lab3_solution

December 12, 2018

1 COMP3222/6246 Machine Learning Technologies (2018/19)

1.1 Week 6 – Decision Trees, Random Forests, Ensemble Learning

Follow each code block at your own pace, you can have a look at the book or ask questions to demonstrators if you find something confusing.

2 Chapter 6 - Decision Trees

"*Decision Trees* are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multioutput tasks." [Geron2017]

```
In [88]: %matplotlib inline


         import numpy as np

         np.random.seed(42) # to ensure our results exactly like the book
```

2.1 6.1 Training and Visualizing a Decision Tree

First, let's load the iris dataset from sci-kit learn library.

```
In [89]: from sklearn.datasets import load_iris
         from sklearn.tree import DecisionTreeClassifier

         iris = load_iris()
```

2.1.1 6.1.1 Determine Targets

Let's determine which columns will be in our interest and print them.

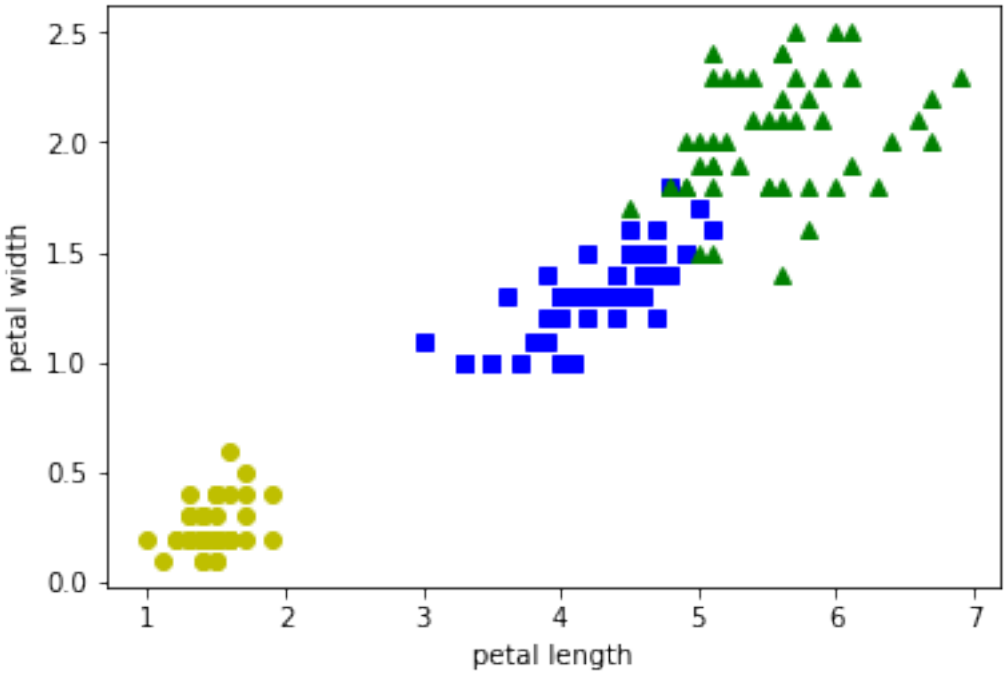
```
In [90]: X = iris.data[:, 2:] # only focus on petal length and width
         Y = iris.target
         feature_names = iris.feature_names[2:]
         print("given:",feature_names,
               "\npredict whether:", iris.target_names)
```

```
given: ['petal length (cm)', 'petal width (cm)']
predict whether: ['setosa' 'versicolor' 'virginica']
```

2.1.2 Exercise 6.1.1: Plot the data set

Plot the data set and have a look at the two features that are selected.

```
In [91]: # use matplotlib as you did on previous labs
         color_map = ["yo", "bs", "g^"]
         for target_index, target_name in enumerate(iris.target_names):
             plt.plot(X[:, 0][Y==target_index], # petal length on X axis (the ones that equal
                      X[:, 1][Y==target_index], # petal width on Y axis (the ones that equal t
                      color_map[target_index],
                      label=target_name)
         plt.xlabel("petal length")
         plt.ylabel("petal width")
         plt.show()
```



2.1.3 6.1.2 Train the dataset

Without separating the dataset as we did in previous labs, let's use all the data set and train the decision tree.

```
In [92]: tree_clf = DecisionTreeClassifier(max_depth=2)
         tree_clf.fit(X,Y)
```

```
Out[92]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=2,
                                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=None,
                                splitter='best')
```

There are many hyperparameters that a decision tree classifier has. You can see from the ouput above the parameters that will be used for predictions. Two criterions that you can use with decision trees in sci-kit learn. These metrics are calculated in each node of decision tree. * **Gini impurity** `criterion='gini'` is a measure of how often a randomly chosen element from a set would be incorrectly labeled. Formally it is computed by:

$$I_G(p) = \sum_{i=1}^{J} p_i \sum_{k \neq i} p_k$$

where J denotes classes and p_i is the fraction of items which are labeled with class i. For a concrete example have a look: https://stats.stackexchange.com/a/339514 * **Information Gain** `criterion='entropy'` is a mesaure of entropy, which is used in thermodynamics as a measure of molecular disorder. Entropy=0 means the molecules are well ordered.

$$H(T) = I_E(p) = -\sum_{i=1}^{J} p_i \log_2 p_i$$

$p_1, p_2, ...$ as in gini impurity are the fractions that add up to 1.

These two metrics are used for deciding the splits while training a decision tree.

2.1.4 Exercise 6.1.2: Gini or Entropy

- **Should you use Gini impurity or entropy?**
  "The most of time it does not make a big difference: they lead to similar trees.". [page=175, Geron2017]
- **Which one is faster to compute and why?**
  "Gini impurity is slightly faster to compute.", since logarithmic functions, which can be computationally intensive. In the cases when there can be difference in terms of result: "Gini impurity tenst to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly balanced trees." [page=175, Geron2017]
- **Visualize the tree that you trained above in 6.1.2.**
  Solution is in 6.1.2.
- **Why does *gini impurity* metric decrease in deeper nodes?**
  Gini impurity in the visualization decreases when the node becomes more "pure". As Wikipedia mentions: "Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset." Thus, in short, the chance of picking a randomly chosen element from a node decreases, as gini impurity decreases.
- **Which cases do you see that the metric is zero?** If the node is completely pure (all instances belong to the same class), the gini impurity becomes zero.

2.1.5 6.1.2 Visualization

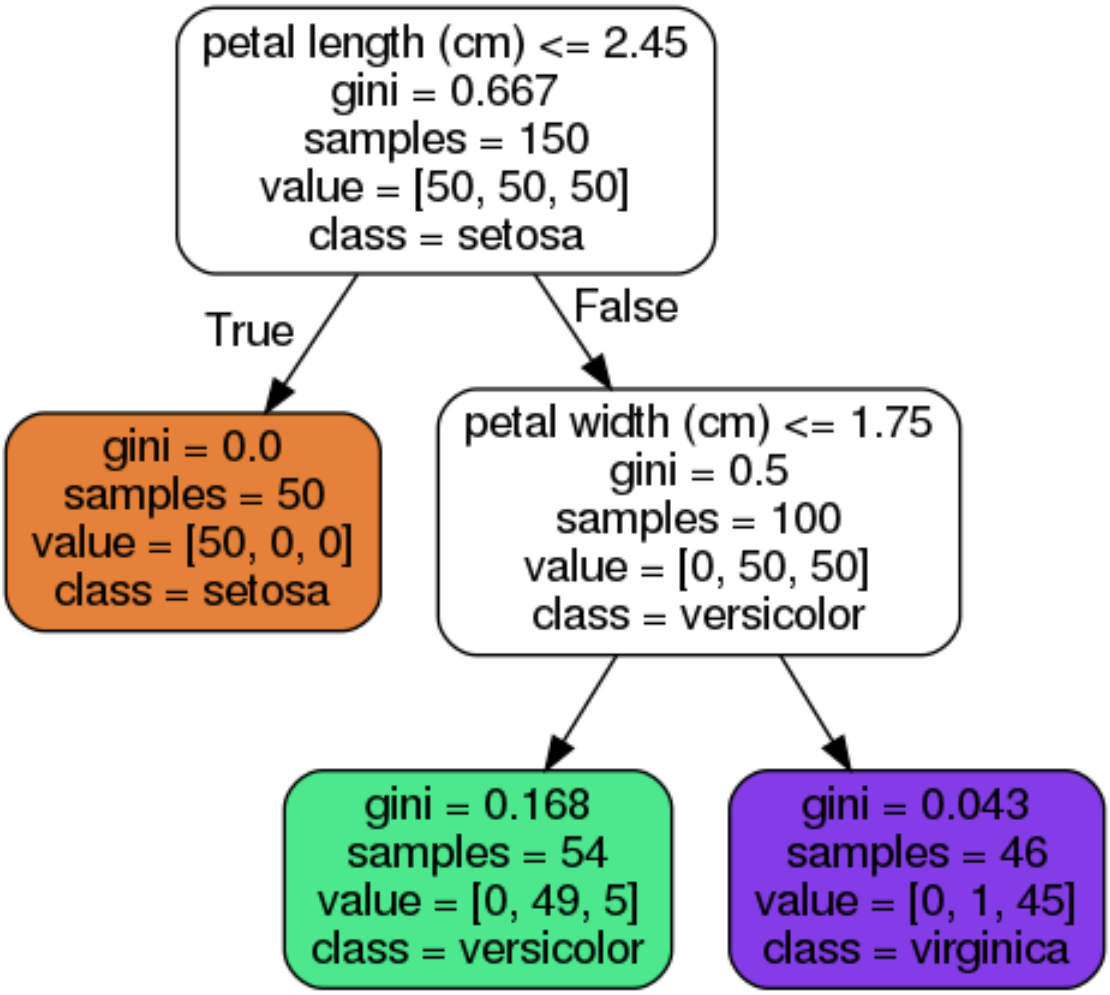
You can export the decision tree as a `dot` file from sci-kit learn. You can convert `dot` to `png` image by installing `graphviz`.

```
In [93]: from sklearn.tree import export_graphviz
         export_graphviz(tree_clf,
                         out_file="iris_tree.dot",
                         feature_names=feature_names,
                         class_names=iris.target_names,
                         rounded=True,
                         filled=True
                        )
```

```
In [ ]: # Make sure you installed graphviz (exclamation mark is for shell commands)
        !apt install graphviz

        # Convert dot file to png file.
        !dot -Tpng iris_tree.dot -o iris_tree.png
```

```
In [95]: from IPython.display import Image
         Image(filename='iris_tree.png')
```

```
Out[95]:
```



To see a better visualization example of decision trees, have a look at this page.

2.1.6 Extra: Another way to visualize decision trees

There is a brand new visualization library from creators of ANTLR (parser generator) for decision trees called `dtreeviz`. You can find some other examples from their repository for better visualization. Follow the steps below:

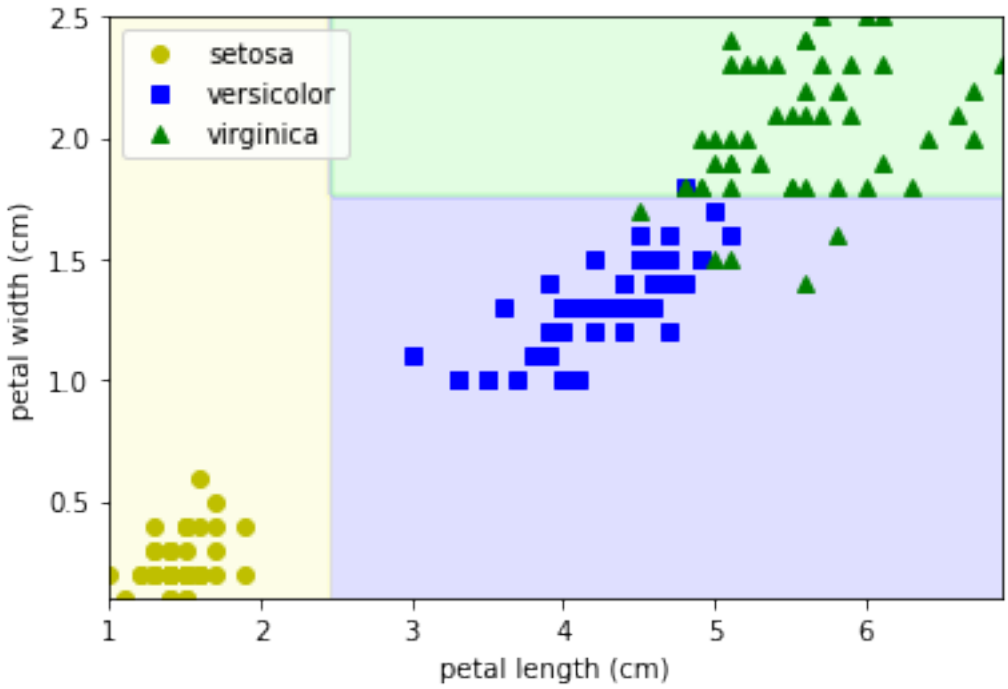
```
In [ ]: # install the package
        !pip install dtreeviz
        # (optional)
        !apt-get install msttcorefonts -qq
```

```
In [97]: from dtreeviz.trees import dtreeviz
         import matplotlib as mpl

         mpl.rcParams['axes.facecolor'] = 'white'
         viz = dtreeviz(tree_clf,
```

```
                    X,
                    Y,
                    target_name='flower type',
                    feature_names=feature_names,
                    class_names=list(iris.target_names),
                    fancy=True,
                    orientation ='TD')

        # uncomment this
        #viz
```

2.1.7 6.1.3 Plot training data set

```
In [98]: import matplotlib.pyplot as plt
         from matplotlib.colors import ListedColormap
         CUSTOM_CMAP = ListedColormap(['#fafab0','#9898ff','#a0faa0'])

         # helper function to plot the boundaries
         def plot_decision_boundary(clf, x, y):
             color_map = ["yo", "bs", "g^"]
             for target_index, target_name in enumerate(iris.target_names):
                 plt.plot(x[:, 0][y==target_index], # petal length on X axis (the ones that eq
                          x[:, 1][y==target_index], # petal width on Y axis (the ones that equ
                          color_map[target_index],
                          label=target_name)

             x1s = np.linspace(np.min(x[:, 0]), np.max(x[:, 0]), 100)
             x2s = np.linspace(np.min(x[:, 1]), np.max(x[:, 1]), 100)
             x1, x2 = np.meshgrid(x1s, x2s)
             x_test = np.c_[x1.ravel(), x2.ravel()]
             y_pred = clf.predict(x_test).reshape(x1.shape)
             plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=CUSTOM_CMAP)

         plot_decision_boundary(tree_clf, X, Y)
         plt.legend()
         plt.xlabel(feature_names[0]) # petal length (cm)
         plt.ylabel(feature_names[1]) # petal width (cm)
         plt.show()
```

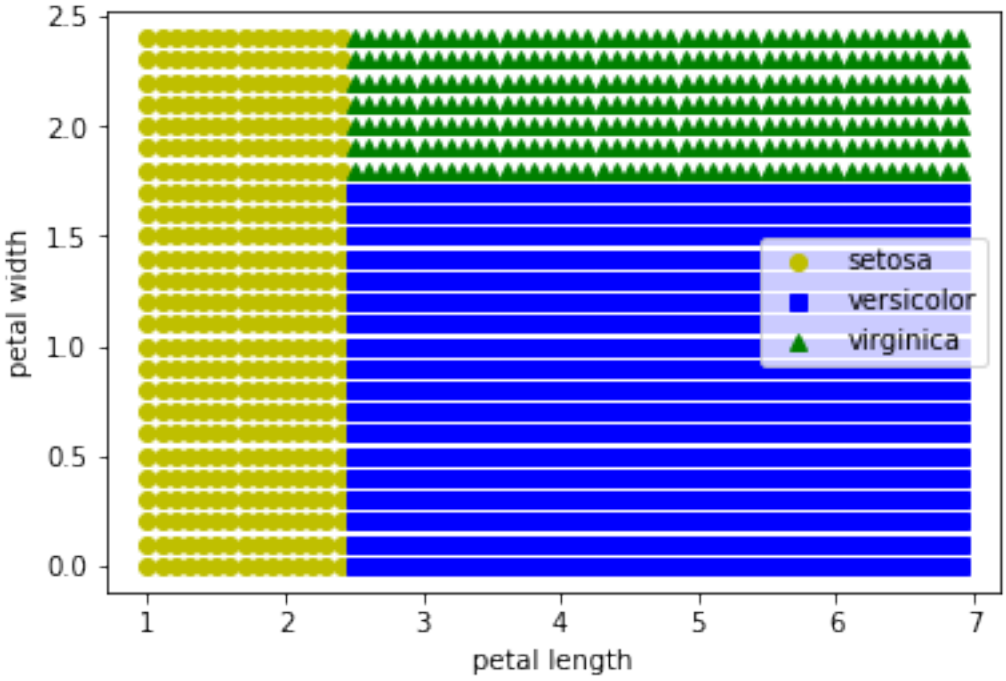


2.1.8 Exercise 6.1: Plot decision boundaries

1. **Write down which colour correspond to which node in the decision tree.**
   By looking at the visulization: * Purple node denotes the green shaded area. * Green node denotes the blue shaded area. * Orange node denotes the yellow shaded area.
2. **Try to increase/decrease `max_depth` of decision tree classifier and observe the changes in the decision boundaries. What would you set `max_depth`?**
   Go to 6.1.2 and change `max_depth` parameter and rerun the blocks until here.
3. **If the helper function (`plot_decision_boundary`) was not available to you, how would you visualize the decision boundaries? Tip: try to create X's that ranges from $[1,0]$ to $[7,2.5]$ where. You can use:**

```
In [99]: # check np.mgrid[minX1:maxX1:increment, minX2:maxX2:increment]
         X = np.mgrid[1:7:0.1, 0:2.5:0.1].reshape(2,-1).T

         color_map = ["yo", "bs", "g^"]
         Y = tree_clf.predict(X)
         for target_index, target_name in enumerate(iris.target_names):
             plt.plot(X[:, 0][Y==target_index], # petal length on X axis (the ones that equal
                      X[:, 1][Y==target_index], # petal width on Y axis (the ones that equal t
                      color_map[target_index],
                      label=target_name)
         plt.legend()
         plt.xlabel("petal length")
```

```
plt.ylabel("petal width")
plt.show()
```



2.1.9 6.1.4 Estimating Class Probabilities

To estimate the probability of an instance belongs to a class, you can use `predict_proba`, to determine the class that an instance will be assigned to use `predict`.

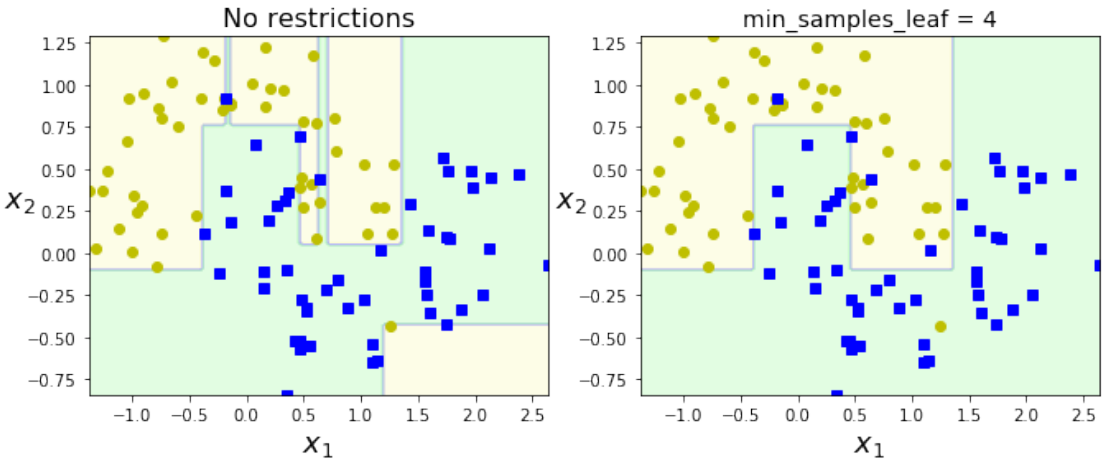
```
In [100]: tree_clf.predict_proba([[5, 1.5]])

Out[100]: array([[0.        , 0.90740741, 0.09259259]])

In [101]: tree_clf.predict([[5, 1.5]])

Out[101]: array([1])
```

2.1.10 6.1.5 Regularization Hyperparameters

"Constraining a model to make it simpler and reduce the risk of overfitting is called _regularization" [Geron2017, page 27] To avoid overfitting, you can limit the generation of a node by `min_samples_leaf` (the minimum samples that a node must have to able to be splitted.).

```
In [102]: from sklearn.datasets import make_moons
          Xm, ym = make_moons(n_samples=100, noise=0.25, random_state=53)
```

```
deep_tree_clf1 = DecisionTreeClassifier(random_state=42)
deep_tree_clf2 = DecisionTreeClassifier(min_samples_leaf=4, random_state=42)
deep_tree_clf1.fit(Xm, ym)
deep_tree_clf2.fit(Xm, ym)

plt.figure(figsize=(11, 4))
plt.subplot(121)

plt.xlabel(r"$x_1$", fontsize=18)
plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
plot_decision_boundary(deep_tree_clf1, Xm, ym)
plt.title("No restrictions", fontsize=16)
plt.subplot(122)

plt.xlabel(r"$x_1$", fontsize=18)
plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
plot_decision_boundary(deep_tree_clf2, Xm, ym)
plt.title("min_samples_leaf = {}".format(deep_tree_clf2.min_samples_leaf), fontsize=

plt.show()
```

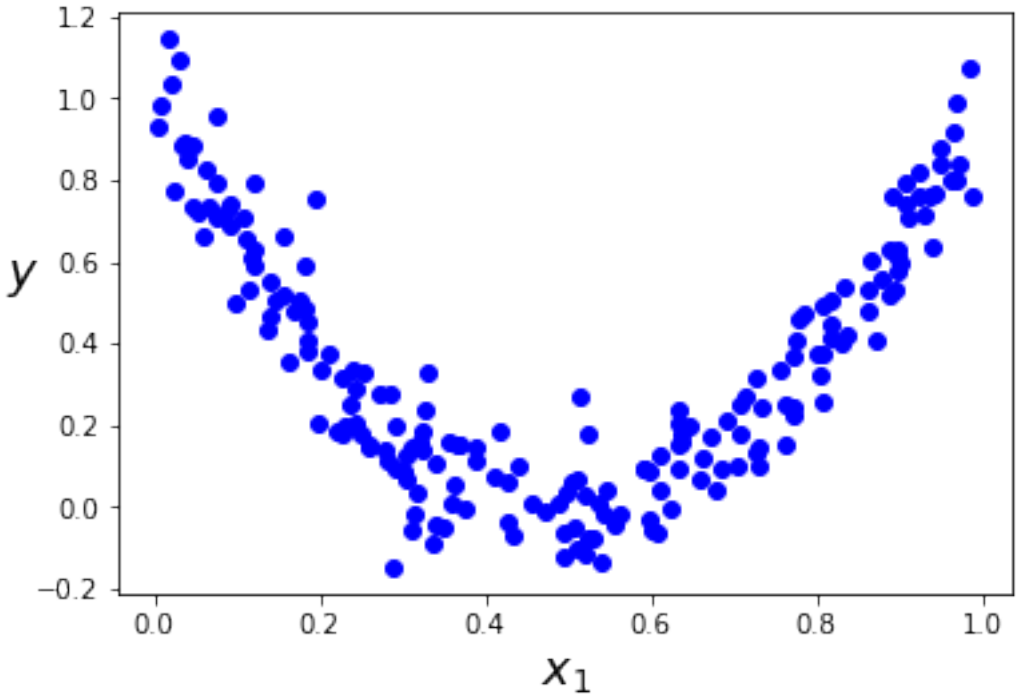


2.2 6.2 Regression

Decision trees can be used for regression tasks too. Instead of predicting a class, in regression tasks, the aim is to predict a numberic value (such as the price of a car). Assume that we have this quadratic data set with some noise:

```
In [103]: # Quadratic training set + noise
          np.random.seed(42)
          m = 200
```

```
X = np.random.rand(m, 1)
y = 4 * (X - 0.5) ** 2
Y = y + np.random.randn(m, 1) / 10

plt.plot(X, Y, "bo")
plt.xlabel("$x_{1}$", fontsize=18)
plt.ylabel("$y$", fontsize=18, rotation=0)
plt.show()
```




```
In [104]: from sklearn.tree import DecisionTreeRegressor
          tree_reg = DecisionTreeRegressor(max_depth=2)
          tree_reg.fit(X,Y)
```

```
Out[104]: DecisionTreeRegressor(criterion='mse', max_depth=2, 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=None, splitter='best')
```

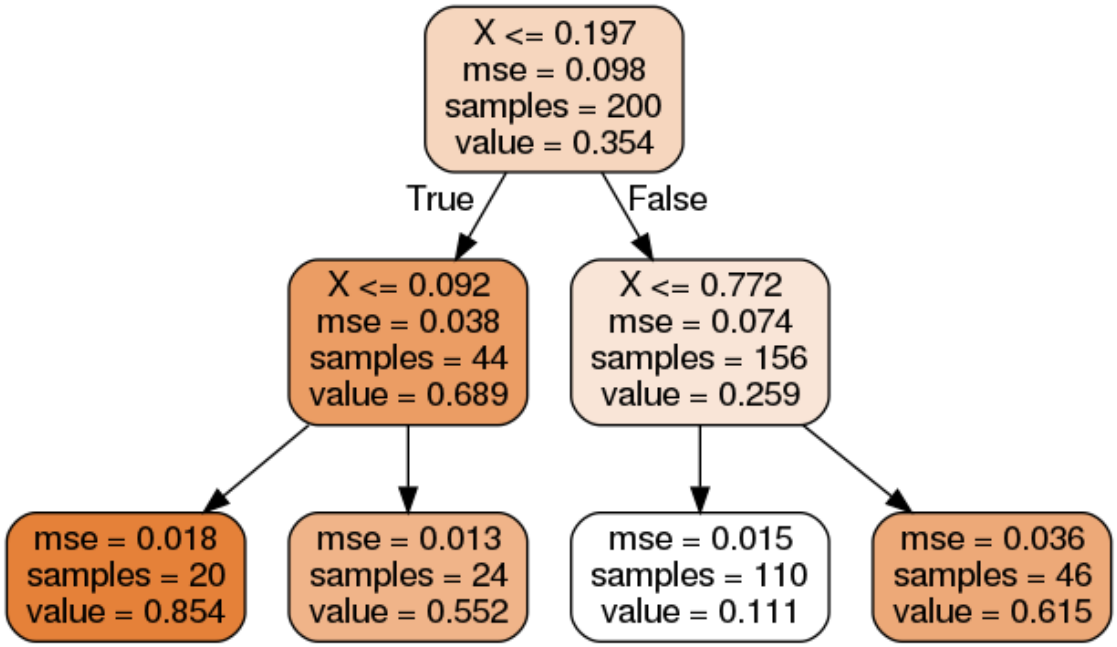
2.2.1 Exercise 6.2: Visualize the regression tree

1. Visualize the regression tree same as before with *graphviz*.

```
In [105]: from sklearn.tree import export_graphviz
          export_graphviz(tree_reg,
```

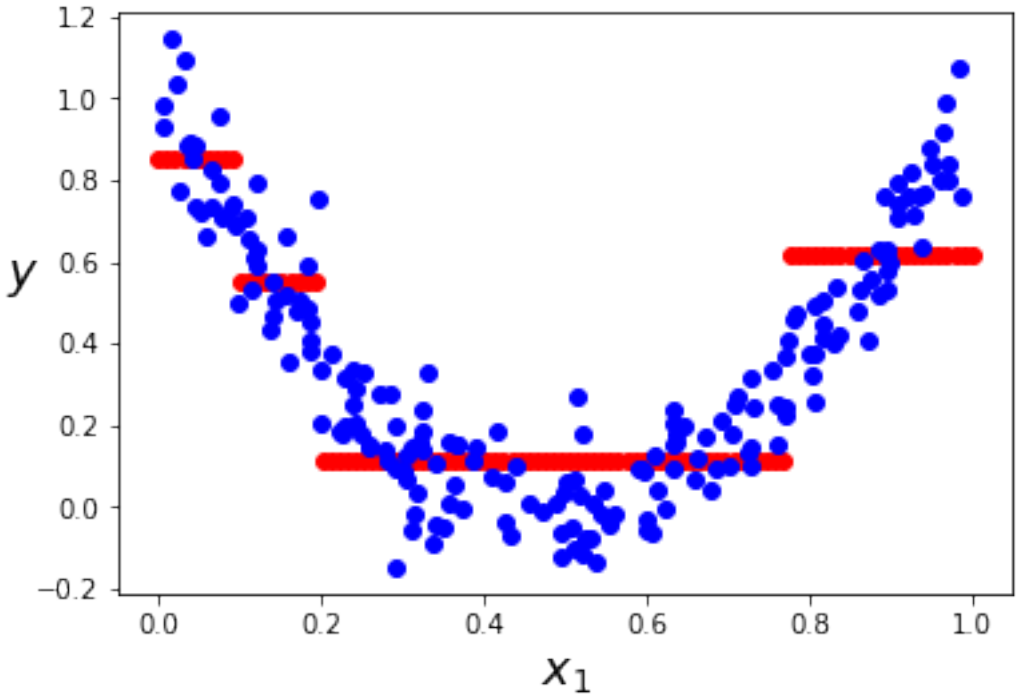
```
                        out_file="tree_reg.dot",
                        feature_names=["X"],
                        class_names=["Y"],
                        rounded=True,
                        filled=True
                       )
# Convert dot file to png file.
!dot -Tpng tree_reg.dot -o tree_reg.png
from IPython.display import Image
Image(filename='tree_reg.png')
```

Out[105]:



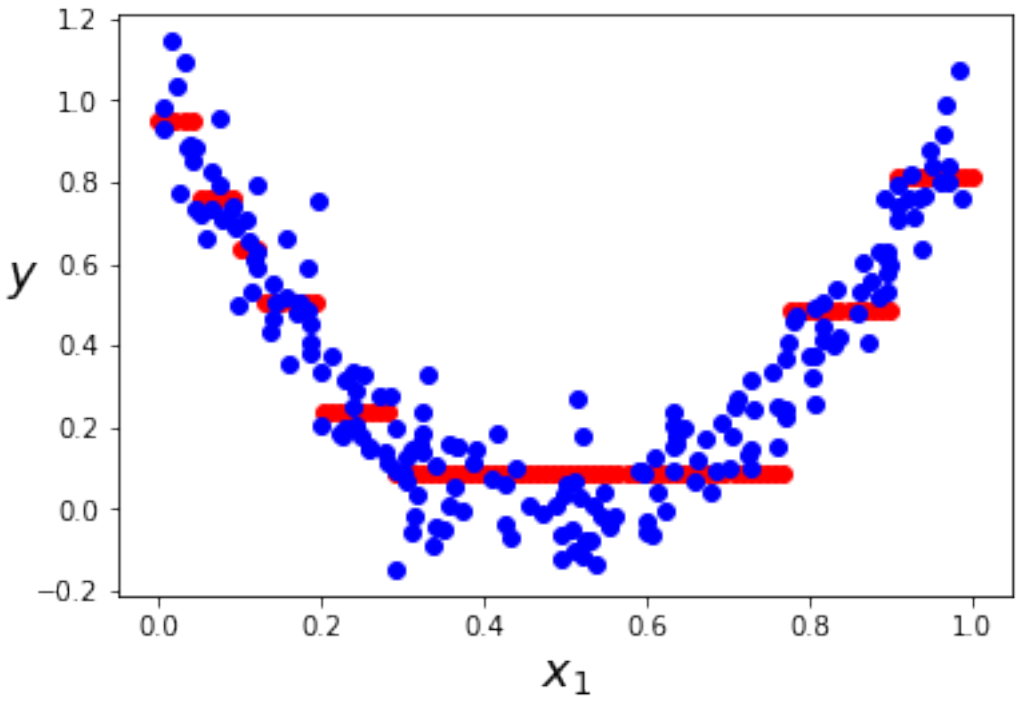
2. Plot this regression tree (Tip: try many values for x (e.g. `np.linspace(min, max, noOfPoints)`)

```
In [106]: Xs = np.linspace(0, 1, 100).reshape(-1, 1)
          Ys = tree_reg.predict(Xs)
          plt.plot(Xs, Ys, "ro")
          plt.plot(X, Y, "bo")
          plt.xlabel("$x_{1}$", fontsize=18)
          plt.ylabel("$y$", fontsize=18, rotation=0)
          plt.show()
```



3. Plot the decision boundaries of `max_depth=2` and `max_depth=3` regression trees (also try `min_samples_leaf=10`)

```
In [107]: # max_depth=2 is at above
          tree_reg = DecisionTreeRegressor(max_depth=3)
          tree_reg.fit(X,Y)
          Xs = np.linspace(0, 1, 100).reshape(-1, 1)
          Ys = tree_reg.predict(Xs)
          plt.plot(Xs, Ys, "ro")
          plt.plot(X, Y, "bo")
          plt.xlabel("$x_{1}$", fontsize=18)
          plt.ylabel("$y$", fontsize=18, rotation=0)
          plt.show()
```



4. Compare the differences on the difference plots. Notice that the average is taken in the regions which are separated by the decision tree regressor.

Check the plots from 2 and 3.

3 Chapter 7: Ensemble Learning and Random Forests

Instead of using a single predictor, to improve our predictions we use now use *an ensemble*: **a group of predictors**. It is as if you are asking a number of experts opinion about a problem and you aggregate their answers.

3.1 7.1 Voting Classifiers

A brief comparision between `soft` voting and `hard` voting with using three predictors.

```
In [108]: from sklearn.model_selection import train_test_split
          from sklearn.datasets import make_moons

          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)
```

Helper function for printing accuracies on test set

```
In [109]: from sklearn.metrics import accuracy_score
          def test_clfs(*clfs): # clf -> classifier
              for clf in clfs:
```

```
    clf.fit(X_train, Y_train) # train the classifier
    Y_pred = clf.predict(X_test)
    print(clf.__class__.__name__ + ":", accuracy_score(Y_test, Y_pred))
```

Let's test `hard` voting first.

```
In [110]: from sklearn.ensemble import RandomForestClassifier
          from sklearn.ensemble import VotingClassifier
          from sklearn.linear_model import LogisticRegression
          from sklearn.svm import SVC

          # Don't worry about the warnings,
          # sci-kit community will be fixing it in the next major version 0.20.0

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

          voting_clf = VotingClassifier(estimators=[('lr', log_clf),
                                                    ('rf', rnd_clf),
                                                    ('svc', svm_clf)],
                                        voting='hard')
          test_clfs(log_clf, rnd_clf, svm_clf, voting_clf)
```

```
LogisticRegression: 0.864
RandomForestClassifier: 0.872
SVC: 0.888
VotingClassifier: 0.896


/home/tdgunes/Projects/COMP6246-2018Fall/.venv/lib/python3.6/site-packages/sklearn/linear_model
  FutureWarning)
/home/tdgunes/Projects/COMP6246-2018Fall/.venv/lib/python3.6/site-packages/sklearn/ensemble/for
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/home/tdgunes/Projects/COMP6246-2018Fall/.venv/lib/python3.6/site-packages/sklearn/svm/base.py
  "avoid this warning.", FutureWarning)
/home/tdgunes/Projects/COMP6246-2018Fall/.venv/lib/python3.6/site-packages/sklearn/linear_model
  FutureWarning)
/home/tdgunes/Projects/COMP6246-2018Fall/.venv/lib/python3.6/site-packages/sklearn/svm/base.py
  "avoid this warning.", FutureWarning)
```

3.1.1 Exercise 7.1: Which voting to pick?

1. Check the `soft` voting and compare the results. Why do you think it is different?

If you don't know the difference have a look at your book, page 186.

From the book: "If all classifiers are able to estimate class probabilites, then you can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*". The hard voting in this case is only majority voting for deciding the prediction.

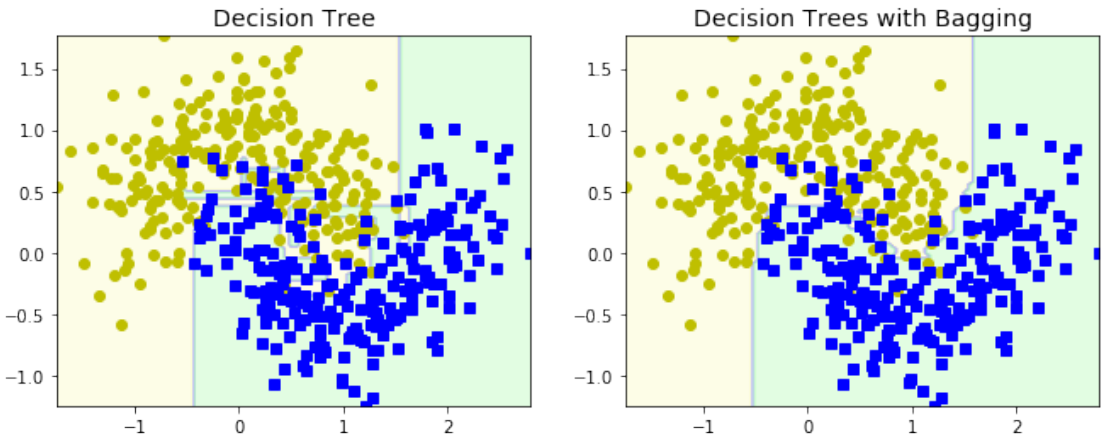
3.2 7.2 Bagging & Pasting

Instead of giving training set to each predictor in our ensemble, another approach to gain more accuracy is to separate the training set and give a different training subset to each predictor. There are two ways: * **Bagging:** (*bootstrapping in statistics*) picking a random subset from training set and not removing this selected set from training set for each predictor, i.e. **sampling with replacement** * **Pasting:** picking a random subset from training set and removing this selected set from the training set for each predictor, i.e. **sampling without replacement** If you are still confused, have a look at here.

```
In [111]: from sklearn.ensemble import BaggingClassifier
          from sklearn.tree import DecisionTreeClassifier

          # define our decision tree classifier
          tree_clf = DecisionTreeClassifier(random_state=42)
          # 500 copies of the predictor, which has 100 samples from training set
          # n_jobs=-1 for utilizing all cores
          bag_clf = BaggingClassifier(tree_clf,
                                      n_estimators=500,
                                      max_samples=100,
                                      bootstrap=True,
                                      n_jobs=-1,
                                      random_state=42)

          # fit the bagging classifier
          bag_clf.fit(X_train, Y_train)
          tree_clf.fit(X_train, Y_train)

          Y_pred_bag = bag_clf.predict(X_test)
          Y_pred_tree = tree_clf.predict(X_test)

          from sklearn.metrics import accuracy_score
          print("Bagging Classifier")
          print(accuracy_score(Y_test, Y_pred_bag))
          print("Decision Tree Classifier")
          print(accuracy_score(Y_test, Y_pred_tree))

          plt.figure(figsize=(11,4))
          plt.subplot(121)
          plot_decision_boundary(tree_clf, X, Y)
          plt.title("Decision Tree", fontsize=14)
          plt.subplot(122)
          plot_decision_boundary(bag_clf, X, Y)
          plt.title("Decision Trees with Bagging", fontsize=14)
          plt.show()
```

```
Bagging Classifier
0.904
Decision Tree Classifier
```

```
0.856
```



3.2.1 7.2 Exercise: Bagging or Pasting

- **What is the difference between these two techniques which select a subset of a training set?**

The explanation above answers this question.

- **When do you think it is better to use bagging or pasting?**

The book page 189: Bootstraping (bagging) "introduces a bit more diversity in the subsets that each predictor is trained on ...". "Overall, bagging often results in better models, ..."

3.3 7.3 Gradient Boosting

In Gradient Boosting, we start with a single predictor. Sequentially, we add new predictors to *an ensemble* that corrects its predecessor. In detail:

The algorithm is for `3 predictors` in our ensemble: 1. Pick a set of weak predictors (e.g. DecisionTreeRegressor(max_depth=2)) - For this example, it is 3. 2. Train the initial predictor on training set 3. Train the second predictor on residual errors (use only errors, i.e. `y2 = y - predictor1.predict(X)`) 3. Train the third predictor on the residual errors of the second predictor 4. Use all predictors and sum their predictions (if regression)

3.3.1 Exercise 7.3: Implement, Plot and Compare

1. Implement this boosting technique that is explained above with `n` predictors.
   - Use the `quadratic training set` as a regression task.
   - Use `DecisionTreeRegressor(max_depth=2)` as a weak predictor
2. For $n = 3$, plot step by step the evolution of the ensemble predictions

- First figure with only one predictor
- Second figure with two predictors
- Third figure with three predictors

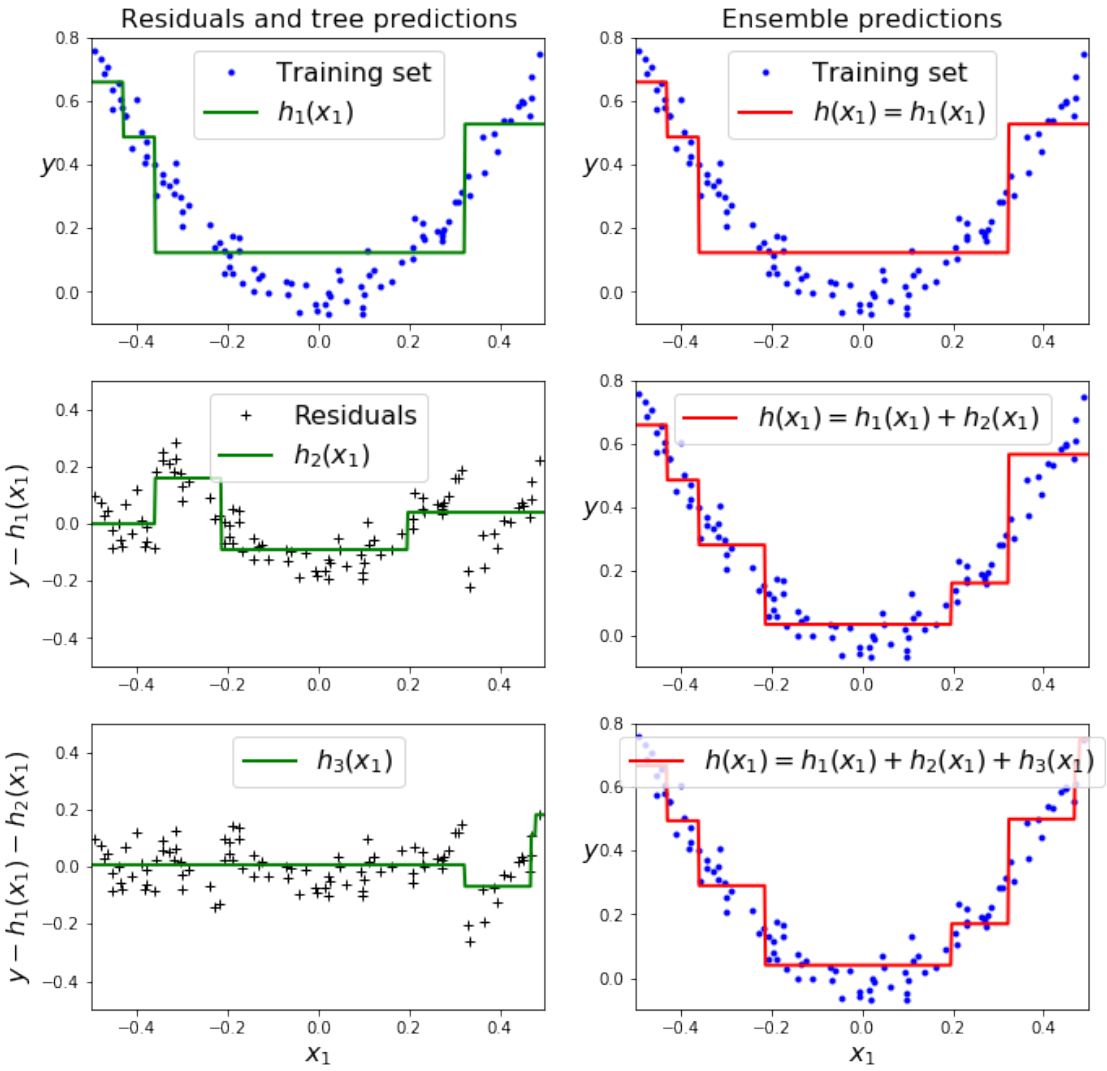
3. Use sci-kit learn's `GradientBoostingRegressor` and compare if your results are similiar.
4. **Do you think this boosting technique is scalable in practice?**
   We sequentially add predictors and train by the residual errors. For this reaseon, it cannot be parallelized (or only partially.) It does not scale well as bagging or pasting.

Tips: - **Plotting**: this is similar as before in Regression section 6.2, as explained earlier. - **GradientBoostingRegressor**: use this code snippet:

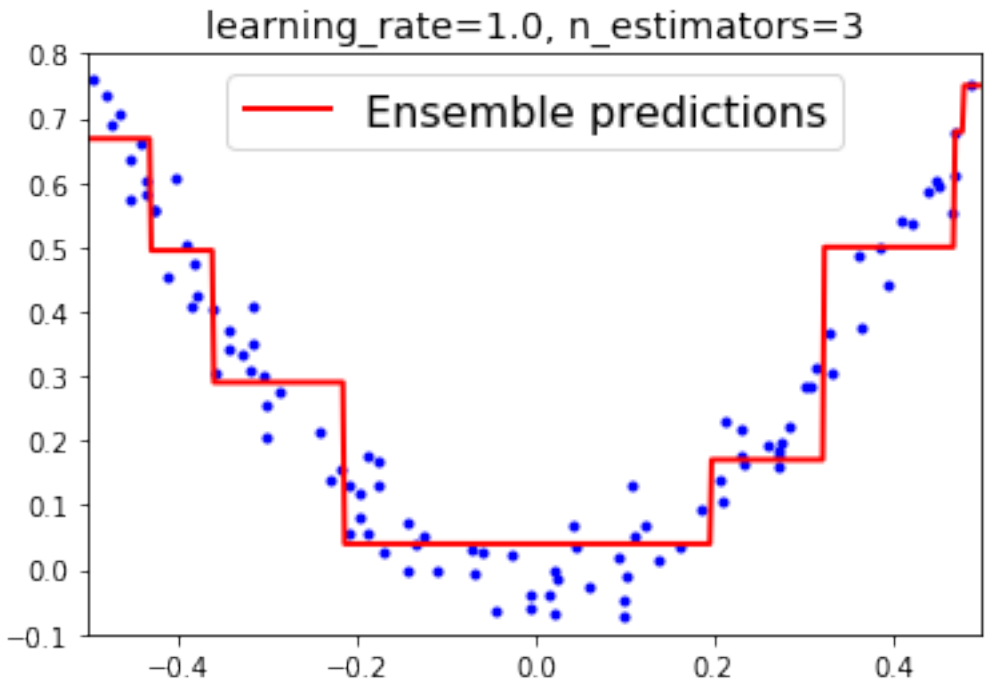
```
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X,Y)
```

```
In [112]: # Your implementation here
          # Quadratic training set + noise

          # 7.3.1
          np.random.seed(42)
          X = np.random.rand(100, 1) - 0.5
          y = 3*X[:, 0]**2 + 0.05 * np.random.randn(100)

          # 7.3.2
          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)

          def plot_predictions(regressors, X, y, axes, label=None, style="r-", data_style="b."
              x1 = np.linspace(axes[0], axes[1], 500)
              y_pred = sum(regressor.predict(x1.reshape(-1, 1)) for regressor in regressors)
              plt.plot(X[:, 0], y, data_style, label=data_label)
              plt.plot(x1, y_pred, style, linewidth=2, label=label)
              if label or data_label:
                  plt.legend(loc="upper center", fontsize=16)
              plt.axis(axes)


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

          plt.subplot(321)
```

```
plot_predictions([tree_reg1], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="$h_1(x_1)$",
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.title("Residuals and tree predictions", fontsize=16)

plt.subplot(322)
plot_predictions([tree_reg1], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="$h(x_1) = h_
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.title("Ensemble predictions", fontsize=16)

plt.subplot(323)
plot_predictions([tree_reg2], X, y2, axes=[-0.5, 0.5, -0.5, 0.5], label="$h_2(x_1)$"
plt.ylabel("$y - h_1(x_1)$", fontsize=16)

plt.subplot(324)
plot_predictions([tree_reg1, tree_reg2], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="$
plt.ylabel("$y$", fontsize=16, rotation=0)

plt.subplot(325)
plot_predictions([tree_reg3], X, y3, axes=[-0.5, 0.5, -0.5, 0.5], label="$h_3(x_1)$"
plt.ylabel("$y - h_1(x_1) - h_2(x_1)$", fontsize=16)
plt.xlabel("$x_1$", fontsize=16)

plt.subplot(326)
plot_predictions([tree_reg1, tree_reg2, tree_reg3], X, y, axes=[-0.5, 0.5, -0.1, 0.8
plt.xlabel("$x_1$", fontsize=16)
plt.ylabel("$y$", fontsize=16, rotation=0)

plt.show()
```



In [113]:
```
# 7.3.3

from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0, rand
gbrt.fit(X, y)

plot_predictions([gbrt], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="Ensemble predictio
plt.title("learning_rate={}, n_estimators={}".format(gbrt.learning_rate, gbrt.n_estim
plt.show()
```



3.4 Recap

At this point, we demonstrated these concepts:

- training and visualizing various models from Decision Trees and Ensemble Learning
- details about some hyperparameters and how to regularize the models
- different training methods for ensemble learning
- how a boosting technique works and its properties.

Of course, there are some material that we have not be able to cover. In your free time, it can be better to have a look at:

- **Decision Trees**
- How hard is it to find the best split to construct the *optimal tree*?
- Why is CART algorithm called *a greedy algorithm*?
- **Ensemble Learning and Random Forests**
- Difference between Extra-Trees and Random Forests
- How does AdaBoost work?
- Learning Feature Importance with Random Forests
- What is `Stacking`?

In [ ]: