lab6-exercises

November 21, 2022

1 Lab 6: Decision Trees and Random Forests

In this lab you'll train a decision tree classifier and a random forest classifier. You'll do so on both synthetic and real data.

Run the code cell below to import the required packages.

[143]:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import sklearn
import sklearn.tree          # For DecisionTreeClassifier class
import sklearn.ensemble      # For RandomForestClassifier class
import sklearn.datasets      # For make_circles
import sklearn.metrics       # For accuracy_score
```

2 1. Fitting a Random Forest classifier to synthetic data

Exercises 1.1–1.4 ask you to apply scikit-learn's decision tree classifier (**sklearn.tree.DecisionTreeClassifier**) and random forest classifier (**sklearn.ensemble.RandomForestClassifier**) to synthetic data.

Run the code cell below to define some useful functions for plotting data and predictions.

[144]:

```
def plot_data(X, y):
    """Plots a toy 2D data set. Assumes values in range [-3,3] and at most 3␣
 ↪classes."""
    plt.plot(X[y==0,0], X[y==0,1], 'ro', markersize=6)
    plt.plot(X[y==1,0], X[y==1,1], 'bs', markersize=6)
    plt.plot(X[y==2,0], X[y==2,1], 'gx', markersize=6, markeredgewidth=2)
    plt.xlim([-3, 3])
    plt.ylim([-3, 3])
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.gca().set_aspect('equal')

def plot_predict(model):
    """
    Plots the model's predictions over all points in range 2D [-3, 3].
    Assumes at most 3 classes.
    """
    extent = (-3, 3, -3, 3)
    x1min, x1max ,x2min, x2max = extent
    x1, x2 = np.meshgrid(np.linspace(x1min, x1max, 100), np.linspace(x2min,␣
 ↪x2max, 100))
    X = np.column_stack([x1.ravel(), x2.ravel()])
    y = model.predict(X).reshape(x1.shape)
    cmap = matplotlib.colors.ListedColormap(['r', 'b', 'g'])
    plt.imshow(y, extent=extent, origin='lower', alpha=0.4, vmin=0, vmax=2,␣
 ↪cmap=cmap, interpolation='nearest')
    plt.xlim([x1min, x1max])
    plt.ylim([x2min, x2max])
    plt.gca().set_aspect('equal')

def plot_class_probability(model, class_index):
    """
    Plots the model's class probability for the given class {0,1,2}
    over all points in range 2D [-3, 3]. Assumes at most 3 classes.
    """
    extent = (-3, 3, -3, 3)
    x1min, x1max ,x2min, x2max = extent
    x1, x2 = np.meshgrid(np.linspace(x1min, x1max, 100), np.linspace(x2min,␣
 ↪x2max, 100))
    X = np.column_stack([x1.ravel(), x2.ravel()])
    p = model.predict_proba(X)[:,class_index].reshape(x1.shape)
    colors = [[1, 0, 0], [0, 0, 1], [0, 1, 0]]
    cmap = matplotlib.colors.ListedColormap(np.linspace([1, 1, 1],␣
 ↪colors[class_index], 50))
    plt.imshow(p, extent=extent, origin='lower', alpha=0.4, vmin=0, vmax=1,␣
 ↪cmap=cmap)
    plt.xlim([x1min, x1max])
    plt.ylim([x2min, x2max])
    plt.gca().set_aspect('equal')
```

2.0.1 Exercise 1.1 — Train and inspect a small decision tree (2 points, 2 classes)

Read the documentation for the *DecisionTreeClassifier*'s fit method. Notice that the y vector should contain integer class labels. You are asked to build the small 2D training set below:

$$X = \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Write a few lines of code to 1. Define the training set above in two variables X and y. 2. Train a decision tree classifier on X and y. Use argument *random_state*=0. 3. Plot the decision tree predictions and the data (use *plot_predict* and *plot_data* from preamble). 4. Plot the decision tree itself (use **sklearn.tree.plot_tree**); pass `feature_names=['x1', 'x2']` as an argument.

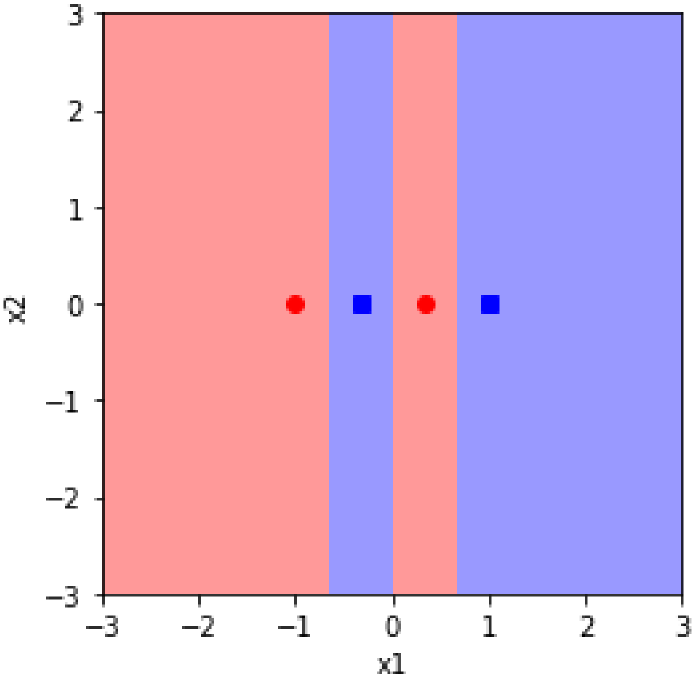
You should end up with a plot showing the data and the decision surface between classes 0 (red) and class 1 (blue). You should see a binary decision tree diagram depicting a tree of height 1 that splits the feature space using the first variable (`x1`) at threshold 0.

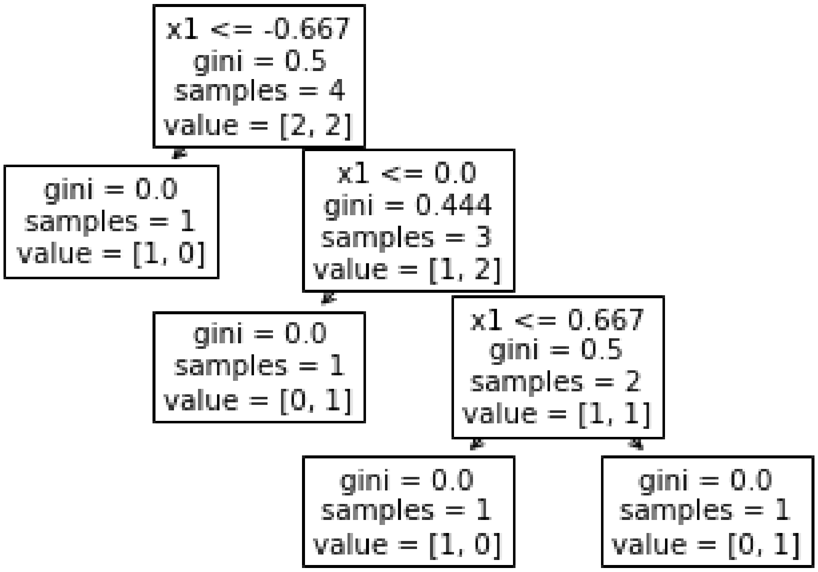
Tip 1: If you want a single code cell to generate multiple figures, call `plt.figure()` to tell Matplotlib that you want the subsequent plotting commands to generate a new, separate figure from any previous plotting commands.

Tip 2: If the last line of your code cell returns a value, it will be printed as the `Out` of the cell before the plots are shown. Some times you don't care about this 'final' value, for example if it is a string or some object you don't need printed. If you want to suppress the cell's `Out` value, add a semicolon (`;`) to the end of the last line of code in the cell.

[161]:

```python
X, y = np.array([[-1, 0], [1, 0], [-1/3, 0], [1/3, 0]]), np.array([0, 1, 1, 0])

dt = sklearn.tree.DecisionTreeClassifier(random_state=0).fit(X, y)

plot_predict(dt)
plot_data(X, y)

plt.figure()
sklearn.tree.plot_tree(dt, feature_names=["x1", "x2"]);
```





Once you have things working, **add two data points** to your training set: * $\mathbf{x}_3 = (-\frac{1}{3}, 0)$ with class label $y_3 = 1$ (blue), and * $\mathbf{x}_4 = (\frac{1}{3}, 0)$ with class label $y_4 = 0$ (red).

Re-run your code cell above and make sure you understand how the splits and thresholds you see in the tree correspond to the decision region shown.

Note: If a decision tree node is shown as having *value*=[1,2], it means that node's region (before splitting) contains exactly one training point from class 0 and two training points from class 1. The root note thus 'contains' the entire training set.

2.0.2 Exercise 1.2 — Train and inspect a small decision tree (3 points, 3 classes)

Repeat Exercise 1.1 but with the following changes: 1. To the original $\mathbf{x}_1$ and $\mathbf{x}_2$, add a third training point $\mathbf{x}_3 = (0, -2)$ with class label $y_3 = 2$. 2. Print the **feature_importances_** attribute of your trained decision tree. Intuitively, the feature importance is 0.0 if a feature is not used at all, or 1.0 if it is the only feature needed to make decisions.

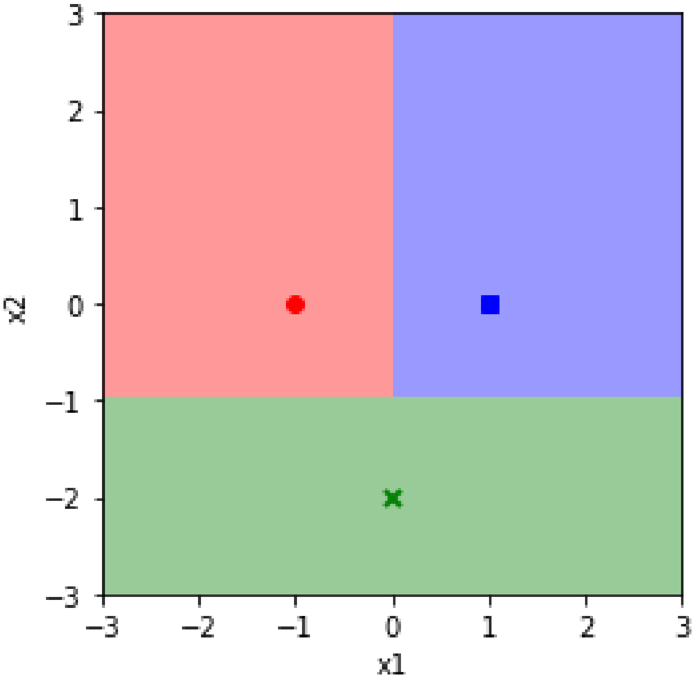
This time you should see a binary decision tree of height 2, where the first split is done by thresholding the second feature (`X[1]`) and the second split is done by thresholding the first feature (`X[0]`). If a node has *value*=[1,1,0], it means that node's region (say, the red+blue region) contains exactly one training point from class 0, one training point from class 1, and zero training points from class 2.

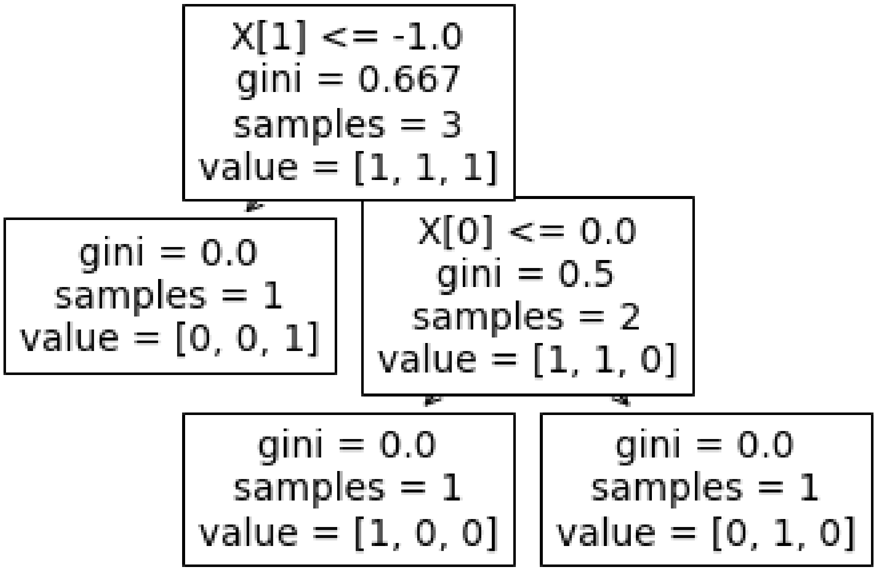
3. Try incrementing *random_state* from 0 up to 9. How many distinct decision trees did you observe? Do the 'feature importances' make intuitive sense, given the trees that you observed?

[146]:
```
X, y = np.array([[-1, 0], [1, 0], [0, -2]]), np.array([0, 1, 2])

dt = sklearn.tree.DecisionTreeClassifier(random_state=0).fit(X, y)
```

```
print("Feature importances:", dt.feature_importances_)

plot_predict(dt)
plot_data(X, y)

plt.figure()
sklearn.tree.plot_tree(dt);
```

```
Feature importances: [0.5 0.5]
```





2.0.3 Exercise 1.3 — Train and inspect a small random forest (3 points, 3 classes)

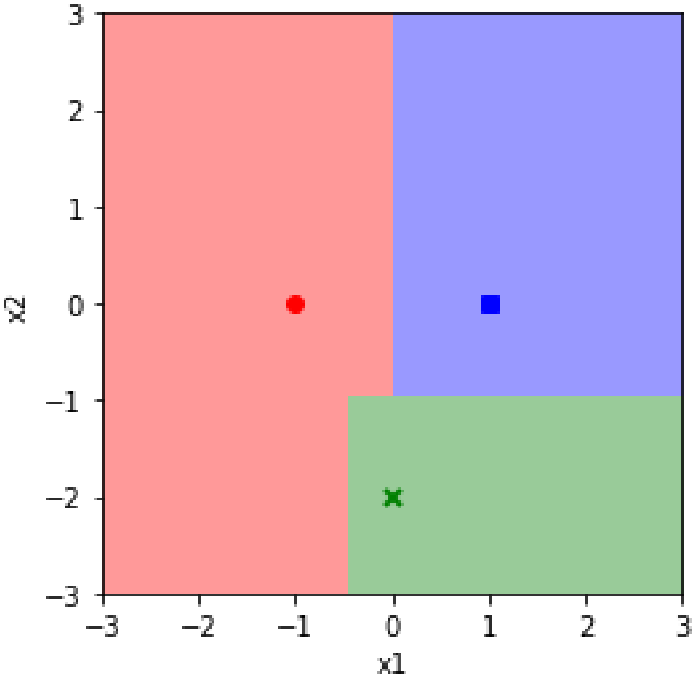
The goal of this exercise is to show you how a random forest is a collection of decision trees.

Repeat Exercise 1.2 but this time train using a random forest classifier (**sklearn.ensemble.RandomForestClassifier**) on the same 3-point data set. Use *random_state*=0 and *n_estimators*=3. You should see a decision region that still has axis-aligned boundaries, but different from Exercise 1.2.

[147]:

```
X, y = np.array([[-1, 0], [1, 0], [0, -2]]), np.array([0, 1, 2])

rf = sklearn.ensemble.RandomForestClassifier(random_state=0, n_estimators=3).
 ↪fit(X, y)

print("Feature importances:", rf.feature_importances_)

plot_predict(rf)
plot_data(X, y)
```

```
Feature importances: [0.66666667 0.33333333]
```

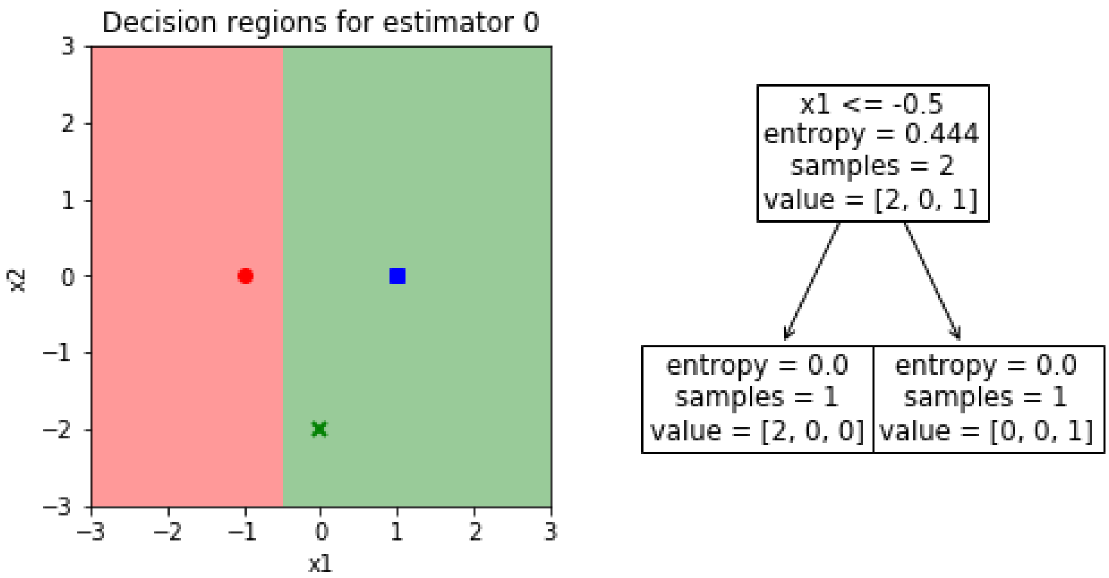


A random forest builds multiple decision trees where each decision tree (each 'estimator') is trained on a different "re-samplings" of the training data. Specifically, decision tree j is trained on a new training set $\mathcal{D}_j$ that is built by sampling N pairs $(\mathbf{x}_i, y_i)$ from the original N training examples in $\mathcal{D}$. The sampling is done "with replacement," meaning that the new training set $\mathcal{D}_j$ may contain duplicates and/or be missing some of the original data.

With the *random_state* you have chosen, the three re-samplings of $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), (\mathbf{x}_3, y_3)\}$ are: * $\mathcal{D}_1 = \{(\mathbf{x}_1, y_1), (\mathbf{x}_1, y_1), (\mathbf{x}_3, y_3)\}$ (red, red, green). * $\mathcal{D}_2 = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), (\mathbf{x}_2, y_2)\}$ (red, red, blue). * $\mathcal{D}_3 = \{(\mathbf{x}_2, y_2), (\mathbf{x}_2, y_2), (\mathbf{x}_3, y_3)\}$ (blue, blue, green).

Each individual decision tree is not a good classifier. To make a prediction, the random forest collects a prediction from each tree, and returns the class with the most "votes." The winner of these votes tends to be a good classification.

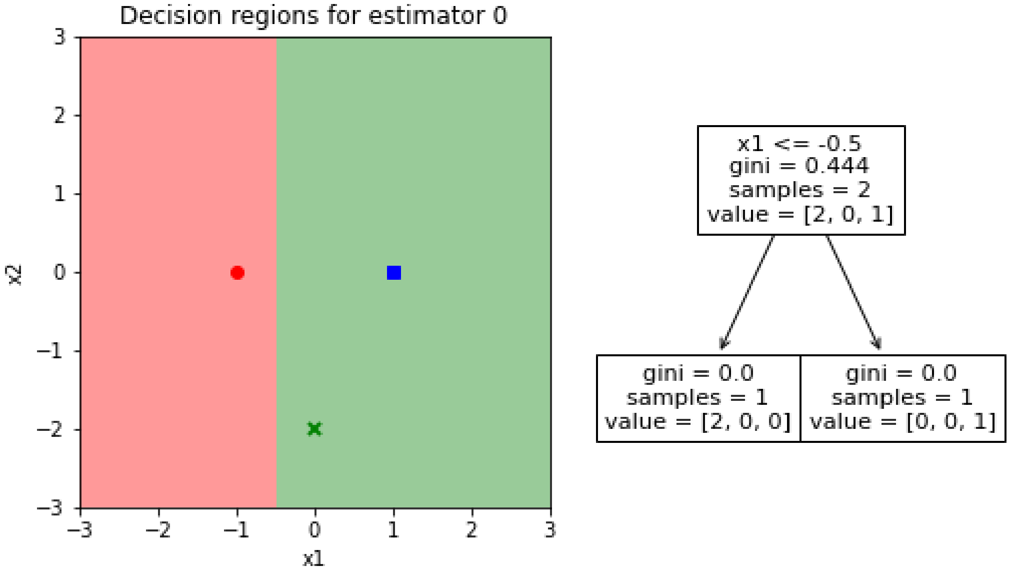
Plot the decision region and decision tree for each of the three decision trees that comprise the random forest. Read about the *estimators_* attribute in the *RandomForestClassifier* documentation. Your answer should use a for-loop *over the estimators*, generating a plot for each one. Your code should generate three figures, where each figure contains two subplots (use Matplotlib's subplot function). Your first pair should look like this:

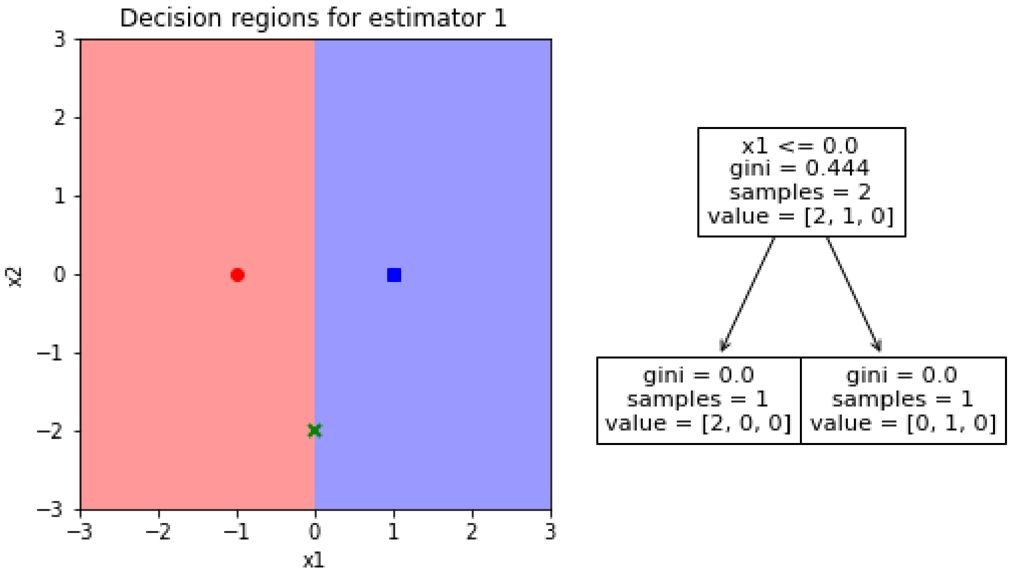


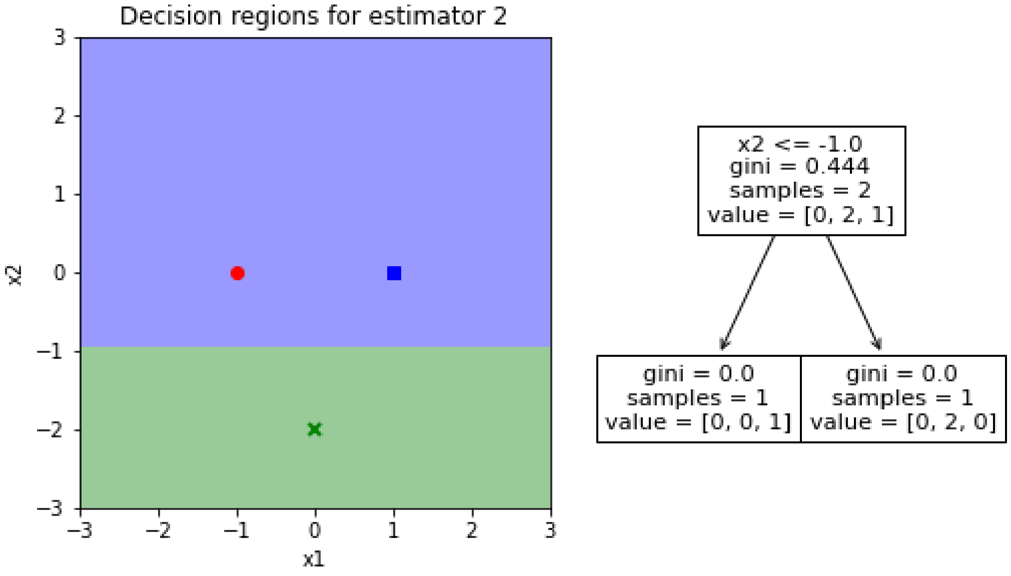
Hint: Remember to use `plt.figure()` to start a new figure. Pass argument `figsize=(8,4)` to the *figure* function to make a figure that's twice as wide as it is tall.

[148]:

```
for i in range(len(rf.estimators_)):
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plot_predict(rf.estimators_[i])
    plot_data(X, y)
    plt.title("Decision regions for estimator {}".format(i))
    plt.subplot(1, 2, 2)
    sklearn.tree.plot_tree(rf.estimators_[i], feature_names=["x1", "x2"])
    plt.tight_layout()
```

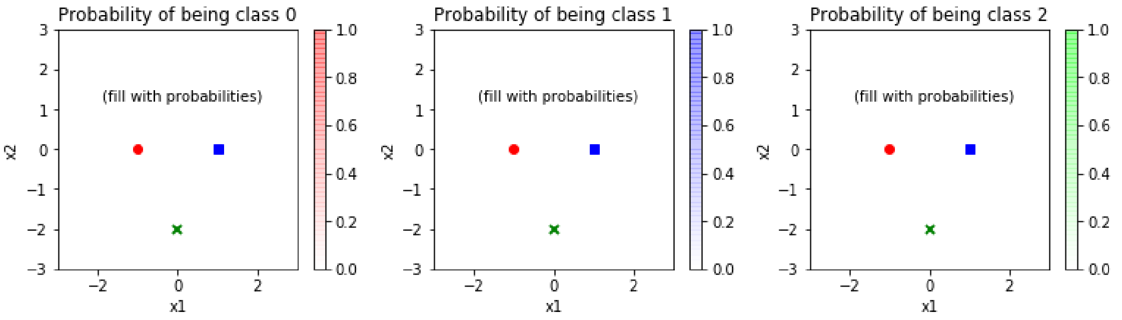






Each of the above trees ‘votes’ to determine the final class for every point in input space. The next step asks you to visualize how the “votes” from these trees add up and contribute to “class probabilities.”

Plot the class probabilities reported by the *RandomForestClassifier* instance that you trained. Use Matplotlib’s *figure*, *subplot*, and *colorbar* functions to create a single figure with three rows (one per class). For each subplot, use the *plot_class_probability* function (see preamble) to plot the heatmap of probablities. (This function calls *RandomForestClassifier*'s **predict_proba** method.) Your plot should look like this, but with the probabilities filled in:

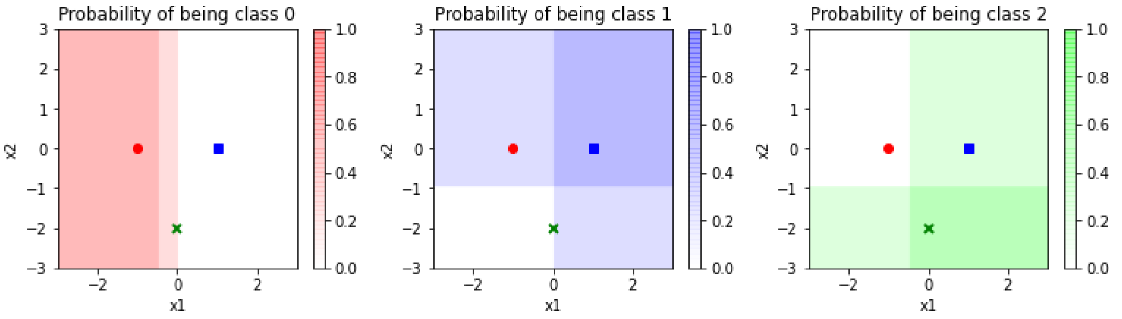


It's important not to confuse the number of trees with the number of classes, which in this case are both three.

Tip: As the figure you need to generate has three subplots in a row, I suggest setting *figsize=(13,3)*.

[149]:

```
plt.figure(figsize=(13, 3))

for i in range(3):
    plt.subplot(1, 3, i + 1)
    plot_class_probability(rf, i)
    plt.colorbar()
    plot_data(X, y)
    plt.title("Probability of being class {}".format(i))
```



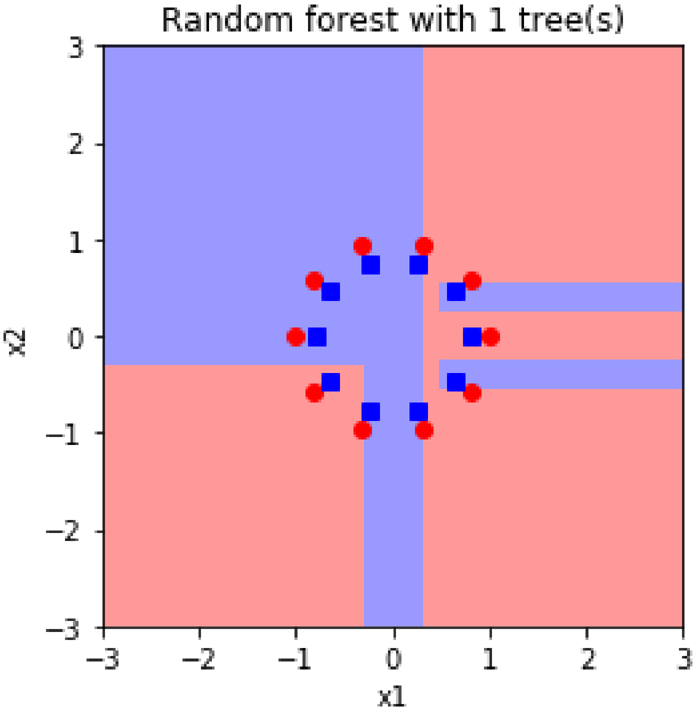
It is important to understand how the decision tree predictions lead to these probabilities. Inspect the shape of the decision regions proposed by each individual decision tree, and then ensure that you understand their correspondence with the class probabilities above.

Finally, once you understand how the three decision trees are combined, **re-run all code cells of Exercise 1.3** but use *n_estimators=10* so that there are now 10 decision trees. Notice how the final class probabilities change.

2.0.4 Exercise 1.4 — Train and inspect a small random forest on a synthetic pattern

You are asked to train a random forest classifier on a synthetic binary classification data set. Use the **sklearn.datasets.make_circles** function to create two concentric circles.

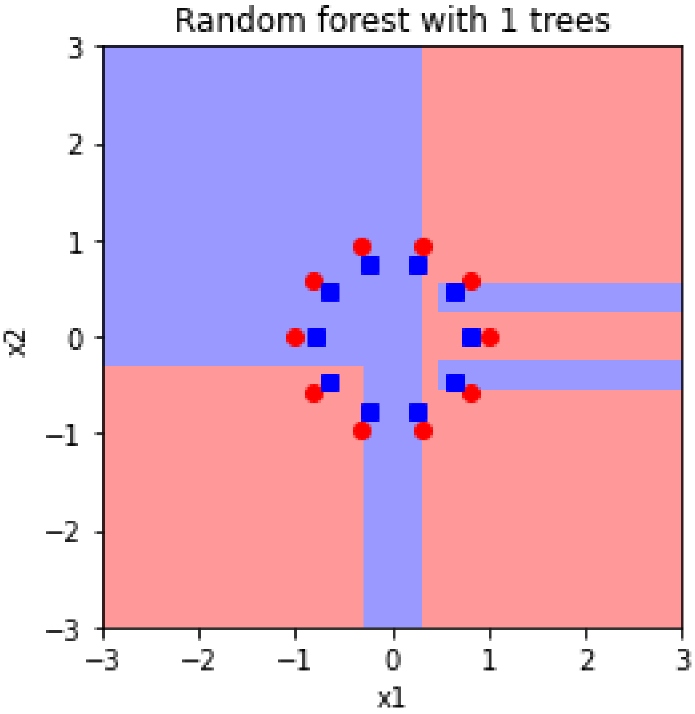
Write a few lines of code to: 1. Generate the circle data, with each circle made from 10 points. Use *random_state*=0. 2. For each *n_estimators* in $\{1, 2, 4, 8, 16, 32, 64\}$, train a random forest (use *random_state*=0) and plot its decision regions.

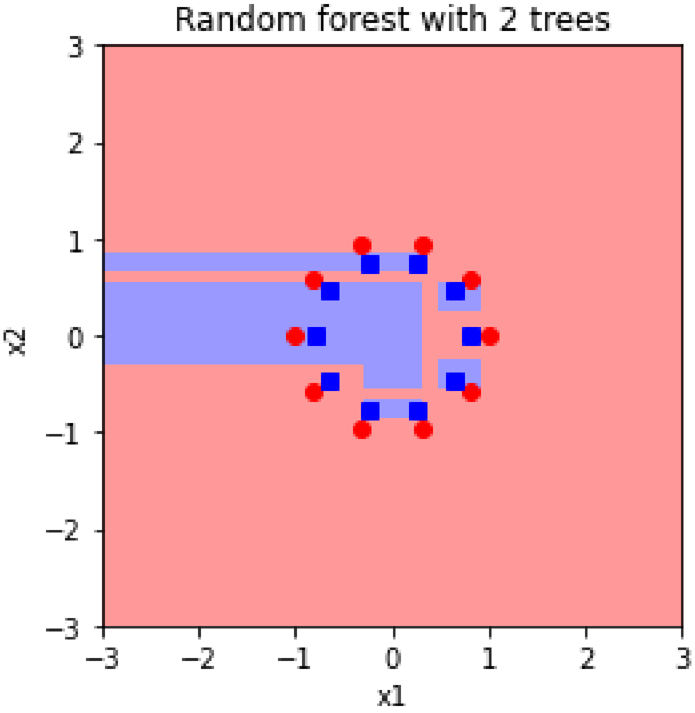


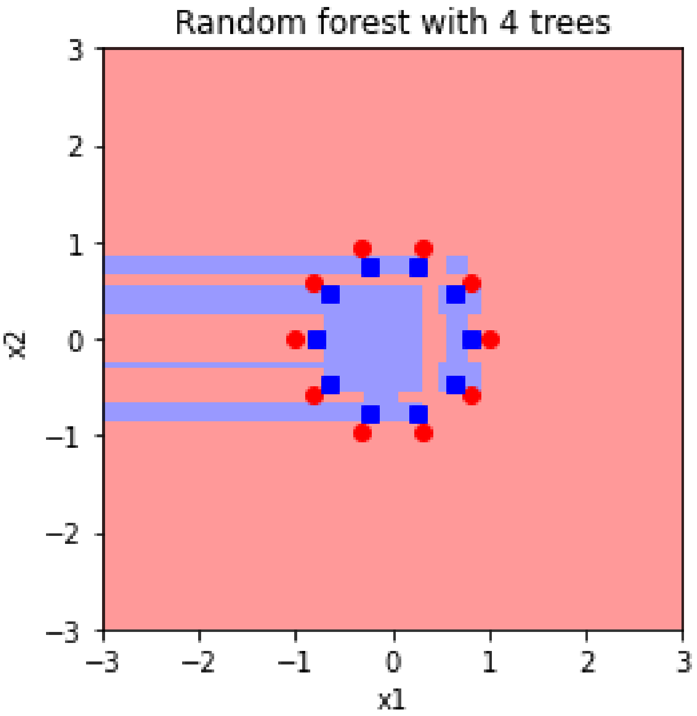
You first plot should look like this:

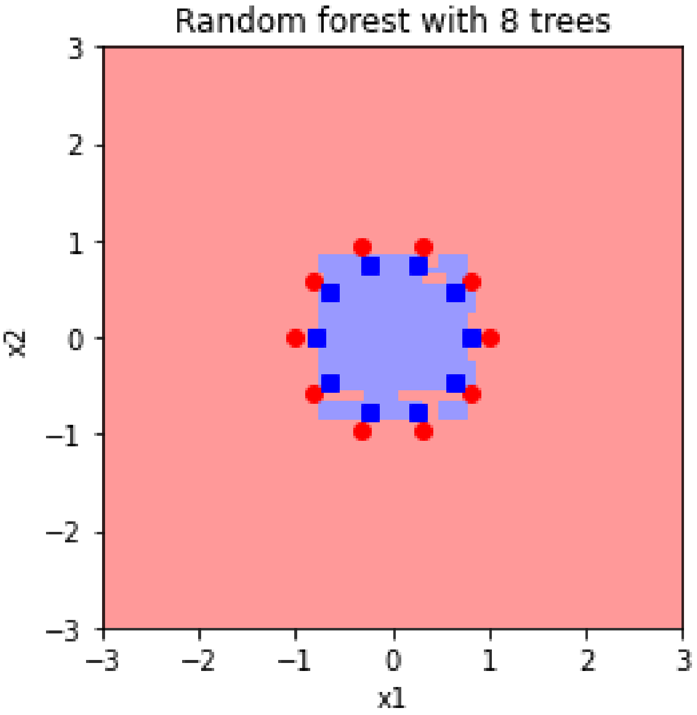
You should see a progression, where as the predictions from "more trees" are averaged together, the resulting decision regions tend to be better.

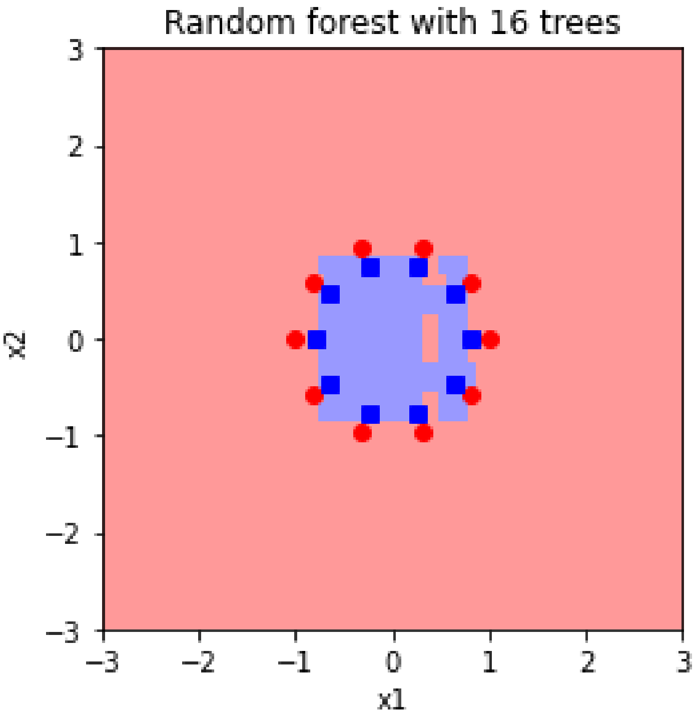
```
[150]: (X, y) = sklearn.datasets.make_circles(n_samples=20, random_state=0)

for n in [2 ** i for i in range(7)]:
    rf = sklearn.ensemble.RandomForestClassifier(random_state=0,␣
 ↪n_estimators=n).fit(X, y)
    plt.figure()
    plot_predict(rf)
    plot_data(X, y)
    plt.title("Random forest with {} trees".format(n))
```

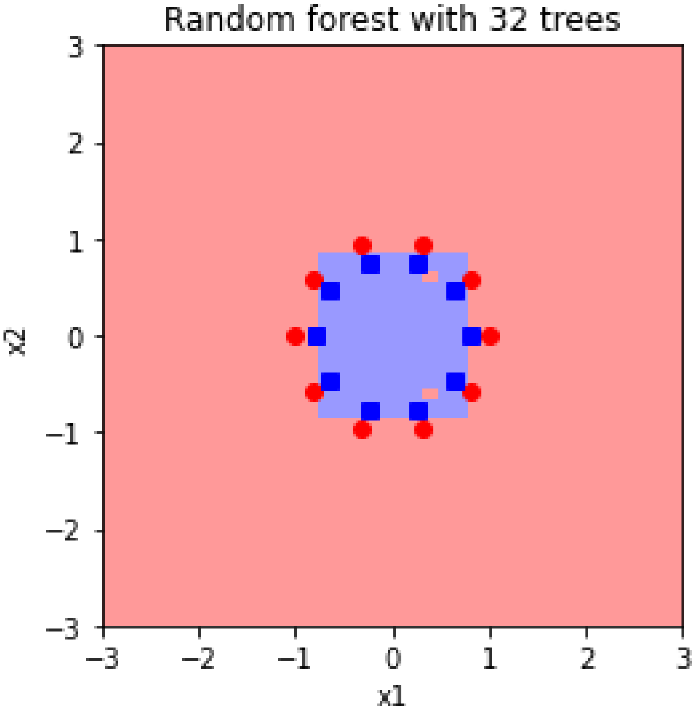

Random forest with 1 trees
3
2
1
0
−1
−2
−3
x2
−3
−2
−1
0
1
2
3
x1

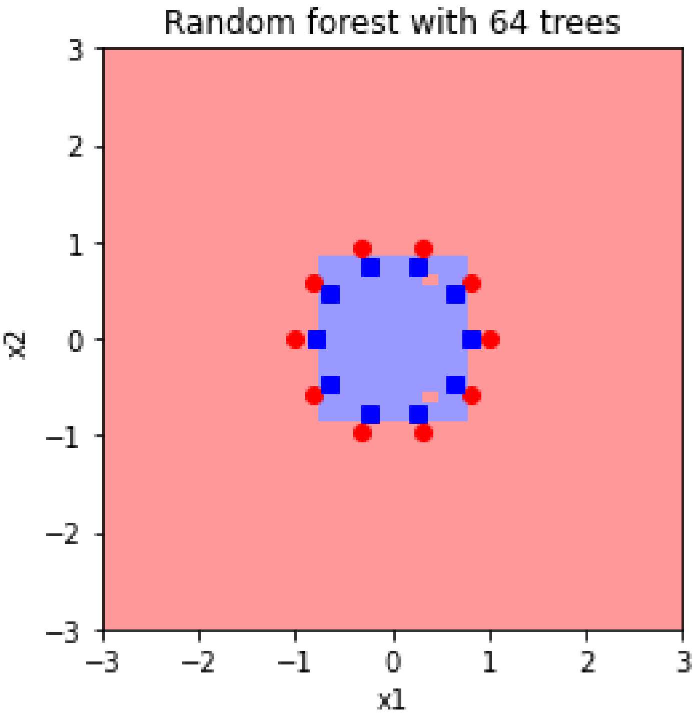

Random forest with 2 trees
3
2
1
0
−1
−2
−3
x2
−3
−2
−1
0
1
2
3
x1


Random forest with 4 trees
3
2
1
0
−1
−2
−3
x2
−3
−2
−1
0
1
2
3
x1

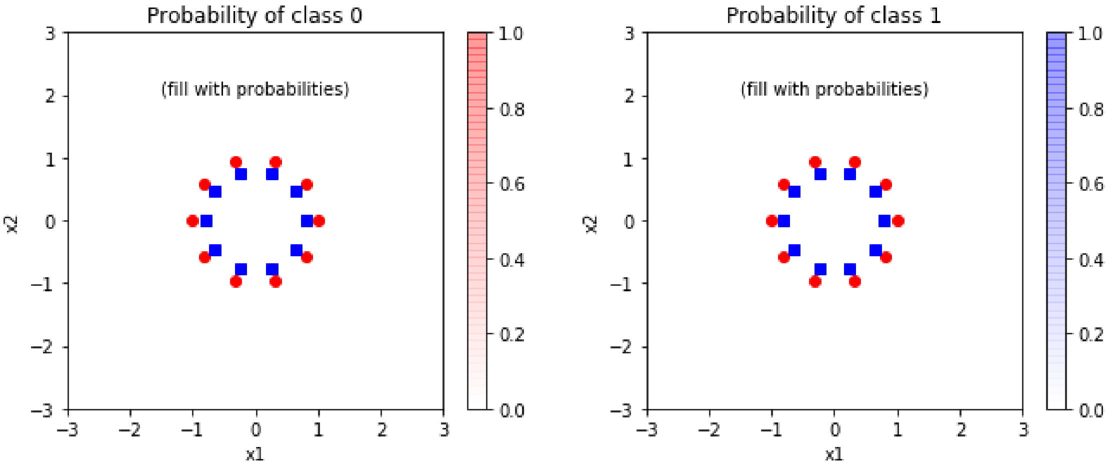

Random forest with 8 trees
3
2
1
0
−1
−2
−3
x2
−3
−2
−1
0
1
2
3
x1


Random forest with 16 trees
3
2
1
0
−1
−2
−3
x2
−3
−2
−1
0
1
2
3
x1


Random forest with 32 trees
3
2
1
0
−1
−2
−3
x2
−3
−2
−1
0
1
2
3
x1

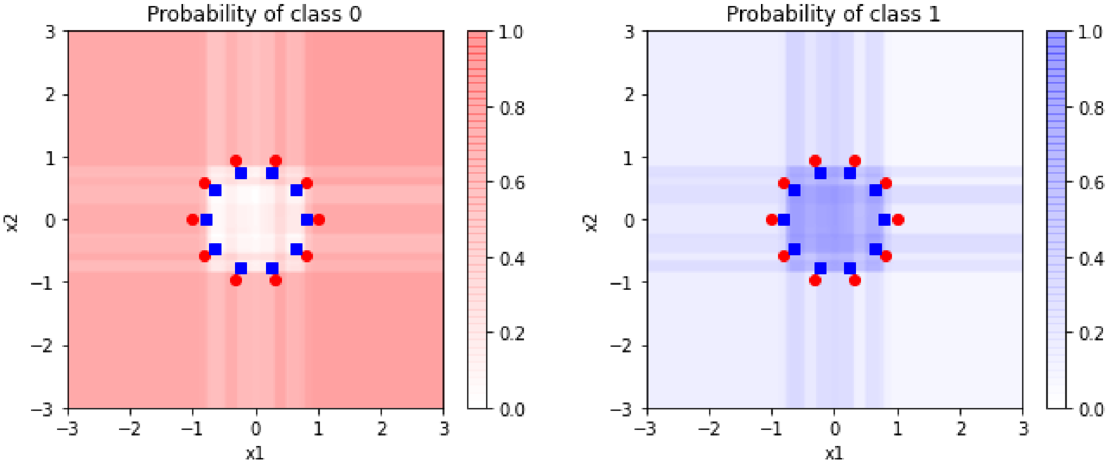


Plot the class probabilities of the final random forest (the one with *n_estimators*=64) using the *plot_class_probability* function. You should generate one figure with two subplots, each with a colour bar, similar to Exercise 1.3. Your figure should look like this:



```
[151]: plt.figure(figsize=(11, 4))

       for i in range(2):
           plt.subplot(1, 2, i + 1)
           plot_class_probability(rf, i)
```

```
    plt.colorbar()
    plot_data(X, y)
    plt.title("Probability of class {}".format(i))
```



Notice how even though the decision region is compacy, the class probabilities have axis-aligned structure extending far away from the data.

Finally, **scale the second feature by a factor of two and re-run all code cells of Exercise 1.4**. You can scale the second feature by multiplying the second column in you X matrix by 2. You should see your circle vertically stretched.

After seeing all the results, make note of whether the decision regions simply scaled with the data or whether the decision regions qualitatively changed in any way.

3 2. Fitting a Random Forest classifier to real data

Exercises 2.1–2.5 ask you to train and evaluate decision tree and random forest classifiers across multiple hyperparameters on real data.

3.0.1 Exercise 2.1 — Load data from a CSV file and plot it

In this exercise you'll be loading the `data_train.csv` and `data_test.csv` files accompanying this lab. Here's a preview of the training data file:

```
sepal_length,sepal_width,label
4.9,2.4,1
4.8,3.0,0
5.1,3.3,0
7.7,3.0,2
6.2,2.8,2
...
```

This is part of the classic Iris flower data set. The first two comman-separated columns are features. They encode characteristics of flowers. The last column is the class label, where each integer represents one of *Iris Setosa* (0), *Iris Versicolour* (1), and *Iris Virginica*.

Write a few lines of code to load each CSV file from disk. From the first file you should create variables *X_train* and *y_train* to refer to the training features (float64) and training labels (int32) respectively. From the second file you should likewise create variables *X_test* and *y_test*. Use the **np.loadtxt** function to load the CSV like you did in Lab 4.

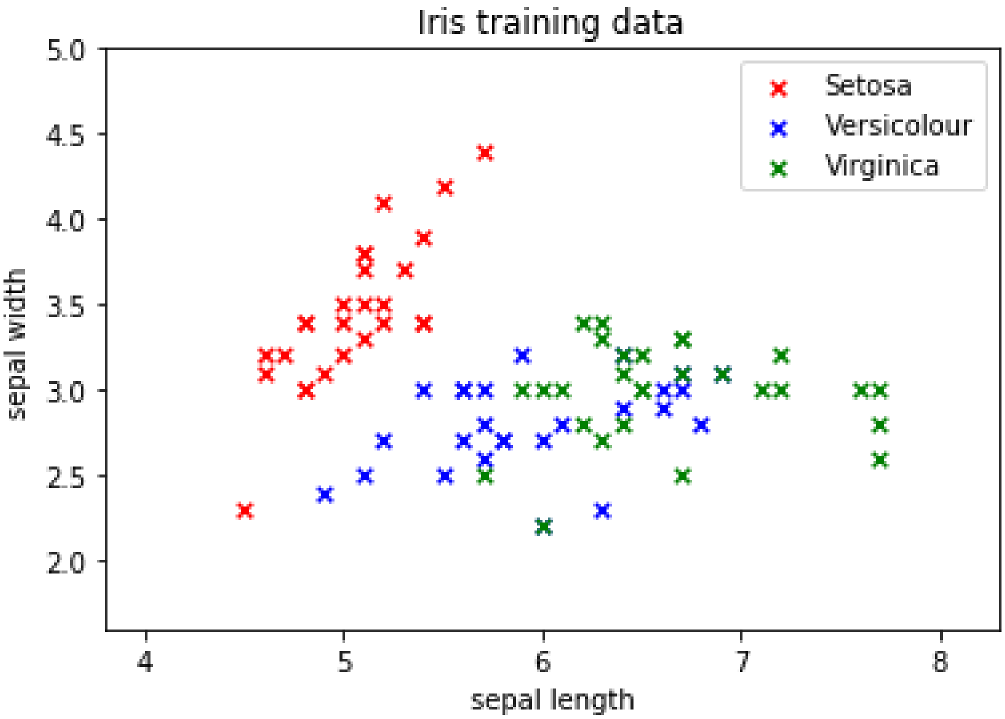
[152]:
```
data_train, data_test = np.loadtxt("data_train.csv", delimiter=",", skiprows=1), \
    np.loadtxt("data_test.csv", delimiter=",", skiprows=1)

X_train, y_train = data_train[:, :2].astype(np.float64), \
    data_train[:, -1].astype(np.int32)

X_test, y_test = data_test[:, :2].astype(np.float64), \
    data_test[:, -1].astype(np.int32)
```

Check your answer by running the code cell below.

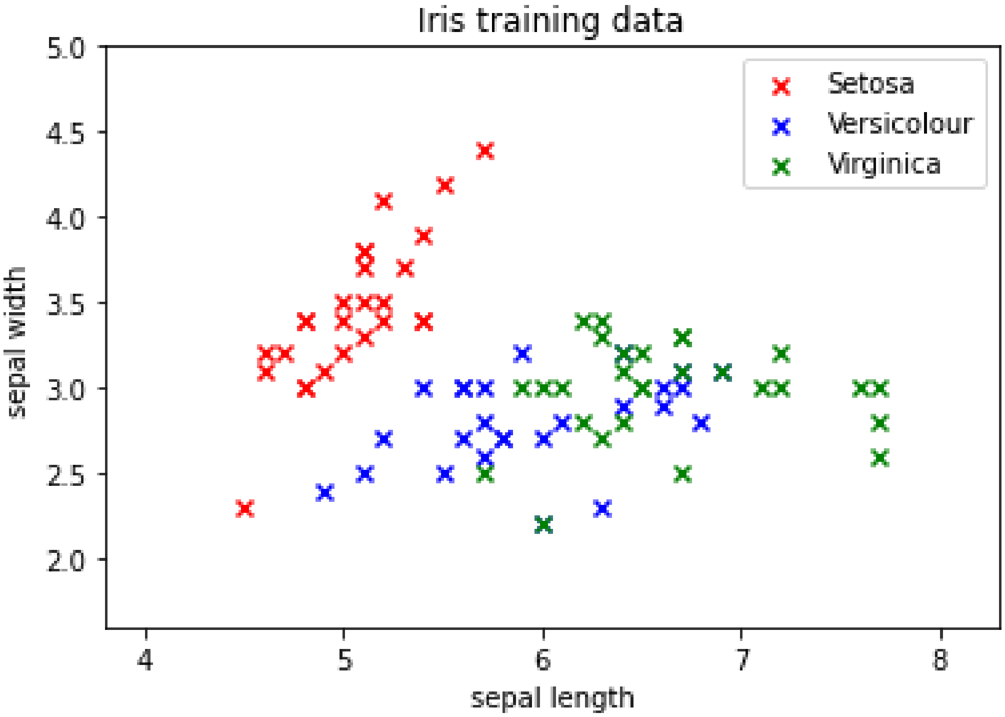
[153]:
```
assert 'X_train' in globals(), "No X_train variable!"
assert 'y_train' in globals(), "No y_train variable!"
assert 'X_test' in globals(), "No X_test variable!"
assert 'y_test' in globals(), "No y_test variable!"
assert isinstance(X_train, np.ndarray)
assert isinstance(y_train, np.ndarray)
assert isinstance(X_test, np.ndarray)
assert isinstance(y_test, np.ndarray)
assert X_train.shape == (80,2), "X_train was wrong shape!"
assert X_train.dtype in (np.float32, np.float64), "X_train was wrong data type!"
assert y_train.shape == (80,), "y_train was wrong shape!"
assert y_train.dtype == np.int32, "y_train was wrong data type!"
assert X_test.shape == (70,2), "X_test was wrong shape!"
assert X_test.dtype in (np.float32, np.float64), "X_test was wrong data type!"
assert y_test.shape == (70,), "y_test was wrong shape!"
assert y_test.dtype == np.int32, "y_test was wrong data type!"
print("Correct!")
```

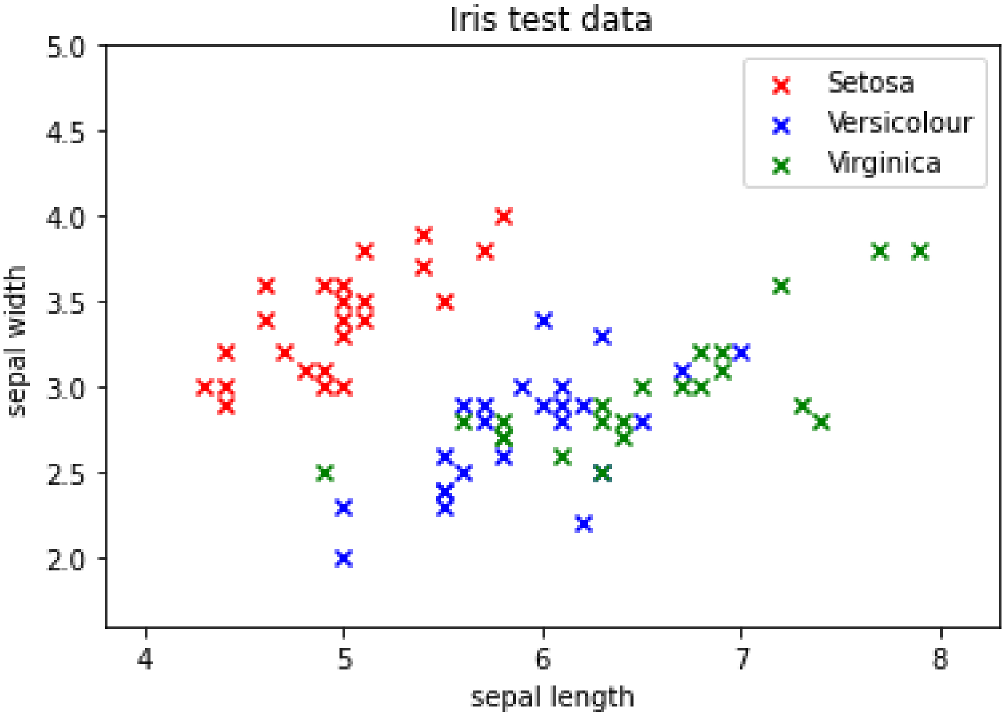
```
Correct!
```

Plot the training and testing data by completing the plotting code below, much like you did in Exercise 2.1 of Lab 4. When you run the code cell it will generate two figures, and the first one should look like this:



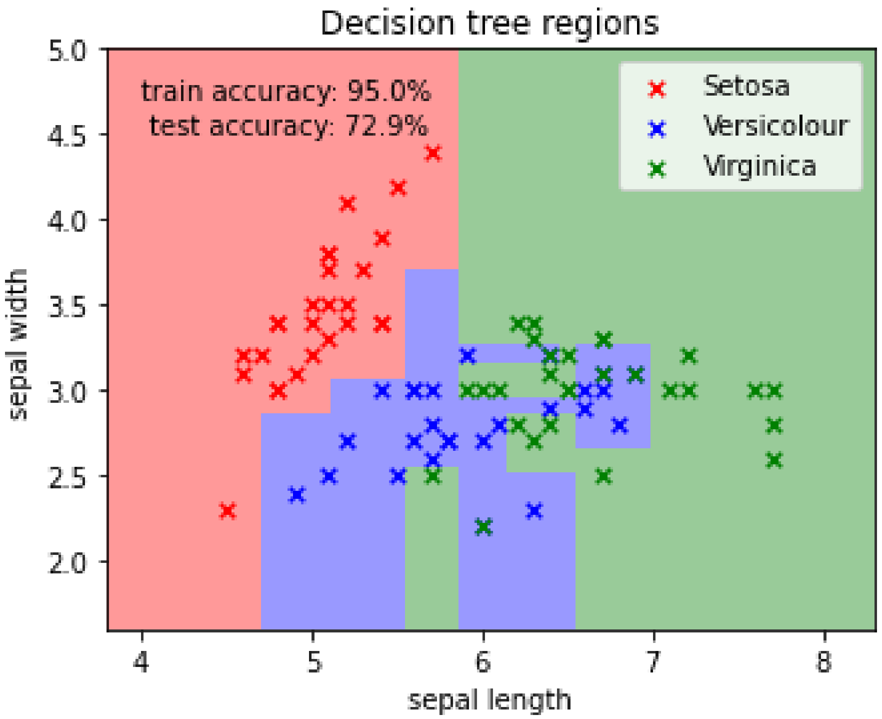
[154]:
```
iris_extent = (3.8, 8.3, 1.6, 5)    # A good (x1min, x1max, x2min, x2max) for␣
 ↪plotting iris data

def plot_iris_data(X, y, title):
    plt.scatter(X[y == 0, 0], X[y == 0, 1], c="r", marker="x", label="Setosa")
    plt.scatter(X[y == 1, 0], X[y == 1, 1], c="b", marker="x",␣
 ↪label="Versicolour")
    plt.scatter(X[y == 2, 0], X[y == 2, 1], c="g", marker="x",␣
 ↪label="Virginica")

    plt.xlabel("sepal length")
    plt.xlim([3.8, 8.3])

    plt.ylabel("sepal width")
    plt.ylim([1.6, 5])

    plt.title(label=title)
    plt.legend()

plt.figure()
plot_iris_data(X_train, y_train, 'Iris training data')

plt.figure()
plot_iris_data(X_test, y_test, 'Iris test data')
```


Iris training data
5.0
4.5
4.0
3.5
3.0
2.5
2.0
sepal width
4
5
6
7
8
sepal length
Setosa
Versicolour
Virginica


Iris test data
5.0
4.5
4.0
3.5
3.0
2.5
2.0
sepal width
4
5
6
7
8
sepal length
Setosa
Versicolour
Virginica

3.0.2 Exercise 2.2 — Train a decision tree on the Iris data and plot it

You must train a decision tree on the Iris training data, and plot the decision regions. Your final result should look like this:



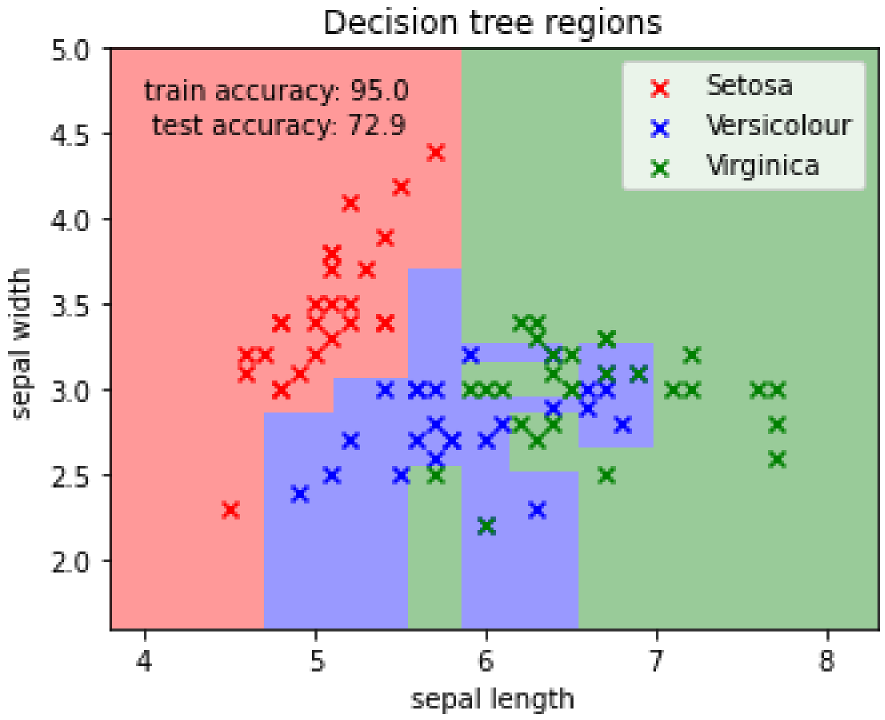
Write a few lines of code to: 1. Train a *DecisionTreeClassifier* object on the training data. 2. Plot the resulting predictor. Use *random_state*=0. 3. Add text to your plot showing the accuracy of the classifier when predicting on the training data and when predicting on the held-out test data. Use the **sklearn.metrics.accuracy_score** function and Matplotlib's **text** function.

For plotting, you cannot use the *plot_predict* function from Exercise 1 because it assumes extents (-3, 3, -3, 3), but you can adapt that code to work for the range of Iris data below.

Tip: When you are formatting a string like `"%.1f" % accuracy` and you want to add a `%` symbol to the string, use two `%%` symbols in a row as in `"%.1f%%" % accuracy`. This lets Python know not to expect a second value to substitute into the string, and to just print `%` in that spot.

[155]:

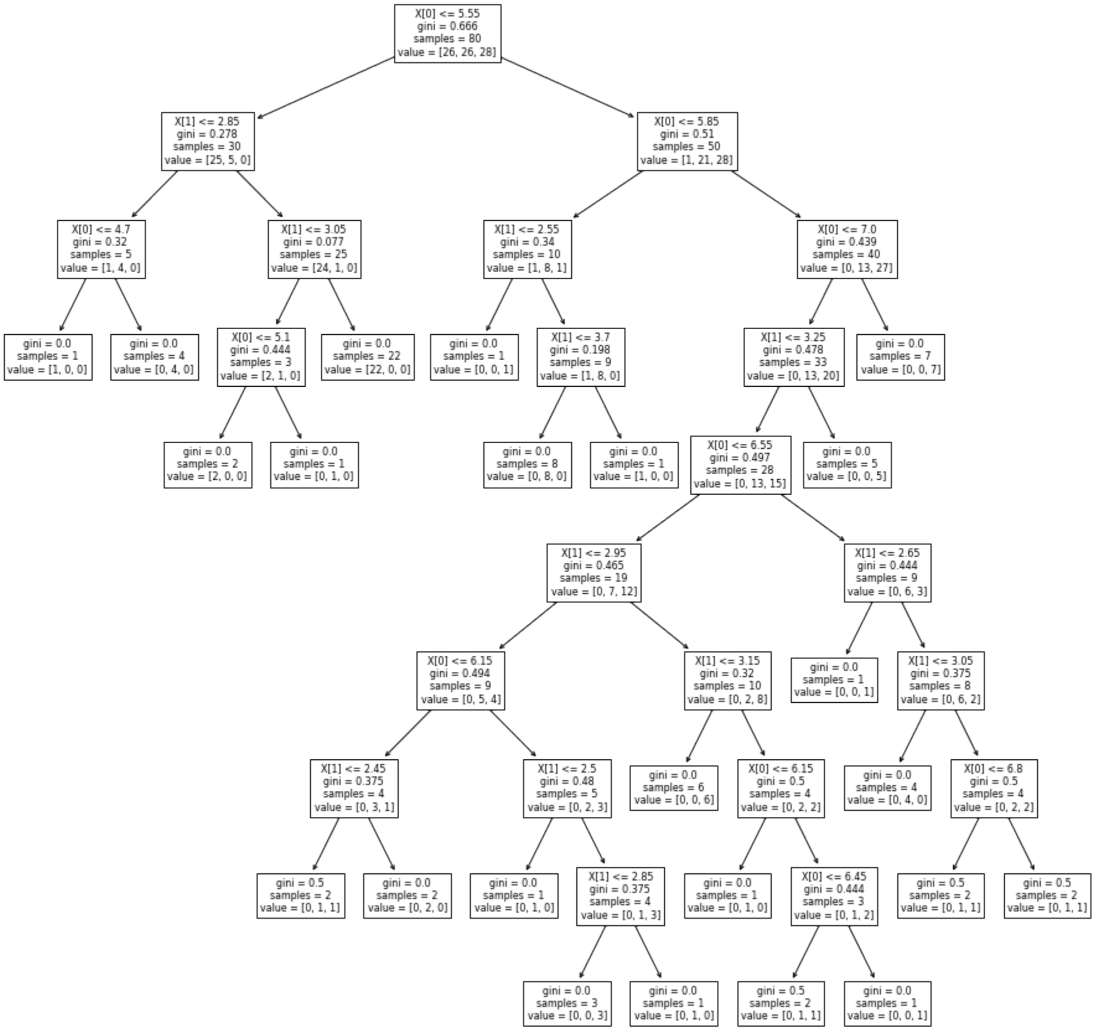
```
def plot_iris_predict(model):
    res = np.meshgrid(
        np.linspace(iris_extent[0], iris_extent[1], 100),
        np.linspace(iris_extent[2], iris_extent[3], 100),
    )

    X = np.column_stack([res[0].ravel(), res[1].ravel()])
    y = model.predict(X).reshape(res[0].shape)

    plt.xlim([iris_extent[0], iris_extent[1]])
    plt.ylim([iris_extent[2], iris_extent[3]])

    plt.imshow(y, alpha=0.4, cmap=matplotlib.colors.ListedColormap(["r", "b",␣
 ↪"g"]), \
        extent=iris_extent, interpolation="nearest", origin="lower", vmin=0,␣
 ↪vmax=2)

# aux function
def _plot_tree_with_metrics(dt, title="Decision tree regions"):
    plt.figure()
    plot_iris_predict(dt)
    plot_iris_data(X_train, y_train, title)

    plt.text(4.00, 4.70, "train accuracy: {:.1f}".format(
            sklearn.metrics.accuracy_score(y_train, dt.predict(X_train)) * 100
        ),
    )

    plt.text(4.05, 4.50, "test accuracy: {:.1f}".format(
            sklearn.metrics.accuracy_score(y_test, dt.predict(X_test)) * 100
        ),
    )

dt = sklearn.tree.DecisionTreeClassifier(random_state=0).fit(X_train, y_train)
_plot_tree_with_metrics(dt)
```



Plot the decision tree using the *plot_tree* function. You'll need to use the *figure* function with *figsize*=(16,16) in order to make the figure large enough to see all the details. Remember the hint about ending a line with a semicolon (;).

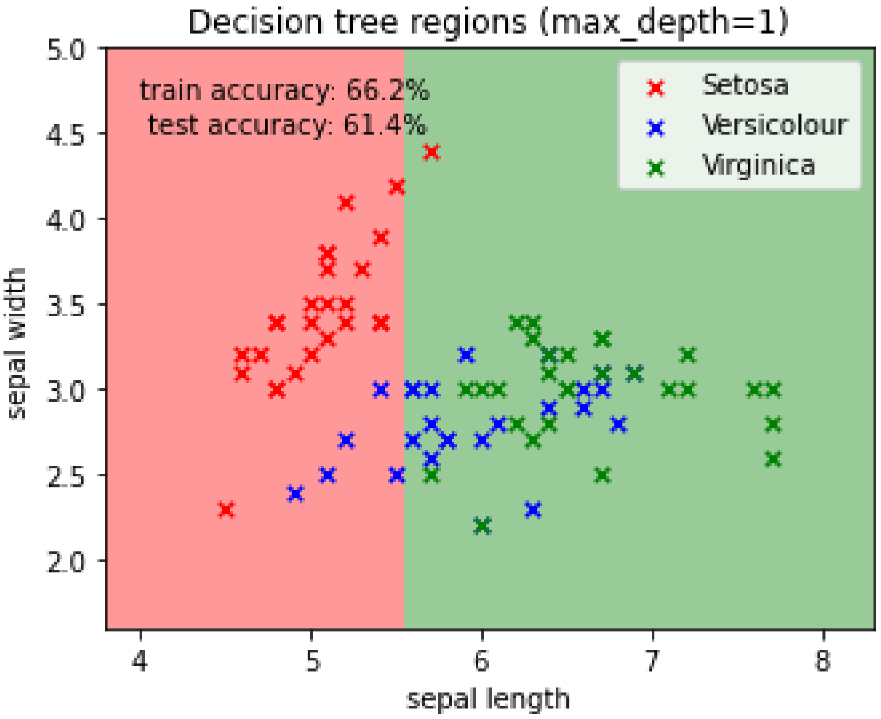
Ask yourself: is this tree reasonably "interpretable" in your view?

[156]:
```
plt.figure(figsize=(16, 16))
sklearn.tree.plot_tree(dt);
```



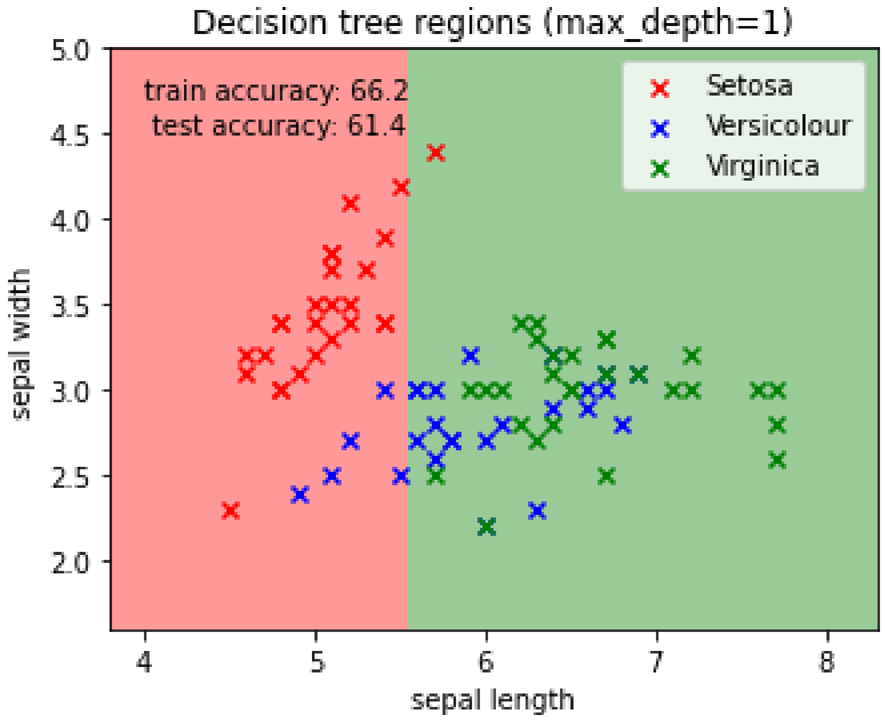
3.0.3 Exercise 2.3 — Train decision trees of different depths

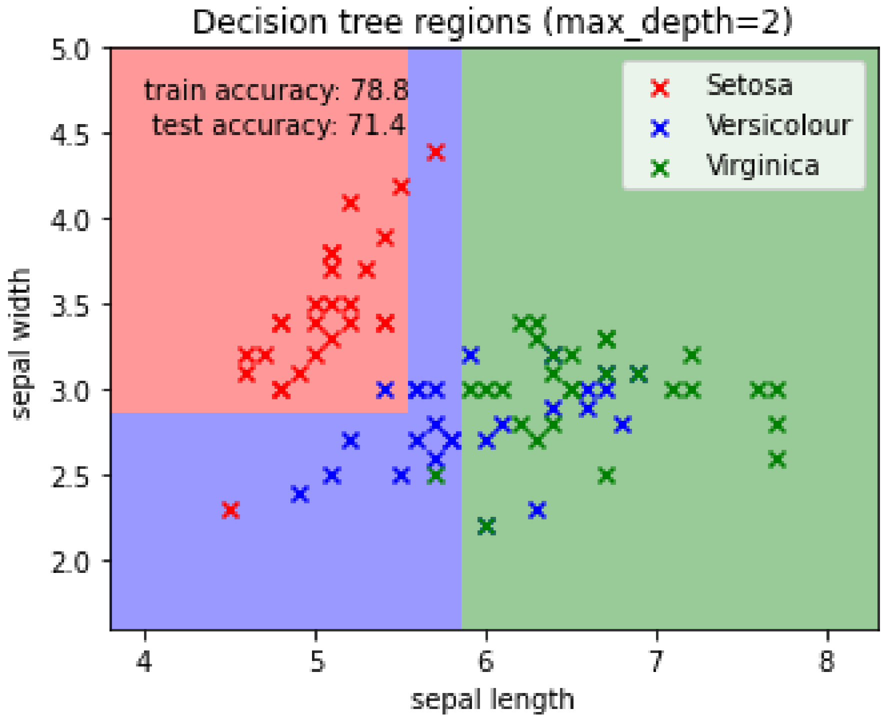
You are asked to train multiple decision trees, each with a different *max_depth* parameter, and plot the resulting decision regions and training/test accuracy. The maximum depth of a decision tree controls how finely the tree is allowed to split the feature space before it must predict a class label. A depth of 1 means it is only allowed one split. Your first figure should look like this:

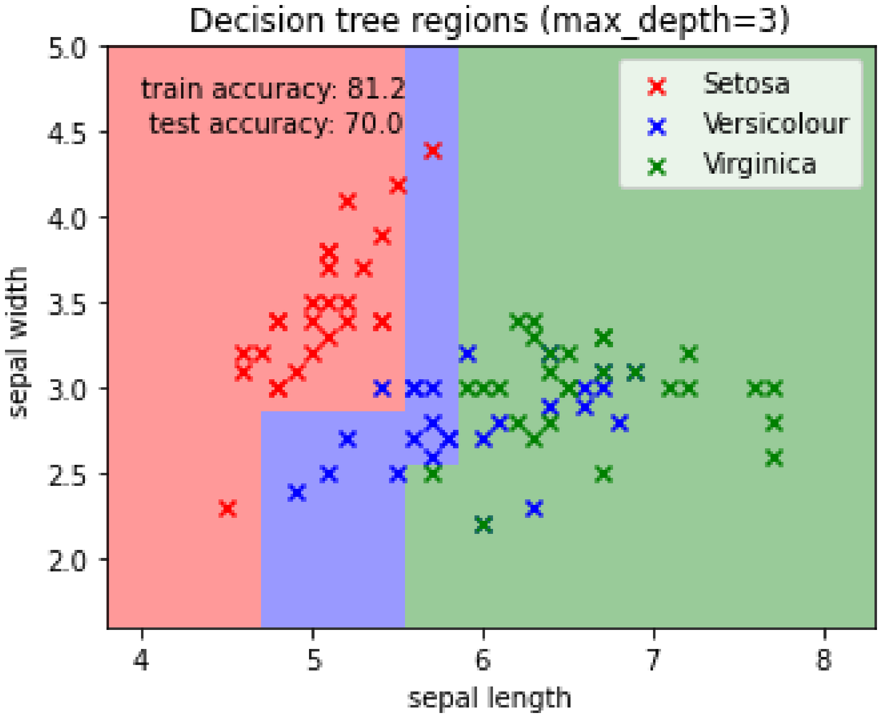


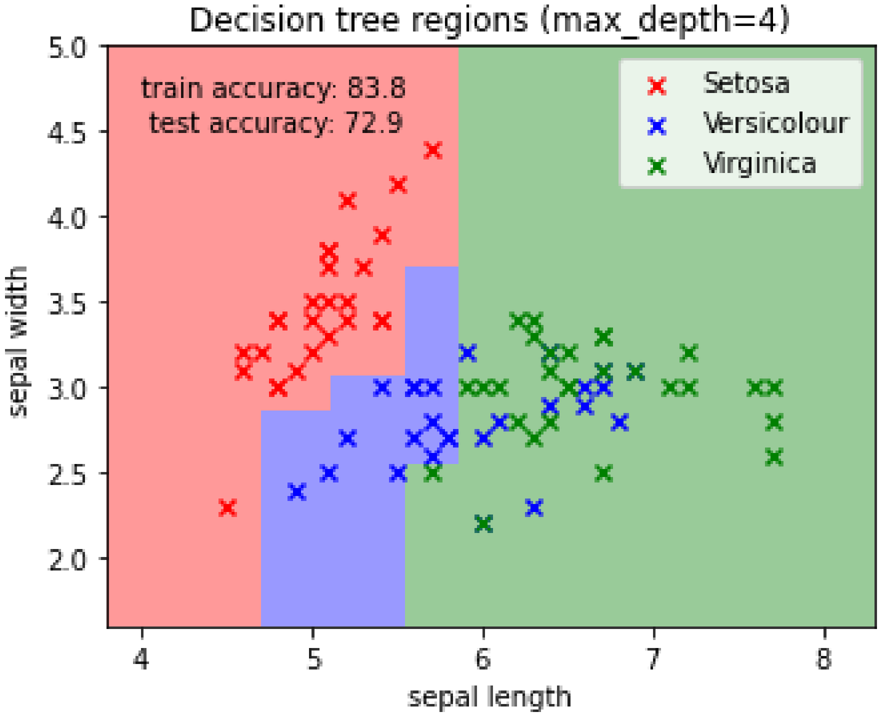
Write a few lines of code to create nine figures: one for each value of *max_depth* $\in \{1, \ldots, 9\}$. Use a for-loop in your answer. On each iteration, train a new classifier using *random_state*=0. Use the *plot_iris_data* and your *plot_iris_predict* functions.

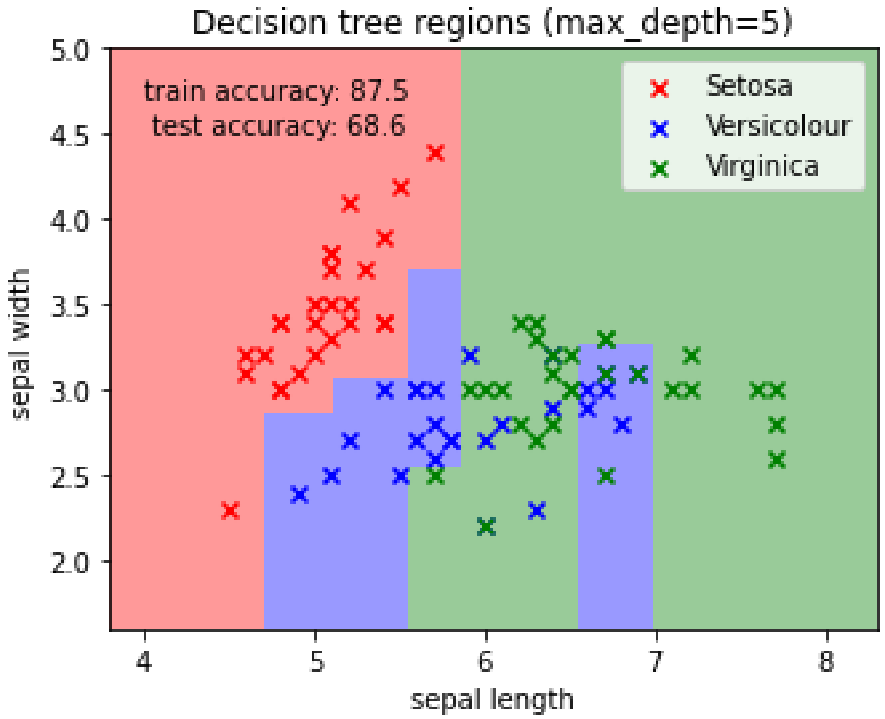
```
[157]: for i in range(1, 10):
           dt = sklearn.tree.DecisionTreeClassifier(random_state=0, max_depth=i).
        ↪fit(X_train, y_train)
           _plot_tree_with_metrics(dt, "Decision tree regions (max_depth={})".
        ↪format(i))
```

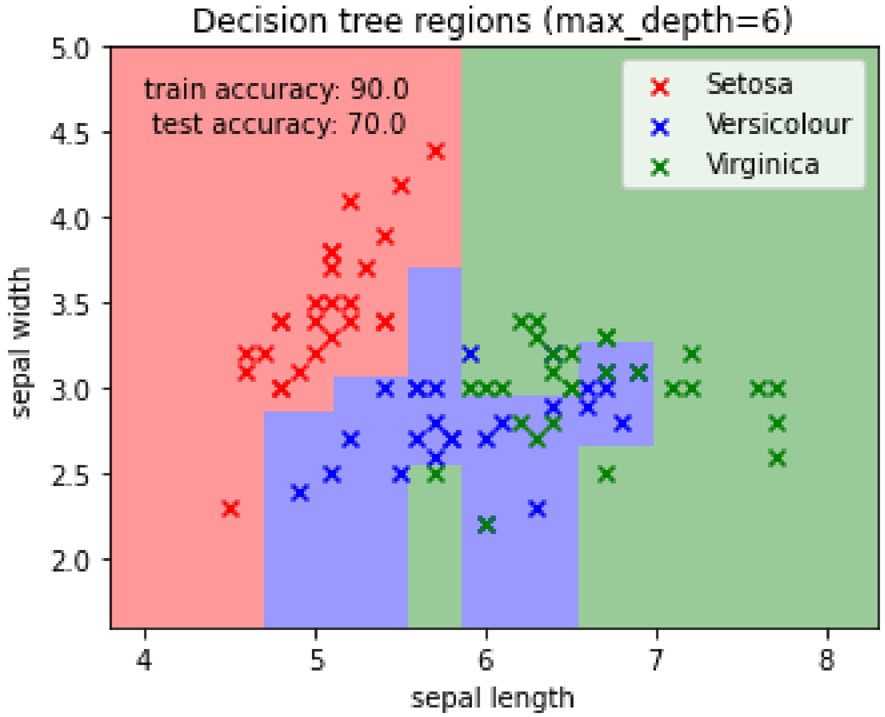

Decision tree regions (max_depth=1)
train accuracy: 66.2
test accuracy: 61.4
Setosa
Versicolour
Virginica
sepal width
sepal length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8

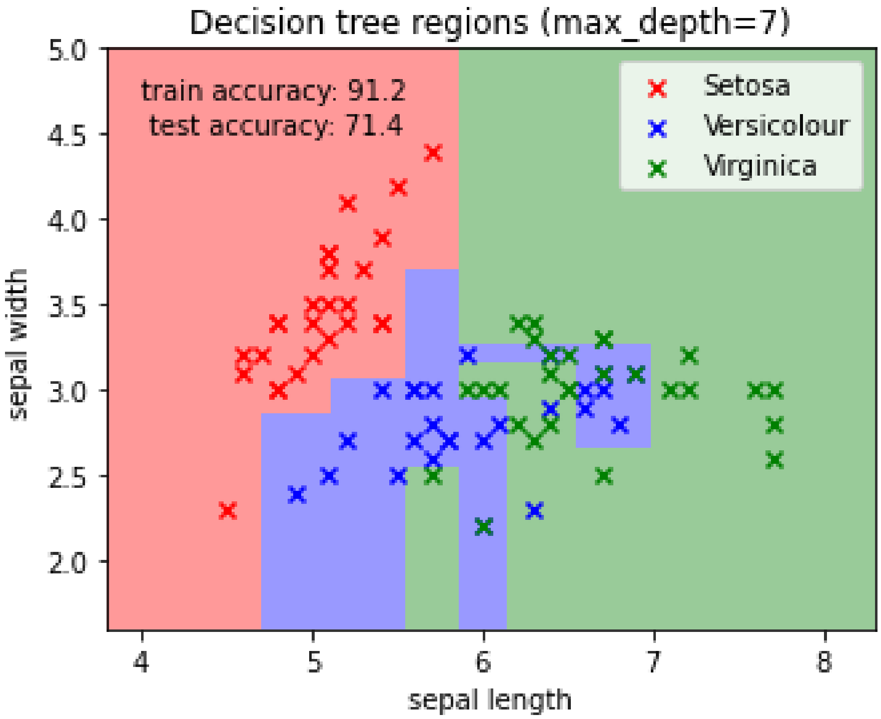

Decision tree regions (max_depth=2)
train accuracy: 78.8
test accuracy: 71.4
Setosa
Versicolour
Virginica
sepal width
sepal length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8

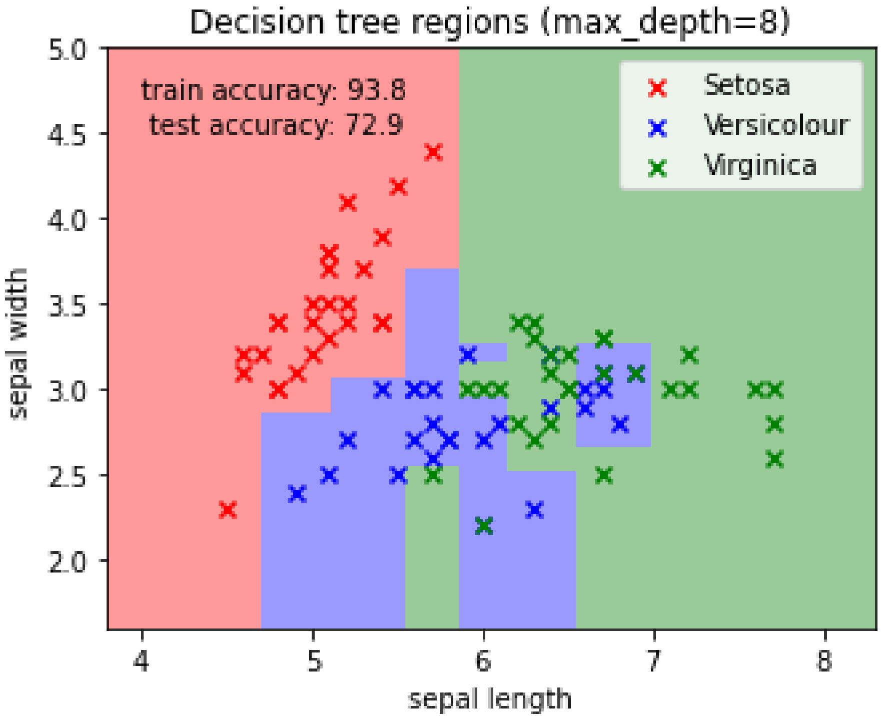

Decision tree regions (max_depth=3)
train accuracy: 81.2
test accuracy: 70.0
Setosa
Versicolour
Virginica
sepal width
sepal length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8

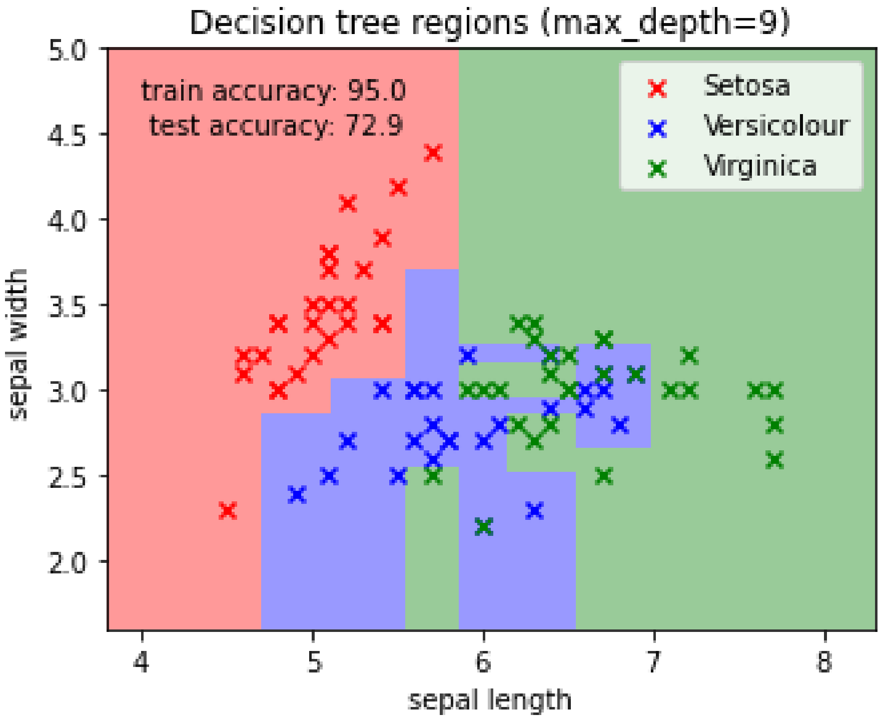

Decision tree regions (max_depth=4)
train accuracy: 83.8
test accuracy: 72.9
Setosa
Versicolour
Virginica
sepal width
sepal length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8


Decision tree regions (max_depth=5)
train accuracy: 87.5
test accuracy: 68.6
Setosa
Versicolour
Virginica
sepal width
sepal length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8


Decision tree regions (max_depth=6)
train accuracy: 90.0
test accuracy: 70.0
Setosa
Versicolour
Virginica
sepal width
sepal length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8


Decision tree regions (max_depth=7)
train accuracy: 91.2
test accuracy: 71.4
Setosa
Versicolour
Virginica
5.0
4.5
4.0
3.5
3.0
2.5
2.0
sepal width
4
5
6
7
8
sepal length


Decision tree regions (max_depth=8)
train accuracy: 93.8
test accuracy: 72.9
Setosa
Versicolour
Virginica
5.0
4.5
4.0
3.5
3.0
2.5
2.0
sepal width
4
5
6
7
8
sepal length

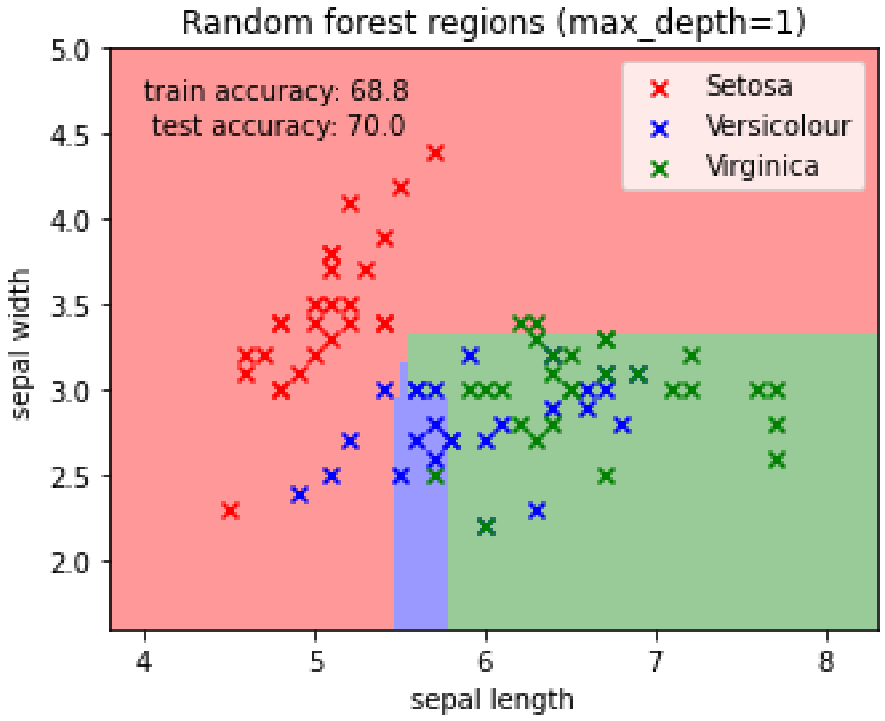


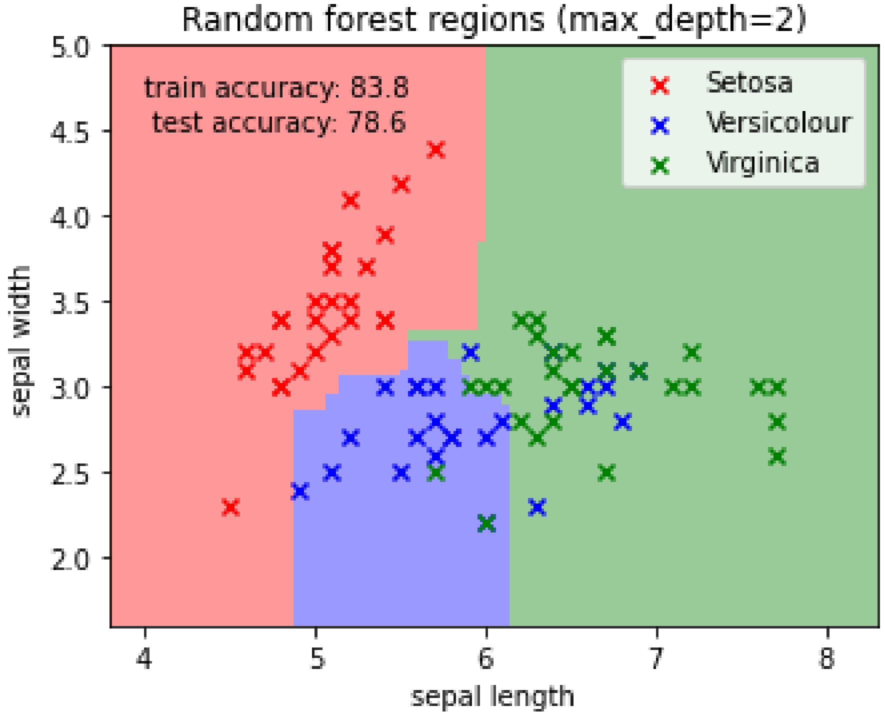
What happens to the training accuracy as *max_depth* increases? What is the maximum test accuracy of a decision tree classifier on this data set?

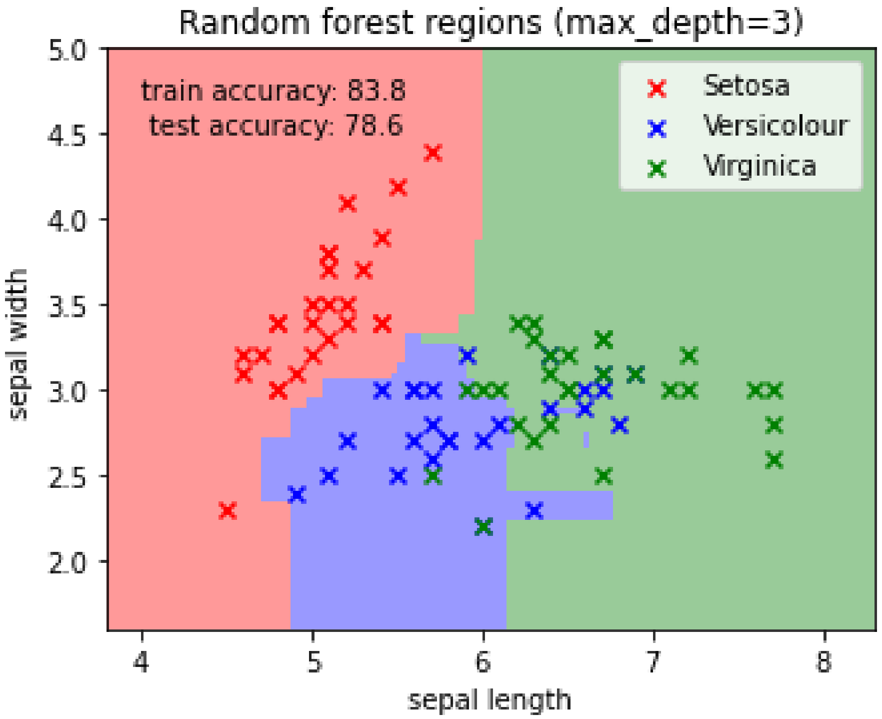
3.0.4 Exercise 2.4 — Train random forests of different depths

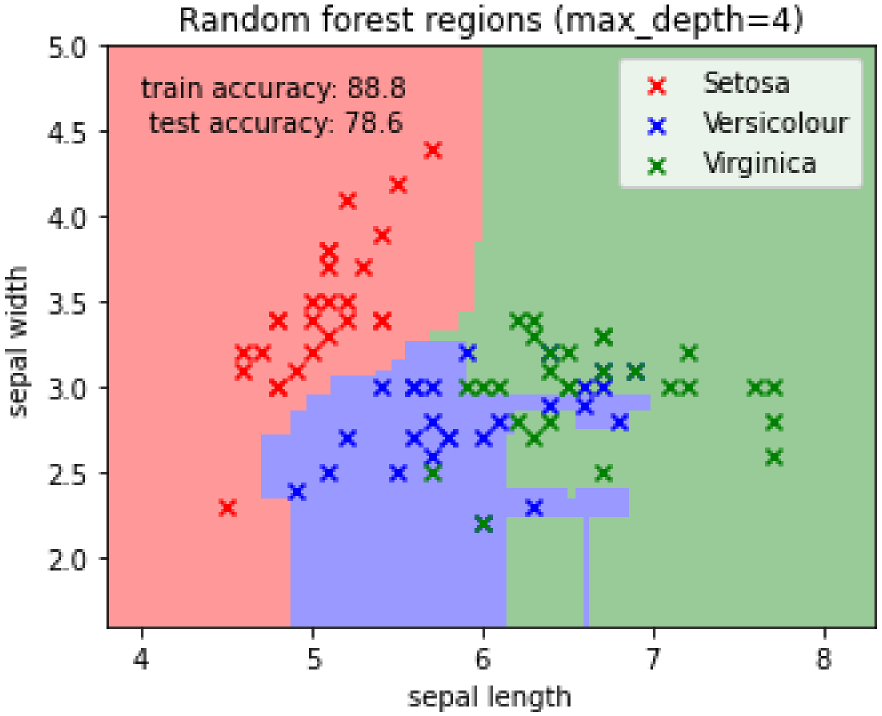
Repeat Exercise 2.3 but this time train a *RandomForestClassifier* instead of a *DecisionTreeClassifier*. Use *random_state*=0 and *n_estimators*=100.

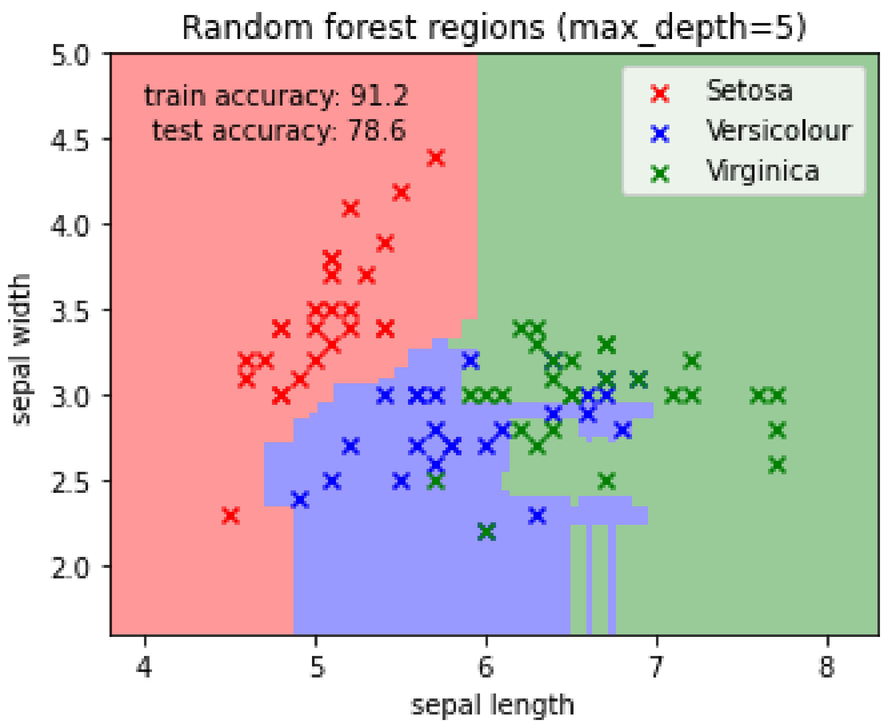
[158]:
```
for i in range(1, 10):
    rf = sklearn.ensemble.RandomForestClassifier(random_state=0, max_depth=i,␣
 ↪n_estimators=100).fit(X_train, y_train)
    _plot_tree_with_metrics(rf, "Random forest regions (max_depth={})".
 ↪format(i))
```

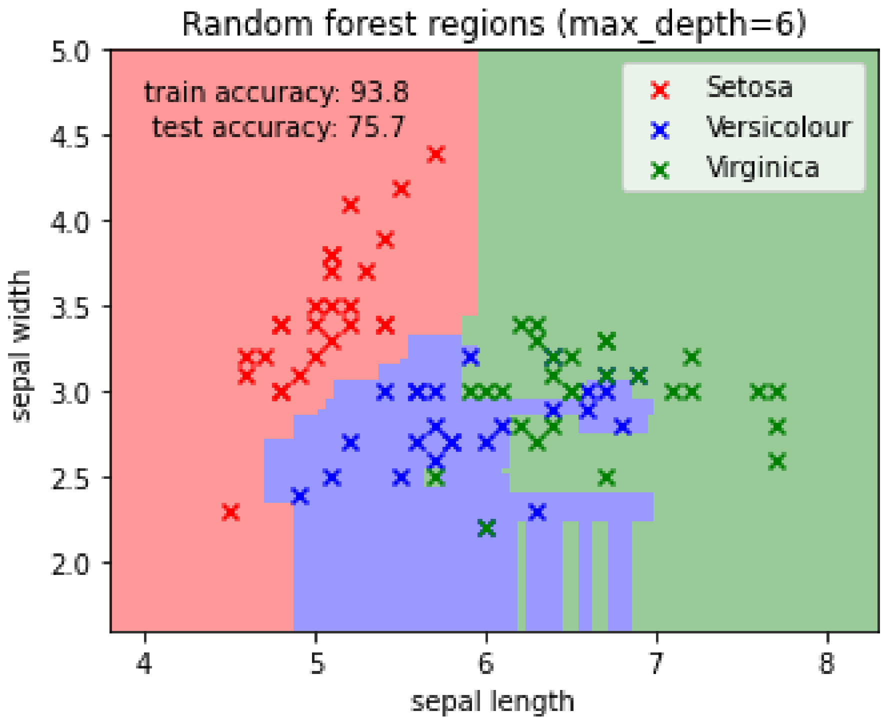

Random forest regions (max_depth=1)
train accuracy: 68.8
test accuracy: 70.0
Setosa
Versicolour
Virginica
sepal width
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8
sepal length

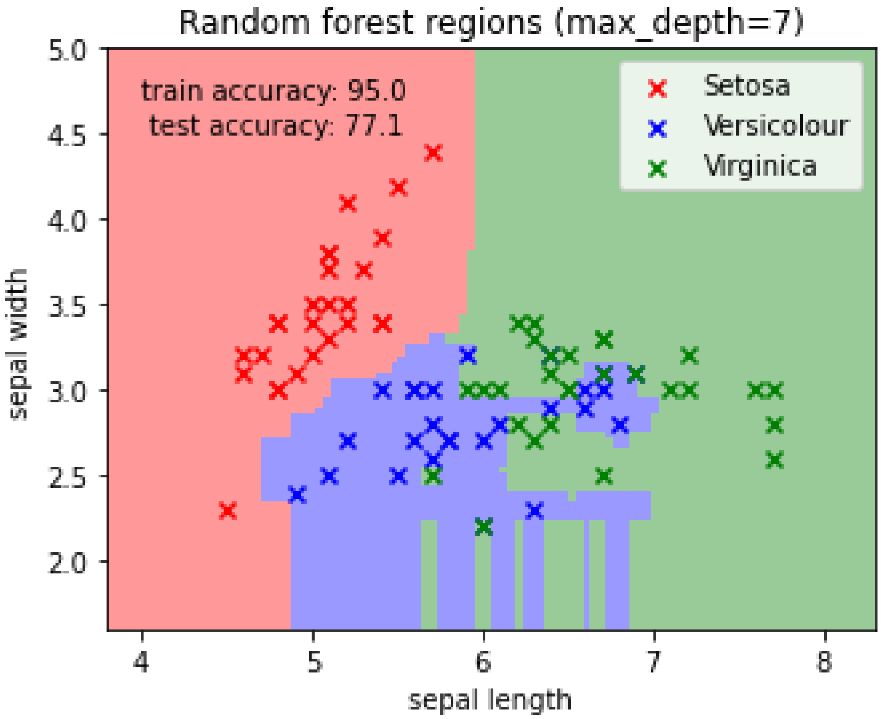

Random forest regions (max_depth=2)
train accuracy: 83.8
test accuracy: 78.6
Setosa
Versicolour
Virginica
sepal width
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8
sepal length

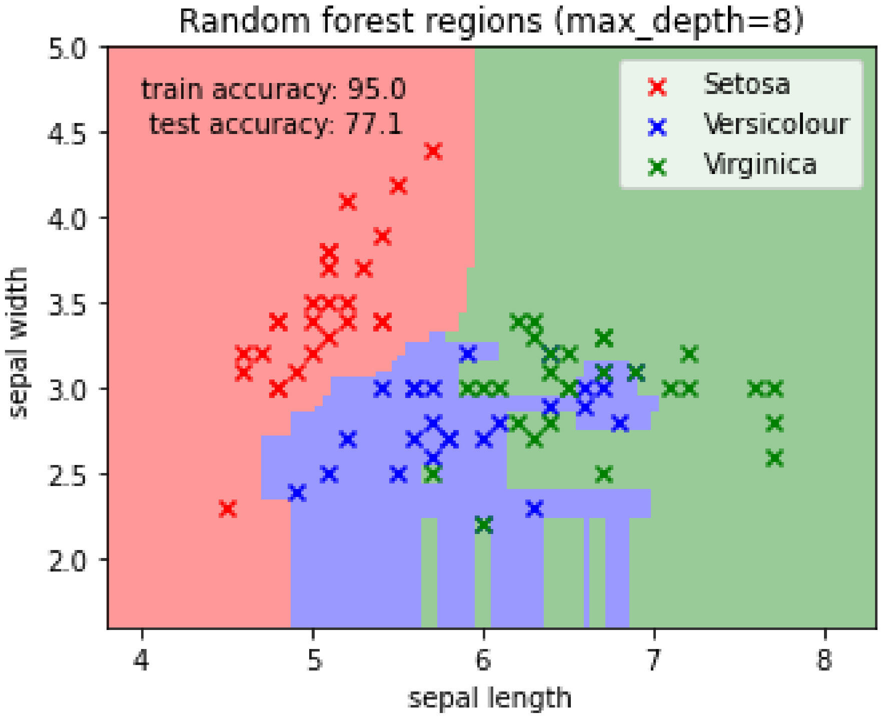

Random forest regions (max_depth=3)
train accuracy: 83.8
test accuracy: 78.6
Setosa
Versicolour
Virginica
5.0
4.5
4.0
3.5
3.0
2.5
2.0
sepal width
4
5
6
7
8
sepal length

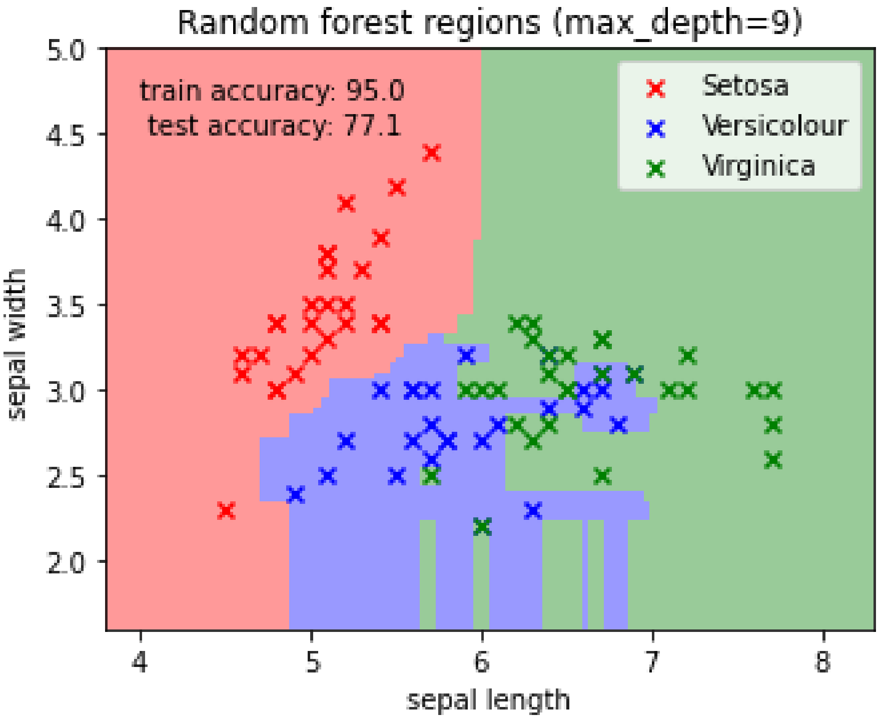

Random forest regions (max_depth=4)
train accuracy: 88.8
test accuracy: 78.6
Setosa
Versicolour
Virginica
5.0
4.5
4.0
3.5
3.0
2.5
2.0
sepal width
4
5
6
7
8
sepal length


Random forest regions (max_depth=5)
train accuracy: 91.2
test accuracy: 78.6
Setosa
Versicolour
Virginica
5.0
4.5
4.0
3.5
3.0
2.5
2.0
sepal width
4
5
6
7
8
sepal length


Random forest regions (max_depth=6)
train accuracy: 93.8
test accuracy: 75.7
Setosa
Versicolour
Virginica
5.0
4.5
4.0
3.5
3.0
2.5
2.0
sepal width
4
5
6
7
8
sepal length


Random forest regions (max_depth=7)
train accuracy: 95.0
test accuracy: 77.1
Setosa
Versicolour
Virginica
sepal width
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8
sepal length


Random forest regions (max_depth=8)
train accuracy: 95.0
test accuracy: 77.1
Setosa
Versicolour
Virginica
sepal width
5.0
4.5
4.0
3.5
3.0
2.5
2.0
4
5
6
7
8
sepal length



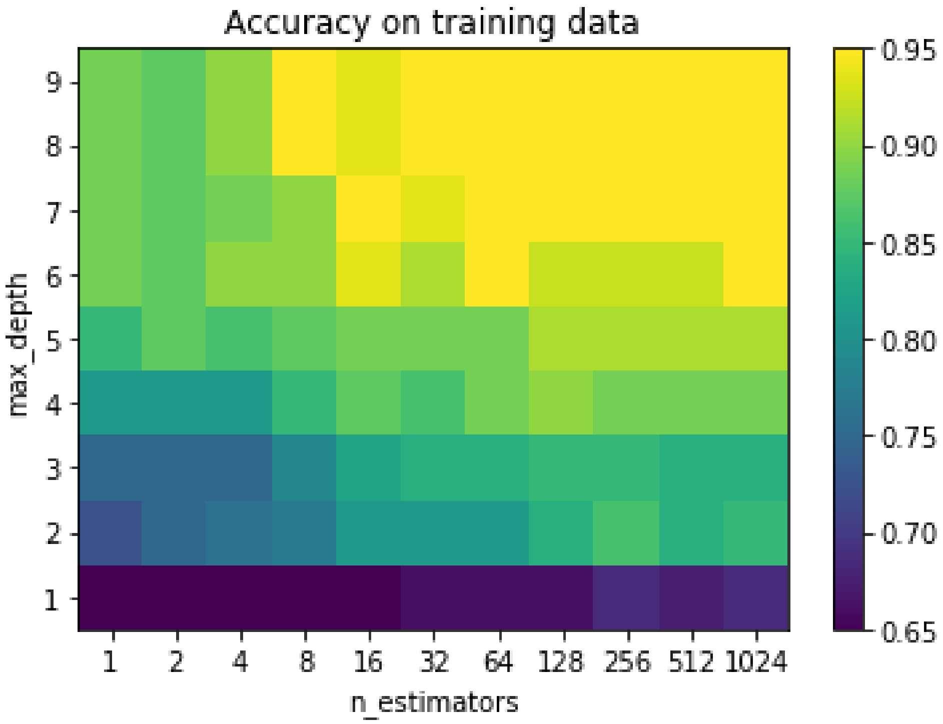
What happens to the training accuracy as *max_depth* increases? What is the maximum test accuracy of a random forest classifier on this data set?

3.0.5 Exercise 2.5 — Evaluate training and test accuracy across two hyperparameters

The performance of *RandomForestClassifier* depends on many hyperparameters. Here you'll perform a sweep over *max_depth* (as you did in Exercise 2.4) and also *n_estimators* (as you did in Exercise 1.4). Instead of plotting the decisions in input space, you'll plot a heatmap of the accuracy for each parameter setting.

You must evaluate the training and test accuracy of *RandomForestClassifier* for every combination of $* \ max\_depth \in \{1,2,3,4,5,6,7,8,9\} * \ n\_estimators$ \$ $\in \{1,2,4,8,16,32,64,128,256,512,1024\}$\$

The results should be compiled into a two-dimensional ndarray and plotted as a heatmap with properly labeled axes. Your training accuracy plot should look like this:

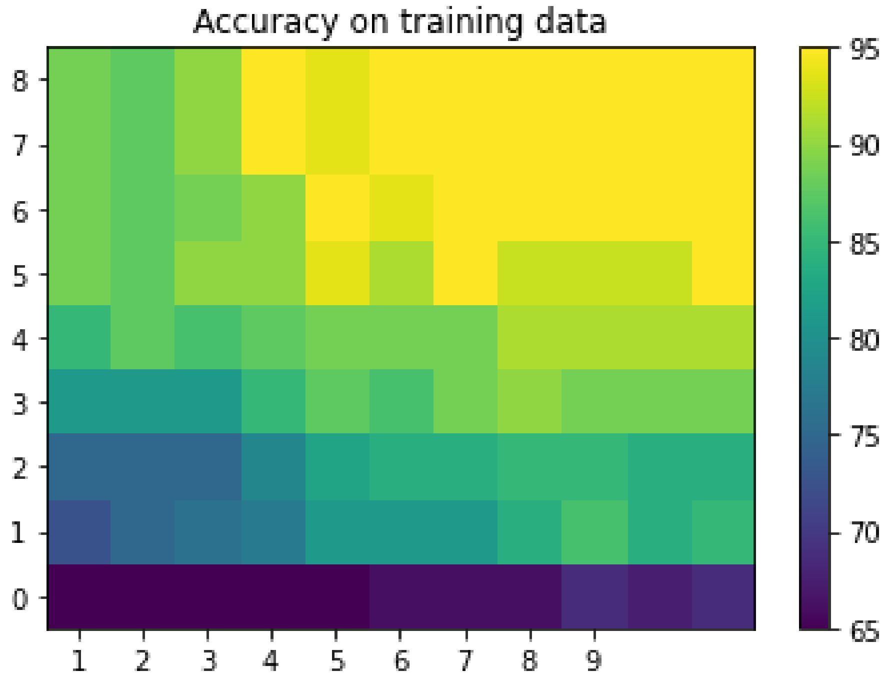


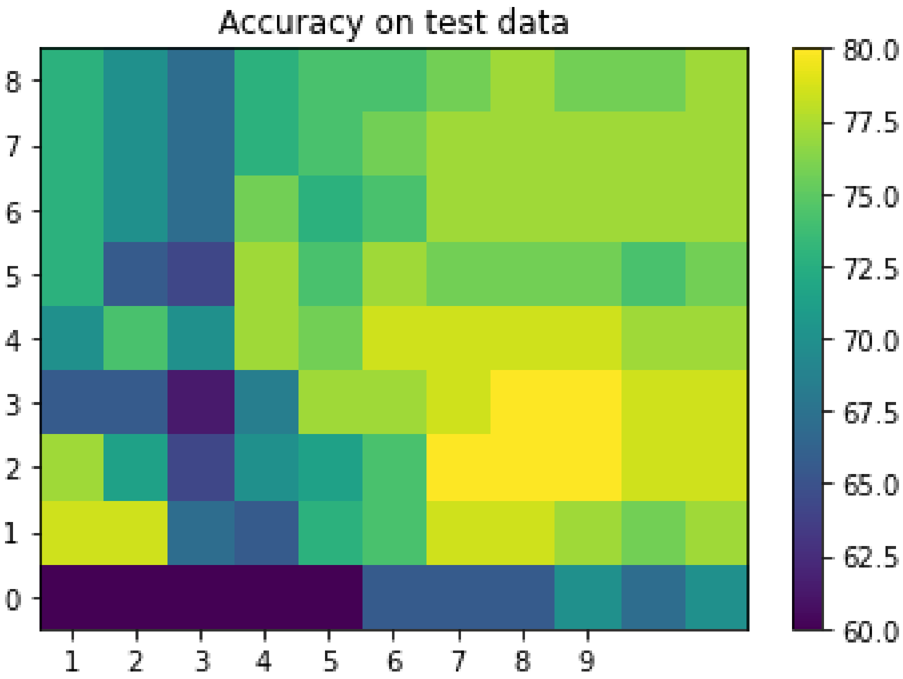
Write code to evaluate the training and testing accuracy for all the above combinations. Since training many models may take a few seconds, the plotting should be done in a subsequent code cell, without having to re-run the experiment.

Tip: If you build a 9×11 array of accuracies and plot them using Matplotlib's *imshow* function, Matplotlib does not know which values of *max_depth* each row $0, \ldots, 8$ corresponds to, nor does it know which value of *n_estimators* each column $0, \ldots, 10$ corresponds to. You can specify the values to use via the **xticks** function, where for example `plt.xticks([0,1,2], [10,20,40])` would cause Matplotlib to display labels *10, 20, 40* at the x-axis positions, instead of its default "guess" of displaying *0, 1, 2*.

[159]:
```
max_depth, n_estimators, accuracies = [i for i in range(1, 10)], [2 ** j for j␣
↪in range(11)], np.empty((9, 11, 2))

for i, depth in enumerate(max_depth):
    for j, estimators in enumerate(n_estimators):
        rf = sklearn.ensemble.RandomForestClassifier(random_state=0,␣
↪max_depth=depth, n_estimators=estimators).fit(X_train, y_train)
        accuracies[i][j][0], accuracies[i][j][1] = (
    sklearn.metrics.accuracy_score(y_train, rf.predict(X_train)) * 100,
    sklearn.metrics.accuracy_score(y_test, rf.predict(X_test)) * 100,
)
```

[160]:
```
plt.figure()
plt.imshow(accuracies[:, :, 0], origin="lower")
plt.colorbar()
plt.title(label="Accuracy on training data")
plt.xticks(range(len(n_estimators)), labels=n_estimators);
plt.xticks(range(len(max_depth)), labels=max_depth);

plt.figure()
plt.imshow(accuracies[:, :, 1], origin="lower")
plt.colorbar()
plt.title(label="Accuracy on test data")
plt.xticks(range(len(n_estimators)), labels=n_estimators);
plt.xticks(range(len(max_depth)), labels=max_depth);
```





For what combination(s) of (*max_depth*, *n_estimators*) does the random forest classifier have highest accuracy **on the test data**? Is the accuracy trend on the test data different than the trend on the training data?