# Ensemble Methods: Random Forest and Extra Trees

## 1. Theoretical Motivation for Random Forest

The bagging procedure can be further enhanced by introducing random forests. As we discussed in bagging, with a set of $n$ **independent** (or uncorrelated) observations $Z_1, Z_2, \cdots, Z_n$, each with variance $\sigma^2$, the variance of the mean $\mathbb{V}ar\bar{Z}$ of the observations is given by $\frac{\sigma^2}{n}$. **In other words, averaging a set of observations reduces variance.**

However, if bootstrapped data are used instead, the observations $Z_1, Z_2, \cdots, Z_b$ will be correlated. In particular, they are identically distributed (but **not independent**) with some positive correlation $\rho$. It then holds that:

$$\mathbb{V}ar\bar{Z}_b = \rho\sigma^2 + \sigma^2\frac{(1-\rho)}{b}$$

**Q**: So what?
**A**: While the second term goes to zero as the number of observations $b$ increases, the first term remains constant.

This issue is particularly relevant for bagging with decision trees. For example, consider a situation in which a feature provides a very good split of the data. What would happen?

**A**: Such feature will be selected and split for every $g^{(*b)}$ at the root level. Consequently, we will end up with highly correlated predictions.

The major idea of **random forest** is to perform bagging in combination with a "**decorrelation**" of the trees by including only **a subset of randomly selected features** during the tree construction. **This simple but powerful idea will decorrelate the trees, since the strong features might not be included in the subset.**

**Remark**: In random forest, $p$ features are chosen as split candidates from the full set of $d$ features. Typically, we choose $p \approx \sqrt{d}$. For example, if we have 13 features in total, each time we will only use 4 randomly selected from the 13 features and only use 1 from the 4 features to split the tree.

## 2. Python Example: Random Forest for Classifying Digits

To demonstrate the capability of random forest, let's consider one piece of the optical character recognition problem: **the identification of hand-written digits**. In the wild, this problem involves locating and identifying characters in an image. Here, we'll take a shortcut and use Scikit-Learn's set of pre-formatted digits built into the library.

Loading and visualizing the digits data

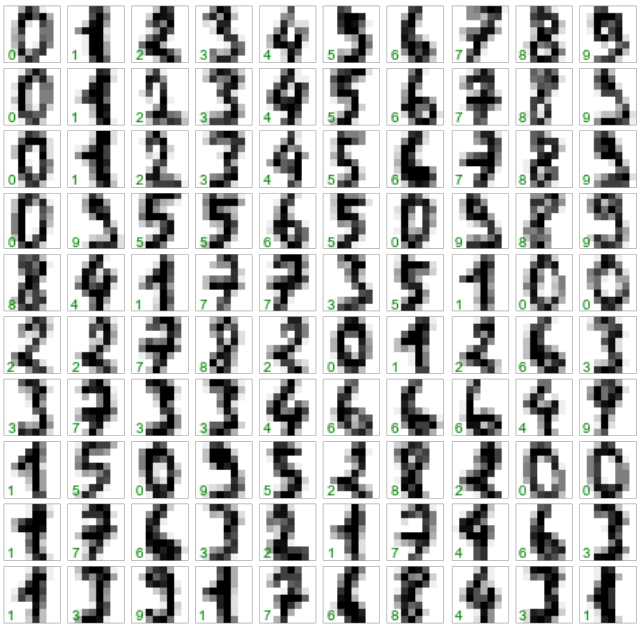
We'll use Scikit-Learn's data access interface and take a look at this data:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.images.shape
```

```
(1797, 8, 8)
```

The image data is a three-dimensional array: 1,797 samples each consisting of an 8 × 8 grid of pixels. Let's visualize the first hundred of these:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes, color='green')
```



To work with this data within Scikit-Learn, we need a two-dimensional, [n_samples, n_features] representation. We can accomplish this by treating each pixel in the image as a feature: that is, by flattening out the pixel arrays to have a length-64 array of pixel values representing each digit. Additionally, we need the target array, which gives the previously determined label for each digit. These two quantities are built into the digits dataset under the data and target attributes, respectively:

```
X = digits.data
X.shape
```

```
(1797, 64)
```

```
y = digits.target
y.shape
```

```
(1797,)
```

We see here that there are 1,797 samples and 64 features.

We can quickly classify the digits using a random forest as follows:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
model = RandomForestClassifier(n_estimators=1000)
model.fit(Xtrain, ytrain)
ypred = model.predict(Xtest)
```

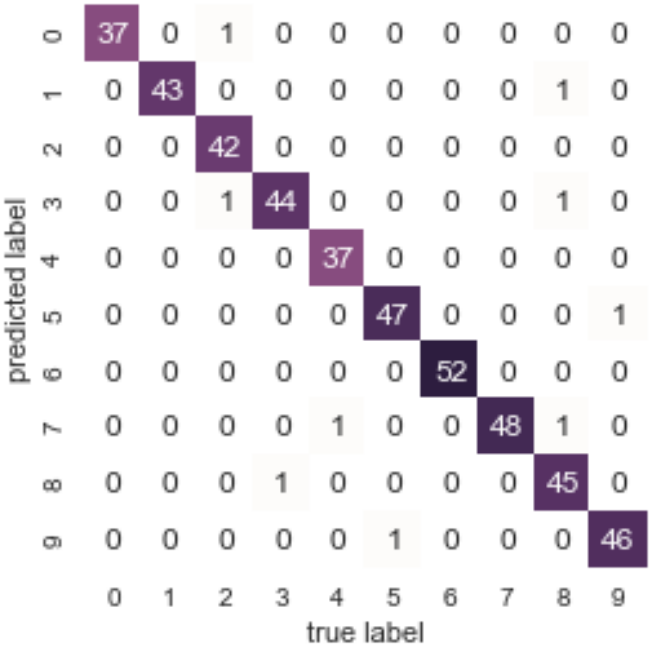
We can take a look at the classification report for this classifier:

```
from sklearn import metrics
print(metrics.classification_report(ytest, ypred))
```

```
              precision    recall  f1-score   support

           0       0.97      1.00      0.99        37
           1       0.95      0.98      0.97        43
           2       1.00      0.95      0.98        44
           3       0.96      0.98      0.97        45
           4       1.00      0.97      0.99        38
           5       0.96      0.98      0.97        48
           6       1.00      1.00      1.00        52
           7       0.96      1.00      0.98        48
           8       0.98      0.94      0.96        48
           9       0.98      0.96      0.97        47

avg / total       0.98      0.98      0.98       450
```

And for good measure, plot the confusion matrix:

```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(ytest, ypred)
cmap = sns.cubehelix_palette(light=1, as_cmap=True)
sns.heatmap(mat.T, square=True, cmap=cmap, annot=True, fmt='d', cbar=False)
plt.xlabel('true label')
plt.ylabel('predicted label');
```




We find that a simple, untuned random forest results in a very accurate classification of the digits data.

## 3. Feature Importance

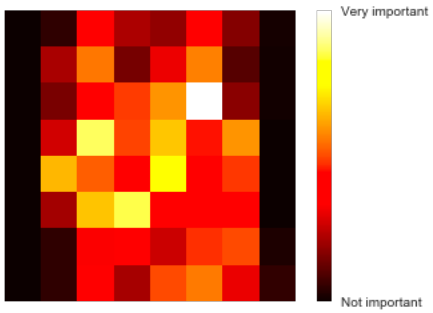
One great quality of Random Forests (and other ensemble tree methods) is that they make it easy to measure the relative importance of each feature (or attribute). Scikit-Learn measures a feature's importance **by looking at how much the tree nodes use that feature to reduce impurity on average** (across all trees in the forest).

**Fun Time**: If a feature can reduce more impurity, the feature is more important in geneal. (1) Yes (2) No

Scikit-Learn computes this score automatically for each feature after training; then, it scales the results so that the sum of all importance equals 1. You can access the result using the feature_importances_ variable. For example, suppose you plot each pixel's importance of a Random Forest classifier on the

previous example of digit images with 8x8 pixels. In that case, you get the image represented in the figure below.

```
import matplotlib
def plot_digit(data):
    image = data.reshape(8, 8)
    plt.imshow(image, cmap = matplotlib.cm.hot,
               interpolation="nearest")
    plt.axis("off")

plot_digit(model.feature_importances_)

cbar = plt.colorbar(ticks=[model.feature_importances_.min(),
model.feature_importances_.max()])
cbar.ax.set_yticklabels(['Not important', 'Very important'])
```




**Remarks**

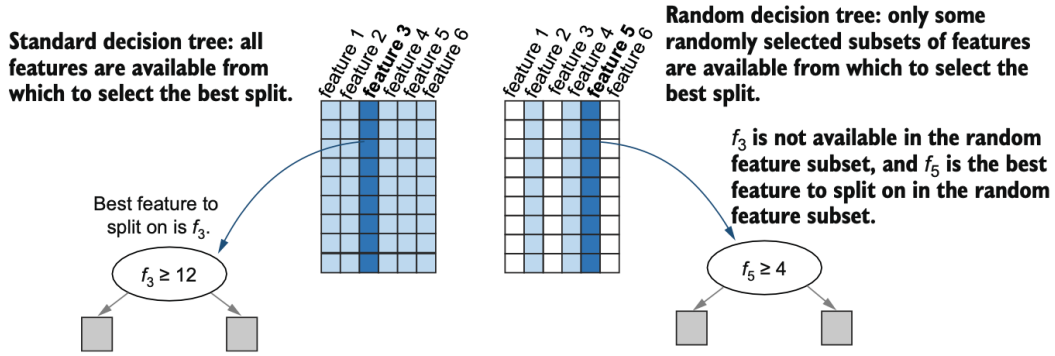
1. One benefit of using random forests is that they also provide a natural mechanism for scoring features based on their importance. This means we can rank features to identify the most important ones and drop less effective features, thus performing **feature selection**!
2. Feature selection, also known as variable subset selection, is a procedure for identifying the most influential or relevant data features/attributes. Feature selection is an important step of the modeling process, especially for high-dimensional data. Dropping the least-relevant features often improves generalization performance and minimizes overfitting. It also often improves the computational efficiency of training. These concerns are consequences of the curse of dimensionality, where a large number of features can inhibit a model's ability to generalize effectively. See *The Art of Feature Engineering: Essentials for Machine Learning* by Pablo Duboue (Cambridge University Press, 2020) to learn more about feature selection and engineering.
3. SHAP (SHapley Additive exPlanations) is a more sophisticated analysis to obtain feature importance for any machine model. SHAP is a feature-attribution technique that computes feature importance based on each feature's contribution to the overall prediction. See *Ensemble Methods for Machine Learning* by Gautam Kunapuli (Manning, 2023) for learn more about SHAP and other explainable methods.

You can download the source codes `RFDigitsRecog.ipynb` to reproduce these two examples from the course website.

## 4. Extra Trees Algorithm[1]

Extremely randomized trees take the idea of randomized decision trees to the extreme by selecting not just the splitting variable from a random subset of features but also **the splitting threshold**!

To understand this more clearly, recall that every node in a decision tree tests a condition of the form "is $f_k$ < threshold?" where $f_k$ is the $k^{\text{th}}$ feature, and threshold is the split value.




- Standard decision-tree learning looks at all the features to determine the best $f_k$ and then looks at **all the values of that feature** to determine the threshold.
- Randomized decision-tree learning looks at a random subset of features to determine the best $f_k$ and then looks at all the values of that feature to determine the threshold. (Random Forest)
- Extremely randomized decision-tree learning also looks at a random subset of features to determine the best $f_k$. **But to be even more efficient, it selects a random splitting threshold.** (Extra Trees)

## 5. Python Example: Bagging, Random Forest, and Extra Trees

We'll train and evaluate the performance of Bagging, Random Forest, and Extra Trees using the Breast Cancer Diagnosis dataset. We'll use the Wisconsin Diagnostic Breast Cancer (WDBC) dataset, a common benchmark data set in machine learning.

---

[1] Extra Trees does not mean more trees. Instead, it should be written as **ExtRa**, which means Extra Randomized. **ExtRa** Trees are Random Forests with an additional source of randomness.

The WDBC data set was created by applying feature extraction techniques on patient biopsy (檢體) medical images. More concretely, for each patient, the data describes the size and texture of the cell nuclei of cells extracted during biopsy. WDBC is available in scikit-learn. In addition, we also create a RandomState so that we can generate randomization in a reproducible manner:

```
from sklearn.datasets import load_breast_cancer
dataset = load_breast_cancer()

# Convert to a Pandas DataFrame so we can visualize nicely
import pandas as pd
import numpy as np
df = pd.DataFrame(data=dataset['data'], columns=dataset['feature_names'])
i = np.random.permutation(len(dataset['target']))
df = df.iloc[i, :7]
df['diagnosis'] = dataset['target'][i]
df = df.reset_index()
df.head()
```

| | index | mean radius | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | diagnosis |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 391 | 8.734 | 16.84 | 55.27 | 234.3 | 0.10390 | 0.07428 | 0.000000 | 1 |
| **1** | 18 | 19.810 | 22.15 | 130.00 | 1260.0 | 0.09831 | 0.10270 | 0.147900 | 0 |
| **2** | 99 | 14.420 | 19.77 | 94.48 | 642.5 | 0.09752 | 0.11410 | 0.093880 | 0 |
| **3** | 276 | 11.330 | 14.16 | 71.79 | 396.6 | 0.09379 | 0.03872 | 0.001487 | 1 |
| **4** | 377 | 13.460 | 28.21 | 85.89 | 562.1 | 0.07517 | 0.04726 | 0.012710 | 1 |

```
X, y = dataset['data'], dataset['target']
rng=np.random.RandomState(seed=4190) # Initialize a random number generator
```

Once we've preprocessed our data set, we'll train and evaluate bagging with decision trees, random forests, and Extra Trees to answer the following questions:

- How does the ensemble performance change with ensemble size? That is, what happens when our ensembles get bigger and bigger?
- How does the ensemble performance change with base learner complexity? That is, what happens when our individual base estimators become more and more complex?

In this case study, since all three ensemble methods considered use decision trees as base estimators, one **"measure" of complexity is tree depth, with deeper trees being more complex.**

**Ensemble Size vs. Ensemble Performance**

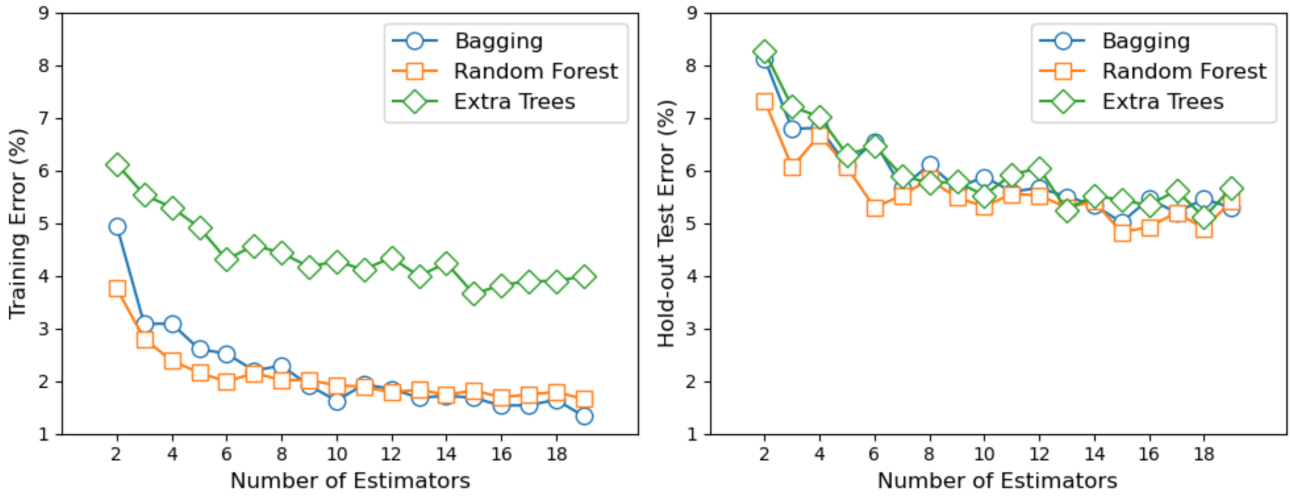
First, let's look at how training and testing performance change with ensemble size by comparing the behavior of the three algorithms as the parameter `n_estimators` increases. As always, we follow good machine-learning practices and split the data set into a training set and a hold-out test set randomly.

Recall that because the test set is held out during training, the test error is generally a useful estimate of how well we'll do on future data, that is, generalize. However, because we don't want our learning and evaluation to be at the mercy of randomness, we'll repeat this experiment 30 times and average the results (or even better, you alter the codes and do cross validation). In the following listing, we'll see how the ensemble size influences model performance.

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier,
ExtraTreesClassifier
from sklearn.metrics import accuracy_score

max_leaf_nodes = 8
n_runs = 30
n_estimator_range = range(2, 20, 1)

bag_trn_error = np.zeros((n_runs, len(n_estimator_range)))  # Train error for the
bagging classifier
rf_trn_error = np.zeros((n_runs, len(n_estimator_range)))   # Train error for the
random forest classifier
xt_trn_error = np.zeros((n_runs, len(n_estimator_range)))   # Train error for the
extra trees classifier

bag_tst_error = np.zeros((n_runs, len(n_estimator_range)))  # Test error for the
bagging classifier
rf_tst_error = np.zeros((n_runs, len(n_estimator_range)))   # Test error for the
random forest classifier
xt_tst_error = np.zeros((n_runs, len(n_estimator_range)))   # Test error for the extra
trees classifier

for run in range(0, n_runs):

    # Split into train and test sets
    X_trn, X_tst, y_trn, y_tst = train_test_split(X, y, test_size=0.25,
random_state=rng)

    for j, n_estimators in enumerate(n_estimator_range):
        # Train using bagging
        bag_clf =
BaggingClassifier(estimator=DecisionTreeClassifier(max_leaf_nodes=max_leaf_nodes),
                                    n_estimators=n_estimators, max_samples=0.5,
n_jobs=-1, random_state=rng)
        bag_clf.fit(X_trn, y_trn)
        bag_trn_error[run, j] = 1 - accuracy_score(y_trn, bag_clf.predict(X_trn))
        bag_tst_error[run, j] = 1 - accuracy_score(y_tst, bag_clf.predict(X_tst))

        # Train using random forests
        rf_clf = RandomForestClassifier(max_leaf_nodes=max_leaf_nodes,
                                        n_estimators=n_estimators, n_jobs=-1)
        rf_clf.fit(X_trn, y_trn)
        rf_trn_error[run, j] = 1 - accuracy_score(y_trn, rf_clf.predict(X_trn))
        rf_tst_error[run, j] = 1 - accuracy_score(y_tst, rf_clf.predict(X_tst))

        # Train using extra trees
```

```
        xt_clf = ExtraTreesClassifier(max_leaf_nodes=max_leaf_nodes, bootstrap=True,
                                      n_estimators=n_estimators, n_jobs=-1,
random_state=rng)
        xt_clf.fit(X_trn, y_trn)
        xt_trn_error[run, j] = 1 - accuracy_score(y_trn, xt_clf.predict(X_trn))
        xt_tst_error[run, j] = 1 - accuracy_score(y_tst, xt_clf.predict(X_tst))

    results = (bag_trn_error, bag_tst_error, \
               rf_trn_error, rf_tst_error, \
               xt_trn_error, xt_tst_error)
```

```
%matplotlib inline

import matplotlib.pyplot as plt

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
n_estimator_range = range(2, 20, 1)

# Plot the training error
m = np.mean(bag_trn_error*100, axis=0)
ax[0].plot(n_estimator_range, m, linewidth=1.5, marker='o', markersize=9, mfc='w');

m = np.mean(rf_trn_error*100, axis=0)
ax[0].plot(n_estimator_range, m, linewidth=1.5, marker='s', markersize=9, mfc='w');

m = np.mean(xt_trn_error*100, axis=0)
ax[0].plot(n_estimator_range, m, linewidth=1.5, marker='D', markersize=9, mfc='w');

ax[0].legend(['Bagging', 'Random Forest', 'Extra Trees'], fontsize=12)
ax[0].set_xlabel('Number of Estimators', fontsize=12)
ax[0].set_ylabel('Training Error (%)', fontsize=12)
ax[0].set_xticks(range(2, 20, 2))
ax[0].axis([0, 21, 1, 9])
# ax[0].grid()

# Plot the test error
m = np.mean(bag_tst_error*100, axis=0)
ax[1].plot(n_estimator_range, m, linewidth=1.5, marker='o', markersize=9, mfc='w');

m = np.mean(rf_tst_error*100, axis=0)
ax[1].plot(n_estimator_range, m, linewidth=1.5, marker='s', markersize=9, mfc='w');

m = np.mean(xt_tst_error*100, axis=0)
ax[1].plot(n_estimator_range, m, linewidth=1.5, marker='D', markersize=9, mfc='w');

ax[1].legend(['Bagging', 'Random Forest', 'Extra Trees'], fontsize=12)
ax[1].set_xlabel('Number of Estimators', fontsize=12)
ax[1].set_ylabel('Hold-out Test Error (%)', fontsize=12)
# ax[1].grid()
ax[1].set_xticks(range(2, 20, 2))
ax[1].axis([0, 21, 1, 9]);
plt.tight_layout()
```




We can now visualize the averaged training and test errors on the WDBC data set.

- As expected, the training error for all the approaches decreases steadily as the number of estimators increases.
- The test error also decreases with ensemble size and then stabilizes. As the test error is an estimate of the generalization error, our experiment confirms our intuition about the performance of these ensemble methods in practice.
- All three approaches greatly outperform single decision trees (where the plot begins). This shows that, in practice, even if single decision trees are unstable, ensembles of decision trees are robust and can generalize well.

## Base Learner Complexity vs. Ensemble Performance

Next, we compare the behavior of the three algorithms as the complexity of the base learners increases. There are several ways to control the complexity of the base decision trees: **maximum depth**, **maximum number of leaf nodes**, **impurity criteria**, and so on. Here, we compare the performance of the three ensemble methods with complexity of each base learner determined by `max_leaf_nodes`.

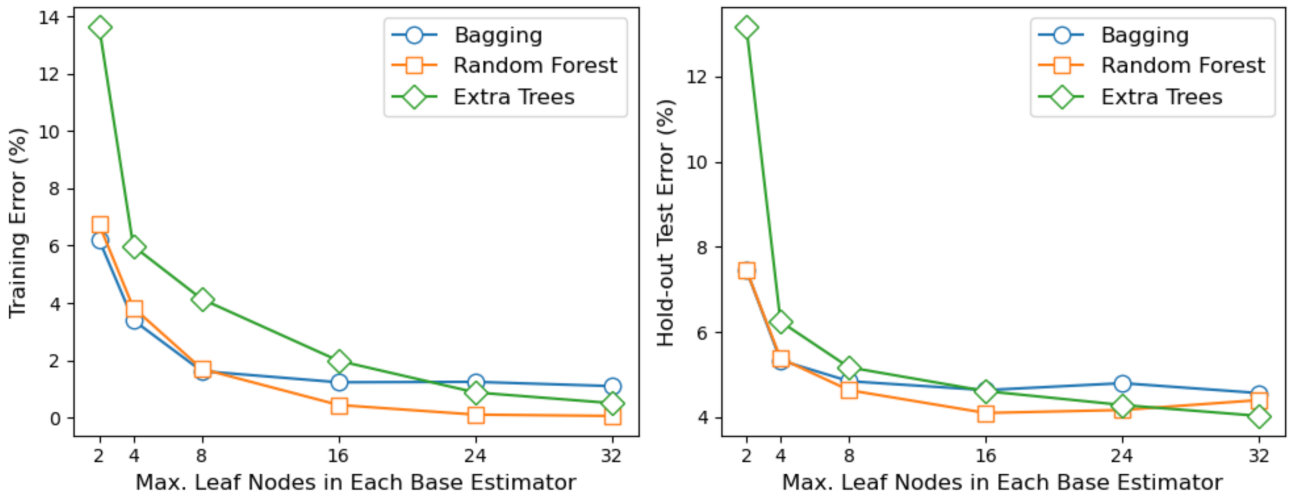
This comparison can be performed in a manner similar to the previous one. To allow each ensemble method to use increasingly complex base learners, we can steadily increase the number of in `max_leaf_nodes` for each base decision tree. That is, in each of BaggingClassifier, RandomForestClassifier, and ExtraTreesClassifier, we set `max_leaf_nodes` = 2, 4, 8, 16, and 32. Again, we'll repeat this experiment 30 times and average the results (or even better, you can alter the codes and do cross validation).

```
n_estimators = 16
n_runs = 30
n_leaf_range = [2, 4, 8, 16, 24, 32]
```

```
bag_trn_error = np.zeros((n_runs, len(n_leaf_range)))  # Train error for the bagging
classifier
rf_trn_error = np.zeros((n_runs, len(n_leaf_range)))   # Train error for the random
forest classifier
xt_trn_error = np.zeros((n_runs, len(n_leaf_range)))   # Train error for the extra
trees classifier

bag_tst_error = np.zeros((n_runs, len(n_leaf_range)))  # Test error for the bagging
classifier
rf_tst_error = np.zeros((n_runs, len(n_leaf_range)))   # Test error for the random
forest classifier
xt_tst_error = np.zeros((n_runs, len(n_leaf_range)))   # Test error for the extra
trees classifier

for run in range(0, n_runs):

    # Split into train and test sets
    X_trn, X_tst, y_trn, y_tst = train_test_split(X, y, test_size=0.25,
random_state=rng)

    for j, max_leaf_nodes in enumerate(n_leaf_range):
        # Train using bagging
        bag_clf =
BaggingClassifier(estimator=DecisionTreeClassifier(max_leaf_nodes=max_leaf_nodes),
                                        n_estimators=n_estimators, max_samples=0.5,
n_jobs=-1, random_state=rng)
        bag_clf.fit(X_trn, y_trn)
        bag_trn_error[run, j] = 1 - accuracy_score(y_trn, bag_clf.predict(X_trn))
        bag_tst_error[run, j] = 1 - accuracy_score(y_tst, bag_clf.predict(X_tst))

        # Train using random forests
        rf_clf = RandomForestClassifier(max_leaf_nodes=max_leaf_nodes,
                                            n_estimators=n_estimators, n_jobs=-1,
random_state=rng)
        rf_clf.fit(X_trn, y_trn)
        rf_trn_error[run, j] = 1 - accuracy_score(y_trn, rf_clf.predict(X_trn))
        rf_tst_error[run, j] = 1 - accuracy_score(y_tst, rf_clf.predict(X_tst))

        # Train using extra trees
        xt_clf = ExtraTreesClassifier(max_leaf_nodes=max_leaf_nodes, bootstrap=True,
                                          n_estimators=n_estimators, n_jobs=-1,
random_state=rng)
        xt_clf.fit(X_trn, y_trn)
        xt_trn_error[run, j] = 1 - accuracy_score(y_trn, xt_clf.predict(X_trn))
        xt_tst_error[run, j] = 1 - accuracy_score(y_tst, xt_clf.predict(X_tst))

    results = (bag_trn_error, bag_tst_error, \
                   rf_trn_error, rf_tst_error, \
                   xt_trn_error, xt_tst_error)
```

```
%matplotlib inline

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
n_leaf_range = [2, 4, 8, 16, 24, 32]

# Plot the training error
m = np.mean(bag_trn_error*100, axis=0)
ax[0].plot(n_leaf_range, m, linewidth=1.5, marker='o', markersize=9, mfc='w');

m = np.mean(rf_trn_error*100, axis=0)
ax[0].plot(n_leaf_range, m, linewidth=1.5, marker='s', markersize=9, mfc='w');

m = np.mean(xt_trn_error*100, axis=0)
ax[0].plot(n_leaf_range, m, linewidth=1.5, marker='D', markersize=9, mfc='w');

ax[0].legend(['Bagging', 'Random Forest', 'Extra Trees'], fontsize=12)
ax[0].set_xlabel('Max. Leaf Nodes in Each Base Estimator', fontsize=12)
```

```
ax[0].set_ylabel('Training Error (%)', fontsize=12)
ax[0].set_xticks(n_leaf_range)
# ax[0].grid()

# Plot the test error
m = np.mean(bag_tst_error*100, axis=0)
ax[1].plot(n_leaf_range, m, linewidth=1.5, marker='o', markersize=9, mfc='w');

m = np.mean(rf_tst_error*100, axis=0)
ax[1].plot(n_leaf_range, m, linewidth=1.5, marker='s', markersize=9, mfc='w');

m = np.mean(xt_tst_error*100, axis=0)
ax[1].plot(n_leaf_range, m, linewidth=1.5, marker='D', markersize=9, mfc='w');

ax[1].legend(['Bagging', 'Random Forest', 'Extra Trees'], fontsize=12)
ax[1].set_xlabel('Max. Leaf Nodes in Each Base Estimator', fontsize=12)
ax[1].set_ylabel('Hold-out Test Error (%)', fontsize=12)
ax[1].set_xticks(n_leaf_range)
# ax[1].grid();
plt.tight_layout()
```




**Remarks:**

- We can perform a similar performance evaluation by changing the **maximum depth** instead of the **maximum number of leaf nodes**.
- Recall that highly complex trees are inherently unstable and sensitive to small perturbations in the data. This means that, in general, if we increase the complexity of the base learners, we'll need a lot more of them to reduce the variability of the ensemble overall successfully. Here, we've fixed `n_estimators=16`. This can be altered, too.

To reproduce these results, you can download the source code `Bagging_RF_ET.ipynb` from the course website.