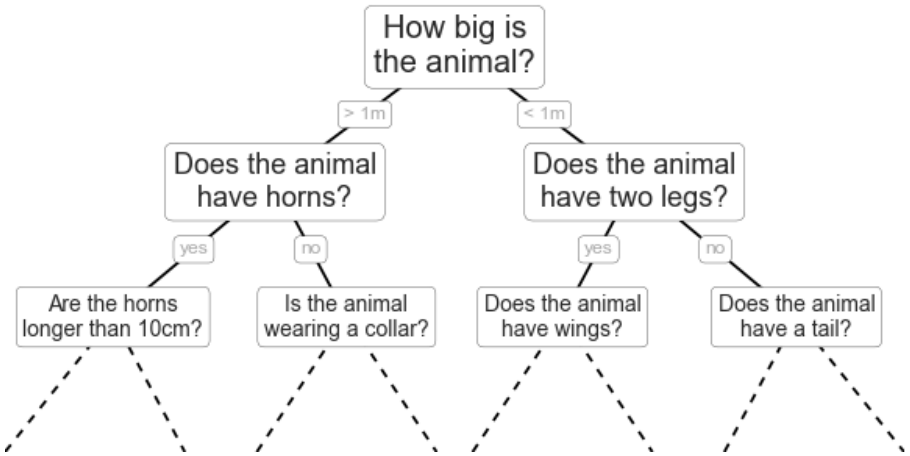
# Chapter 4:
# *Supervised Learning* II: *Decision Tree, Ensamble Learning* dan *Random Forests*

October 16, 2020

## 1 *Decision Tree*

*Decision Tree* (DT) merupakan algoritma *Machine Learning* (ML) yang dapat digunakan untuk klasifikasi maupun regresi, dan bahkan bisa digunakan pada masalah-masalah dengan *multioutput*. DT merupakan algoritma yang cukup *powerful* mampu melakukan *fitting* untuk dataset yang cukup kompleks. DT juga merupakan bagian fundamental dari *Random Forrest*, yang merupakan salah satu algoritma ML tersedia dan cukup powerful.

Sebagai ilustrasi dari DT, contoh pada Gambar 4.1 adalah penggunaan DT untuk melakukan konstruksi klasifikasi jenis binatang yang mungkin anda temui ketika sedang berjalan-jalan ke hutan. Pemisahan secara biner membuat algoritma DT sangat efisien. Pada DT yang dikonstruksi dengan baik, setiap pertanyaan akan mengurangi pilihan kurang lebih setengahnya, sehingga akan sangat cepat mengurangi kemungkinan-kemungkinan pilihan meskipun jumlah *class yang sangat besar*. *Trick*-nya tentunya adalah untuk memberikan pertanyaan yang paling tepat pada setiap langkah. Pada implementasi *machine learning* dari DT, pertanyaan-pertanyaan tersebut pada setiap node membagi data ke dalam dua grup menggunakan sebuah harga *cutoff* pada salah satu *features* yang ada.




Gambar 4.1. Ilustrasi decision tree untuk melakukan klasifikasi jenis binatang

Untuk memvisualisasikan DT, kita akan membuat sebuat contoh DT dan melihatnya bagaimana melakukan prediksi. Kode Python berikut melakukan training dari `DecionTreeClassifier` pada dataset iris.

- Load dataset iris

[17]:
```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
```

- Training *classifier*

[18]:
```
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42, criterion='gini')
tree_clf.fit(X, y)
```

[18]:
```
DecisionTreeClassifier(ccp_alpha=0.0, 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='deprecated',
                       random_state=42, splitter='best')
```

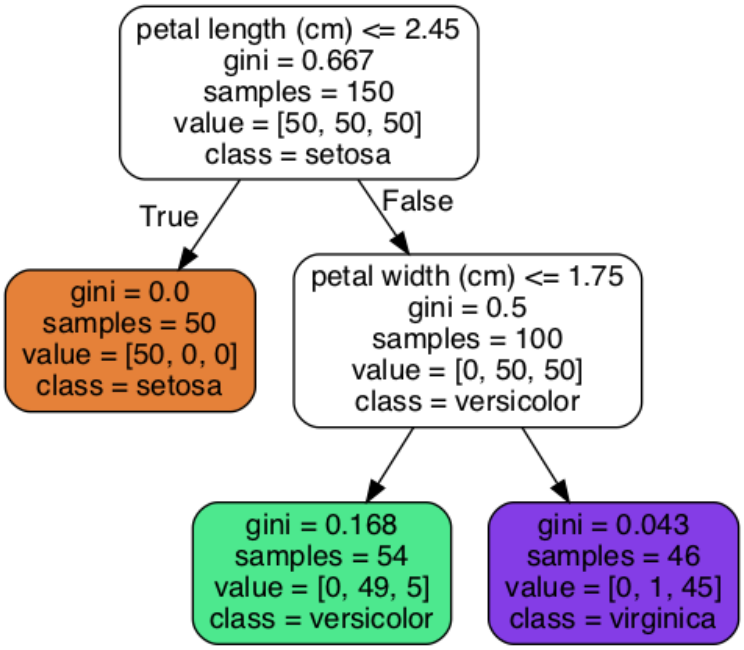
- Untuk memvisualisasikan DT yang sudah ditraining, kita bisa memakai metode `export_graphviz()` sehingga dihasilkan file `.dot` dan file image.

[73]:
```
from pydotplus import graph_from_dot_data
from sklearn.tree import export_graphviz

dot_data = export_graphviz(
        tree_clf,
        out_file=None,
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
graph = graph_from_dot_data(dot_data)
graph.write_png('tree_4.jpg')
```

[73]:
```
True
```

Maka hasil training algoritma DT dapat dilihat pada file `tree_4.jpg` yang telah tersimpan, yang diperlihatkan juga pada Gambar 4.2.

Jika dimisalkan kita menemukan sebuah bunga iris dan ingin mengklasifikasikannya, apakah dari jenis sentosa, versicolor atau virginica, maka proses klasifikasi yang ditunjukkan pada Gambar 4.2 adalah sebagai berikut. 1. Dimulai pada *root node* (*depth* = 0,paling atas), dengan pertanyaan apakah *petal length* $\leq$ 2.45? Jika ya, maka kita turun ke node anak *root node* sebelah kiri (*depth* = 1, kiri), yang merupakan *leaf node* karena tidak mempunyai lagi node anak. Jika tidak maka turun ke node anak sebelah kanan (*depth* = 1, kanan), yang bukan *leaf node* karena masih mempunyai




Gambar 4.2. Decision Tree untuk bunga Iris.

node anak. 2. Pada node anak sebelah kanan, pertanyaan selanjutnya apakah *petal width* $\leq 1.75$? Jika ya, maka bunga tersebut kemungkinan besar adalah *Iris versicolor* ($depth = 2$, kiri), dan jika tidak maka kemungkinan besar bunga tersebut adalah *Iris virginica* ($depth = 2$, kanan).

Pada Gambar 4.2, atribut `samples` menghitung jumlah sampel training yang termasuk kategori tersebut. Misalkan, 100 sampel training mempunyai *petal length* $> 2.46$ cm ($depth = 1$, kanan), dan diantara 100 sampel tersebut, 54 sampel training mempunyai *petal width* $\leq 1.75$ cm ($depth = 2$, kiri). Atribut `value` menyatakan berapa training sampel dari tiap *class* yang berada pada node tersebut. Misalkan untuk node di kanan bawah terdapat 0 *Iris sentosa*, 1 *Iris versicolor* dan 45 *Iris virginica*. Dan yang terakhir, atribut `gini` menyatakan ukuran *impurity*. Node dikatakan *pure* jika *gini* = 0 yang berarti semua sampel training yang lewat node tersebut berasal dari satu *class* yang sama. Sebagai contoh, node ($depth = 1$, kiri) hanya dilalui oleh sampel training *class Iris sentosa*, maka dikategorikan sebagai *pure* dan `gini` = 0. Perhitungan skor `gini` menggunakan persamaan (4.1). Sehingga untuk node ($depth = 2$, kiri), skore gini sama dengan $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$

> ***Persamaan (4.1).*** Perhitungan *gini impurity*
>
> $G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$
>
> dimana $p_{i,k}$ menyatakan rasio dari sampel-sampel training *class* $k$ diantara sampel-sampel training keseluruhan yang sampai di node ke-$i$.

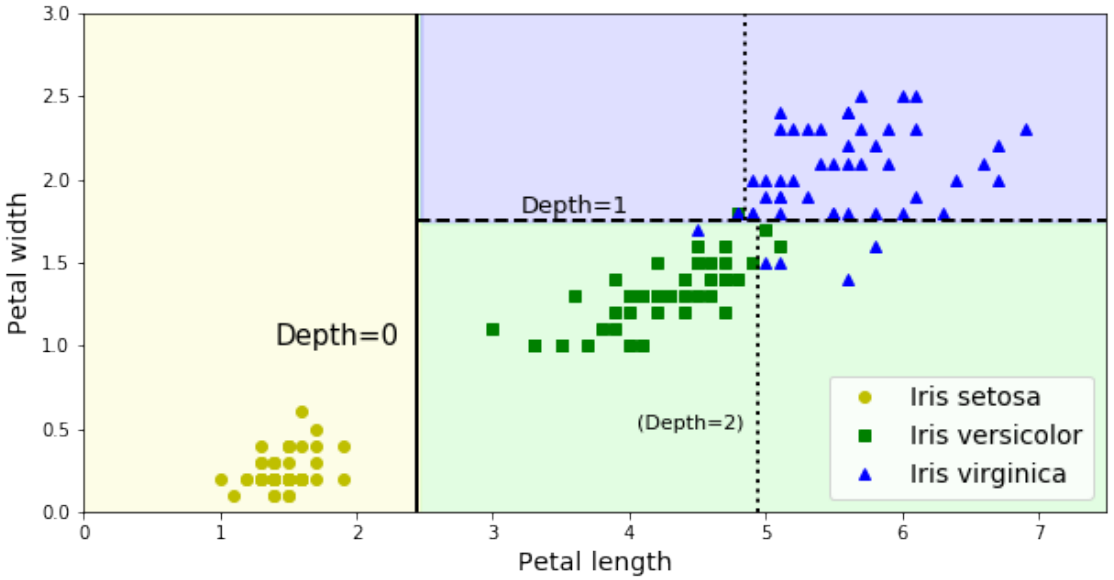
**Catatan.** *Scikit-Learn* menggunakan algoritma *Classification and Regression Tree* (CART) yang hanya menghasilkan *binary trees*. Node-node *Non-leaf* selalu mempunyai dua cabang (pertanyaan selalu dengan jawaban ya/tidak). Tetapi, algoritma lain seperti *ID3* dapat menghasilkan DT dengan node-node yang mempunyai lebih dari dua cabang.

Program Python berikut menghasilkan gambar yang menunjukan batas keputusan dari DT (DT *decision boundaries*). Garis tebal merepresentasikan batas keputusan dari node *root* ($depth = 0$)

dengan *petal length* = 2.45 cm. Karena area sebelah kiri mempunyai *impurity* = *0* (hanya *Iris sentosa*), maka tidak bisa dibagi lebih jauh ke dalam cabang lain. Tetapi area sebelah kanan bersifat *impure* sehingga node *depth* = 1 sebelah kanan dibagi lagi dengan batas *petal width* = 1.75 cm, yang direpresentasikan dengan garis putus-putus. Karena `max_depth` dibatasi 2, maka DT selanjutnya akan berhenti. Jika `max_depth` dibatasi dengan 3, maka percabangan akan dilanjutkan sekali lagi dimana node *depth* = 2 akan terbagi masing-masing menjadi dua daerah (garis-garis putus titik-titik).

[20]:
```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

def plot_decision_boundary(clf, X, y, axes=[0, 7.5, 0, 3], iris=True,␣
 ↪legend=False, plot_training=True):
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#a0faa0','#9898ff'])
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap)
    if not iris:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
        plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
    if plot_training:
        plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris setosa")
        plt.plot(X[:, 0][y==1], X[:, 1][y==1], "gs", label="Iris versicolor")
        plt.plot(X[:, 0][y==2], X[:, 1][y==2], "b^", label="Iris virginica")
        plt.axis(axes)
    if iris:
        plt.xlabel("Petal length", fontsize=14)
        plt.ylabel("Petal width", fontsize=14)
    else:
        plt.xlabel(r"$x_1$", fontsize=18)
        plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
    if legend:
        plt.legend(loc="lower right", fontsize=14)

plt.figure(figsize=(10, 5))
plot_decision_boundary(tree_clf, X, y,legend=True)
plt.plot([2.45, 2.45], [0, 3], "k-", linewidth=2)
plt.plot([2.45, 7.5], [1.75, 1.75], "k--", linewidth=2)
plt.plot([4.95, 4.95], [0, 1.75], "k:", linewidth=2)
plt.plot([4.85, 4.85], [1.75, 3], "k:", linewidth=2)
plt.text(1.40, 1.0, "Depth=0", fontsize=15)
plt.text(3.2, 1.80, "Depth=1", fontsize=13)
```

```
plt.text(4.05, 0.5, "(Depth=2)", fontsize=11)

plt.show()
```




### 1.1 Estimasi probabilitas dari *class*

DT bisa digunakan untuk melakukan estimasi probabilitas sebuah data termasuk pada *class* k tertentu. Pertama, algoritma akan memeriksa data tersebut melalui cabang-cabang untuk menentukan *leaf node* data tersebut. Setelah ditemukan maka akan dihitung rasio sampel-sampel training yang termasuk pada *class* k tersebut pada node ini, dan mengembalikannya sebagai probabilitas *class* k untuk data tersebut. Sebagai contoh, jika terdapat bunga dengan panjang *petal* 5 cm dan lebar 1.5 cm, maka *leaf node* untuk data tersebut adalah *depth* $= 2$ sebelah kiri, sehingga DT akan menghasilkan probabilitas berikut: 0% 0/54 untuk *Iris setosa*, 90.7% 49/54 untuk *Iris virginica*, dan 9.3% (5/54) untuk *Iris versicolor*.

[21]:
```
tree_clf.predict_proba([[5, 1.5]])
```

[21]:
```
array([[0.        , 0.90740741, 0.09259259]])
```

Maka, jika algoritma DT digunakan untuk klasifikasi, data tersebut dikategorikan sebagai *class* *Iris virginica* (*class* 1), karena mempunyai probabilitas terbesar.

[22]:
```
tree_clf.predict([[5, 1.5]])
```

[22]:
```
array([1])
```

**Catatan.** Algoritma akan menghasilkan probabilitas yang sama untuk setiap data pada daerah kotak yang sama pada gambar di atas.

## 1.2 Algoritma Training *Classification and Regression Training* (CART)

`Scikit-Learn` menggunakan algoritma CART untuk training DT. Algoritma bekerja dengan melakukan pemisahan (*splitting*) training set menjadi 2 subset menggunakan satu *feature* k dan sebuah threshold $t_k$ (mis. *petal length* $\leq$ 2.45 cm). Pertanyaannya kemudian adalah, bagaimana memilih *feature* k dan juga threshold ($t_k$)? Maka DT akan mencari pasangan $(k, t_k)$ yang menghasilkan subset yang paling kecil *impurity*-nya (diboboti dengan ukurannya). Persamaan (4.2) memberikan *cost-function* yand diminimalkan pada saat mencari pasangan $(k, t_k)$.

**Persamaan (4.2)** *Cost function* dari CART untuk klasifikasi

$$J(k, t_k) = \frac{m_{kiri}}{m} G_{kiri} + \frac{m_{kanan}}{m} G_{kanan}$$

dimana, $G_{kiri/kanan}$ merupakan ukuran *impurity* dari subset kiri/kanan dan $m_{kiri/kanan}$ adalah jumlah data pada subset *kiri/kanan*.

Setelah algoritma CART berhasil memisahkan training set menjadi dua subset, maka kemudian CART akan melakukan pemisahan subset tersebut menggunakan cara yang sama. Demikian seterusnya dilakukan secara rekursif, dan akan berhenti setelah mencapai *maximum depth* (didefinisikan dengan *hyperparameter* `max_depth`), atau sampai tidak ditemukan lagi pemisahan yang akan mengurangi *impurity*.

## 1.3 *Impurity* dengan Gini atau *Entropy*

Untuk perhitungan *impurity* secara default digunakan *Gini Impurity* pada Persamaan (4.1), tetapi kita juga bisa menggunakan *entropy impurity* dengan memilih *hyperparameter* `criterion` ke `entropy`. Konsep *entropy* berasal dari termodinamika sebagai ukuran ketiakberaturan molekuler. Ketika *entropy* mendekati nilai nol maka molekul-molekul berada pada keteraturan. *Entropy* juga menyebar ke domain yang lain, seperti pada Teori Informasi dari Shannon, dimana istilah *entropy* digunakan untuk ukuran rata-rata informasi sebuah pesan. *Entropy* bernilai nol jika semua pesan identik.

Pada *Machine Learning*, ukuran *entropy* sering digunakan sebagai ukuran *impurity*. Jika sebuah node hanya berisi data-data dari satu kelas saja maka *entropy* dikatakan nol. Seperti pada Gambar 4.2, pada node dengan $depth = 2$ sebelah kiri mempunyai *entropy* sama dengan $-(49/54)\log_2(49/54) - (5/54)\log_2(5/54) = 0.445$.

**Persamaan (4.3)** *Entropy*

$$H_i = \sum_{k=1, p_{ik} \neq 0}^{n} p_{i,k} \log_2(p_{i,k})$$

Pertanyaannya mana yang lebih baik kita gunakan, ukuran *impurity* dengan Gini atau *Entropy*? Pengalaman menunjukan keduanya akan menghasilkan kinerja yang tidak jauh berbeda. Keduanya biasanya akan menghasilkan *trees* yang sama. Gini *impurity* cenderung lebih cepat dibandingkan dengan *entropy impurity*. Tetapi, ketika keduanya berbeda maka Gini mempunyai kecenderungan untuk mengisolasi *class* yang paling banyak muncul pada satu cabang tersendiri dari sebuah *tree*, sedangkan *Entropy* lebih cenderung menghasilkan *trees* yang lebih seimbang.

### 1.4 Regresi dengan Decision Tree

*Decision tree* juga mempunyai kemampuan untuk melakukan regresi. Kita dapat membangun *regression tree* dengan menggunakan *Scikit-Learn* dengan *class* `DecisionTreeRegressor`, melakukan training pada dataset kuadratik yang bernoise dengan `max_depth=2`. Berikut contoh implementasi sederhana dari Regresi dengan DT.

- Membangkitkan data training $X$ dan target $y$

[23]:
```
# Quadratic training set + noise
import numpy as np
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
```

- Karena menggunakan DT untuk regresi, maka import DecisionTreeRegressor dari `Scikit Learn`.

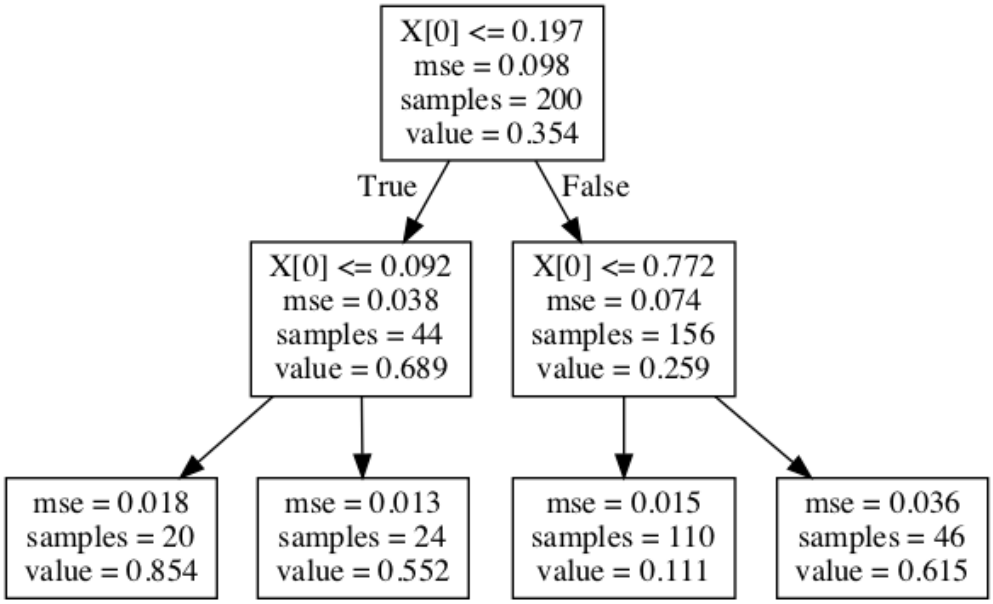
[24]:
```
from sklearn.tree import DecisionTreeRegressor

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

[24]:
```
DecisionTreeRegressor(ccp_alpha=0.0, 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='deprecated',
                      random_state=42, splitter='best')
```

- Visualisasi hasil DT seperti contoh sebelumnya

[25]:
```
from pydotplus import graph_from_dot_data
from sklearn.tree import export_graphviz

dot_data = export_graphviz(
        tree_reg,
        out_file=None,
        #feature_names=iris.feature_names[2:],
        #class_names=iris.target_names,
        #rounded=True,
        #filled=True
    )
graph = graph_from_dot_data(dot_data)
graph.write_png('tree_reg.jpg')
```

[25]:
```
True
```




Gambar 4.3. Decision tree untuk regresi.

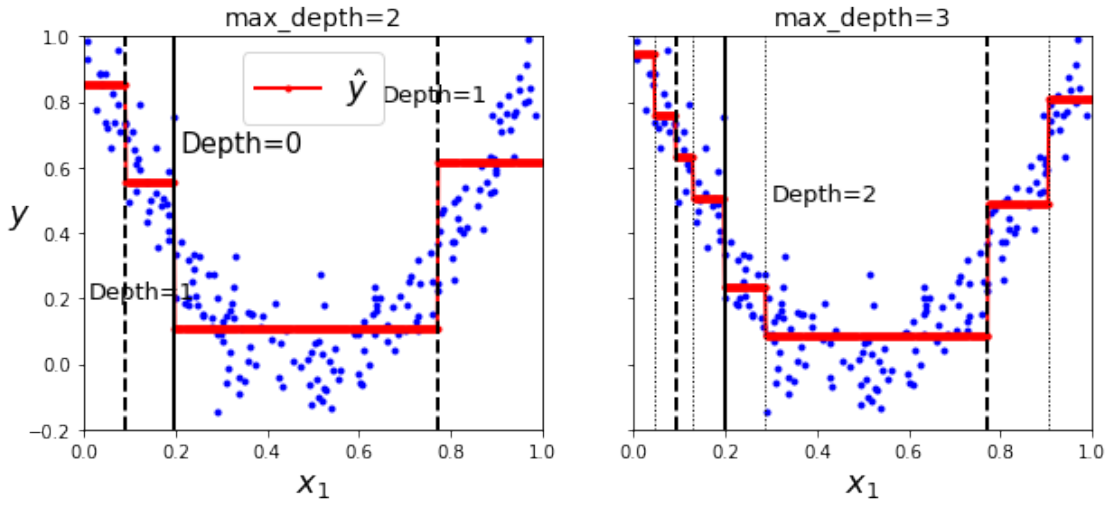
*Tree* yang dihasilkan terlihat mirip dengan *tree* klasifikasi pada contoh sebelumnya. Perbedaan utamanya adalah DT untuk regresi akan menghasilkan sebuah harga, sedangkan DT pada klasifikasi akan memprediksi *class* dari data.

Sebagai contoh bagaimana algoritma regresi bekerja, dimisalkan kita ingin melakukan prediksi untuk data baru dengan $x_1 = 0.6$. Jika kita lihat Gambar 4.3 dan mengikuti percabangan yang melalui *tree* maka kita akan sampai pada node terbawah *depth* = 2 kedua dari kanan dengan hasil prediksi value = 0.111. Value tersebut merupakan perata-rataan 110 sampel yang mencapai *leaf-node*, dengan harga *mean squared error* (MSE) sama dengan 0.015 pada 110 sampel tersebut.

Hasil dari model prediksi bisa dilihat pada gambar di bawah sebelah kiri. Jika digunakan `max_depth=3` maka akan diperoleh gambar di sebelah kanan. Jika diperhatikan maka pada kedua gambar terlihat bahwa harga prediksi untuk setiap *region* merupakan harga rata-rata target pada *region* tersebut. Algoritma bekerja dengan cara melakukan pemisahan (*splitting*) sehingga membuat kebanyakan *training instance* sedekat mungkin terhadap harga terprediksi.

[26]:
```
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt

tree_reg1 = DecisionTreeRegressor(random_state=42, max_depth=2)
tree_reg2 = DecisionTreeRegressor(random_state=42, max_depth=3)
tree_reg1.fit(X, y)
tree_reg2.fit(X, y)

def plot_regression_predictions(tree_reg, X, y, axes=[0, 1, -0.2, 1],␣
 ↪ylabel="$y$"):
    x1 = np.linspace(axes[0], axes[1], 500).reshape(-1, 1)
    y_pred = tree_reg.predict(x1)
    plt.axis(axes)
```

```
    plt.xlabel("$x_1$", fontsize=18)
    if ylabel:
        plt.ylabel(ylabel, fontsize=18, rotation=0)
    plt.plot(X, y, "b.")
    plt.plot(x1, y_pred, "r.-", linewidth=2, label=r"$\hat{y}$")

fig, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
plt.sca(axes[0])
plot_regression_predictions(tree_reg1, X, y)
for split, style in ((0.1973, "k-"), (0.0917, "k--"), (0.7718, "k--")):
    plt.plot([split, split], [-0.2, 1], style, linewidth=2)
plt.text(0.21, 0.65, "Depth=0", fontsize=15)
plt.text(0.01, 0.2, "Depth=1", fontsize=13)
plt.text(0.65, 0.8, "Depth=1", fontsize=13)
plt.legend(loc="upper center", fontsize=18)
plt.title("max_depth=2", fontsize=14)

plt.sca(axes[1])
plot_regression_predictions(tree_reg2, X, y, ylabel=None)
for split, style in ((0.1973, "k-"), (0.0917, "k--"), (0.7718, "k--")):
    plt.plot([split, split], [-0.2, 1], style, linewidth=2)
for split in (0.0458, 0.1298, 0.2873, 0.9040):
    plt.plot([split, split], [-0.2, 1], "k:", linewidth=1)
plt.text(0.3, 0.5, "Depth=2", fontsize=13)
plt.title("max_depth=3", fontsize=14)

#save_fig("tree_regression_plot")
plt.show()
```




Algoritma CART pada kasus ini bekerja seperti sebelumnya, kecuali objektif dari *cost function*

untuk melakukan *splitting* adalah meminimalkan *mean square error* (MSE), bukan meminimalkan *impurity*. Persamaan (4.4) merupakan *cost function* yang diminimalkan pada regresi.

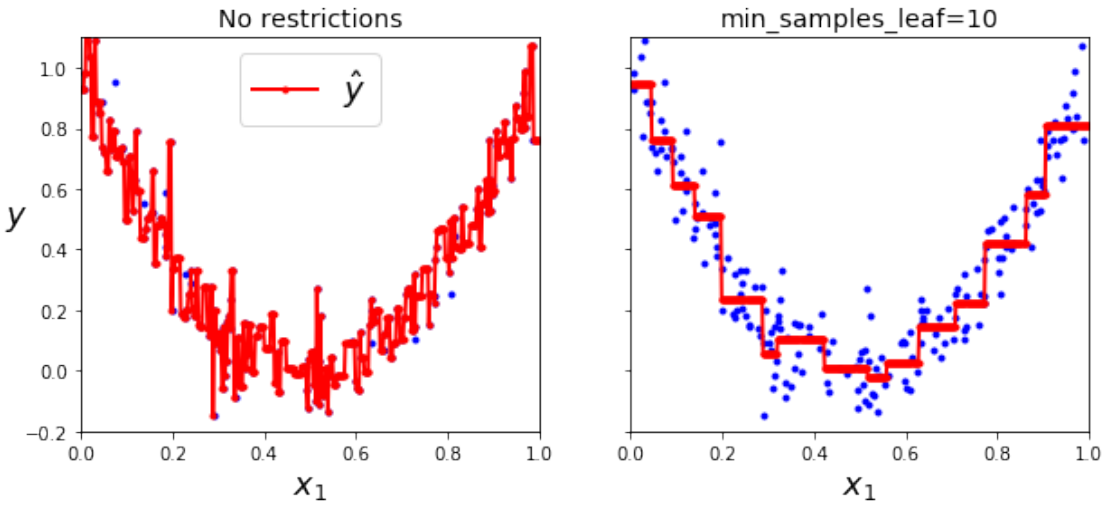
**Persamaan (4.4)** *Cost function* untuk regresi

$$J(k, t_k) = \frac{m_{kiri}}{m}\text{MSE}_{kiri} + \frac{m_{kanan}}{m}\text{MSE}_{kanan}$$

$$\text{dimana} \begin{cases} \text{MSE}_{node} = \sum_{i \in node} (\hat{y}_{node} - y^{(i)})^2 \\ \hat{y}_{node} = \frac{1}{m_{node}} \sum_{i \in node} y^{(i)} \end{cases}$$

**Catatan**. *Decision Tree* yang biasanya digunakan untuk klasifikasi, menjadi sangat mudah mengalami *overfitting* ketika digunakan untuk regresi. Tanpa menggunakan regularisasi (sebagai default) maka akan diperoleh gambar di sebelah kiri dimana *overfitting* terhadap training set terjadi. Dengan mengatur *hyperparameter* `min_samples_leaf=10` akan menghasilkan model yang lebih baik seperti ditunjukan pada gambar sebelah kanan.

[27]:
```
tree_reg1 = DecisionTreeRegressor(random_state=42)
tree_reg2 = DecisionTreeRegressor(random_state=42, min_samples_leaf=10)
tree_reg1.fit(X, y)
tree_reg2.fit(X, y)

x1 = np.linspace(0, 1, 500).reshape(-1, 1)
y_pred1 = tree_reg1.predict(x1)
y_pred2 = tree_reg2.predict(x1)

fig, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)

plt.sca(axes[0])
plt.plot(X, y, "b.")
plt.plot(x1, y_pred1, "r.-", linewidth=2, label=r"$\hat{y}$")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", fontsize=18, rotation=0)
plt.legend(loc="upper center", fontsize=18)
plt.title("No restrictions", fontsize=14)

plt.sca(axes[1])
plt.plot(X, y, "b.")
plt.plot(x1, y_pred2, "r.-", linewidth=2, label=r"$\hat{y}$")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.title("min_samples_leaf={}".format(tree_reg2.min_samples_leaf), fontsize=14)

#save_fig("tree_regression_regularization_plot")
plt.show()
```




## 2 *Ensemble Learning* dan *Random Forests*

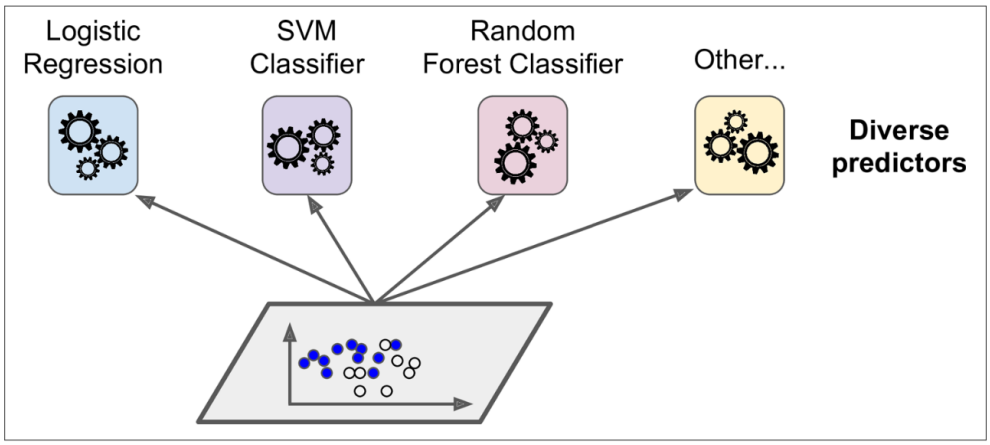
Jika dimisalkan kita menanyakan pertanyaan yang cukup kompleks ke banyak orang yang dipilih secara random, kemudian jawaban-jawaban mereka dikumpulkan dan digabungkan (agregasi). Pada banyak kasus kita akan menemukan jawaban hasil agregasi dari banyak orang tadi lebih baik dibanding dengan jawaban seorang ahli. Permasalahan ini disebut dengan *wisdom of crowd*. Hal yang serupa akan terjadi pada kasus prediksi di *machine learning*. Jika kita mengagregasikan prediksi-prediksi hasil dari satu grup prediktor (*classifier* atau *regressor*) maka kita akan sering mendapatkan hasil prediksi yang lebih baik dibandingkan dengan prediktor individu yang dianggap terbaik. Sebuah grup prediktor dapat kita sebut sebagai ***ensemble***. Sehingga teknik ini disebut dengan ***Ensemble Learning***, dan sebuah algoritma *ensemble learning* disebut dengan metode *ensemble*.

Sebagai contoh dari metode *ensemble*, dimisalkan kita melakukan training satu grup DT *classifier* dan masing-masing menggunakan subset data training yang random dan berbeda-beda. Untuk membuat prediksi, kita akan memperoleh hasil-hasil prediksi dari setiap individu DT, kemudian memutuskan kategori *class* yang mendapatkan *voting* terbanyak dari individu-individu DT tersebut. *Ensemble* dari DT seperti ini disebut dengan ***Random Forest***, yang kendatipun sederhana, tetapi termasuk salah satu algoritma *machine learning* yang sangat *powerful* sekarang ini. Pada bagian ini kita akan mempelajari metode-metode *ensemble* yang paling populer, diantaranya *bagging*, *boosting*, dan *stacking*. Sebagai catatan, *bagging* dan *boosting* akan dibahas pada Chapter ini, sedangkan *stacking* tidak akan dibahas (dapat dilihat dari buku-buku referensi atau di `Scikit-Learn:` Ensemble Methods).

### 2.1 *Voting Classifier*

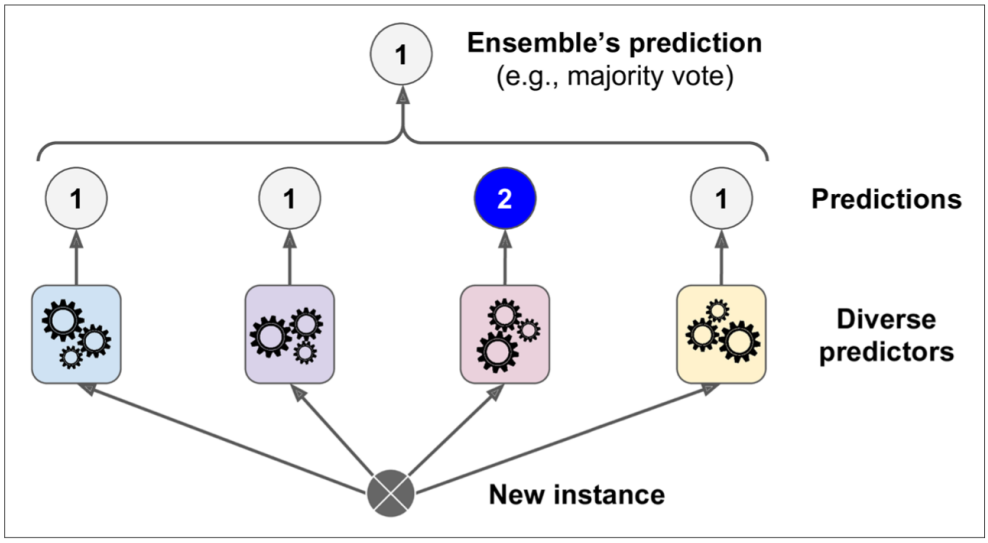
Dimisalkan kita telah melakukan training untuk beberapa *classifier* dan masing-masing mencapai akurasi 80%. *Classifier-classifier* tersebut misalkan adalah *classifier* regresi logistik, support vector machine (svm), *random forest*, *K-Nearest Neighbors*, dan mungkin ditambah yang lain, seperti yang ditunjukan pada Gambar 4.4. Cara yang paling sederhana untuk membuat *classifier* yang

lebih baik adalah dengan melakukan agregasi prediksi-prediksi dari setiap *classifier*, dan kemudian memutuskan kategori *class* yang mendapatkan *voting* terbanyak dari individu-individu *classifier* tersebut. *Classifier* yang berbasis *voting* paling banyak (*majority vote*) disebut dengan *hard voting classifier*, yang diilustrasikan pada Gambar 4.5.




Gambar 4.4. Training untuk satu grup classifier dari berbagai macam algoritma

*Hard voting classifier* seperti pada Gambar 4.5 sering mencapai keakuratan yang lebih baik dibandingkan dengan *classifier* individual terbaik pada grup tersebut. Bahkan pada kenyataannya, meskipun setiap *classifier* individu merupakan *weak learner* (hanya sedikit lebih baik dari *random guessing*), hasil *ensamble*-nya masih dapat menjadi *strong learner* (mencapai keakuratan tinggi) dengan syarat jumlah *weak learner* cukup banyak dan beragam.




Gambar 4.5. Ilustrasi hard voting classifier

**Catatan.** Metode *ensemble* akan bekerja dengan baik ketika prediktor-prediktor yang digunakan dalam grup tersebut se-independen mungkin (seberagam mungkin) satu sama lain. Untuk memperoleh keberagaman dapat digunakan algoritma-algoritma klasifikasi yang sangat berbeda. Cara ini akan menambah kemungkinan masing-masing membuat jenis-jenis error yang berbeda sehingga akan meningkatkan keakuratan algoritma hasil *ensemble* secara keseluruhan.

Kode program berikut membuat dan melatih (training) sebuah *voting classifier* pada `Scikit-Learn`, yang terdiri dari 3 *classifier* yang beragam.

[40]:
```
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)
```

[41]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(solver="lbfgs", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
svm_clf = SVC(gamma="scale", random_state=42)

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

[42]:
```
voting_clf.fit(X_train, y_train)
```

[42]:
```
VotingClassifier(estimators=[('lr',
                              LogisticRegression(C=1.0, class_weight=None,
                                                 dual=False, fit_intercept=True,
                                                 intercept_scaling=1,
                                                 l1_ratio=None, max_iter=100,
                                                 multi_class='auto',
                                                 n_jobs=None, penalty='l2',
                                                 random_state=42,
                                                 solver='lbfgs', tol=0.0001,
                                                 verbose=0, warm_start=False)),
                             ('rf',
                              RandomForestClassifier(bootstrap=True,
                                                     ccp_alpha=0.0,
                                                     class_weight=None,
                                                     crit...
                                                     oob_score=False,
                                                     random_state=42, verbose=0,
                                                     warm_start=False)),
                             ('svc',
                              SVC(C=1.0, break_ties=False, cache_size=200,
                                  class_weight=None, coef0=0.0,
                                  decision_function_shape='ovr', degree=3,
                                  gamma='scale', kernel='rbf', max_iter=-1,
```

```
                                probability=False, random_state=42,
                                shrinking=True, tol=0.001, verbose=False))],
                    flatten_transform=True, n_jobs=None, voting='hard',
                    weights=None)
```

[43]:
```
from sklearn.metrics import accuracy_score

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))
```

```
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.896
VotingClassifier 0.912
```

Dari hasil di atas terlihat, *voting* classifier mempunyai skor keakuratan tertinggi (91, 2%) dibandingkan dengan keakuratan *classifier-classifier* individu, yaitu regresi logistik = 86, 4%, *random forest* = 89, 6%, dan SVC = 89, 6%. Selain *hard voting classifier*, ada juga yang disebut *soft voting classifier* (silahkan cari pengertiannya dan cara melakukannya di Scikit-Learn).

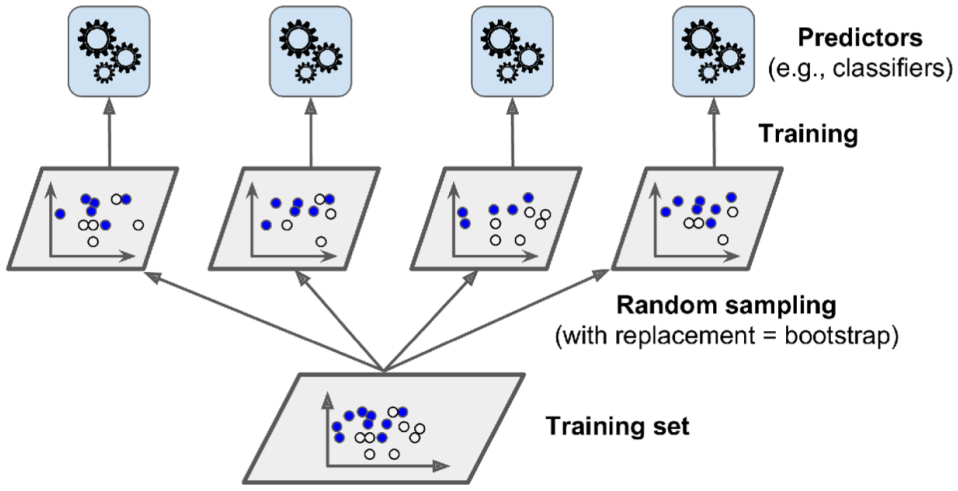
## 2.2 *Bagging* dan *Pasting*

Salah satu cara untuk mendapatkan keberagaman dari satu set *classifiers* adalah menggunakan algoritma-algoritma training yang berbeda, seperti yang telah didiskusikan sebelumnya. Pendekatan lain adalah dengan menggunakan algoritma yang sama untuk setiap prediktor yang ada, kemudian masing-masing ditraining dengan subset random dari dataset training. Ketika sampling dilakukan dengan pengganti (*sampling with replacement*), maka metode ini disebut dengan ***bagging*** (*bootstrap aggregating*). Ketika sampling dilakukan tanpa pengganti (*sampling without replacement*), maka metode ini disebut dengan ***pasting***.

*Bagging* dan *pasting* keduanya memperbolehkan *training instances* untuk disampel beberapa kali oleh beberapa prediktor, tetapi hanya *bagging* yang memperbolehkan *training instances* disampel beberapa kali oleh prediktor yang sama. Ilustrasi dari *bagging* dan *pasting* dapat dilihat pada Gambar 4.6.

Setelah semua prediktor ditraining, *ensamble* prediktor tersebut dapat melakukan prediksi untuk data baru (*new instance*) dengan mengagregasikan hasil-hasil prediksi dari semua prediktor pada *ensamble* tersebut. Fungsi agregasi yang digunakan biasanya adalah berupa mode statistik untuk kasus klasifikasi (seperti hasil prediksi yang sering muncul pada *hard voting classifier*), atau rata-rata untuk kasus regresi. Masing-masing prediktor individu mempunyai bias yang lebih tinggi dibandingkan jika masing-masing ditraining menggunakan keseluruhan dataset training (*original training set*). Tetapi, agregasi akan mengurangi bias dan variansi. Secara umum, hasil agregasi *ensamble* akan memiliki bias yang serupa tetapi variansi yang lebih rendah dibandingkan dengan prediktor-prediktor individu.

***Bagging* dan *Pasting* menggunakan *Scikit-Learn*** `Scikit-Learn` menawarkan API yang cukup sederhana untuk *bagging* dan *pasting* menggunakan *class* `BaggingClassifier` atau




Gambar 4.6. Bagging dan pasting melakukan traning beberapa prediktor menggunakan sampel-sampel yang dipilih secara acak dari training set

`BaggingRegressor` untuk regresi. Kode berikut menunjukan training dari sebuah *ensemble* yang terdiri dari 500 *decision tree* (DT) *classifiers*, dimana masing-masing ditraining menggunakan 100 *training instance* yang disampel (*with replacement*) secara random dari training set. Contoh ini menunjukan penggunaan `Scikit-Learn` untuk *bagging* (jika ingin menggunakan *pasting*, maka tinggal merubah `bootstrap=False`).

[44]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, random_state=42)# Untuk pasting --> set␣
↪bootstrap = False
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

[45]:
```
from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))
```

```
0.904
```

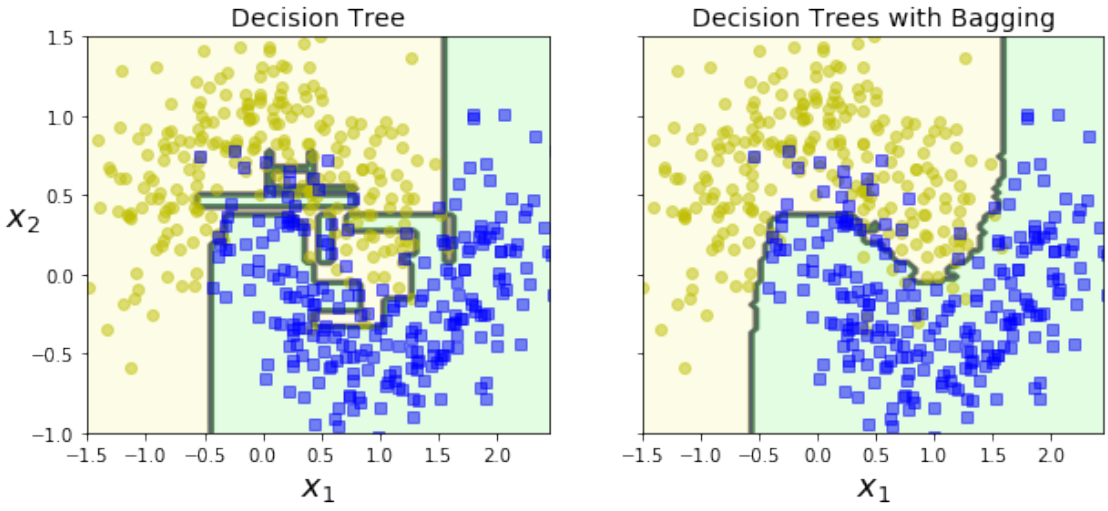
[46]:
```
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_tree))
```

```
0.856
```

[47]:
```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
```

```
import numpy as np

def plot_decision_boundary(clf, X, y, axes=[-1.5, 2.45, -1, 1.5], alpha=0.5,␣
 ↪contour=True):
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap)
    if contour:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
        plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", alpha=alpha)
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", alpha=alpha)
    plt.axis(axes)
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
```

[48]:
```
fix, axes = plt.subplots(ncols=2, figsize=(10,4), sharey=True)
plt.sca(axes[0])
plot_decision_boundary(tree_clf, X, y)
plt.title("Decision Tree", fontsize=14)
plt.sca(axes[1])
plot_decision_boundary(bag_clf, X, y)
plt.title("Decision Trees with Bagging", fontsize=14)
plt.ylabel("")
plt.show()
```



Gambar di atas membandingkan batas keputusan (*decision boundary*) dari satu *classifier* DT saja dengan *ensamble* yang terdiri dari 500 *classifier* DT, dimana keduanya ditraining dengan *moons* dataset. Seperti yang dapat kita lihat, hasil prediksi dari *ensemble* akan memiliki hasil yang jauh lebih baik dibandingkan dengan satu DT. Hasil *ensemble* mempunyai *bias* yang sebanding tetapi variansi yang lebih kecil (secara kasar mempunyai error yang sama pada data training tetapi *decision boundary* lebih beraturan).

*Bootstrapping* menghasilkan keragaman (*diversity*) pada subset-subset yang digunakan untuk mentraining setiap prediktor, sehingga *bagging* akan menghasilkan bias yang sedikit lebih tinggi dibandingkan dengan *pasting*. Tetapi keragaman yang lebih juga menghasilkan korelasi antara prediktor lebih kecil, sehingga variansi *ensemble* dapat dikurangi. Secara umum, *bagging* menghasilkan model yang lebih baik dibandingkan dengan *pasting*, sehingga *bagging* lebih banyak digunakan.

### 2.3 *Random Forests*

*Random forests* merupakan *ensemble* dari *decision tree* (DT) yang umumnya ditraining dengan metode *bagging* (sebagian kecil *pasting*), biasanya dengan melakukan setting `max_samples` sama dengan ukuran dari training set. Bisa saja kita membuat `BaggingClassifier` yang diberikan masukan `DecisionTreeClassifier` untuk menghasilkan algoritma *random forests*, tetapi `Scikit-Learn` sudah menyediakan *class* `RandomForestClassifier` yang lebih mudah digunakan dan dioptimasi untuk algoritma DT (terdapat *class* untuk melakukan regresi juga).

Algoritma *random forests* (RF) memberikan keacakan (*randomness*) lebih ketika mengembangkan *trees*. Tidak seperti algoritma DT sendiri yang mencari *feature* terbaik pada saat melakukan *splitting* node, RF akan mencari *best feature* diantara subset *feature* yang acak. Algoritma RF menghasilkan keberagaman *tree* yang lebih tinggi, yang akhirnya memperbesar bias untuk mendapatkan variansi yang lebih rendah. Hal ini menghasilkan model terbaik secara keseluruhan.

- ***Random Forests* dengan `BaggingClassifier` dan `DecisionTreeClassifier`**

[37]:
```
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)

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16,␣
 ↪random_state=42),
    n_estimators=500, max_samples=1.0, bootstrap=True, random_state=42)
```

[38]:
```
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

[53]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y_pred, y_test)
```

[53]:
```
0.904
```

- ***Random Forests* dengan *class* `RandomForestClassifier`**

*Class* `RandomForestClassifier` mempunyai semua *hyperparameter* pada algoritma `DecisionTreeClassifier` untuk mengontrol bagaimana *tree* berkembang, dan juga mempunyai semua *hyperparameter* pada *class* `BaggingClassifier` untuk mengontrol *ensemble*.

[54]:
```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,␣
↪random_state=42)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

[55]:
```
accuracy_score(y_pred_rf, y_test)
```

[55]:
```
0.912
```

Dari dua cara di atas, saat melakukan klasifikasi dengan metode *random forests*, kita bisa lihat bahwa menggunakan *class* `RandomForestClassifier` secara langsung jauh lebih sederhana dibandingkan dengan menggunakan *class* `BaggingClassifier` dan `DecisionTreeClassifier`. Hasil dari klasifikasi untuk dataset `make_moons`, *class* `RandomForestClassifier` menghasilkan akurasi 91,2% sedangkan dengan menggunakan `BaggingClassifier` dan `DecisionTreeClassifier` menghasilkan akurasi yang tidak jauh berbeda yaitu 90,4%.

Ketika *tree* dibuat berkembang pada *random forest*, hanya subset *feature* acak yang dipertimbangkan untuk *splitting* (seperti yang didiskusikan sebelumnya) pada setiap node. Adalah hal yang mungkin untuk membuat *trees* lebih acak lagi dengan menggunakan *threshold* random untuk setiap *feature* dibandingkan harus mencari threshold-threshold terbaik yang memungkinkan (seperti yang dilakukan DT biasa). *Forest* dengan keacakan *trees* yang lebih ekstrim disebut dengan *Extremely Randomized Trees Ensemble* (atau disingkat *Extra-Trees*). Sekali lagi, teknik ini akan menghasilkan bias yang lebih besar untuk memperoleh variansi yang lebih kecil. Hal ini juga membuat *Extra-Trees* akan ditraining lebih cepat dibandingkan dengan *random forest* biasa, karena mencari *best possible threshold* untuk setiap feature pada setiap node merupakan salah satu yang paling banyak mengkonsumsi waktu saat mengembangkan *tree*. Pada `Scikit-Learn`, *Extra-Trees* dapat didefinisikan dengan *class* `ExtraTreesClassifier` untuk klasifikasi dan `ExtraTreesRegressor` untuk regresi.

**Catatan.** Sangat sulit untuk menentukan dari awal mana yang lebih bagus, apakah `RandomForestClassifier` atau `ExtraTreesClassifier` untuk kasus tertentu. Secara umum, untuk mengetahuinya maka dicoba keduanya dan dibandingkan hasilnya dengan *cross validation* (tuning parameter bisa menggunakan *grid search*).

Keuntungan lain *random forests* adalah adanya kemungkinan untuk mengukur seberapa penting dari sebuah *feature* dibandingkan dengan *feature-feature* yang lain. `Scikit-Learn` mengukur level seberapa penting dari sebuah *feature* dengan cara melihat seberapa banyak node-node *tree* yang menggunakan *feature* tersebut dapat mengurangi *impurity* secara rata-rata (untuk keseluruhan semua *trees* pada *forest*). Lebih tepatnya adalah perata-rataan dengan cara pembobotan, dimana setiap bobot dari node sama dengan jumlah sampel training yang diasosiasikan dengannya. `Scikit-Learn` menghitung skor-skor ini secara otomatis untuk setiap feature setelah training,

kemudian menskalakan hasil sehingga penjumlahan semua level penting sama dengan 1 (normalisasi). Kita dapat mengakses hasilnya pada variabel `feature_importance_`. Contoh program berikut melakukan training untuk `RandomForestClassifier` dengan dataset Iris dan outputnya berupa level penting tidaknya sebuah *feature*. Terlihat bahwa, *feature-feature* yang paling penting adalah panjang *petal* (44%) dan lebar *petal* (42%), sementara panjang dan lebar *sepal* merupakan *feature* yang relatif tidak penting, berturut-turut 11% dan 2%.

[56]:
```
from sklearn.datasets import load_iris
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
rnd_clf.fit(iris["data"], iris["target"])
for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
    print(name, score)
```

```
sepal length (cm) 0.11249225099876375
sepal width (cm) 0.02311928828251033
petal length (cm) 0.4410304643639577
petal width (cm) 0.4233579963547682
```

[57]:
```
rnd_clf.feature_importances_
```

[57]:
```
array([0.11249225, 0.02311929, 0.44103046, 0.423358  ])
```

Jika kita juga melakukan training *random forest classifier* dengan dataset MNIST, dan plotting level penting tidaknya setiap piksel (*feature* dalam hal ini) maka akan diperoleh gambar berikut.

[58]:
```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1)
mnist.target = mnist.target.astype(np.uint8)
```

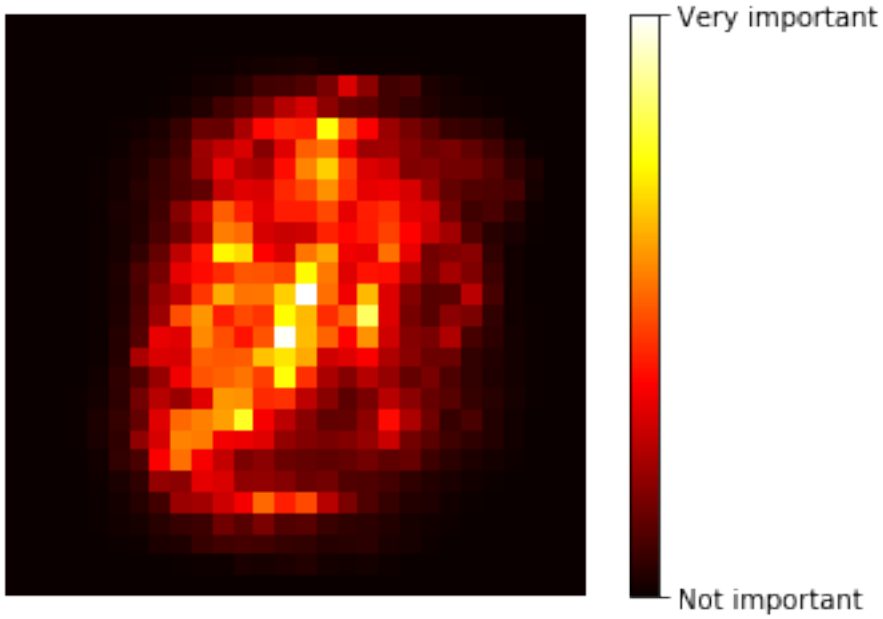
[59]:
```
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
rnd_clf.fit(mnist["data"], mnist["target"])
```

[59]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=42, verbose=0,
                       warm_start=False)
```

[62]:
```
import matplotlib as mpl
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = mpl.cm.hot,
               interpolation="nearest")
```

```
    plt.axis("off")
```

[64]:
```
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'])
plt.show()
```




Terlihat pada gambar di atas, *feature-feature* (dalam kasus ini piksel-piksel) yang paling penting berada di tengah-tengah, sedangkan yang berada di pinggir dianggap tidak penting. *Random forests* sangat praktis untuk mengetahui secara cepat *feature-feature* apa yang sebetulnya penting, terutama jika kita perlu melakukan seleksi *feature* (*feature selection*).

## 2.4 *Boosting*

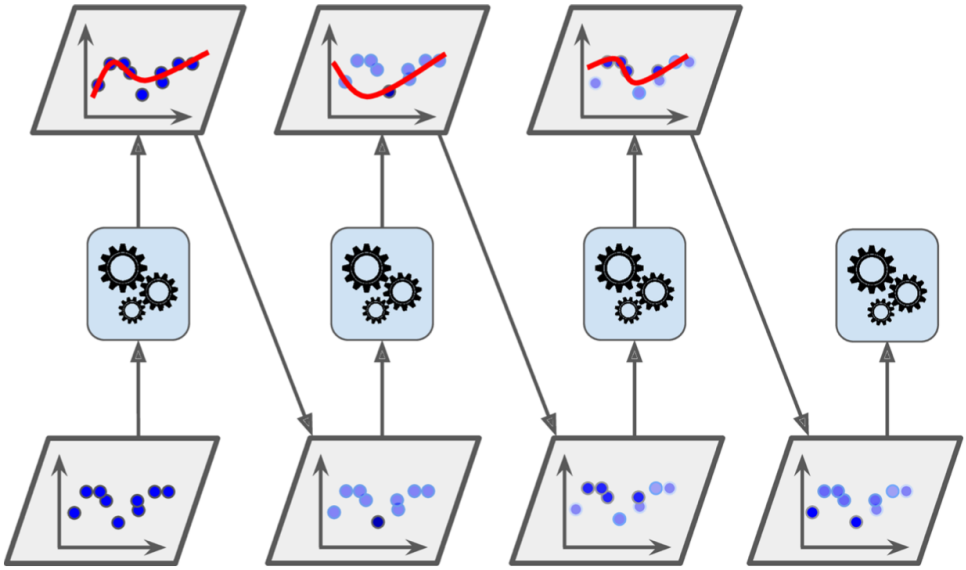
*Boosting* (atau disebut *hypothesis boosting*) mengacu pada metode-metode *ensemble* yang mengkombinasikan beberapa pembelajar yang lemah (*weak learners*) menjadi pembelajar yang kuat (*strong learner*). Ide dari metode *boosting* adalah melakukan training prediktor-prediktor secara sekuensial, dimana masing-masing akan memperbaiki hasil prediktor sebelumnya. Banyak metode-metode *boosting* yang ada, tetapi dua yang paling populer adalah *AdaBoost (Adaptive Boosting)* dan *Gradient Boosting*. Pada buku ini hanya akan dijelaskan algoritma *AdaBoost*. Untuk *gradient boosting* dapat dilihat pada buku-buku referensi.

- ***AdaBoost***

Salah satu cara sebuah prediktor baru untuk memperbaiki prediktor sebelumnya adalah dengan memberikan perhatian lebih pada *training instances* dimana prediktor sebelumnya mengalami *un-*

*derfitting*. Sehingga akan membuat prediktor baru lebih fokus pada kasus-kasus yang sulit. Teknik inilah yang digunakan oleh *AdaBoost*.

Sebagai contoh, ketika melakukan training *classifier AdaBoost*, algoritma pertama kali akan memberikan training untuk *classifier* basis (seperti *decision tree*) dan menggunakannya untuk memprediksi pada sebuah training set. Kemudian algoritma melanjutkan dengan menambahkan bobot relatif dari *training instances* yang diklasifikasikan salah (*misclassified*). Selanjutnya, training dilakukan untuk *classifier* kedua menggunakan bobot yang telah diupdate tadi, dan membuat prediksi lagi pada training set, update lagi bobot relatif dari *training instances*. Proses tersebut dilakukan berulang seperti yang ditunjukan oleh Gambar 4.7.



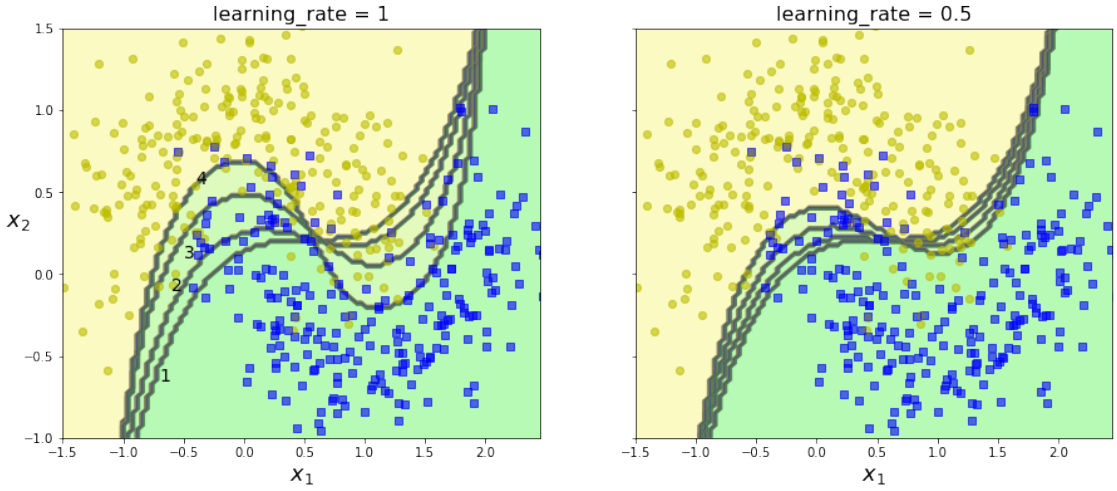
Gambar 4.7. Training secara sekuensial pada AdaBoost dengan update bobot relatif dari training instances

Program berikut adalah implementasi metode dari *AdaBoost* dimana *classifier* yang digunakan adalah *regularized support vector machine* (SVM) dengan kernel RBF (lihat Chapter V).

Program berikut mengilustrasikan proses pada Gambar 4.7 dengan menggunakan *regularized support vector machine* (SVM) dengan kernel RBF (lihat Chapter 5). Di bawah dapat dilihat juga gambar hasil eksekusi yang menunjukan batas keputusan (*decision boundary*) dari 4 prediktor berurutan hasil training secara sekuensial. *Classifier* pertama akan mendapatkan banyak *instances* yang diklasifikasikan salah, sehingga bobot akan di-*boosting*. *Classifier* kedua akan mendapatkan kinerja yang lebih baik untuk *instances* tadi, dan seterusnya. Plotting sebelah kanan merepresentasikan hal yang sama, tetapi menggunakan *learning rate* setengahnya, yaitu bobot *instances* yang diklasifikasikan salah akan di-*boosting* setengah kali dari yang sebelumnya pada setiap iterasi. Seperti yang dapat kita lihat, teknik *learning* secara sekuensial ini mirip dengan *gradient descent*, bedanya *gradient descent* melakukan update parameter dari prediktor untuk meminimalkan *cost function* pada setiap iterasi. Sedangkan *AdaBoost* menambahkan prediktor pada *ensemble* setiap kali iterasi dan menghasilkan kinerja yang lebih baik.

[77]:
```
m = len(X_train)

fix, axes = plt.subplots(ncols=2, figsize=(15,6), sharey=True)
```

```
for subplot, learning_rate in ((0, 1), (1, 0.5)):
    sample_weights = np.ones(m)
    plt.sca(axes[subplot])
    for i in range(4):
        svm_clf = SVC(kernel="rbf", C=0.05, gamma="scale", random_state=42)
        svm_clf.fit(X_train, y_train, sample_weight=sample_weights)
        y_pred = svm_clf.predict(X_train)
        sample_weights[y_pred != y_train] *= (1 + learning_rate)
        plot_decision_boundary(svm_clf, X, y, alpha=0.2)
        plt.title("learning_rate = {}".format(learning_rate), fontsize=16)
    if subplot == 0:
        plt.text(-0.7, -0.65, "1", fontsize=14)
        plt.text(-0.6, -0.10, "2", fontsize=14)
        plt.text(-0.5,  0.10, "3", fontsize=14)
        plt.text(-0.4,  0.55, "4", fontsize=14)
        #plt.text(-0.3,  0.90, "5", fontsize=14)
    else:
        plt.ylabel("")
plt.show()
```




Kita akan melihat sedikit lebih jauh dari algoritma *AdaBoost*. Setiap bobot *instance* $w^{(i)}$ awalnya akan diberi harga $1/m$. Prediktor pertama akan ditraining, dan bobot *error rate* $r_1$ dihitung pada training set seperti pada Persamaan (4.1).

***Persamaan (4.1).*** Pembobotan *error rate* dari prediktor ke-$j$

$$r_j = \frac{\sum_{i=1, \hat{y}_j^{(i)} \neq y^{(i)}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

dimana $\hat{y}_j^{(i)}$ adalah prediksi hasil prediktor ke-$j$ untuk *instance* ke-$i$

Setelah itu, bobot prediktor $\alpha_j$ akan dihitung dengan Persamaan (4.2), dimana $\eta$ adalah *hyper-paremeter* dari learning rate* (default = 1). Semakin akurat sebuah prediktor, maka bobotnya akan semakin besar. Jika prediktor berfungsi seperti *random guessing* (tidak akurat) maka bobot akan mendekati nol. Tetapi, jika lebih banyak salah (lebih tidak akurat dibanding *random guessing*) maka bobot akan negatif.

***Persamaan (4.2).*** Pembobotan *error rate* dari prediktor ke-$j$

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Selanjutnya, algoritma *AdaBoost* melakukan update bobot *instance* menggunakan Persamaan (4.3) yang melakukan update dengan *boosting* bobot *instance-instance* yang salah diklasifikasikan

***Persamaan (4.3).*** Aturan update untuk bobot

for $i = 1, 2, ..., m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{jika } \hat{y}_j^{(i)} = y_j^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{jika } \hat{y}_j^{(i)} \neq y_j^{(i)} \end{cases}$$

Kemudian semua bobot *instance-instance* akan dinormalisasi (dibagi dengan jumlah total bobot $\sum_{i=1}^{m} w^{(i)}$). Akhirnya, prediktor yang baru tergabung akan ditraining menggunakan bobot-bobot yang telah diupdate, dan semua proses keseluruhan akan diulang. Algoritma akan berhenti ketika jumlah prediktor yang diinginkan telah tercapai, atau prediktor yang dianggap sempurna telah ditemukan. Untuk melakukan prediksi *AdaBoost* menghitung prediksi-prediksi dari semua prediktor dan membobotinya dengan bobot prediktor $\alpha_j$ dari Persamaan (4.2). *Class* hasil klasifikasi adalah *class* yang mendapatkan vote terboboti secara mayoritas, seperti pada Persamaan (4.4).

***Persamaan (4.4).*** Prediksi *AdaBoost*

$$\hat{y}(x) = \underset{k}{\operatorname{argmax}} \sum_{j=1, \hat{y}_j(x)=k}^{N} \alpha_j$$

dimana $N$ adalah jumlah prediktor.

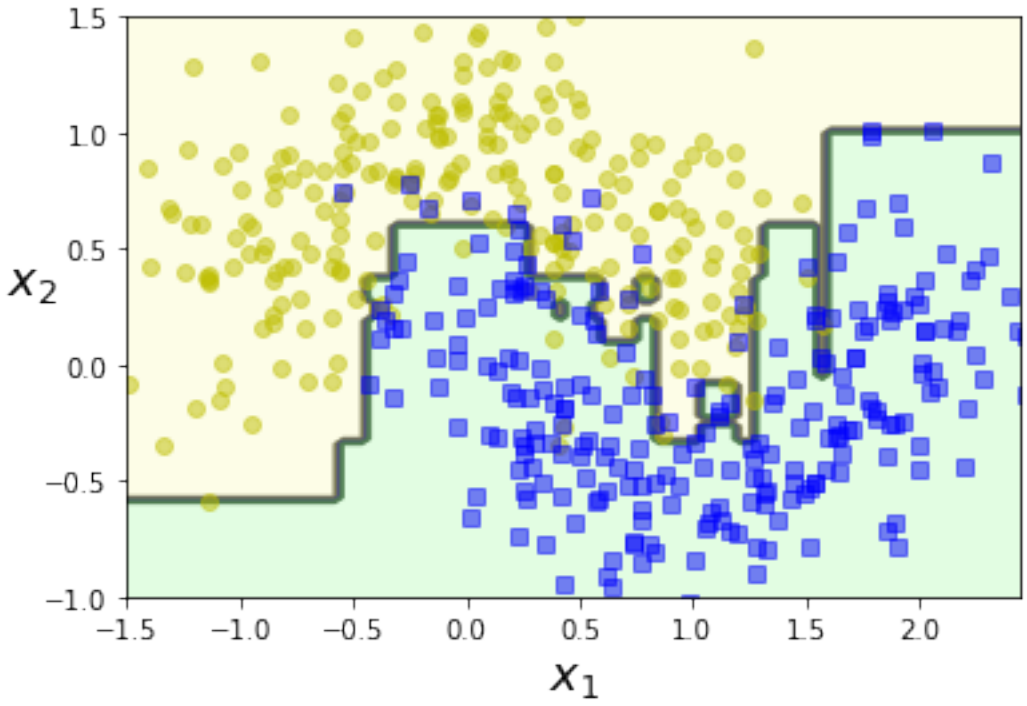
Program berikut melakukan training sebuat *classifier AdaBoost* berdasarkan 200 *classifier decision stumps* (*class* `DecisionTreeClassifier`). Maksud dari *decision stumps* adalah *Decision Tree* dengan `max_depth = 1`(lihat bagian A pada Chapter ini). Dengan kata lain *Tree* yang dibuat hanya mempunyai sebuah *decision node* dan dua *leaf node*.

[78]:
```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

[78]:
```
AdaBoostClassifier(algorithm='SAMME.R',
                   base_estimator=DecisionTreeClassifier(ccp_alpha=0.0,
                                                         class_weight=None,
                                                         criterion='gini',
```

```
                                                    max_depth=1,
                                                    max_features=None,
                                                    max_leaf_nodes=None,
    min_impurity_decrease=0.0,
    min_impurity_split=None,
                                                    min_samples_leaf=1,
                                                    min_samples_split=2,
    min_weight_fraction_leaf=0.0,
                                                    presort='deprecated',
                                                    random_state=None,
                                                    splitter='best'),
                       learning_rate=0.5, n_estimators=200, random_state=42)
```

[79]:
```
plot_decision_boundary(ada_clf, X, y)
```




`Scikit-Learn` menggunakan versi *multiclass* dari *AdaBoost* yang disebut dengan SAMME (*Stage-wise Additive Modeling using a Multiclass Exponential Loss Function*). Ketika hanya dua *class* untuk diklasifikasikan, maka SAMME ekivalen dengan *AdaBoost*.