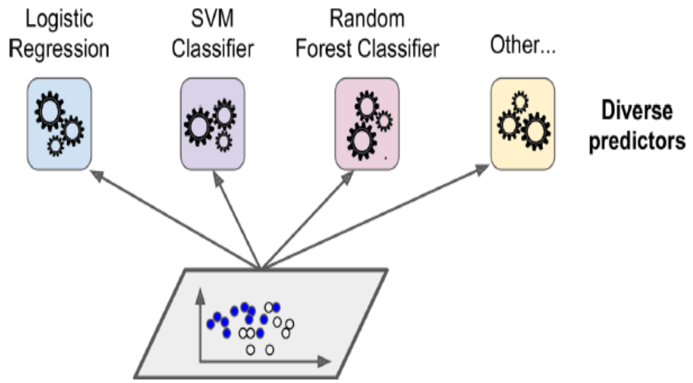
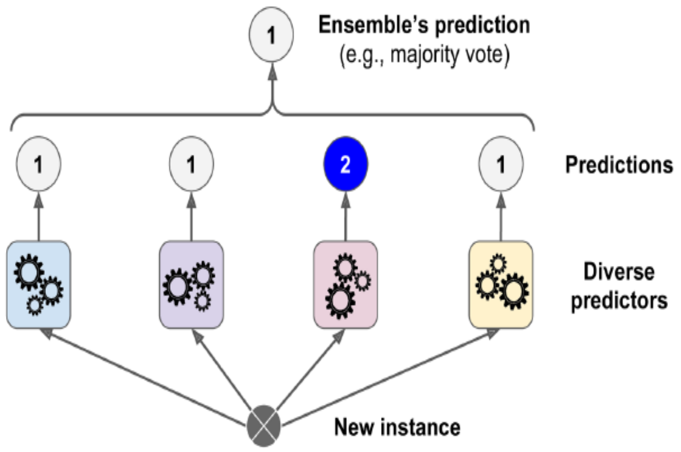
# Ensemble Learning and Random Forest

# Ensemble Learning



- If there are many classifiers, each having an accuracy of about 80%.
- Then a better classifier can be built by aggregating their predictions
- Aggregation works by predicting a class that gets most votes.
- This is called hard voting classifier.

# Ensemble Learning

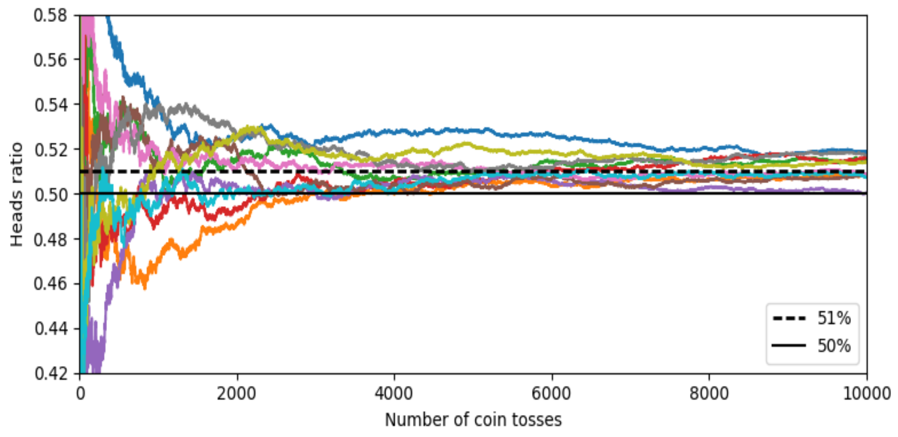


- Some what surprizingly, the voting classifier achieves higher accuracy than the best classifier in the ensemble.
- Even if each classifier is a weak learner (slightly better than random guessing), the ensemble can still be a strong learner, provided they are sufficient number of weak learners and they are sufficiently diverse.

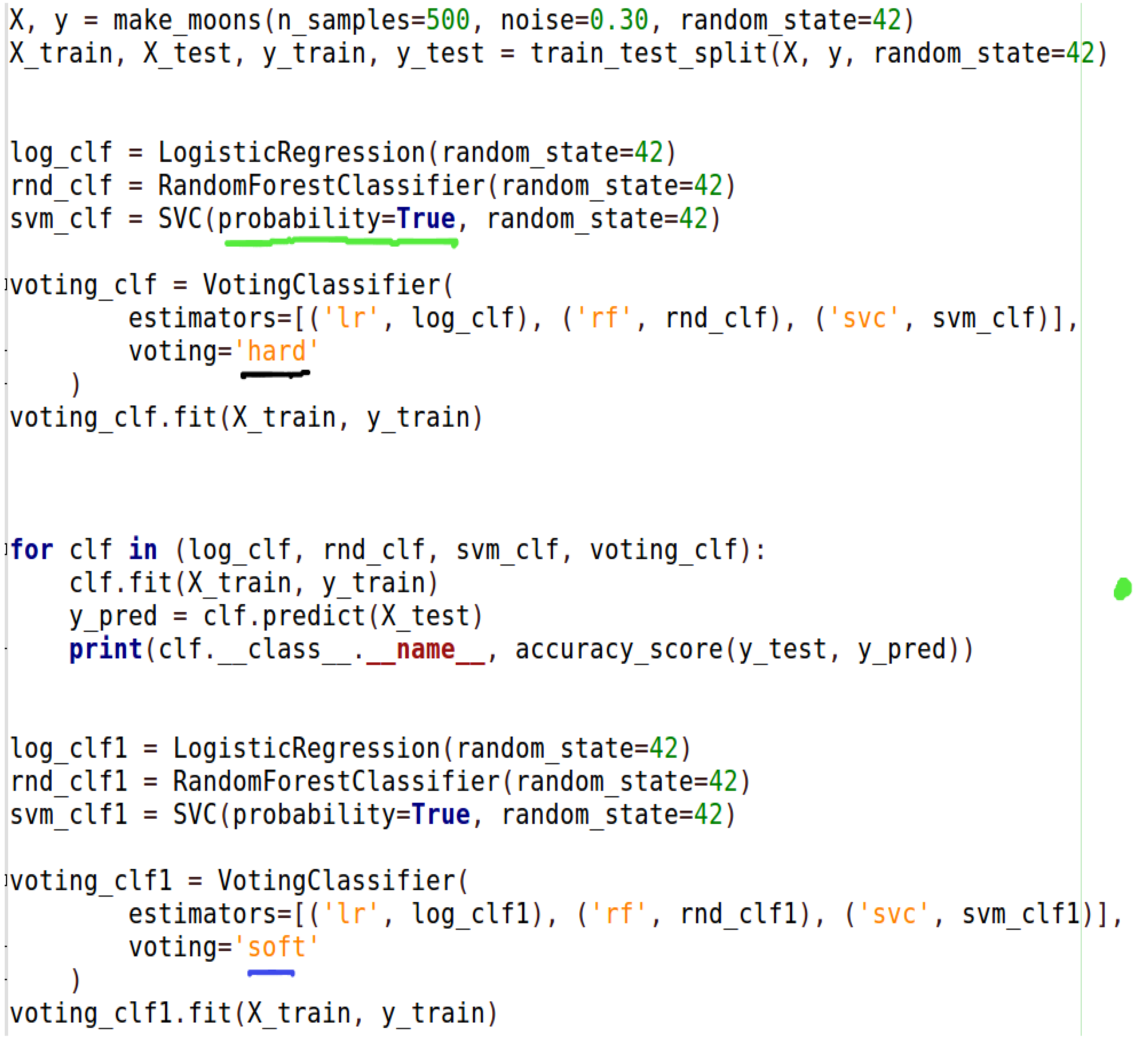
# Ensemble Learning

```
heads_proba = 0.51
coin_tosses = (rnd.rand(10000, 10) < heads_proba).astype(np.int32)
cumulative_heads_ratio = np.cumsum(coin_tosses, axis=0) / np.arange(1, 10001).reshape(-1,1)

plt.figure(figsize=(8,3.5))
plt.plot(cumulative_heads_ratio)
plt.plot([0, 10000], [0.51, 0.51], "k--", linewidth=2, label="51%")
plt.plot([0, 10000], [0.5, 0.5], "k-", label="50%")
plt.xlabel("Number of coin tosses")
plt.ylabel("Heads ratio")
plt.legend(loc="lower right")
plt.axis([0, 10000, 0.42, 0.58])
fl.save_fig("law_of_large_numbers_plot")
plt.show()
```

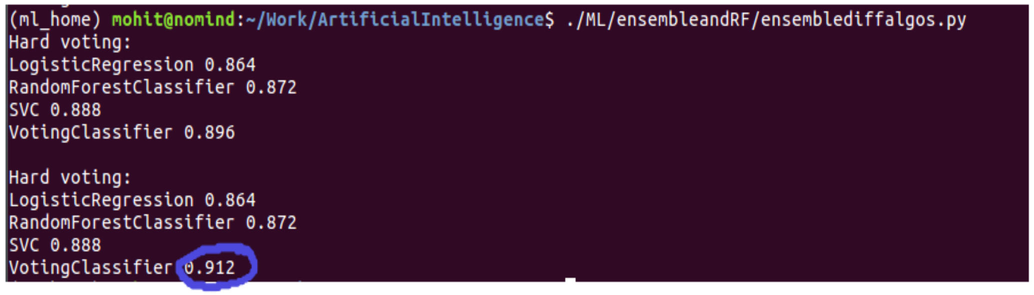
- Theory of Large numbers.
- 1000 % classifeirs of 51% accuracy (barely better than random guessing), with ensemble voting, it usually reaches an accuracy of 75%.
- But they must be perfectly independant and make uncorrelated errors.



# Ensemble Learning



```python
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)


log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(random_state=42)
svm_clf = SVC(probability=True, random_state=42)

voting_clf = VotingClassifier(
        estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
        voting='hard'
    )
voting_clf.fit(X_train, y_train)



for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))


log_clf1 = LogisticRegression(random_state=42)
rnd_clf1 = RandomForestClassifier(random_state=42)
svm_clf1 = SVC(probability=True, random_state=42)

voting_clf1 = VotingClassifier(
        estimators=[('lr', log_clf1), ('rf', rnd_clf1), ('svc', svm_clf1)],
        voting='soft'
    )
voting_clf1.fit(X_train, y_train)
```

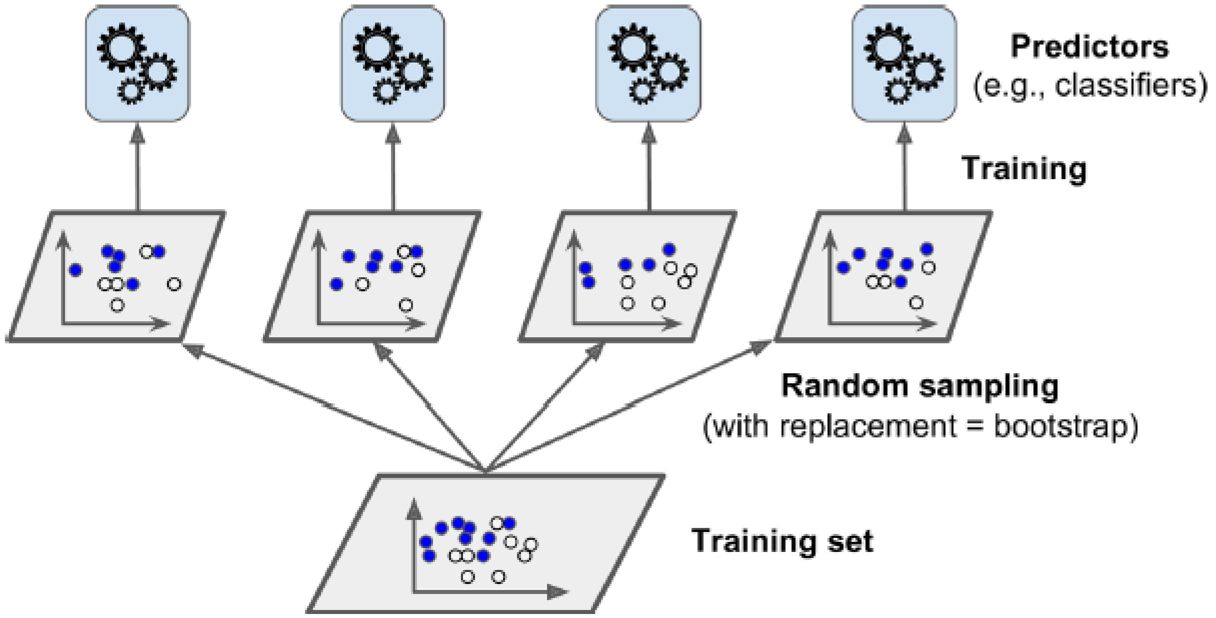


```
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./ML/ensembleandRF/ensemblediffalgos.py
Hard voting:
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896

Hard voting:
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.912
```

- Hard voting – majority vote
- Soft voting – pick the vote with highest confidence (probability)
- SVC by default does not output probability.

# Ensemble Learning:Bagging and Pasting

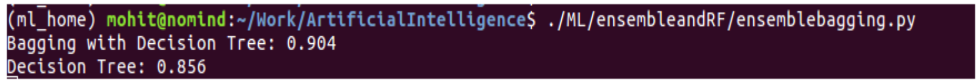
- Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set.
- When sampling is performed with replacement, this method is called bagging (short for bootstrap aggregating).
- When sampling is performed without replacement, it is called pasting.



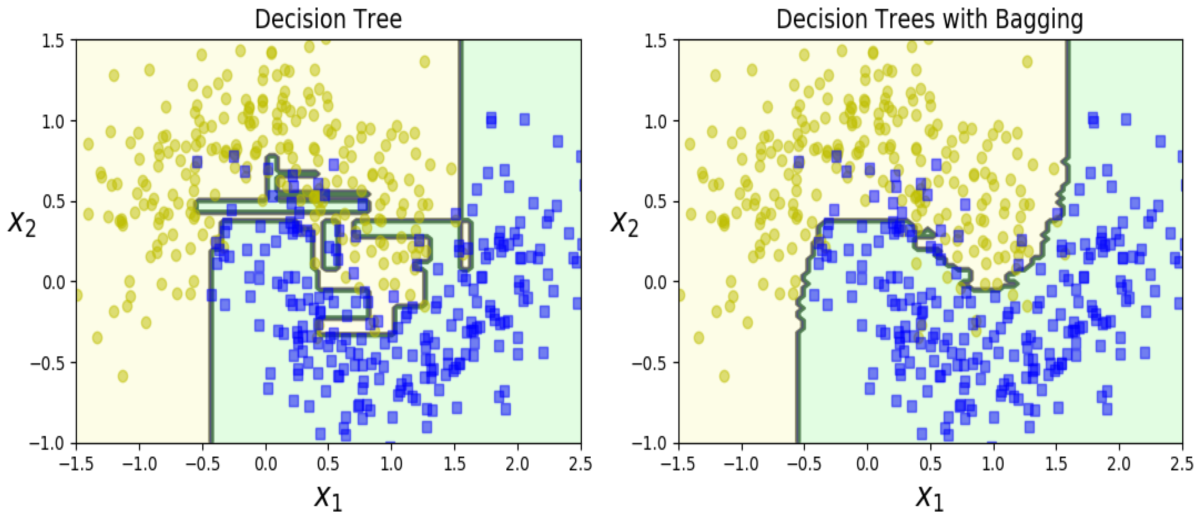
# Ensemble Learning:Bagging and Pasting

```
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

bag_clf = BaggingClassifier(
        DecisionTreeClassifier(random_state=42), n_estimators=500,
        max_samples=100, bootstrap=True, n_jobs=-1, random_state=42
    )
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
print("Bagging with Decision Tree:",accuracy_score(y_test, y_pred))

tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print("Decision Tree:",accuracy_score(y_test, y_pred_tree))
```

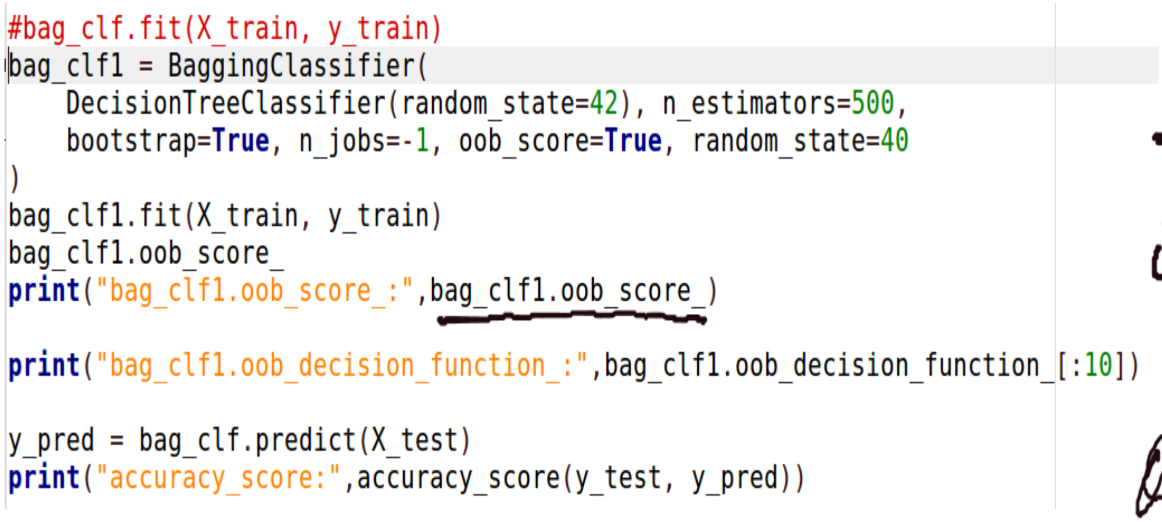
- Ensemble of 500 decision trees.
- Size of sample, example of bagging.
- For Pasting set bootstrap to false.
- number of CPUs to use (-1 is for all available cores)



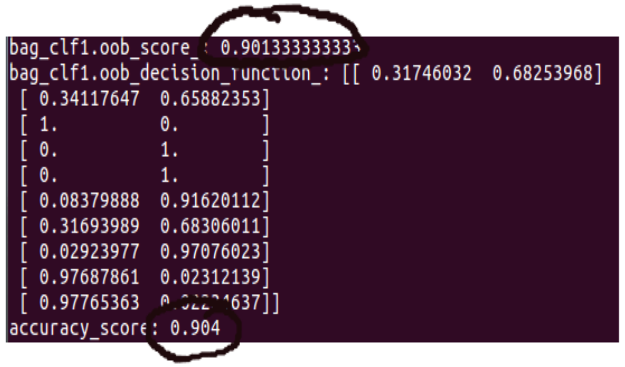
```
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./ML/ensembleandRF/ensemblebagging.py
Bagging with Decision Tree: 0.904
Decision Tree: 0.856
```



# Ensemble Learning:OOBs



```
#bag_clf.fit(X_train, y_train)
bag_clf1 = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    bootstrap=True, n_jobs=-1, oob_score=True, random_state=40
)
bag_clf1.fit(X_train, y_train)
bag_clf1.oob_score_
print("bag_clf1.oob_score_:",bag_clf1.oob_score_)

print("bag_clf1.oob_decision_function_:",bag_clf1.oob_decision_function_[:10])

y_pred = bag_clf.predict(X_test)
print("accuracy_score:",accuracy_score(y_test, y_pred))
```

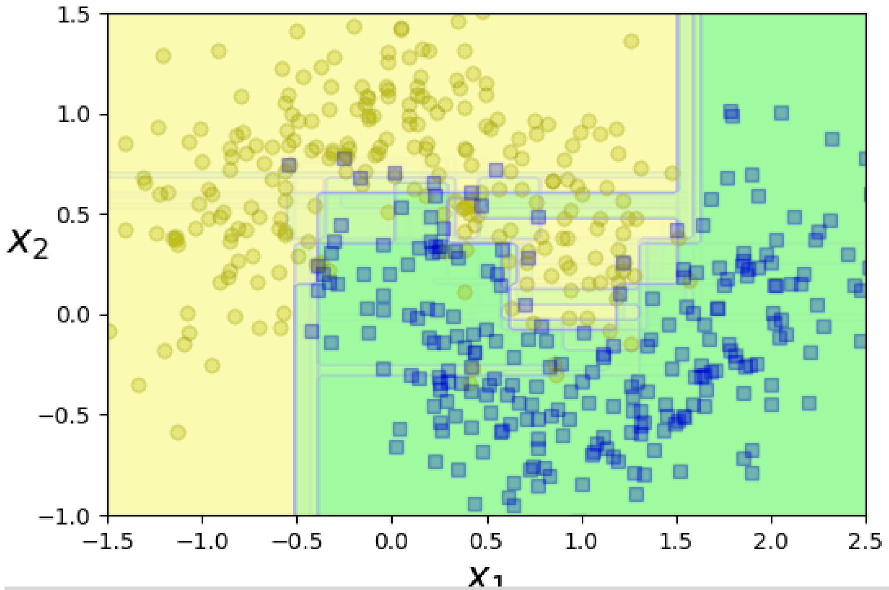


```
bag_clf1.oob_score_: 0.901333333333
bag_clf1.oob_decision_function_: [[ 0.31746032  0.68253968]
 [ 0.34117647  0.65882353]
 [ 1.          0.        ]
 [ 0.          1.        ]
 [ 0.          1.        ]
 [ 0.08379888  0.91620112]
 [ 0.31693989  0.68306011]
 [ 0.02923977  0.97076023]
 [ 0.97687861  0.02312139]
 [ 0.97765363  0.02234637]]
accuracy_score: 0.904
```

- Some percentage of sample is never seen by predictor.
- These are called OOB or Out of bag instances.
- Hence the predictor can be evaluated on these instances without the need for seperate evaluation or cross-validation.

# Ensemble Learning:OOBs

```
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)


plt.figure(figsize=(6, 4))

for i in range(15):
    tree_clf = DecisionTreeClassifier(max_leaf_nodes=16, random_state=42+i)
    indices_with_replacement = rnd.randint(0, len(X_train), len(X_train))
    tree_clf.fit(X[indices_with_replacement], y[indices_with_replacement])
    plot_decision_boundary(tree_clf, X, y, axes=[-1.5, 2.5, -1, 1.5], alpha=0.02, contour=False)

plt.show()
```



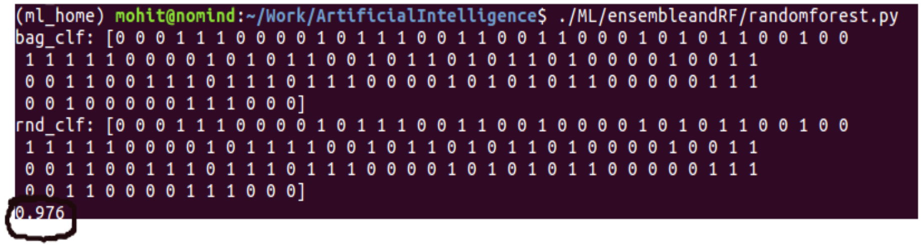
## Ensemble Learning:Random Patches and Random Subspaces

- The BaggingClassifier class supports sampling the features as well. This is controlled by two hyperparameters: max_features and bootstrap_features.
- They work the same way as max_samples and bootstrap, but for feature sampling instead of instance sampling.
- Thus, each predictor will be trained on a random subset of the input features.

# Random Forest

```python
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)


bag_clf = BaggingClassifier(
        DecisionTreeClassifier(splitter="random", max_leaf_nodes=16, random_state=42),
        n_estimators=500, max_samples=1.0, bootstrap=True,
        n_jobs=-1, random_state=42
    )
bag_clf.fit(X_train, y_train)
y_pred =bag_clf.predict(X_test)
print("bag_clf:",y_pred)

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)
y_pred_rf=rnd_clf.predict(X_test)
print("rnd_clf:",y_pred_rf)

print(np.sum(y_pred == y_pred_rf) / len(y_pred))
```

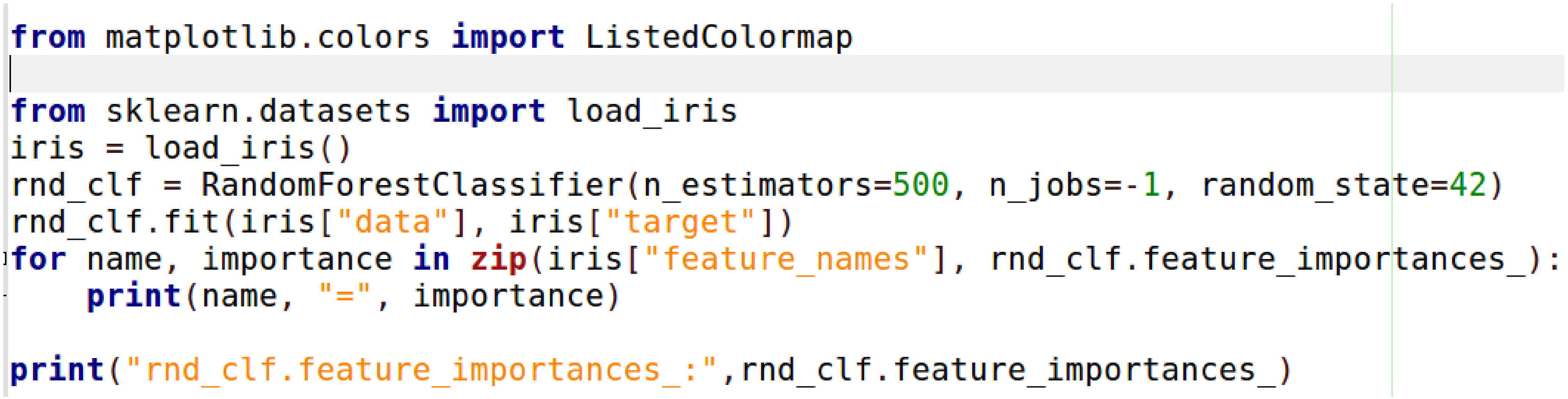
- These 2 are almost identical. With almost identical prediction.



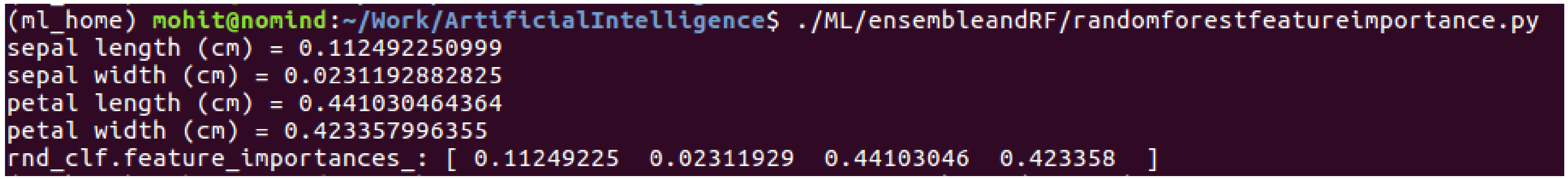
```
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./ML/ensembleandRF/randomforest.py
bag_clf: [0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 0 0 1 1 0 0 1 1 0 0 0 1 0 1 0 1 1 0 0 1 0 0
 1 1 1 1 1 0 0 0 0 1 0 1 0 1 1 0 0 1 0 1 1 0 1 0 1 1 0 1 0 0 0 0 1 0 0 1 1
 0 0 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0 1 0 1 0 1 0 1 1 0 0 0 0 0 1 1 1
 0 0 1 0 0 0 0 0 1 1 1 0 0 0]
rnd_clf: [0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 0 0 1 1 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 1 0 0
 1 1 1 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 1 0 1 0 1 1 0 1 0 0 0 0 1 0 0 1 1
 0 0 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0 1 0 1 0 1 0 1 1 0 0 0 0 0 1 1 1
 0 0 1 1 0 0 0 0 1 1 1 0 0 0]
0.976
```

- Random Forest introduces extra randomness when growing trees.
- Instead of searching for best feature, it searches for best feature from a random subset.
- The result is a greater tree diversity and trades higher bias for lower variance, for generally better results.

# Random Forest:Feature importance



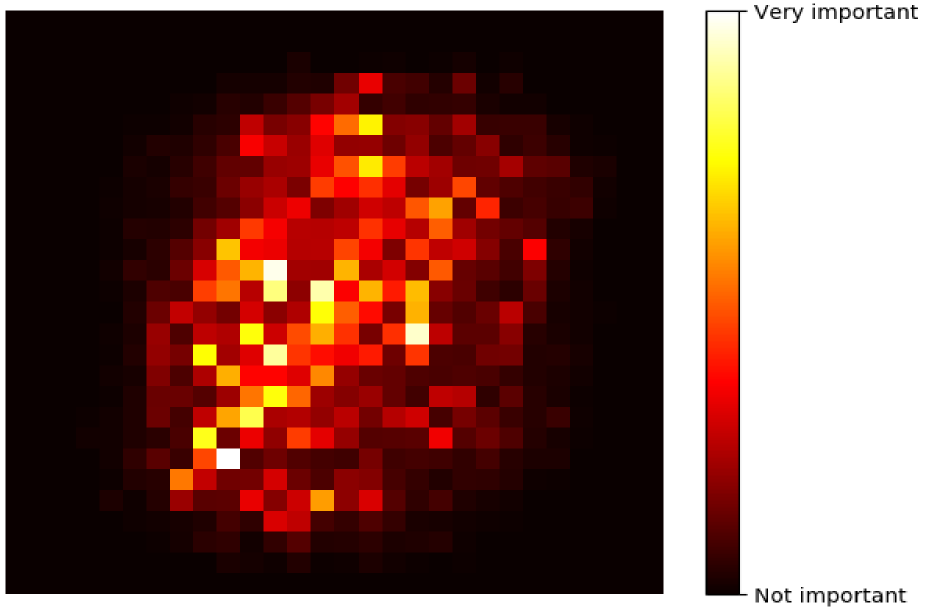
```python
from matplotlib.colors import ListedColormap

from sklearn.datasets import load_iris
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state=42)
rnd_clf.fit(iris["data"], iris["target"])
for name, importance in zip(iris["feature_names"], rnd_clf.feature_importances_):
    print(name, "=", importance)

print("rnd_clf.feature_importances_:",rnd_clf.feature_importances_)
```



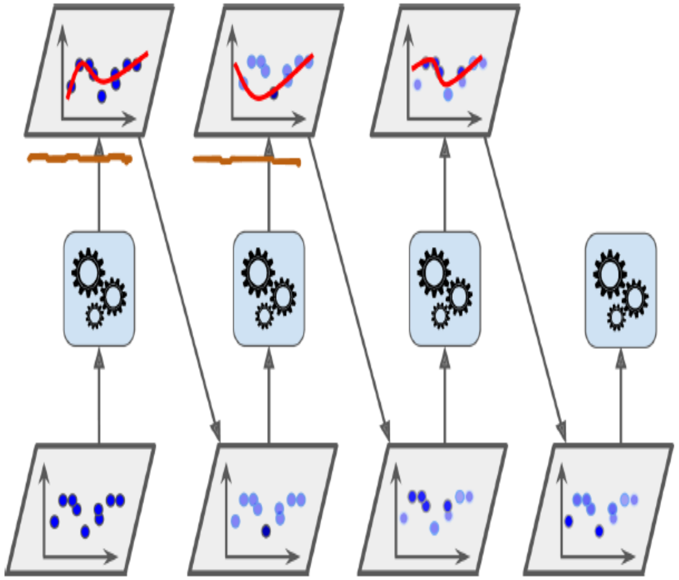
```
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./ML/ensembleandRF/randomforestfeatureimportance.py
sepal length (cm) = 0.112492250999
sepal width (cm) = 0.0231192882825
petal length (cm) = 0.441030464364
petal width (cm) = 0.423357996355
rnd_clf.feature_importances_: [ 0.11249225  0.02311929  0.44103046  0.423358   ]
```

# Random Forest:Feature importance

```python
mnist_raw=fl1.load_ml_data()
mnist = {
    "data": mnist_raw["data"].T,
    "target": mnist_raw["label"][0],
    "COL_NAMES": ["label", "data"],
    "DESCR": "mldata.org dataset: mnist-original",
}

X, y = mnist["data"], mnist["target"]
rnd_clf = RandomForestClassifier(random_state=42)
rnd_clf.fit(X,y)

print("rnd_clf:",rnd_clf)

plot_digit(rnd_clf.feature_importances_)

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

fl1.save_fig("mnist_feature_importance_plot")
plt.show()
```



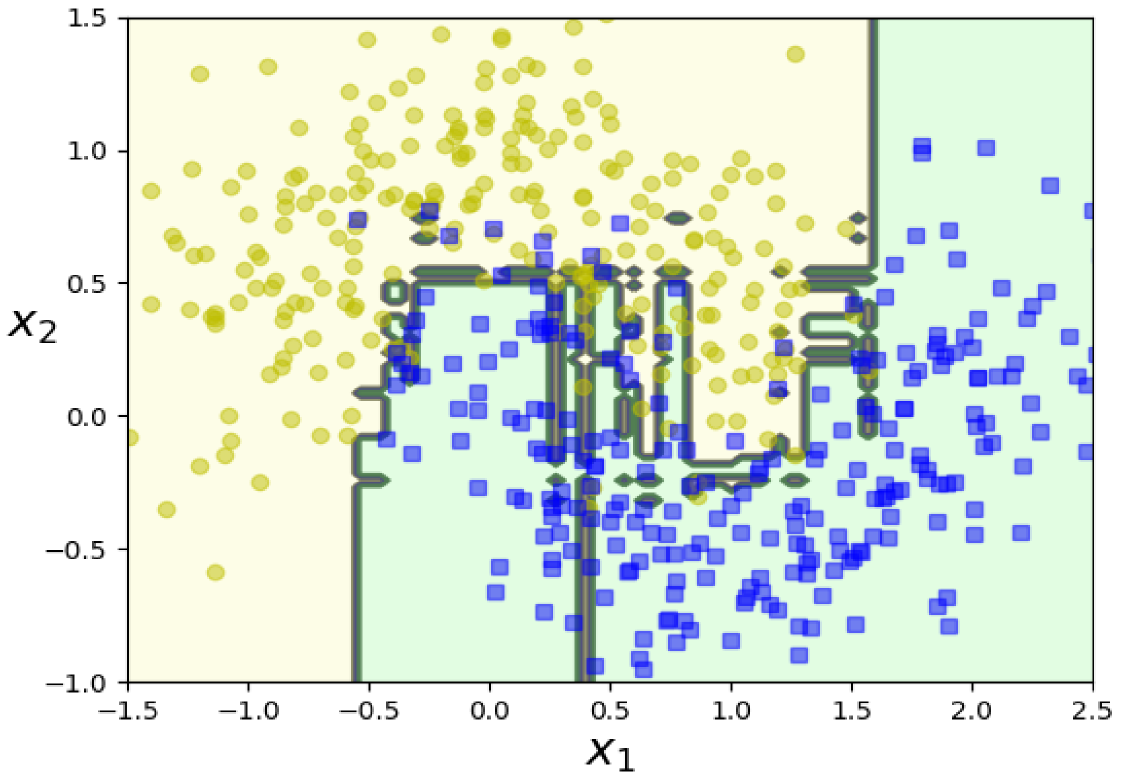
# Random Forest:Ada boosting



- General idea of most boosting methods is to train predictors sequentially.
- Each trying to correct the predecessor.
- Ada Boosting shows the relative weights of misclassified instances being increased subsequent classifiers.

# Random Forest:Ada boosting

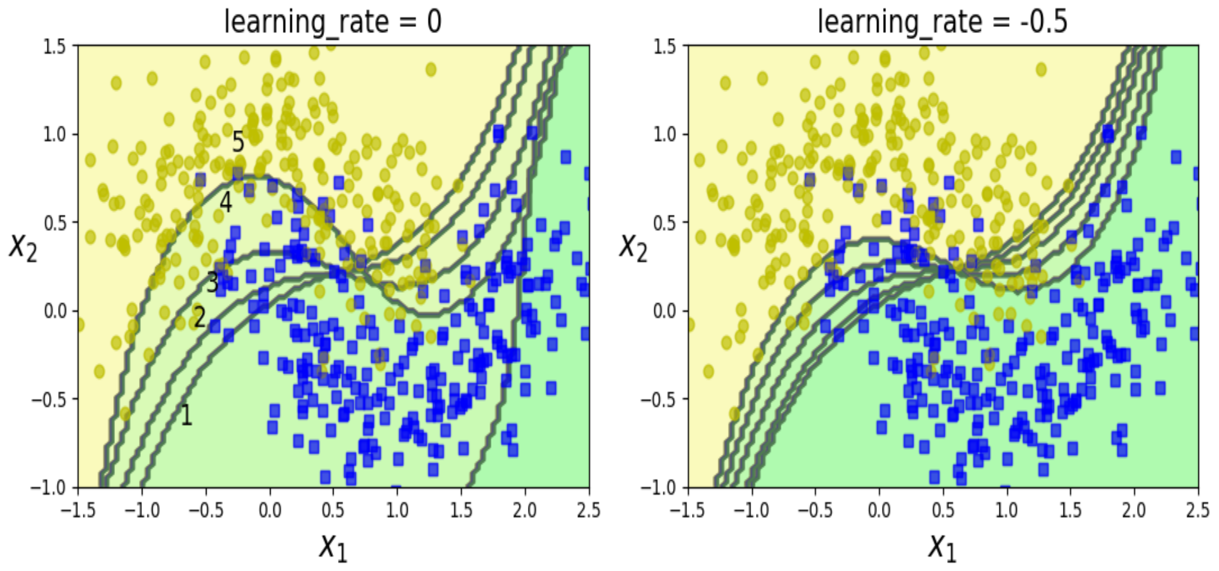
```
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

ada_clf = AdaBoostClassifier(
        DecisionTreeClassifier(max_depth=2), n_estimators=200,
        algorithm="SAMME.R", learning_rate=0.5, random_state=42
    )
ada_clf.fit(X_train, y_train)
plot_decision_boundary(ada_clf, X, y)
plt.show()

print(list(m for m in dir(ada_clf) if not m.startswith("_") and m.endswith("_")))
```



# Random Forest:Ada boosting

```
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

ada_clf = AdaBoostClassifier(
        DecisionTreeClassifier(max_depth=2), n_estimators=200,
        algorithm="SAMME.R", learning_rate=0.5, random_state=42
    )
ada_clf.fit(X_train, y_train)
plot_decision_boundary(ada_clf, X, y)
plt.show()

print(list(m for m in dir(ada_clf) if not m.startswith("_") and m.endswith("_")))
```

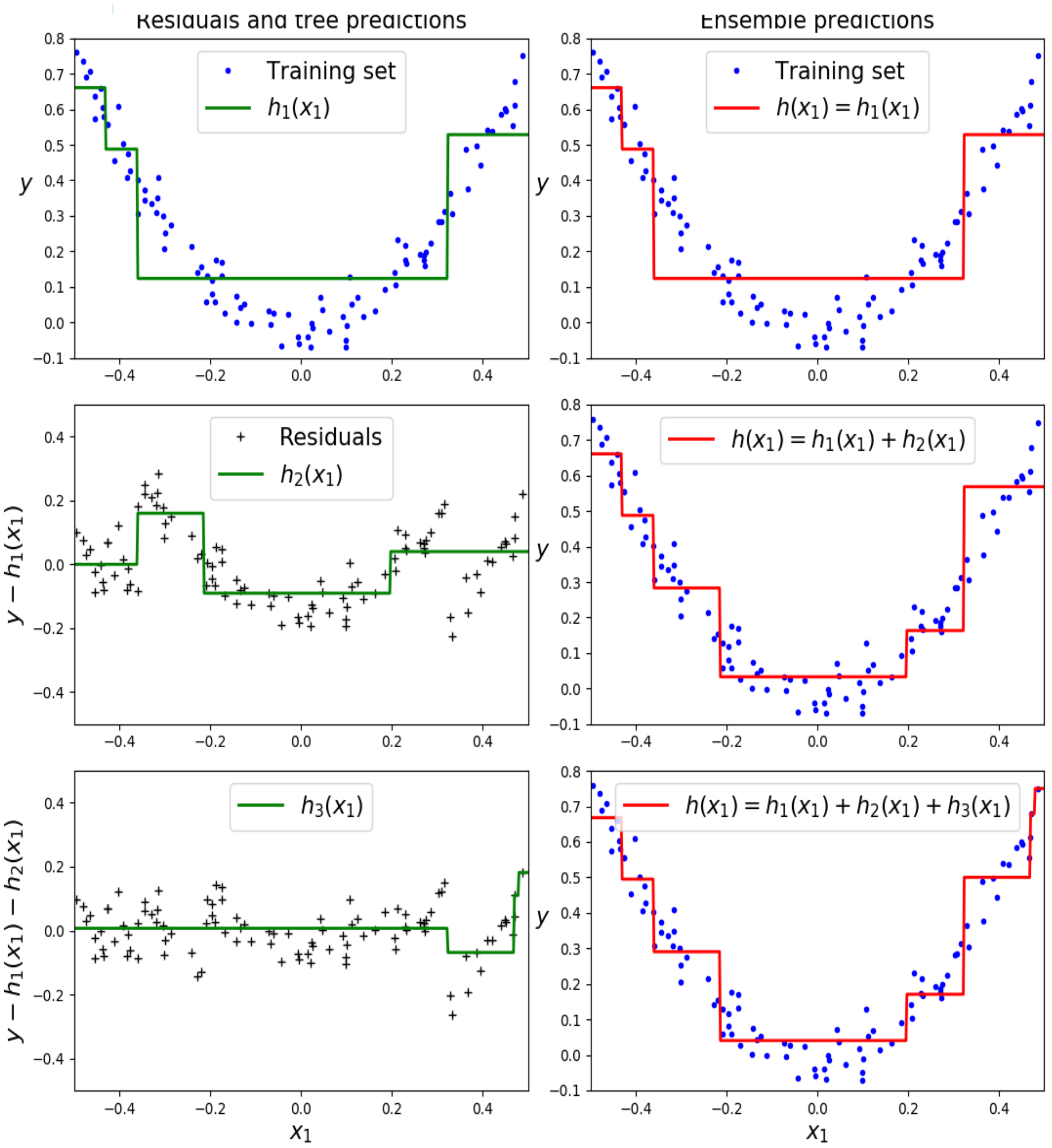
- Ada boosting sequence for SVM.



# Random Forest:Gradient boosting

```
rnd.seed(42)
X = rnd.rand(100, 1) - 0.5
y = 3*X[:, 0]**2 + 0.05 * rnd.randn(100)

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)

y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg2.fit(X, y2)

y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg3.fit(X, y3)

X_new = np.array([[0.8]])
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_r
print(y_pred)
```

- Just like ada boosting, it work by sequentially adding predictors.
- Instead of boosting weights at every iteration, the new predictors try to fit residual errors.

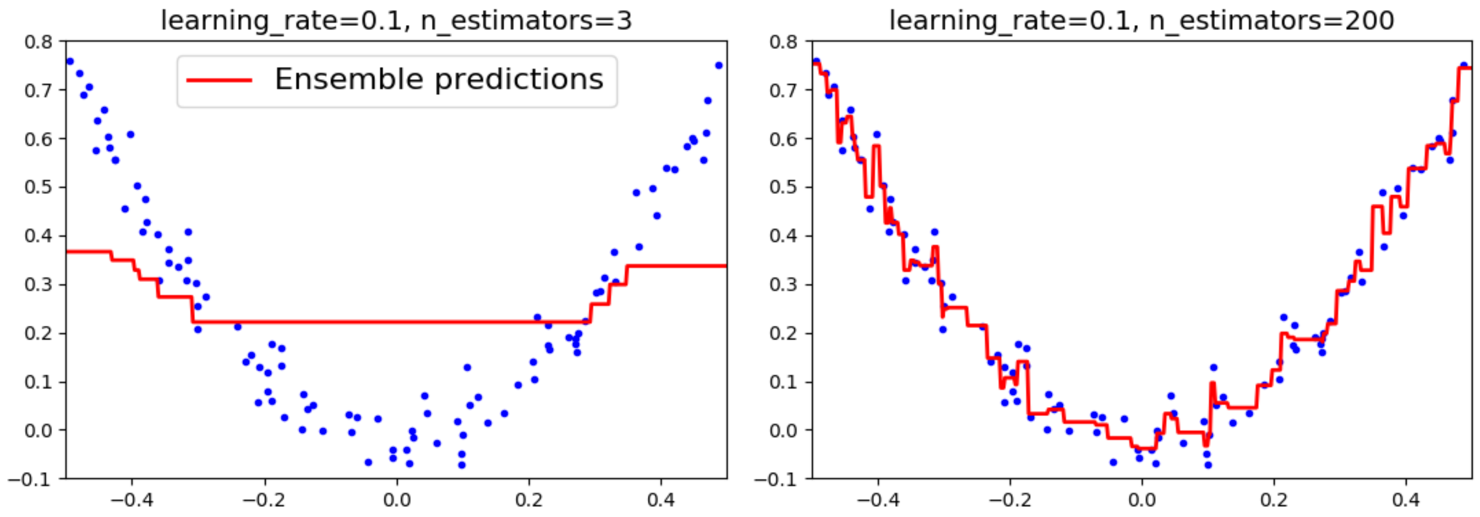


## Random Forest:Gradient boosting

- A simpler way to train GBRT ensembles is to use Scikit-Learn’s GradientBoostingRegressor class.
- Much like the RandomForestRegressor class, it has hyperparameters to control the growth of Decision Trees (e.g., max_depth, min_samples_leaf, and so on), as well as hyperparameters to control the ensemble training, such as the number of trees (n_estimators).

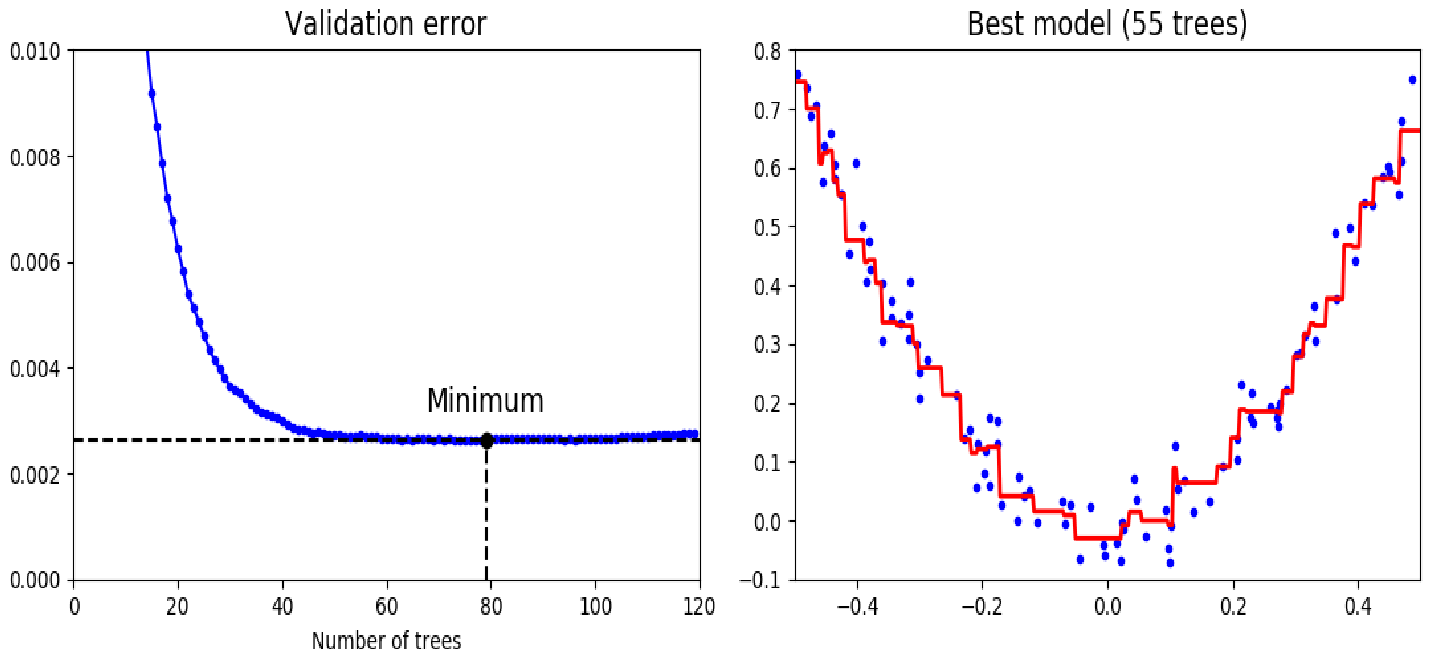
# Random Forest:Gradient boosting

```
rnd.seed(42)
X = rnd.rand(100, 1) - 0.5
y = 3*X[:, 0]**2 + 0.05 * rnd.randn(100)

from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=0.1, random_state=42)
gbrt.fit(X, y)

gbrt_slow = GradientBoostingRegressor(max_depth=2, n_estimators=200, learning_rate=0.1, random_state=42)
gbrt_slow.fit(X, y)

plt.figure(figsize=(11,4))

plt.subplot(121)
plot_predictions([gbrt], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="Ensemble predictions")
plt.title("learning_rate={}, n_estimators={}".format(gbrt.learning_rate, gbrt.n_estimators), fontsize=14)

plt.subplot(122)
plot_predictions([gbrt_slow], X, y, axes=[-0.5, 0.5, -0.1, 0.8])
plt.title("learning_rate={}, n_estimators={}".format(gbrt_slow.learning_rate, gbrt_slow.n_estimators), fontsize=14)

fl.save_fig("gbrt_learning_rate_plot")
plt.show()
```



# Random Forest:Gradient boosting:Early Stopping

```
X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120, learning_rate=0.1, random_state=42)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred) for y_pred in gbrt.staged_predict(X_val)]

best_n_estimators = np.argmin(errors)
min_error = errors[best_n_estimators]

gbrt_best = GradientBoostingRegressor(max_depth=2, n_estimators=best_n_estimators, learning_rate=0.1, random_state=42)
gbrt_best.fit(X_train, y_train)

plt.figure(figsize=(11, 4))
```


Validation error
Minimum
Number of trees
Best model (55 trees)