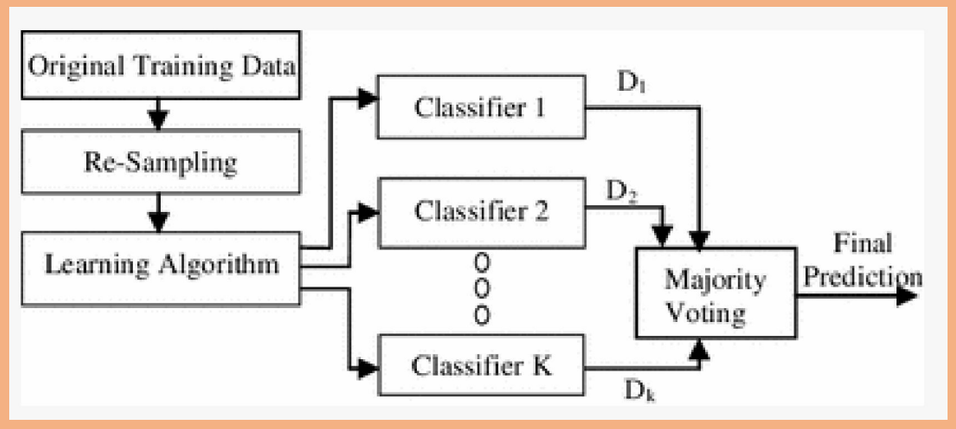
Ensemble Vo

	Bagging	Boosting	Stacking
Partitioning of the data into subsets	Random	Giving mis-classified samples higher preference	Various
Goal to achieve	Minimize variance	Increase predictive force	Both
Methods where this is used	Random subspace	Gradient descent	Blending
Function to combine single models	(Weighted) average	Weighted majority vote	Logistic regression

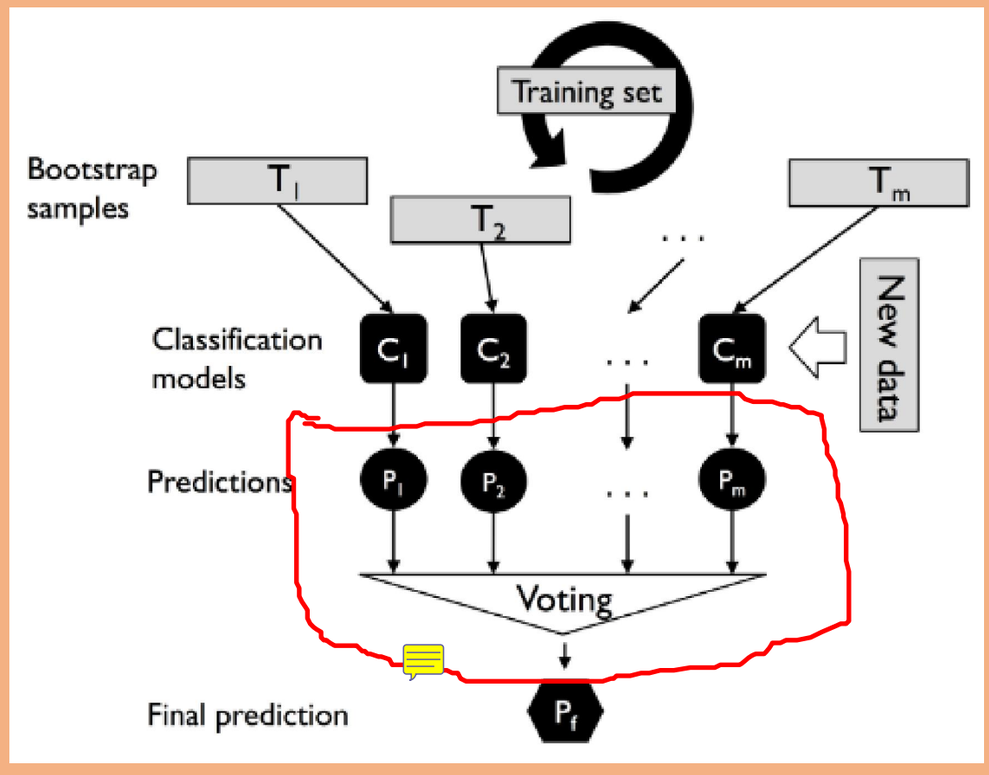


Bagging -- Building an ensemble of classifiers from bootstrap samples

Bagging (Breiman 1996a) is an ensemble method where the same learning algorithm is used for different training datasets to obtain different classifiers. The diversity amongst the training datasets is achieved by a bootstrap technique used to re-sample the training dataset. As shown in Fig., each classifier is then trained on a re-sample of instances, which then assigns a predicted class to this set of instances. The individual classifiers' predictions (having equal weightage) are then combined by taking majority voting.



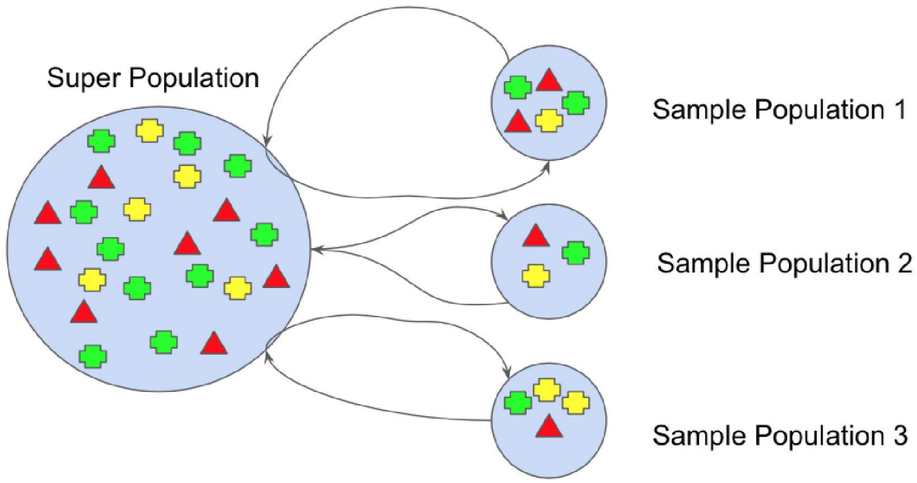
Block diagram of multiple classifier systems based on Bagging. D_i class prediction by the ith classifier



Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

Bagging -- Building an ensemble of classifiers from bootstrap samples

Bootstrap refers to random sampling with replacement. Bootstrap allows us to better understand the bias and the variance with the dataset. Bootstrap involves random sampling of small subset of data from the dataset. This subset can be replace. The selection of all the example in the dataset has equal probability. This method can help to better understand the mean and standand deviation from the dataset.



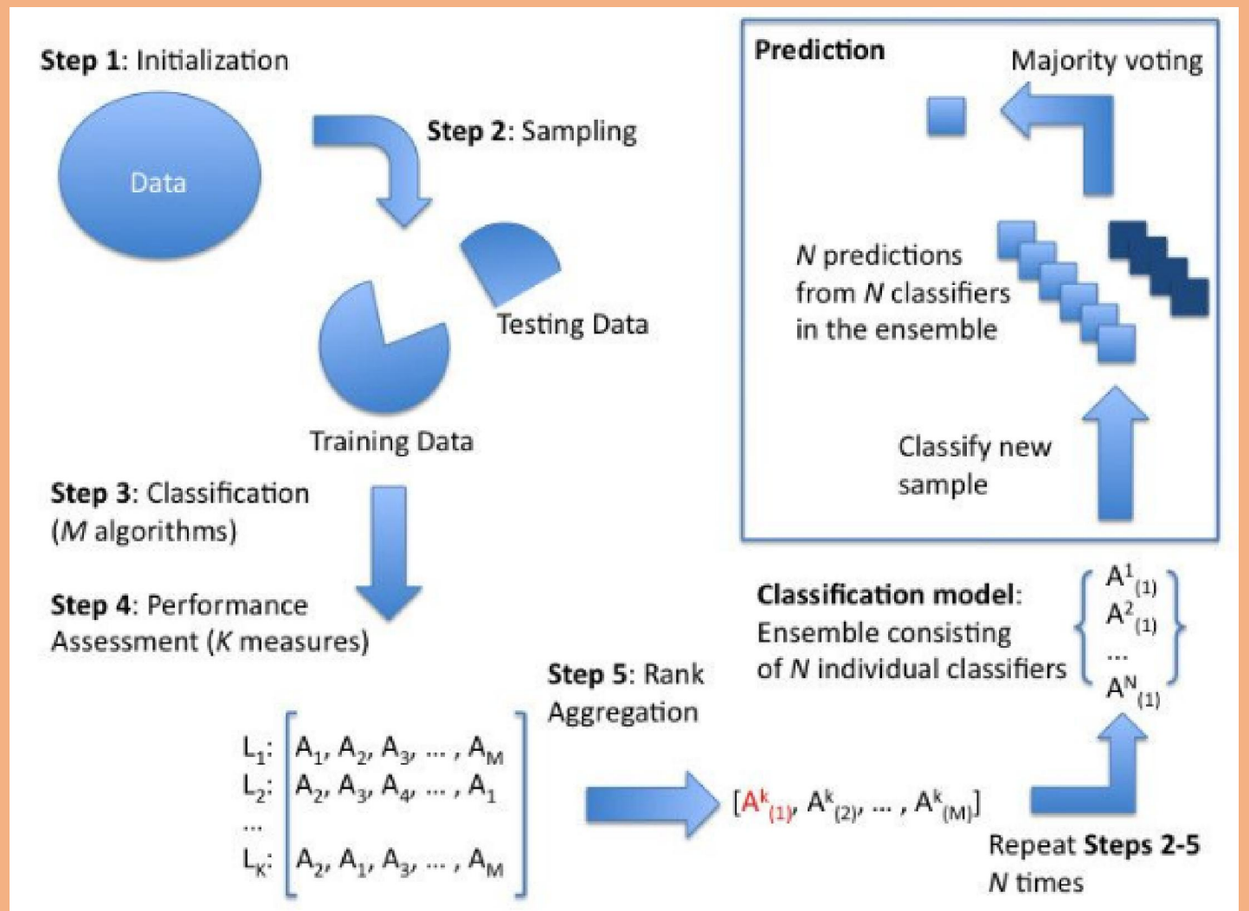
We know that our sample is small and that our mean has error in it. We can improve the estimate of our mean using the bootstrap procedure:

1. Create many (e.g. m) random sub-samples of our dataset with replacement (meaning we can select the same value multiple times).
2. Calculate the mean of each sub-sample.
3. Calculate the average of all of our collected means and use that as our estimated mean for the data.

Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

Bagging -- Building an ensemble of classifier

Ensemble combination method



1. **Initialization**. Set N, the number of bootstrap samples to draw. Let $j = 1$. Select the M classification algorithms along with K performance measures to be optimized.

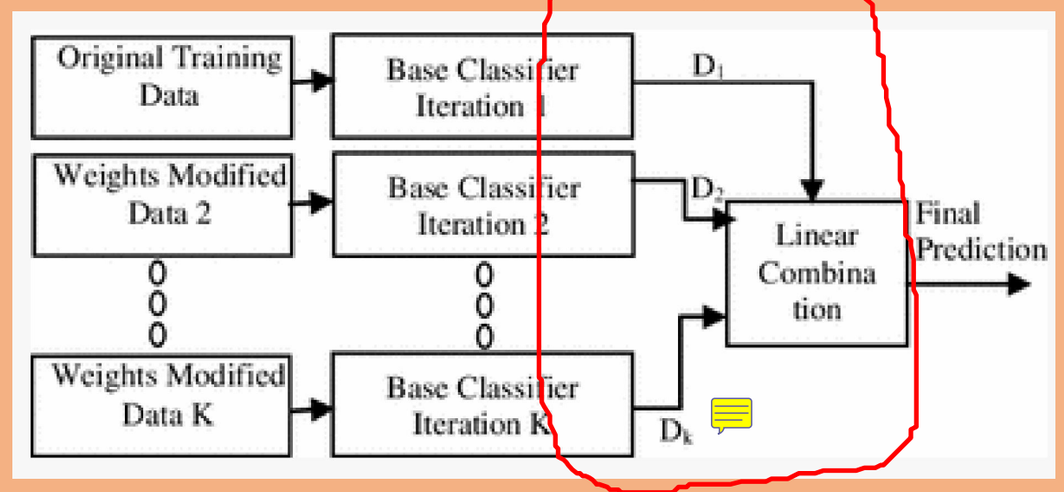
2. **Sampling**. Draw the j^{th} bootstrap sample of size n from training samples using simple random sampling with replacement to obtain $\left\{X_j^*, y_j^*\right\}$. Sampling is repeated until samples from all classes are present in a training set. Please note that some samples will be repeated more than once, while others will be left out of the bootstrap sample. Samples which are left out of the bootstrap samples are called out-of-bag (OOB) samples.
3. **Classification**. Using the j^{th} bootstrap sample train the M classifiers.

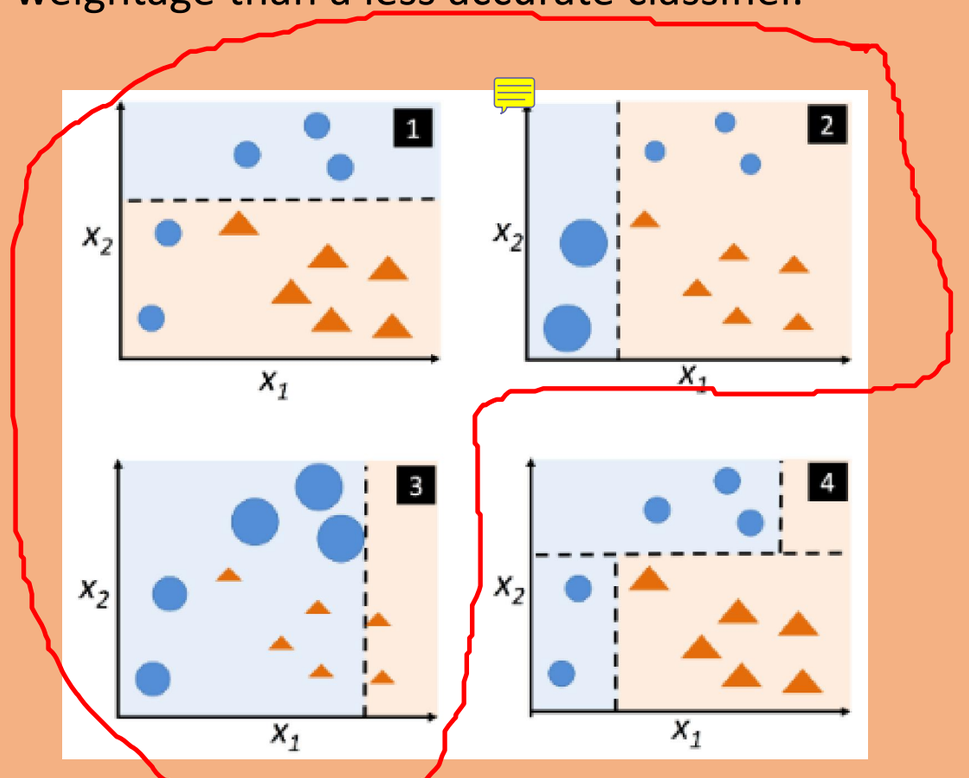
4. **Performance assessment**. The M models fitted in the Classification step are then used to predict class labels on the OOB cases which were not included into the j^{th} bootstrap sample, $\left\{X_j^{oob^*}, y_j^{oob^*}\right\}$. Since the true class labels are known, we can compute the K performance measures. Each performance measure will rank classification algorithms according to their performance under that measure, producing K ordered lists of size $M, L_1, ..., L_K$.

5. **Rank aggregation**. The ordered lists $L_1, ..., L_K$ are aggregated using the weighted rank aggregation procedure which determines the best performing classification algorithm $A_{(1)}^j$. Steps Sampling through Rank aggregation are repeated N times.

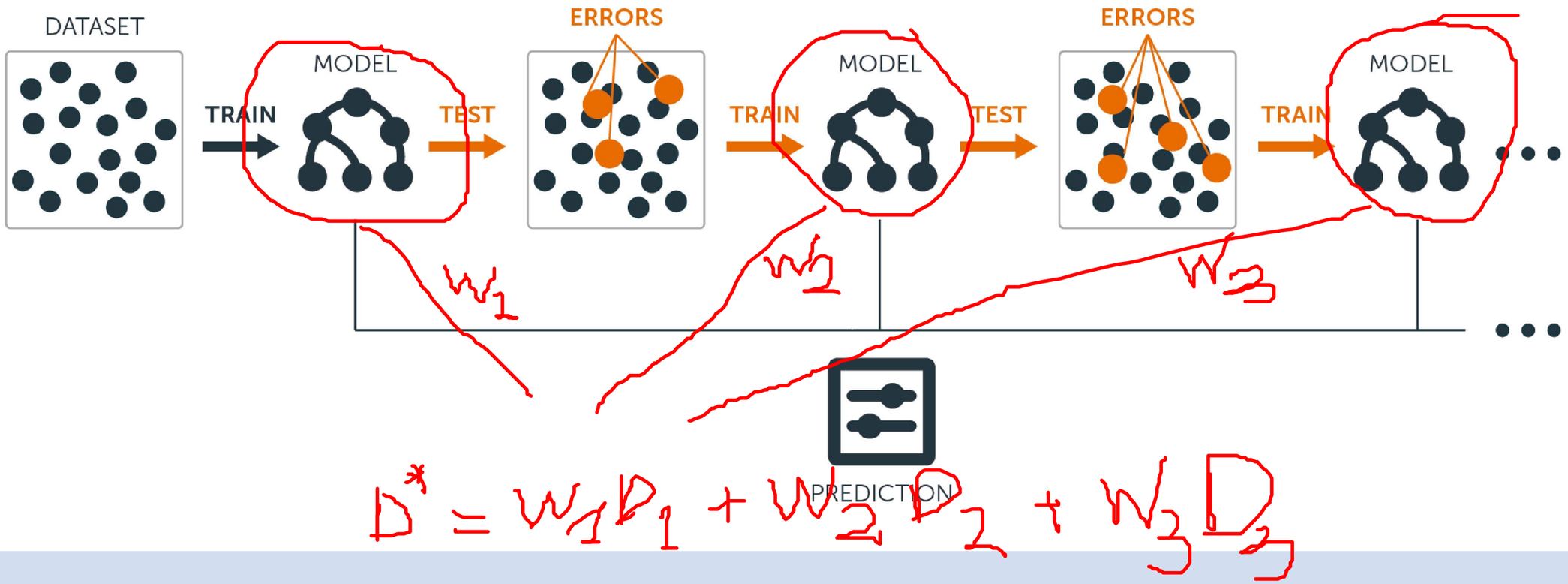
Boosting

Boosting (Freund and Schapire 1996) uses a re-sampling technique different from Bagging. In this case, a new training dataset is generated according to its sample distribution. The first classifier is constructed from the original dataset where every sample has an equal weight (Fig.). In the succeeding training dataset, the weight is reduced if the sample has been correctly classified, otherwise it is increased if the samples are misclassified. In the committee decision, a weighted voting method is used so that a more accurate classifier is given greater weightage than a less accurate classifier.





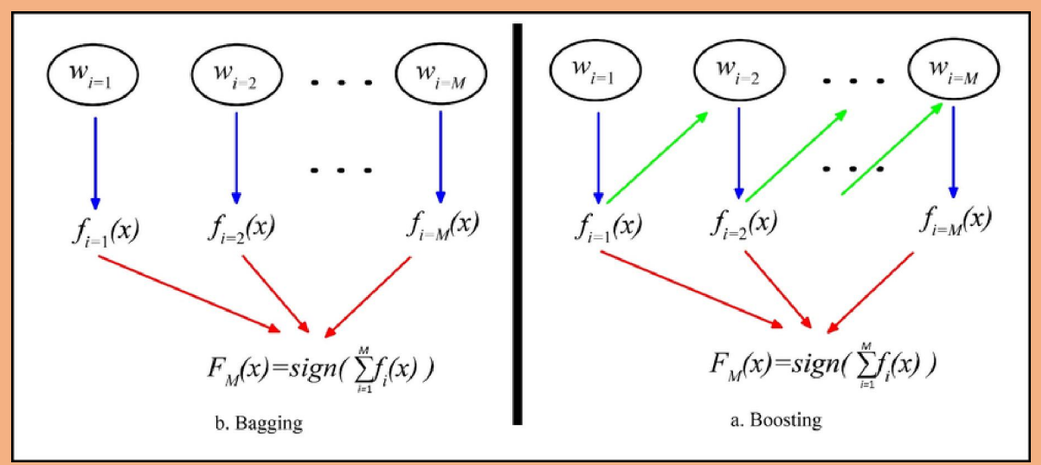
Boosting



Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

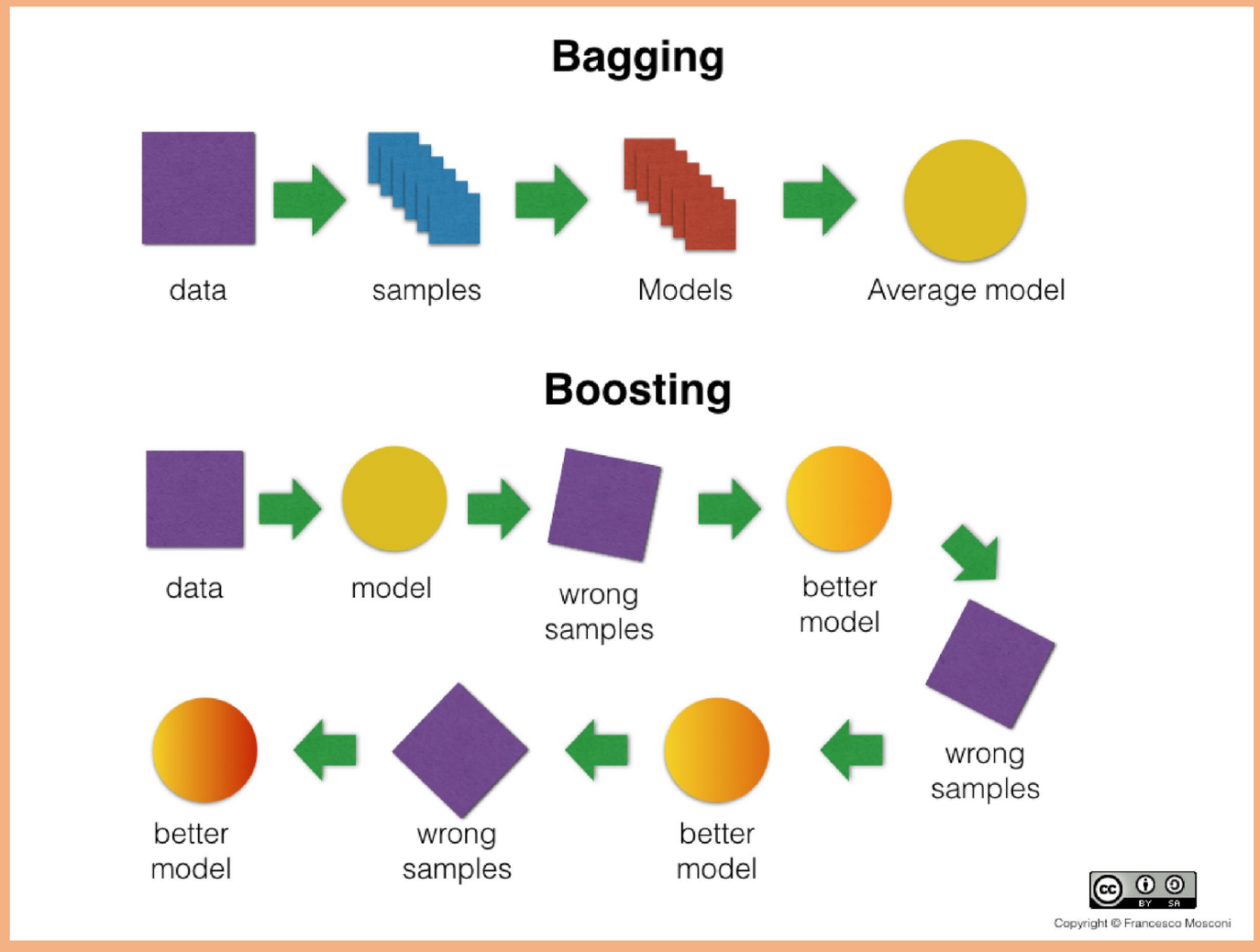
Boosting

Which is the best, Bagging or Boosting?
There's not an outright winner; it depends on the data, the simulation and the circumstances.
Bagging and Boosting decrease the variance of your single estimate as they combine several estimates from different models. So the result may be a model with **higher stability**.
If the problem is that the single model gets a very low performance, Bagging will rarely get a **better bias**. However, Boosting could generate a combined model with lower errors as it optimises the advantages and reduces pitfalls of the single model.
By contrast, if the difficulty of the single model is **over-fitting**, then Bagging is the best option. Boosting for its part doesn't help to avoid over-fitting; in fact, this technique is faced with this problem itself. For this reason, Bagging is effective more often than Boosting.



Boosting

Which is the best, Bagging or Boosting?



Stacked generalization

Stacked generalization (or stacking), proposed by Wolpert (1992), performs its task in two phases (Fig. 6): (i) the *layer*-1 base classifiers are trained using bootstrapped samples of the *level*-0 training dataset and (ii) the outputs of *layer*-1 are then used to train a *layer*-2 *meta-classifier*. The purpose is to check whether the training data have been properly learned. For example, if a particular classifier incorrectly learns a certain region of the feature space and hence consistently misclassifies instances coming from that region, then the *level*-2 classifier may be able to learn this behavior, and along with the learned behaviors of other classifiers, and correct such improper training. Polikar (2008) proposed to use class probabilities rather than class labels as the output in the *level*-1 dataset, so as to improve the stacking performance.

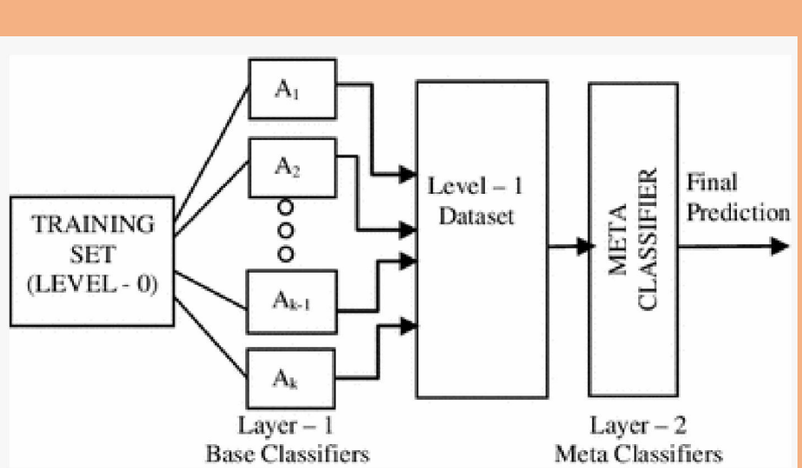
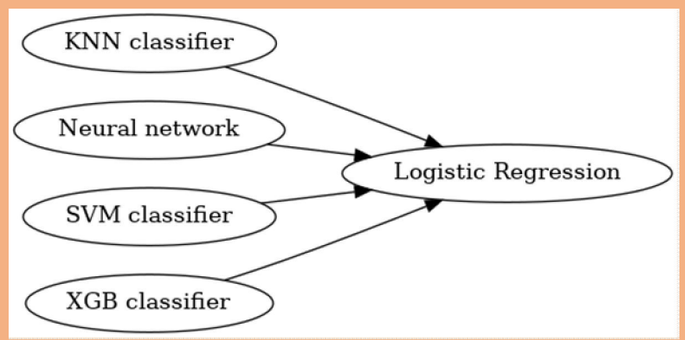


Fig. 6
Multiple classifiers based on stack generalization



Goals

1) Combine statistical learners, and

2) Improve generalization accuracy

Learning error

$$\sum_{i=1}^{n}(y(\boldsymbol{s}_i) - \hat{y}(\boldsymbol{s}_i))^2$$

Generalization error

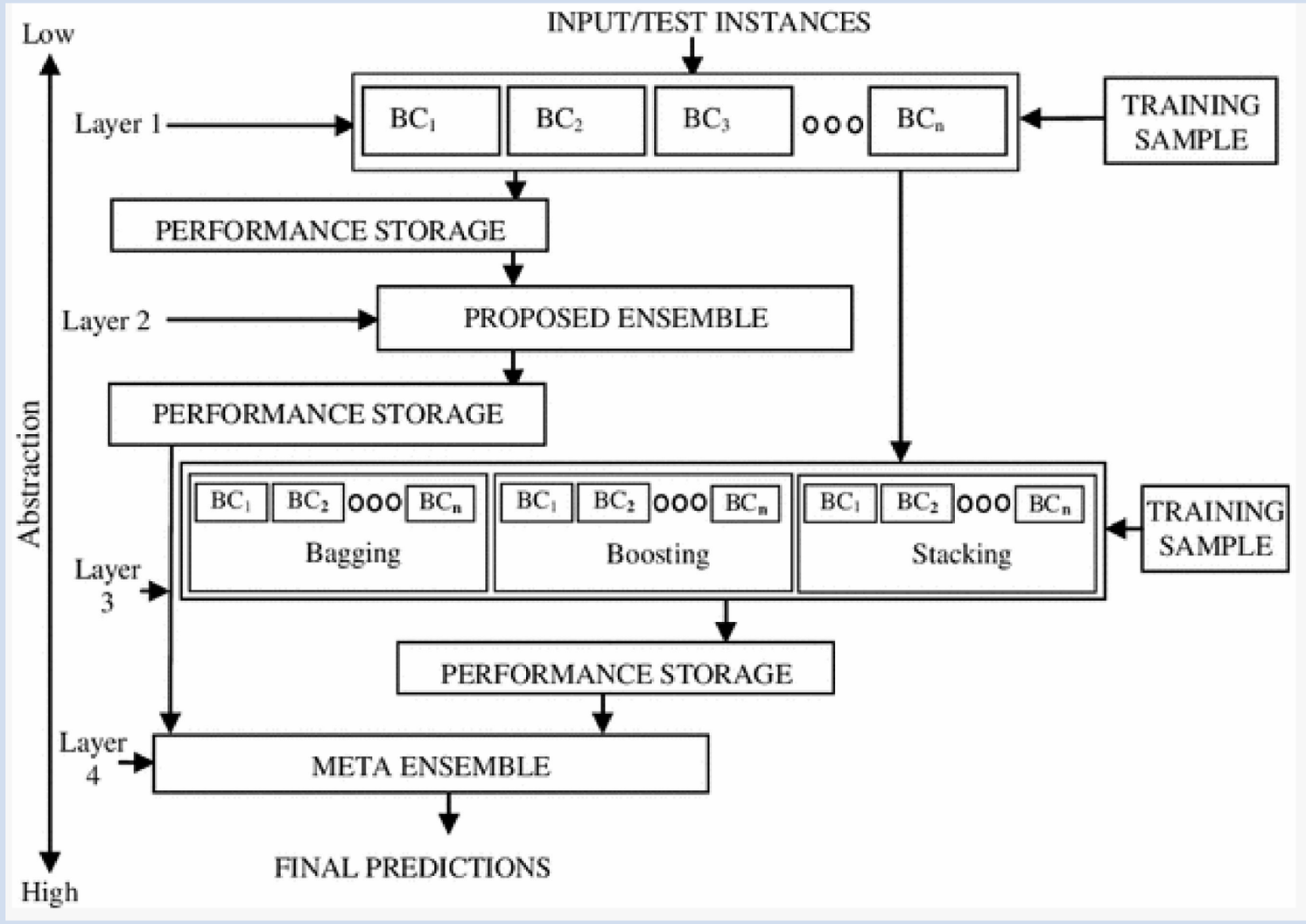
$$\sum_{0}(y(\boldsymbol{s}_0) - \hat{y}(\boldsymbol{s}_0))^2$$

$$\mathrm{RSS}(\boldsymbol{\beta}) = \sum_{i=1}^{n}(y(\boldsymbol{s}_i) - \sum_{j=1}^{m}\boldsymbol{\beta}_j \boldsymbol{f}_j(\boldsymbol{x}(\boldsymbol{s}_i)))^2$$

?

Architecture of the meta-ensemble

From the results of the previous section, it is clear that combining the outputs of different classifiers improves classification accuracy than the best single classifier in the combination, but it does not perform as well as boosting. The advantage of boosting acts directly to reduce the error cases, whereas combining works indirectly. As our proposed model works well to get the best output from the combination, we used this method to combine the results of our ensemble with the results of boosting, stacking and bagging and form a meta-ensemble; the architecture is shown in Fig



Practice

Practice Section:

1/ Iris DataSet

- Weighted Voting
- Majority Voting

2/ Wine DataSet

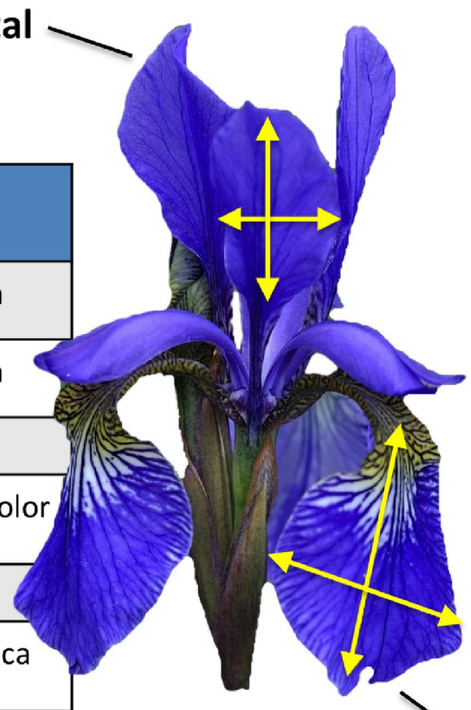
- Bagging
- Boosting

Practice

Dataset : iris

Samples
(instances, observations)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica



Features
(attributes, measurements, dimensions)

Class labels
(targets)

Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

Practice - Classifying Iris Flowers Using Different Classification Models

```
from sklearn import datasets
iris = datasets.load_iris()
X, y = iris.data[:, 1:3], iris.target

from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
import numpy as np

clf1 = LogisticRegression(random_state=1)
clf2 = RandomForestClassifier(random_state=1)
clf3 = GaussianNB()
print('5-fold cross validation:\n')
labels = ['Logistic Regression', 'Random Forest', 'Naive Bayes']
for clf, label in zip([clf1, clf2, clf3], labels):
    scores = model_selection.cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), label))
```



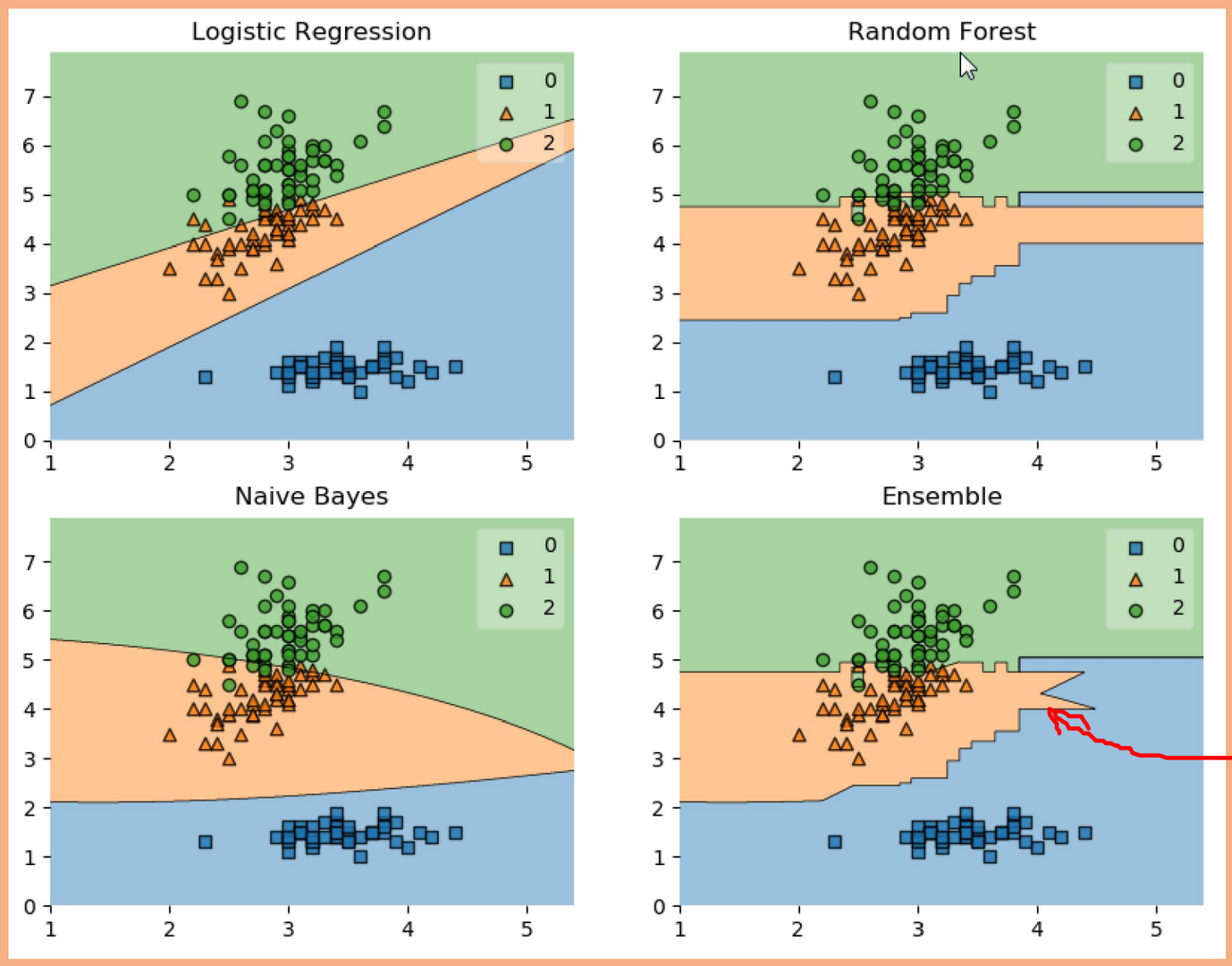
```
5-fold cross validation:

Accuracy: 0.90 (+/- 0.05) [Logistic Regression]
Accuracy: 0.93 (+/- 0.05) [Random Forest]
Accuracy: 0.91 (+/- 0.04) [Naive Bayes]
```

Practice - Classifying Iris Flowers Using Different Classification Models

```
from mlxtend.classifier import EnsembleVoteClassifier
eclf = EnsembleVoteClassifier(clfs=[clf1, clf2, clf3], weights=[1,1,1])

labels = ['Logistic Regression', 'Random Forest', 'Naive Bayes', 'Ensemble']
for clf, label in zip([clf1, clf2, clf3, eclf], labels):
    scores = model_selection.cross_val_score(clf, X, y, cv=5,scoring='accuracy')
    print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), label))

# Plotting Decision Regions
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions
import matplotlib.gridspec as gridspec
import itertools

gs = gridspec.GridSpec(2, 2)
fig = plt.figure(figsize=(10,8))
labels = ['Logistic Regression', 'Random Forest', 'Naive Bayes', 'Ensemble']
for clf, lab, grd in zip([clf1, clf2, clf3, eclf], labels, itertools.product([0, 1], repeat=2)):
    clf.fit(X, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X, y=y, clf=clf)
    plt.title(lab)
plt.show()
```

```
Accuracy: 0.90 (+/- 0.05) [Logistic Regression]
Accuracy: 0.93 (+/- 0.05) [Random Forest]
Accuracy: 0.91 (+/- 0.04) [Naive Bayes]
Accuracy: 0.95 (+/- 0.05) [Ensemble]
```

Practice - Classifying Iris Flowers Using Different Classification Models



Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam

Practice : Using the majority voting principle to make predictions

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator
class MajorityVoteClassifier(BaseEstimator, ClassifierMixin):
    """ A majority vote ensemble classifier
    Parameters
    classifiers : array-like, shape = [n_classifiers]  Different classifiers for the ensemble
    vote : str, {'classlabel', 'probability'} (default='label')
      If 'classlabel' the prediction is based on the argmax of class labels. Else if 'probability', the argmax of the sum of probabilities is used
to predict the class label (recommended for calibrated classifiers).
    weights : array-like, shape = [n_classifiers], optional (default=None)
      If a list of `int` or `float` values are provided, the classifiers are weighted by importance; Uses uniform weights if `weights=None`.
    """
     def __init__(self, classifiers, vote='classlabel', weights=None):
        self.classifiers = classifiers
        self.named_classifiers = {key: value for key, value in _name_estimators(classifiers)}
        self.vote = vote
        self.weights = weights
```

Practice : Using the majority voting principle to make predictions

```
def fit(self, X, y):
    """ Fit classifiers. Parameters
    X : {array-like, sparse matrix}, shape = [n_samples, n_features] Matrix of training samples.
    y : array-like, shape = [n_samples] Vector of target class labels.
    Returns self : object
    """
    if self.vote not in ('probability', 'classlabel'):
        raise ValueError("vote must be 'probability' or 'classlabel'“ "; got (vote=%r)“ % self.vote)
    if self.weights and len(self.weights) != len(self.classifiers):
        raise ValueError('Number of classifiers and weights must be equal''; got %d weights, %d classifiers‘ %
        (len(self.weights), len(self.classifiers)))
# Use LabelEncoder to ensure class labels start with 0, which is important for np.argmax call in self.predict
self.lablenc_ = LabelEncoder()
self.lablenc_.fit(y)
self.classes_ = self.lablenc_.classes_
self.classifiers_ = []
for clf in self.classifiers:
    fitted_clf = clone(clf).fit(X, self.lablenc_.transform(y))
    self.classifiers_.append(fitted_clf)
return self
```

Practice : Using the majority voting principle to make predictions

```
def predict(self, X):
    """ Predict class labels for X.
    Parameters
    ----------
    X : {array-like, sparse matrix}, shape = [n_samples, n_features] Matrix of training samples.
    Returns ----------
    maj_vote : array-like, shape = [n_samples] Predicted class labels.
    """
    if self.vote == 'probability':
        maj_vote = np.argmax(self.predict_proba(X), axis=1)
    else:  # 'classlabel' vote
        #  Collect results from clf.predict calls
        predictions = np.asarray([clf.predict(X) for clf in self.classifiers_]).T
        maj_vote = np.apply_along_axis( lambda x: np.argmax(np.bincount(x, weights=self.weights)),
                                        axis=1,
                                        arr=predictions)
    maj_vote = self.lablenc_.inverse_transform(maj_vote)
    return maj_vote
```

Practice : Using the majority voting principle to make predictions

```
def predict_proba(self, X):
    """ Predict class probabilities for X.
    X : {array-like, sparse matrix}, shape = [n_samples, n_features]
       Training vectors, where n_samples is the number of samples and n_features is the number of features.
    Returns
    avg_proba : array-like, shape = [n_samples, n_classes] Weighted average probability for each class per sample.
    """
    probas = np.asarray([clf.predict_proba(X)  for clf in self.classifiers_])
    avg_proba = np.average(probas, axis=0, weights=self.weights)
    return avg_proba
def get_params(self, deep=True):
    """ Get classifier parameter names for GridSearch"""
    if not deep:
       return super(MajorityVoteClassifier, self).get_params(deep=False)
    else:
       out = self.named_classifiers.copy()
       for name, step in six.iteritems(self.named_classifiers):
          for key, value in six.iteritems(step.get_params(deep=True)):
             out['%s__%s' % (name, key)] = value
       return out
```

4

Practice : Using the majority voting principle to make predictions

```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
X, y = iris.data[50:, [1, 2]], iris.target[50:]
le = LabelEncoder()
y = le.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1, stratify=y)

import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
```

Practice : Using the majority voting principle to make predictions

```
clf1 = LogisticRegression(penalty='l2',  C=0.001, random_state=1)
clf2 = DecisionTreeClassifier(max_depth=1, criterion='entropy', random_state=0)
clf3 = KNeighborsClassifier(n_neighbors=1, p=2, metric='minkowski')

pipe1 = Pipeline([['sc', StandardScaler()], ['clf', clf1]])
pipe3 = Pipeline([['sc', StandardScaler()], ['clf', clf3]])
clf_labels = ['Logistic regression', 'Decision tree', 'KNN']

print('10-fold cross validation:\n')
for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
    scores = cross_val_score(estimator=clf,
                             X=X_train,
                             y=y_train,
                             cv=10,
                             scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
```

```
10-fold cross validation:

ROC AUC: 0.87 (+/- 0.17) [Logistic regression]
ROC AUC: 0.89 (+/- 0.16) [Decision tree]
ROC AUC: 0.88 (+/- 0.15) [KNN]
```

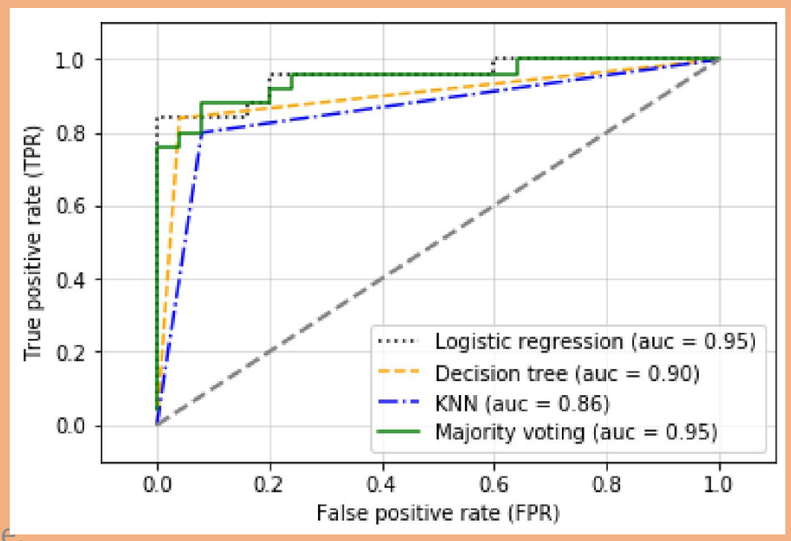
Practice : Using the majority voting principle to make predictions

```
# Majority Rule (hard) Voting

mv_clf = MajorityVoteClassifier(classifiers=[pipe1, clf2, pipe3])

clf_labels += ['Majority voting']
all_clf = [pipe1, clf2, pipe3, mv_clf]

for clf, label in zip(all_clf, clf_labels):
    scores = cross_val_score(estimator=clf,
                             X=X_train,
                             y=y_train,
                             cv=10,
                             scoring='roc_auc')
print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
      % (scores.mean(), scores.std(), label))
```

```
ROC AUC: 0.87 (+/- 0.17) [Logistic regression]
ROC AUC: 0.89 (+/- 0.16) [Decision tree]
ROC AUC: 0.88 (+/- 0.15) [KNN]
ROC AUC: 0.94 (+/- 0.13) [Majority voting]
```

Practice : Evaluating and tuning the ensemble classifier

```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
colors = ['black', 'orange', 'blue', 'green']
linestyles = [':', '--', '-.', '-']
for clf, label, clr, ls in zip(all_clf,  clf_labels, colors, linestyles):
    # assuming the label of the positive class is 1
    y_pred = clf.fit(X_train, y_train).predict_proba(X_test)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_true=y_test, y_score=y_pred)
    roc_auc = auc(x=fpr, y=tpr)
    plt.plot(fpr, tpr, color=clr, linestyle=ls, label='%s (auc = %0.2f)' % (label, roc_auc))
plt.legend(loc='lower right')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray', linewidth=2)

plt.xlim([-0.1, 1.1])
plt.ylim([-0.1, 1.1])
plt.grid(alpha=0.5)
plt.xlabel('False positive rate (FPR)')
plt.ylabel('True positive rate (TPR)')
#plt.savefig('images/04_04', dpi=300)
plt.show()
```

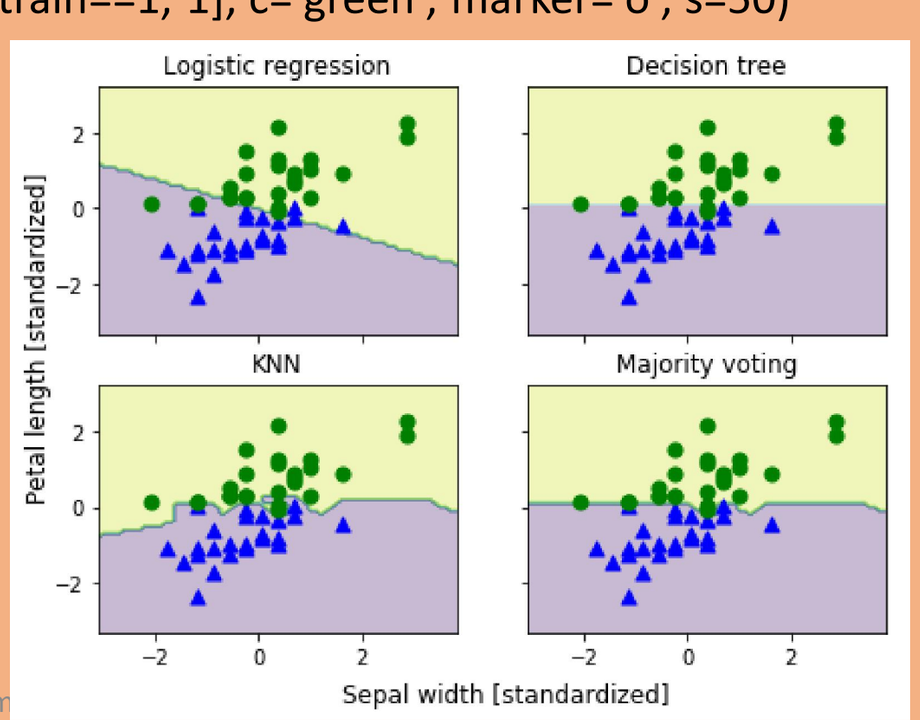


Practice : Evaluating and tuning the ensemble classifier

```
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
from itertools import product

all_clf = [pipe1, clf2, pipe3, mv_clf]

x_min = X_train_std[:, 0].min() - 1
x_max = X_train_std[:, 0].max() + 1
y_min = X_train_std[:, 1].min() - 1
y_max = X_train_std[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

f, axarr = plt.subplots(nrows=2, ncols=2,
                        sharex='col',
                        sharey='row',
                        figsize=(7, 5))
```

Practice : Evaluating and tuning the ensemble classifier

```
for idx, clf, tt in zip(product([0, 1], [0, 1]), all_clf, clf_labels):
    clf.fit(X_train_std, y_train)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0], X_train_std[y_train==0, 1], c='blue', marker='^', s=50)
    axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0], X_train_std[y_train==1, 1], c='green', marker='o', s=50)
    axarr[idx[0], idx[1]].set_title(tt)

  plt.text(-3.5, -5.,
        s='Sepal width [standardized]',
        ha='center', va='center', fontsize=12)
  plt.text(-12.5, 4.5,
        s='Petal length [standardized]',
        ha='center', va='center',
        fontsize=12, rotation=90)

#plt.savefig('images/04_05', dpi=300)
plt.show()
```



Practice : Evaluating and tuning the ensemble classifier

```
print(mv_clf.get_params())
```

```
from sklearn.model_selection import GridSearchCV

params = {'decisiontreeclassifier__max_depth': [1, 2],
          'pipeline-1__clf__C': [0.001, 0.1, 100.0]}

grid = GridSearchCV(estimator=mv_clf,
                    param_grid=params,
                    cv=10,
                    scoring='roc_auc')
grid.fit(X_train, y_train)

for r, _ in enumerate(grid.cv_results_['mean_test_score']):
    print("%0.3f +/- %0.2f %r"
          % (grid.cv_results_['mean_test_score'][r],
             grid.cv_results_['std_test_score'][r] / 2.0,
             grid.cv_results_['params'][r]))
print('Best parameters: %s' % grid.best_params_)
print('Accuracy: %.2f' % grid.best_score_)
```

```
0.933 +/- 0.07 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 0.001}
0.947 +/- 0.07 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 0.1}
0.973 +/- 0.04 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 100.0}
0.947 +/- 0.07 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 0.001}
0.947 +/- 0.07 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 0.1}
0.973 +/- 0.04 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 100.0}
```

```
Best parameters: {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 100.0}
Accuracy: 0.97
```

Practice : Evaluating and tuning the ensemble classifier

Note

By default, the default setting for `refit` in `GridSearchCV` is `True` (i.e., `GridSeachCV(..., refit=True)`), which means that we can use the fitted `GridSearchCV` estimator to make predictions via the `predict` method, for example:

```
grid = GridSearchCV(estimator=mv_clf,
                    param_grid=params,
                    cv=10,
                    scoring='roc_auc')
grid.fit(X_train, y_train)
y_pred = grid.predict(X_test)
```

In addition, the "best" estimator can directly be accessed via the `best_estimator_` attribute.

grid.best_estimator_.classifiers

```
[Pipeline(memory=None,
     steps=[('sc', StandardScaler(copy=True, with_mean=True, with_std=True)), ['clf', LogisticRegression(C=100.0, class_weight
=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=1, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)]]),
 DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=1,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=0,
            splitter='best'),
 Pipeline(memory=None,
     steps=[('sc', StandardScaler(copy=True, with_mean=True, with_std=True)), ['clf', KNeighborsClassifier(algorithm='auto', l
eaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')]])]
```

Practice : Evaluating and tuning the ensemble classifier

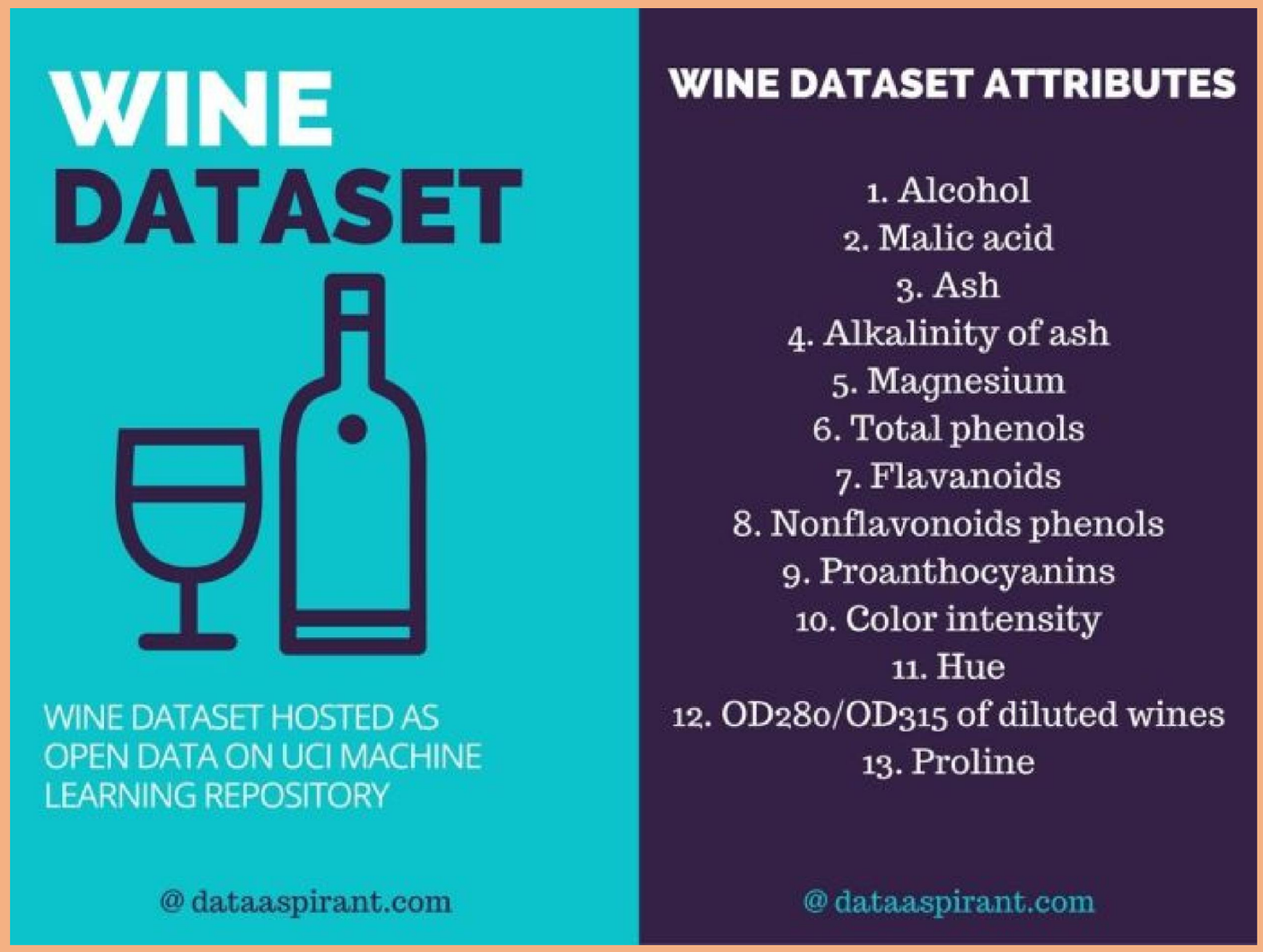
mv_clf = grid.best_estimator_
mv_clf.set_params(**grid.best_estimator_.get_params())
mv_clf

```
MajorityVoteClassifier(classifiers=[Pipeline(memory=None,
     steps=[('sc', StandardScaler(copy=True, with_mean=True, with_std=True)), ('clf', LogisticRegression(C=100.0, class_weight=
None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', ra...ski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'))])],
            vote='classlabel', weights=None)
```

```
MajorityVoteClassifier(classifiers=[Pipeline(memory=None,
     steps=[('sc', StandardScaler(copy=True, with_mean=True, with_std=True)), ('clf', LogisticRegression(C=100.0, class_weight=
None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', ra...ski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'))])],
            vote='classlabel', weights=None)
```

Practice : Bagging - Building an ensemble of classifiers from bootstrap samples

Wine Dataset



Practice : Bagging - Building an ensemble of classifiers from bootstrap samples

```
import pandas as pd
df_wine = pd.read_csv('../InputData/WineData/wine.data',header=None)
df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue', 'OD280/OD315 of diluted wines',
                   'Proline']
# if the Wine dataset is temporarily unavailable from the
# UCI machine learning repository, un-comment the following line
# of code to load the dataset from a local path:
# df_wine = pd.read_csv('wine.data', header=None)

# drop 1 class
df_wine = df_wine[df_wine['Class label'] != 1]

y = df_wine['Class label'].values
X = df_wine[['Alcohol', 'OD280/OD315 of diluted wines']].values
print(X)
print(y)
```

```
array([[ 12.37,   1.82],
       [ 12.33,   1.67],
       [ 12.64,   1.59],
       [ 13.67,   2.46],
       [ 12.37,   2.87],
       [ 12.17,   2.23],
       [ 12.37,   2.3 ],
       [ 13.11,   3.18],
       [ 12.37,   3.48],
       [ 13.34,   1.93],
       [ 12.21,   3.07],
       [ 12.29,   1.82],
       [ 13.86,   3.16],
       [ 13.49,   2.78],
       [ 12.99,   3.5 ],
       [ 11.96,   3.13],
       [ 11.66,   2.14],
       [ 13.03,   2.48],
       [ 11.84,   2.52],
```

```
array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 3])
```

Practice : Bagging - Building an ensemble of classifiers from bootstrap samples

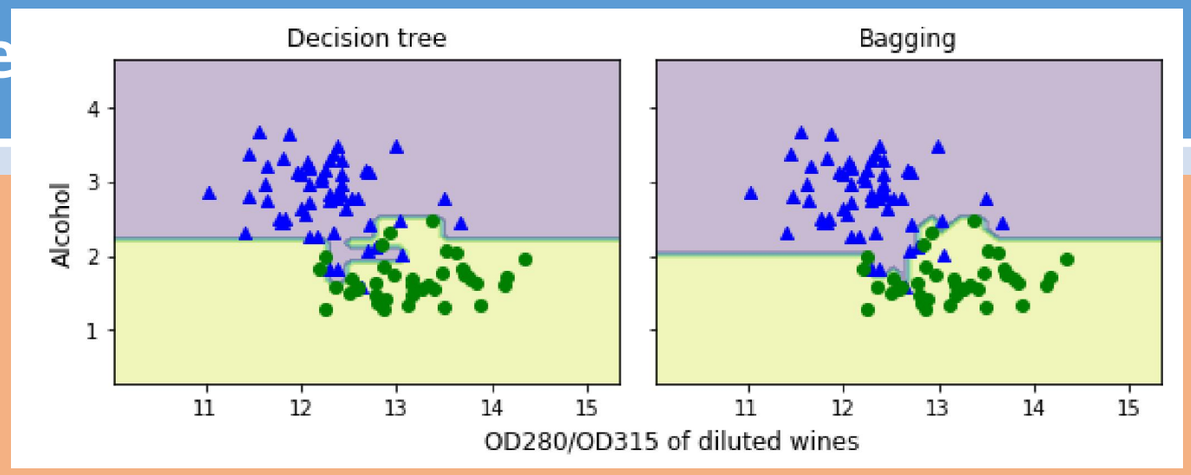
```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

le = LabelEncoder()
y = le.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1, stratify=y)
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(criterion='entropy', max_depth=None, random_state=1)
bag = BaggingClassifier(base_estimator=tree,
                        n_estimators=500,
                        max_samples=1.0,
                        max_features=1.0,
                        bootstrap=True,
                        bootstrap_features=False,
                        n_jobs=1,
                        random_state=1)
```

Practice : Bagging - Building an ensemble of classifiers from bootstrap samples

```
from sklearn.metrics import accuracy_score

tree = tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)

tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print('Decision tree train/test accuracies %.3f/%.3f'
      % (tree_train, tree_test))

bag = bag.fit(X_train, y_train)
y_train_pred = bag.predict(X_train)
y_test_pred = bag.predict(X_test)

bag_train = accuracy_score(y_train, y_train_pred)
bag_test = accuracy_score(y_test, y_test_pred)
print('Bagging train/test accuracies %.3f/%.3f'
      % (bag_train, bag_test))
```

```
Decision tree train/test accuracies 1.000/0.833
Bagging train/test accuracies 1.000/0.917
```

Practice : Bagging - Building an ensemble




```
import numpy as np
import matplotlib.pyplot as plt
x_min = X_train[:, 0].min() - 1
x_max = X_train[:, 0].max() + 1
y_min = X_train[:, 1].min() - 1
y_max = X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
f, axarr = plt.subplots(nrows=1, ncols=2, sharex='col', sharey='row', figsize=(8, 3))
for idx, clf, tt in zip([0, 1], [tree, bag], ['Decision tree', 'Bagging']):
    clf.fit(X_train, y_train)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    axarr[idx].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx].scatter(X_train[y_train == 0, 0], X_train[y_train == 0, 1], c='blue', marker='^')
    axarr[idx].scatter(X_train[y_train == 1, 0], X_train[y_train == 1, 1], c='green', marker='o')
    axarr[idx].set_title(tt)
axarr[0].set_ylabel('Alcohol', fontsize=12)
plt.text(10.2, -0.5, s='OD280/OD315 of diluted wines', ha='center', va='center', fontsize=12)
plt.tight_layout()
plt.show()
```

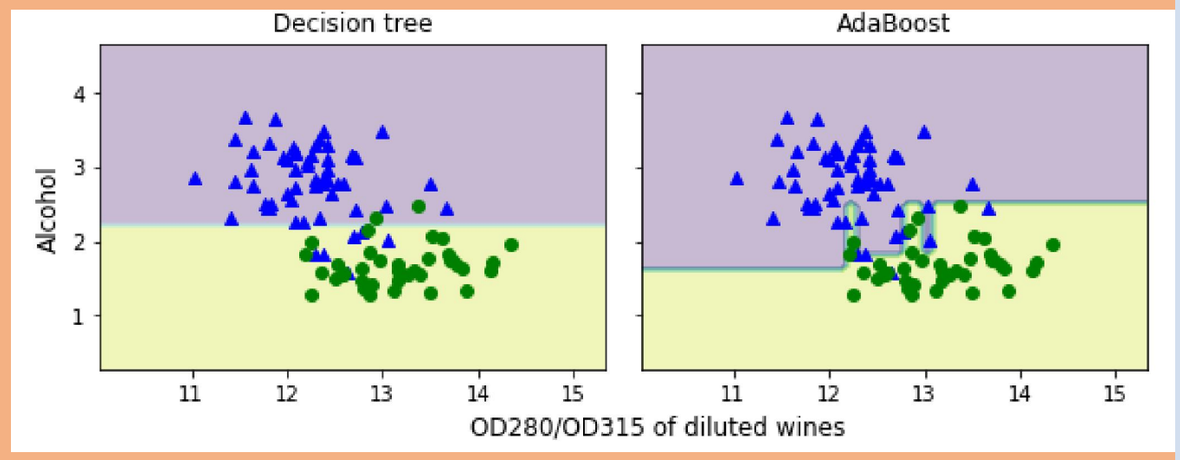
Practice : Leveraging weak learners via adaptive boosting

```
from sklearn.ensemble import AdaBoostClassifier

tree = DecisionTreeClassifier(criterion='entropy', max_depth=1, random_state=1)
ada = AdaBoostClassifier(base_estimator=tree, n_estimators=500, learning_rate=0.1, random_state=1)
tree = tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)

tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print('Decision tree train/test accuracies %.3f/%.3f'% (tree_train, tree_test))

ada = ada.fit(X_train, y_train)
y_train_pred = ada.predict(X_train)
y_test_pred = ada.predict(X_test)

ada_train = accuracy_score(y_train, y_train_pred)
ada_test = accuracy_score(y_test, y_test_pred)
print('AdaBoost train/test accuracies %.3f/%.3f'
      % (ada_train, ada_test))
```

```
Decision tree train/test accuracies 0.916/0.875
AdaBoost train/test accuracies 1.000/0.917
```

Practice : Leveraging weak learners via adaptive boosting

```
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(8, 3))
for idx, clf, tt in zip([0, 1], [tree, ada], ['Decision tree', 'AdaBoost']):
    clf.fit(X_train, y_train)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    axarr[idx].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx].scatter(X_train[y_train == 0, 0], X_train[y_train == 0, 1], c='blue', marker='^')
    axarr[idx].scatter(X_train[y_train == 1, 0], X_train[y_train == 1, 1], c='green', marker='o')
    axarr[idx].set_title(tt)

axarr[0].set_ylabel('Alcohol', fontsize=12)
plt.text(10.2, -0.5, s='OD280/OD315 of diluted wines',
             ha='center', va='center', fontsize=12)

plt.tight_layout()
#plt.savefig('images/07_11.png', dpi=300, bbox_inches='tight')
plt.show()
```



THANK YOU

Dr Tran Anh Tuan, Department of Maths & Computer Science,
HCMUS, VietNam