Lecture 10: Ensemble Learning Methods

ENEE 691 – Machine Learning and Photonics @ UMBC

Ergun Simsek, Ph.D. and Masoud Soroush, Ph.D.

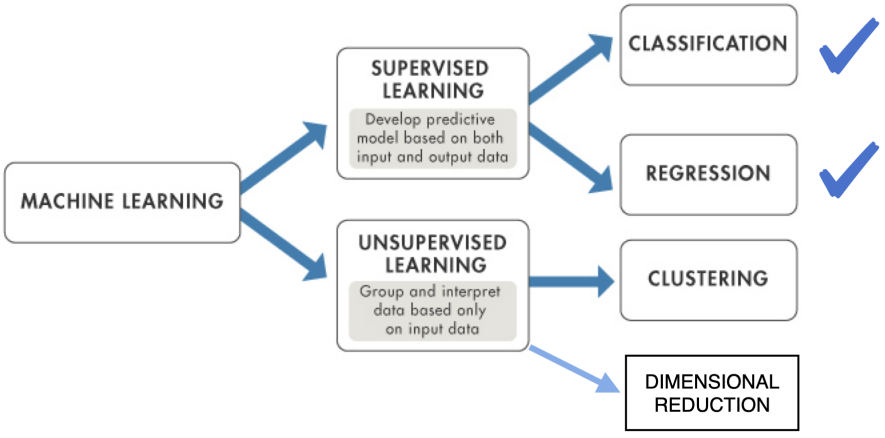
April 4, 2023

1 Ensemble Learning Methods

In today's lecture[1], we introduce a series of techniques known as the ensemble learning methods. Similar to tree-based methods, the ensemble learning techniques can be applied to both classification and regression tasks in the realm of supervised learning.

1.1 Background

The basic idea behind ensemble learning methods is to combine many simple *building block* models to obtain a single but potentially much more powerful (than individual building blocks) model. The individual building block models in this context are called **weak learners**. The idea of ensemble learning can be applied to both classification and regression tasks.



In this lecture, we will introduce three major ensemble learning methods, namely **bagging**, **random forests**, and **boosting**. As we will see, bagging and random forests are among *parallel* ensemble learning techniques, whereas boosting is a *sequential* ensemble learning method. The motivation behind the parallel ensemble

[1] This product is a property of UMBC, and no distribution is allowed. These notes are solely for your own use, as a registered student in this class.

learning methods is to exploit *independence between weak learners*. In contrast, the motivation behind the sequential ensemble learning techniques is to exploit the *dependence between weak learners*.

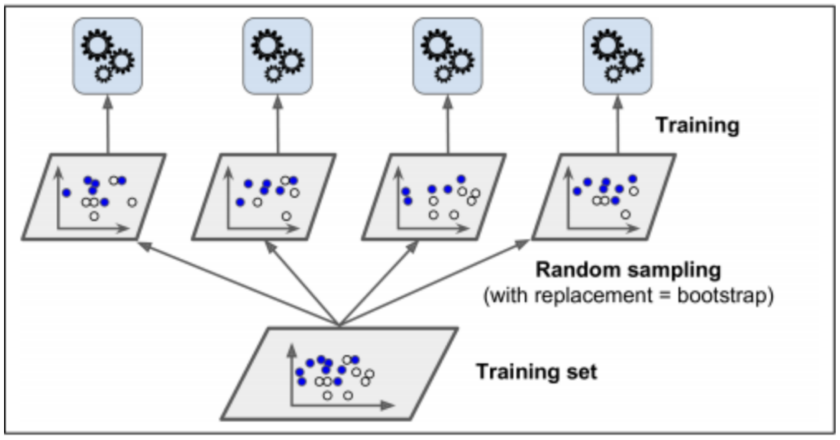
Note: The ensemble learning techniques can be applied to any type of building block (weak learner). However, in this lecture, we primarily assume that the weak learners are (regression and classification) decision trees.

1.2 Bagging

Recall from the previous chapter that one of the drawbacks of decision trees is that they typically suffer from *high variance*. *Bagging* (which is sometimes referred to as the Bootstrap Aggregation method as well) is a procedure to reduce the variance of a statistical learning method. Since decision trees suffer from high variance quite often, the bagging method is consistently applied to trees. To see how the bagging method works, first let us recall a fact from statistics.

A Fact from Statistics: Let $X_1, X_2, \cdots, X_n$ be independent observations, each with a variance σ^2. The variance of the mean variable $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ turns out to be $\frac{\sigma^2}{n}$.

The above facts states that *averaging a set of observations reduces the variance*. The bagging method employs an averaging mechanism to reduce the variance and increase the accuracy of the test subset in a statistical learning method. The question is how this averaging mechanism works in practice. In order to perform the averaging procedure, we need to access multiple training datasets, whereas we only have a single dataset at hand! The answer is that we **draw repeated samples from the single training dataset**. In bagging methods, we allow replacement in the sampling process (*i.e.* an instance can be chosen multiple times to present in many samples of training datasets). The following picture[2] depicts the basic idea behind the bagging method.



Remark: If the sampling process is done *without replacement*, we call the corresponding averaging method **pasting** (instead of bagging).

To formalize the idea of the bagging method suppose we have d features denoted by $\vec{x} \in R \subset \mathbb{R}^d$, and a continuous target variable y. Let f denote the underlying weak learner (*i.g.* the building block regression tree model). From the training dataset, we generate B bootstrapped training datasets (through bootstrapped sampling). We then train the weak learner B times, once on each bootstrapped sample. At the end, in order to make a prediction for an observation $\vec{x}^*$, we use the following average formula

[2] Source: Aurelien Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly Media; 2nd edition, 2019.

$$y^* = f_{bag}(\vec{x}^*) = \frac{1}{B}\sum_{b=1}^{B} f^b(\vec{x}^*)\,, \tag{1}$$

where $f^b(\vec{x}^*)$ denotes the prediction of the weak learner trained on the b-th bootstrapped training dataset.

On the other hand, if the target variable y is a categorical variable, in order to make prediction for the class of a test observation, we record the predicted class from each of the classification trees trained on the B bootstrapped samples, and then take the **majority vote**. This implies that the ultimate predicted class for the test observation is the *most commonly occurring class* among B predictions (coming from the B bootstrapped training samples).

When the weak learners are decision tress, the individual trees are constructed deep. Therefore, each individual tree has high variance, but low bias. Averaging the B trees reduces the variance. Bagging has demonstrated impressive improvements in accuracy of (test) predictions by combining together hundreds or sometimes even thousands of trees into a single algorithm.

1.2.1 Out-of-Bag (OOB) Evaluation

Recall that in the bagging method, bootstrapped samples of the training dataset are generated. It can be shown that on average, each *individual* bagged tree uses of around $\frac{2}{3}$ of the observations (The more accurate estimate is $1 - \frac{1}{e} \simeq 63.21\%$). The remaining $\frac{1}{3}$ of the training observations that were not used to train a given bagged tree are called **out-of-bag** (oob) observations.

Question: What are out-of-bag observations good for? Can we take the advantage of them in any useful manner?

Answer: The out-of-bag observations can be used to estimate the error of the model! Take the i-th (training) observation. We can predict the target for the i-th observation using each of the individual trees (weak learners) for which the i-th observation is out-of-bag! This means that around $\frac{B}{3}$ of the trees can be used to make predictions for the i-th observation. Then to have a final prediction for the i-th observation, we take the average of the targets coming from $\frac{B}{3}$ of trees (in case of regression) or can take the majority vote (in case of classification). In this way, we can obtain a prediction for all n observations if we repeat this process for any observation. From this, one can easily calculate OOB-MSE (for regression tasks) or OOB-classification error (for classification tasks). The resulting OOB error obtained in this manner is a valid estimate of the test error of the bagged model!

1.3 Random Forests

In random forests, the weak learners are all decision trees (This justifies the term *forest* in name of this method). Moreover, random forests operate very similar to the bagging method except that they perform a small tweak. This tweak, however, provides an improvement over bagged trees. The tweak of random forests has the effect of decorrelating the individual trees from one another.

Question: How do random forests exactly work?

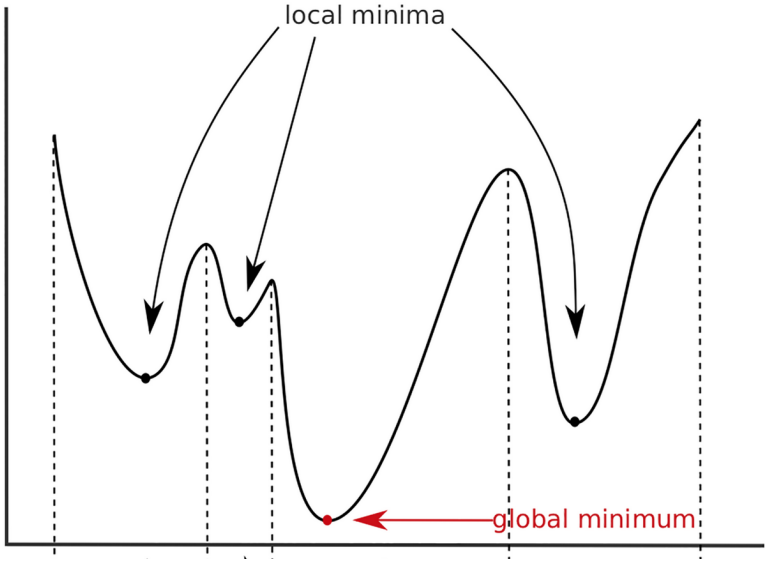
Answer: Just like bagged trees, random forests grow independent trees over bootstrapped samples of the training data. However, in the process of growing trees, for every split that takes place in trees, a *random sample of m features* (rather than the full set of d features) is considered (This justifies the term random in the name of this method). The split is then allowed to use only one of the m features. In random forests, m is typically chosen to be $m \simeq \sqrt{d}$. In other words, in constructing random forests, at each split the algorithm is not allowed to consider even the majority of the features! This may sound a very strange idea at first glance. Suppose the total number of features is 16. Then in constructing a random forest, we have only $\sqrt{16} = 4$ random features available at each splitting point in the trees!

Question: What is the rationale for considering only $m \simeq \sqrt{d}$ random features at each split?

Answer: The short answer is that making only a fraction of the whole features available at splitting points of the bagged trees *decorrelates the trees*. To see the benefit of considering only a random fraction of features for splitting in trees, assume that we have a very significant feature, along with a number of other moderately significant features. If we follow the logic of the bagging method without applying the tweak of random forests, then most of the bagged trees will use the very significant feature at the top splitting point. Therefore, most of the trees will look very similar one another. Thus, the although bagged tree started independently, their performance is very similar and hence, are highly *correlated*! This would then have the effect that the average of highly correlated trees is not very different from the individual trees, and the variance is not reduced very much! However, in the random forest approach, for each tree roughly $(d-m)/d$ of the splits *will not even consider the very significant feature*, and hence a given tree has a greater chance to perform different overall from other trees. In this manner, the bagged trees are decorrelated, and the variance will reduce more easily.

Remark: In the random forest approach, if we allow the number of random features to be chosen to be $m = d$, then the resulting random forest is *exactly* a bagged model.

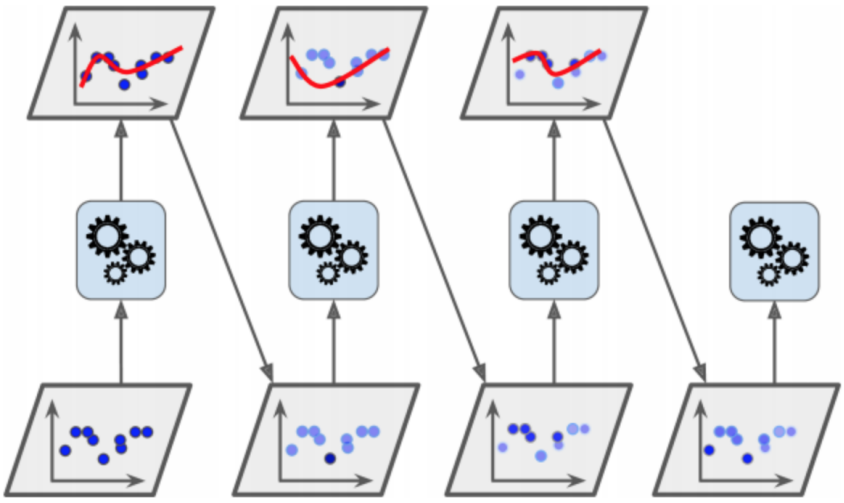
Remark: In the bagging method, the trees are grown independently on random samples of (training) observations. Therefore, the trees tend to be similar to one another. This in turn would have the effect that the bagging method may sometimes get caught in local minima, and can fail to thoroughly explore the parameter space. In contrast, the random forest (because of using a random subset of features at splitting points) has the capability to escape from local minima, and explore a wider region of the parameter space. Hence, if there are local minima, the random forest (in comparison with the bagging method) has a greater chance to reach the global minimum in the parameter space of the model.



1.4 Boosting

Like bagging, boosting is another general approach that can be applied to a variety of weak learners for both regression and classification tasks (In this lecture, we will assume that the weak learners are decision trees though). However, unlike the bagging method, boosting does not take the advantage of the bootstrap sampling. Instead, the trees are constructed in a *sequential manner* in the boosting method. This means that each tree is grown using information from previously grown trees. In other words, *each tree modifies the previous dataset*, and then next tree works on the modified dataset coming from the previous tree in the sequence. The following picture[3] schematically shows how the boosting method works.

[3] Source: Aurelien Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly Media; 2nd edition, 2019.



Now, let us present the boosting algorithm (for regression) which illustrates how it exactly works. We then make some comments on the algorithm, and highlight some important aspects of it. The boosting algorithm consists of three steps where the second step is a loop over B trees.

Boosting Algorithm for Regression Tasks

1. Set $f(\vec{x}) = 0$, and $r_i = y_i$ for all instances i in the training subset.
2. Repeat for $b = 1, 2, \cdots, B$:
   i. Fit a tree $f^b(\vec{x})$ with p nodes (leaves) to the training dataset $\{(\vec{x}_i, r_i) \,|\, i \in \text{train subset}\}$.
   ii. Update f by adding in a weakened version of the current tree:
   $$f(\vec{x}) \longleftarrow f(\vec{x}) + \lambda f^b(\vec{x}) \,. \tag{2}$$
   iii. Update the residual target by
   $$r_i \longleftarrow r_i - \lambda f^b(\vec{x}_i) \,. \tag{3}$$
3. Output the final model
   $$f(\vec{x}) = \lambda \sum_{b=1}^{B} f^b(\vec{x}) \,. \tag{4}$$

As observed in the above algorithm, there exist three hyperparameters in the boosting algorithm (B, λ, and p):

- The total number of trees is B which is a hyperparameter of the model. Note that if B is chosen to be a very large number, then boosting (unlike bagging and random forests) can lead to an overfitting problem.
- The shrinkage coefficient λ is also a hyperparameter of the model. λ is a *positive number* and is typically chosen to be small (typical values are 0.01 and 0.001). Note that a very small value of λ would then require a very large number of trees B in order to achieve a good performance.
- The positive integer p is another hyperparameter of the model that controls the complexity (*i.e.* depth) of the trees. We typically try to keep the trees as simple as possible. As a matter of fact, quite often $p = 1$ (the simplest tree which is nothing but a single split) works well.

A similar boosting algorithm for classification exists. However, in here we omit the details of the boosting algorithm that performs classification tasks. We conclude this section by highlighting an important characteristic of the boosting approach. The boosting method is, in general, a **slow learning** process:

Remark: Note that *by fitting small trees to the residual targets r_i, we slowly improve $f(\vec{x})$ in the regions the performance is not strong*.

Remark: Note that the shrinkage coefficient λ slows down the learning even further which in turn allows more (different) trees to learn the residual targets r_i.

In general, statistical learning methods that tend to learn slowly perform stronger. We will see a similar behavior in the context of neural networks in future weeks.

The above boosting algorithm we introduced above presents the essence of the approach. You should however note that there are several different variations of the boosting method in the machine learning literature. The most popular boosting algorithms include but are not limited to:

- **AdaBoost (Adaptive Boosting):** In the boosting algorithm mentioned above, each the effect of each tree updates the model and the residual target with the same strength. In adaptive boosting, however, weights are assigned to record and update the effect of each tree. In other words, the model learns to assign higher weights to those trees which make more significant improvements. At the end, the final model will be a weighted sum of equation (4).
- **Gradient Boosting:** Gradient boosting is similar to adaptive boosting where weights are assigned to update the model and the residual targets. However, in gradient boosting a differentiable loss function is defined, and the weights are obtained by optimizing the loss function through a gradient descent approach (We will hear more about gradient descent in neural networks lectures in future!).
- **XGBoost (Extreme Gradient Boosting):** Extreme gradient boosting optimizes a differentiable loss function just as the gradient boosting. However, instead of using a gradient descent approach to minimize the loss function, another approach that is based on a second order Taylor approximation is used. Extreme gradient boosting has gained much popularity and attention recently as the algorithm of choice for many winning teams of machine learning competitions!

2 Coding for Ensemble Learning

In here, we present three examples to illustrate the implementation of the ensemble learning techniques in practice. In the first two examples, we construct two toy datasets, one for classification, and one for regression. In the last example, we will reconsider the .....

2.1 A Toy Dataset for Classification

In this example, we construct a toy dataset using make_classification command of the scikit-learn library. We construct a dataset consisting 1000 samples with 10 continuous features, and a categorical target variable with 3 classes.

[1]:
```
# Importing basic libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

[2]:
```

```

```
# Constructing a dataset consisting 1000 samples with 10 features and 3 classes for␣
 ↪the target

from sklearn.datasets import make_classification

X, y = make_classification(n_samples=1000, n_features=10, n_informative=4,␣
 ↪n_redundant=0, n_classes=3,
                           random_state=3, shuffle=True)
```

[3]:
```
# Breaking the data into train and test subsets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=3)
```

2.1.1 Decision Tree Classifier

[4]:
```
# Constructing tree classifier with no specified hyperparameters

from sklearn import tree

tr_clf = tree.DecisionTreeClassifier()
tr_clf.fit(X_train, y_train)
```

[4]:
```
DecisionTreeClassifier()
```

[5]:
```
# Finding the predictions of tree classifier for train and test subsets

train_y_pred = tr_clf.predict(X_train)
test_y_pred = tr_clf.predict(X_test)
```

[6]:
```
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report

train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('Decision Tree Train Classification Report: \n\n', train_report,'\n\n')
print('Decision Tree Test Classification Report: \n\n', test_report)
```

```
Decision Tree Train Classification Report:

              precision    recall  f1-score   support

           0       1.00      1.00      1.00       258
           1       1.00      1.00      1.00       246
           2       1.00      1.00      1.00       246
```

```
    accuracy                           1.00       750
   macro avg       1.00      1.00      1.00       750
weighted avg       1.00      1.00      1.00       750


Decision Tree Test Classification Report:

              precision    recall  f1-score   support

           0       0.65      0.83      0.73        77
           1       0.80      0.68      0.73        87
           2       0.82      0.73      0.77        86

    accuracy                           0.74       250
   macro avg       0.75      0.75      0.74       250
weighted avg       0.76      0.74      0.74       250
```

Question: How do you evaluate the performance of the decision tree classifier? What potential problems do you observe?

2.1.2 Bagged Decision Trees Classifier

In order to reduce the high variance problem in the performance of the above decision tree classifier, we apply bagging on decision tree classifiers.

[7]:
```
# Constructing a bagged-tree classifier

from sklearn.ensemble import BaggingClassifier

bag_clf = BaggingClassifier(base_estimator=tree.DecisionTreeClassifier(),␣
 ↪n_estimators=18, random_state=3)
bag_clf.fit(X_train, y_train)
```

[7]:
```
BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=18,
                  random_state=3)
```

[8]:
```
# Finding the predictions of the bagged-tree classifier for train and test subsets

train_y_pred = bag_clf.predict(X_train)
test_y_pred = bag_clf.predict(X_test)
```

[9]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('Bagged Trees Classifier Train Classification Report: \n\n', train_report,'\n\n')
print('Bagged Trees Classifier Test Classification Report: \n\n', test_report)
```

```
Bagged Trees Classifier Train Classification Report:

              precision    recall  f1-score   support

           0       1.00      1.00      1.00       258
           1       1.00      1.00      1.00       246
           2       1.00      1.00      1.00       246

    accuracy                           1.00       750
   macro avg       1.00      1.00      1.00       750
weighted avg       1.00      1.00      1.00       750


Bagged Trees Classifier Test Classification Report:

              precision    recall  f1-score   support

           0       0.80      0.91      0.85        77
           1       0.86      0.72      0.79        87
           2       0.88      0.92      0.90        86

    accuracy                           0.85       250
   macro avg       0.85      0.85      0.85       250
weighted avg       0.85      0.85      0.85       250
```

Question: How do you evaluate the performance of the bagged-tree classifier? Did the bagging method help reduce the high variance?

2.1.3 Random Forest Classifier

As an another alternative, we now train a random forest classifier.

[10]:
```
# Constructing a random forest classifier

from sklearn.ensemble import RandomForestClassifier

rf_clf = RandomForestClassifier(max_depth=2, random_state=3)
rf_clf.fit(X_train, y_train)
```

[10]:
```
RandomForestClassifier(max_depth=2, random_state=3)
```

[11]:
```
# Finding the predictions of random forest classifier for train and test subsets

train_y_pred = rf_clf.predict(X_train)
test_y_pred = rf_clf.predict(X_test)
```

[12]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data
```

```
print('Random Forest Classifier Train Classification Report: \n\n',␣
 ↪train_report,'\n\n')
print('Random Forest Classifier Test Classification Report: \n\n', test_report)
```

```
Random Forest Classifier Train Classification Report:

               precision    recall  f1-score   support

           0       0.73      0.91      0.81       258
           1       0.86      0.50      0.63       246
           2       0.73      0.84      0.78       246

    accuracy                           0.75       750
   macro avg       0.77      0.75      0.74       750
weighted avg       0.77      0.75      0.74       750



Random Forest Classifier Test Classification Report:

               precision    recall  f1-score   support

           0       0.62      0.90      0.73        77
           1       0.79      0.39      0.52        87
           2       0.71      0.78      0.74        86

    accuracy                           0.68       250
   macro avg       0.70      0.69      0.66       250
weighted avg       0.71      0.68      0.66       250
```

Question: How do you assess the performance of the random forest classifier constructed above?

Let us now take the advantage of some of the hyperparameters of the random forest and construct a better random forest classifier.

[13]:
```
# Constructing a new random forest classifier

rf_clf = RandomForestClassifier(n_estimators=3000, max_depth=6, max_leaf_nodes=8,␣
 ↪random_state=3)
rf_clf.fit(X_train, y_train)
```

[13]:
```
RandomForestClassifier(max_depth=6, max_leaf_nodes=8, n_estimators=3000,
                       random_state=3)
```

[14]:
```
# Finding the predictions of the new random forest classifier for train and test␣
 ↪subsets

train_y_pred = rf_clf.predict(X_train)
test_y_pred = rf_clf.predict(X_test)
```

[15]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test accuracy
```

```
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)      # Generate␣
 ↪classification report for test data

print('Random Forest Classifier Train Classification Report: \n\n',␣
 ↪train_report,'\n\n')
print('Random Forest Classifier Test Classification Report: \n\n', test_report)
```

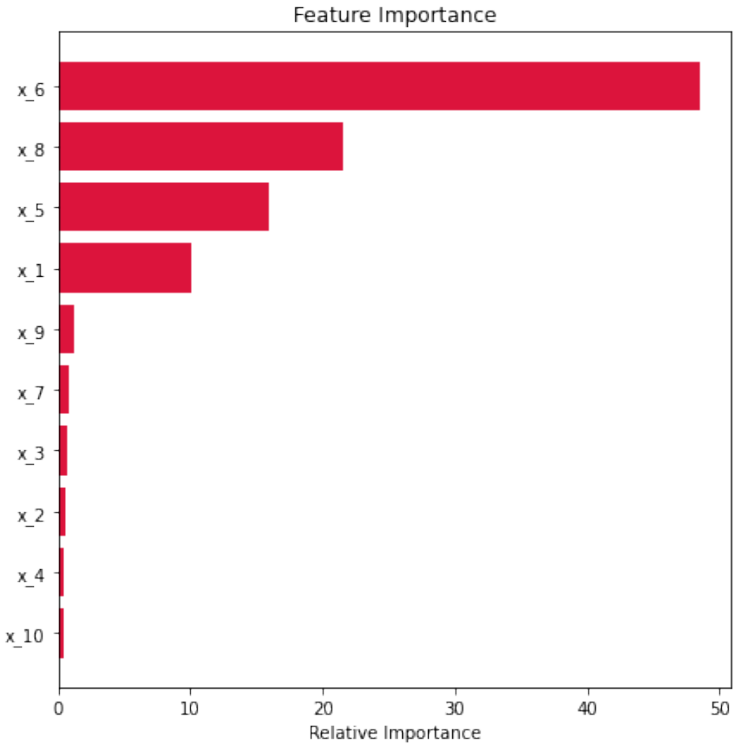
```
Random Forest Classifier Train Classification Report:

              precision    recall  f1-score   support

           0       0.82      0.89      0.85       258
           1       0.87      0.61      0.72       246
           2       0.76      0.91      0.83       246

    accuracy                           0.81       750
   macro avg       0.82      0.81      0.80       750
weighted avg       0.82      0.81      0.80       750



Random Forest Classifier Test Classification Report:

              precision    recall  f1-score   support

           0       0.74      0.88      0.80        77
           1       0.84      0.54      0.66        87
           2       0.74      0.87      0.80        86

    accuracy                           0.76       250
   macro avg       0.77      0.77      0.75       250
weighted avg       0.77      0.76      0.75       250
```

Question: How do you assess the performance of the new random forest classifier?

We can now calculate the feature importance for the dataset exactly in the same manner as trees.

[16]:
```
# Calculating the feature importance

feature_importance = rf_clf.feature_importances_
feature_importance = 100.0*(feature_importance/np.sum(feature_importance))

# Displaying the relative feature importance by a horizontal bar chart

sorted_idx = np.argsort(feature_importance)
pos=np.arange(sorted_idx.shape[0])+0.5
plt.figure(figsize=(7, 7))

plt.title("Feature Importance")
plt.xlabel('Relative Importance')
plt.barh(pos, feature_importance[sorted_idx], color='crimson', align="center")
```

```
plt.yticks(pos, np.array(['x_%d' %(i+1) for i in range(10)])[sorted_idx])
plt.show()
```



Question: What are the most significant features? How many of them do you have? Are you surprised by this number?

2.1.4 Bagged SVC Classifier

Here we show how one can apply the bagging method to other weak learners (other than decisio trees). In this section, we choose our weak learners to be support vector classifiers.

[17]:
```
# We start by training an SVC

from sklearn.svm import SVC

svc_clf = SVC()
svc_clf.fit(X_train, y_train)
```

[17]:
```
SVC()
```

[18]:
```
# Finding the predictions of SVC classifier for train and test subsets

train_y_pred = svc_clf.predict(X_train)
test_y_pred = svc_clf.predict(X_test)
```

[19]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('SVC Classifier Train Classification Report: \n\n', train_report,'\n\n')
print('SVC Classifier Test Classification Report: \n\n', test_report)
```

```
SVC Classifier Train Classification Report:

              precision    recall  f1-score   support

           0       0.91      0.86      0.88       258
           1       0.90      0.87      0.89       246
           2       0.91      0.99      0.95       246

    accuracy                           0.91       750
   macro avg       0.91      0.91      0.91       750
weighted avg       0.91      0.91      0.91       750



SVC Classifier Test Classification Report:

              precision    recall  f1-score   support

           0       0.81      0.84      0.83        77
           1       0.90      0.74      0.81        87
           2       0.84      0.97      0.90        86

    accuracy                           0.85       250
   macro avg       0.85      0.85      0.85       250
weighted avg       0.85      0.85      0.85       250
```

Question: How do you evaluate the performance of the SVC classifier?

Now, let us apply bagging method to SVC classifier.

[20]:
```
# Constructing a bagged-SVC classifier

bag_clf = BaggingClassifier(base_estimator=SVC(), n_estimators=30, random_state=3)
bag_clf.fit(X_train, y_train)
```

[20]:
```
BaggingClassifier(base_estimator=SVC(), n_estimators=30, random_state=3)
```

[21]:
```
# Finding the predictions of the bagged-SVC classifier for train and test subsets

train_y_pred = bag_clf.predict(X_train)
test_y_pred = bag_clf.predict(X_test)
```

[22]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('Bagged SVC Classifier Train Classification Report: \n\n', train_report,'\n\n')
print('Bagged SVC Classifier Test Classification Report: \n\n', test_report)
```

```
Bagged SVC Classifier Train Classification Report:

              precision    recall  f1-score   support

           0       0.89      0.86      0.88       258
           1       0.89      0.86      0.88       246
           2       0.91      0.98      0.94       246

    accuracy                           0.90       750
   macro avg       0.90      0.90      0.90       750
weighted avg       0.90      0.90      0.90       750



Bagged SVC Classifier Test Classification Report:

              precision    recall  f1-score   support

           0       0.80      0.84      0.82        77
           1       0.90      0.71      0.79        87
           2       0.83      0.97      0.89        86

    accuracy                           0.84       250
   macro avg       0.84      0.84      0.84       250
weighted avg       0.85      0.84      0.84       250
```

Question: How do you assess the performance of the bagged-SVC classifier?

Question: From the fact that the high variance did not reduce considerably, what do you conclude about the weak learners (*i.e.* SVC classifiers)?

2.1.5 AdaBoost Classifier

Let us see how AdaBoost is implemented in the current case.

[23]:
```
# Importing AdaBoostClassifier from ensemble module

from sklearn.ensemble import AdaBoostClassifier

# Instantiating the AdaBoostClassifier with 500 sequential trees

adab_clf = AdaBoostClassifier(base_estimator=tree.
 ↪DecisionTreeClassifier(max_leaf_nodes=10),
```

```
                                n_estimators=500, learning_rate=1.5, random_state=0)
adab_clf.fit(X_train, y_train)   # Fitting the train data
```

[23]:
```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_leaf_nodes=10),
                   learning_rate=1.5, n_estimators=500, random_state=0)
```

[24]:
```
# Finding the predictions of the adaboost classifier for train and test subsets

train_y_pred = adab_clf.predict(X_train)
test_y_pred = adab_clf.predict(X_test)
```

[25]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('AdaBoost Classifier Train Classification Report: \n\n', train_report,'\n\n')
print('AdaBoost Classifier Test Classification Report: \n\n', test_report)
```

```
AdaBoost Classifier Train Classification Report:

              precision    recall  f1-score   support

           0       0.99      0.96      0.98       258
           1       0.93      0.98      0.96       246
           2       0.99      0.97      0.98       246

    accuracy                           0.97       750
   macro avg       0.97      0.97      0.97       750
weighted avg       0.97      0.97      0.97       750



AdaBoost Classifier Test Classification Report:

              precision    recall  f1-score   support

           0       0.78      0.79      0.79        77
           1       0.77      0.68      0.72        87
           2       0.81      0.90      0.85        86

    accuracy                           0.79       250
   macro avg       0.79      0.79      0.79       250
weighted avg       0.79      0.79      0.79       250
```

2.2 A Toy Dataset for Regression

In this example, we construct a toy dataset using make_regression command of the scikit-learn library. We construct a dataset consisting 10000 samples with 20 continuous features, and a continuous target variable which itself is a vector in $\mathbb{R}^8$.

[26]:
```
# Constructing a dataset consisting 10000 samples with 20 features and a target in R^8

from sklearn.datasets import make_regression

X, y = make_regression(n_samples=10000, n_features=20, n_informative=5, n_targets=8,
                       random_state=3, shuffle=True)
```

[27]:
```
# Breaking the data into train and test subsets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=3)
```

2.2.1 Decision Tree Regressor

[28]:
```
# Constructing a tree regressor with no specified hyperparameters

from sklearn import tree

tr_reg = tree.DecisionTreeRegressor()
tr_reg.fit(X_train, y_train)
```

[28]:
```
DecisionTreeRegressor()
```

[29]:
```
# Finding the predictions of the tree regressor for train and test subsets

train_y_pred = tr_reg.predict(X_train)
test_y_pred = tr_reg.predict(X_test)
```

[30]:
```
r2_train_score = tr_reg.score(X_train, y_train)    # Calculating R^2 score for train
r2_test_score = tr_reg.score(X_test, y_test)       # Calculating R^2 score for test

print('R^2 score for train dataset = ', round(r2_train_score, 4), '\n')
print('R^2 score for test dataset = ', round(r2_test_score, 4), '\n')
```

```
R^2 score for train dataset =  1.0

R^2 score for test dataset =  0.8063
```

Question: How do you evaluate the performance of the tree regressor? What potential problems do you observe?

2.2.2 Bagged Decision Trees Regressor

Now, to reduce the high variance, we apply the bagging method to decision tree regressors.

[31]:
```
# Importing BaggingRegressor from ensemble module

from sklearn.ensemble import BaggingRegressor

# Instantiating the bagged-tree regressor
```

```
bag_reg = BaggingRegressor(base_estimator=tree.DecisionTreeRegressor(),␣
 ↪n_estimators=100, random_state=3)
bag_reg.fit(X_train, y_train)  # Fitting the train data
```

[31]:
```
BaggingRegressor(base_estimator=DecisionTreeRegressor(), n_estimators=100,
                 random_state=3)
```

[32]:
```
# Finding the predictions of bagged-tree regressor for train and test subsets

train_y_pred = bag_reg.predict(X_train)
test_y_pred = bag_reg.predict(X_test)
```

[33]:
```
r2_train_score = bag_reg.score(X_train, y_train)   # Calculating R^2 score for train
r2_test_score = bag_reg.score(X_test, y_test)      # Calculating R^2 score for test

print('R^2 score for train dataset = ', round(r2_train_score, 4), '\n')
print('R^2 score for test dataset = ', round(r2_test_score, 4), '\n')
```

```
R^2 score for train dataset =  0.9919

R^2 score for test dataset =  0.9476
```

Question: How do interpret the above result?

2.2.3 Random Forest Regressor

We now try a random forest regressor on the dataset and compare the results with the bagging method.

[34]:
```
# Constructing a random forest regressor

from sklearn.ensemble import RandomForestRegressor

rf_reg = RandomForestRegressor(n_estimators=1000, random_state=3)  # Instantiating the␣
 ↪forest
rf_reg.fit(X_train, y_train)                                       # Fitting the train␣
 ↪data
```

[34]:
```
RandomForestRegressor(n_estimators=1000, random_state=3)
```

[35]:
```
# Finding the predictions of the random forest regressor for train and test subsets

train_y_pred = rf_reg.predict(X_train)
test_y_pred = rf_reg.predict(X_test)
```

[36]:
```
r2_train_score = rf_reg.score(X_train, y_train)   # Calculating R^2 score for train
r2_test_score = rf_reg.score(X_test, y_test)      # Calculating R^2 score for test

print('R^2 score for train dataset = ', round(r2_train_score, 4), '\n')
print('R^2 score for test dataset = ', round(r2_test_score, 4), '\n')
```

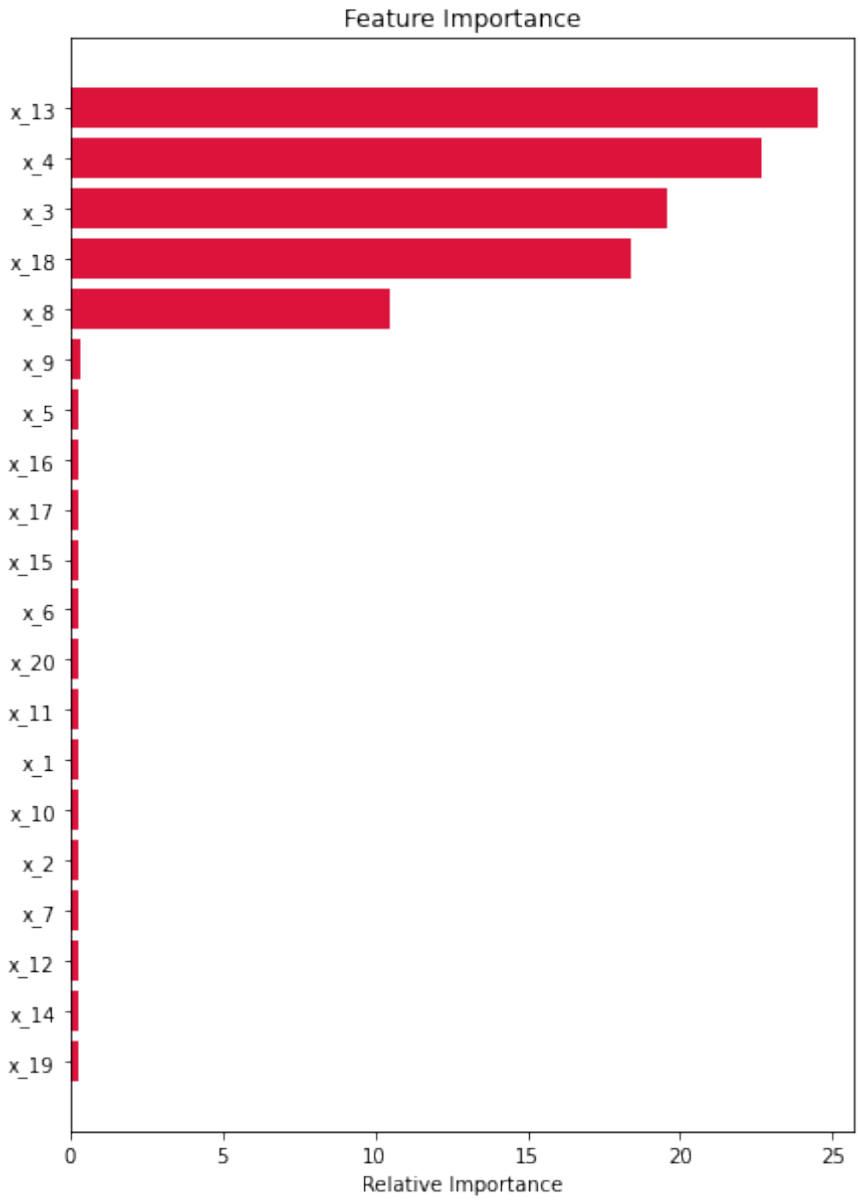
```
R^2 score for train dataset =  0.9927
```

```
R^2 score for test dataset =  0.949
```

Question: How do you compare the above result with the results of the bagged-tree regressor?

We can again calculate the feature importance for the dataset to find the most significant features for the considered target.

[37]:
```
# Calculating the feature importance

feature_importance = rf_reg.feature_importances_
feature_importance = 100.0*(feature_importance/np.sum(feature_importance))

# Displaying the relative feature importance by a horizontal bar chart

sorted_idx = np.argsort(feature_importance)
pos=np.arange(sorted_idx.shape[0])+0.5
plt.figure(figsize=(7, 10))

plt.title("Feature Importance")
plt.xlabel('Relative Importance')
plt.barh(pos, feature_importance[sorted_idx], color='crimson', align="center")
plt.yticks(pos, np.array(['x_%d' %(i+1) for i in range(20)])[sorted_idx])
plt.show()
```



Question: What are the most significant features? How many of them do you have? Are you surprised by this number?