Lab 7: How AdaBoost combines weak classifiers

In this lab you'll train an AdaBoost classifier, and inspect it to learn how it makes predictions.

The goal is to get a sense for how a several estimators can be summed together to make a strong predictor. In a random forest, each estimator is trained independently before averaging. In boosting, the estimators are trained sequentially, with each new estimator asked to "correct" mistakes made by the collection of previous estimators.

You will see that each estimator is a very simple decision tree, also called a "decision stump" because it only has one split (*max_depth=1*). The use of shallow trees is deliberate: their shallowness makes them individually 'weak' at predicting, but easier to combine (to "boost") into a strong predictor.

Run the code cell below to import the required packages.

In [1]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import sklearn
import sklearn.tree        # For DecisionTreeClassifier class
import sklearn.ensemble    # For AdaBoostClassifier class
import sklearn.datasets    # For make_gaussian_quantiles
import sklearn.metrics     # For accuracy_score
```

1. Understanding how AdaBoost makes predictions

Exercises 1.1–1.2 ask you to train scikit-learn's AdaBoost classifier (**sklearn.ensemble.AdaBoostClassifier**) and to inspect the way it makes predictions.

Run the code cell below to define some useful functions for plotting data and predictions. These are the same functions used for plotting in Lab 5.

In [2]:
```
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].
    If argument is already a Numpy array, treats it as predictions.
    Otherwise calls the argument's predict() function to generate predictions.
    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, interpolation='nearest')
    plt.xlim([x1min, x1max])
    plt.ylim([x2min, x2max])
    plt.gca().set_aspect('equal')
```

Exercise 1.1 — Compare an AdaBoost classifier to a simple decision tree

The goal of this exercise is to show you that with multiple shallow "decision stumps" we can still produce the same output as single deep decision tree. Start by training a decision tree classifier like you did in Lab5. (No boosting yet.)

Step 1: Train a decision tree classifier like you did in exercise 1.1 from Lab5.

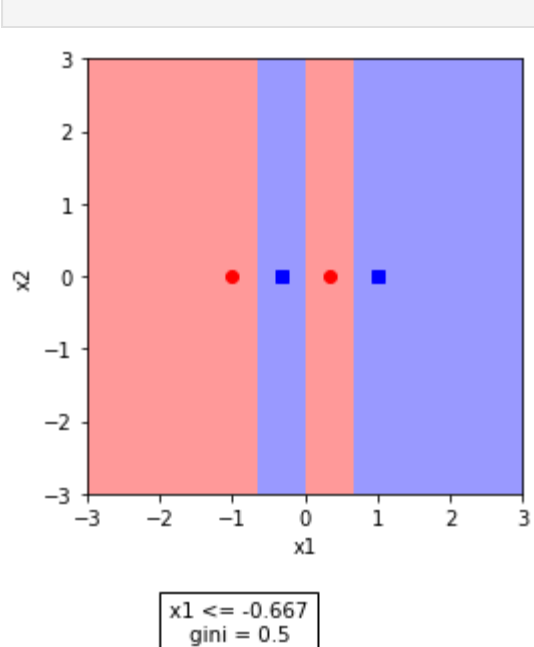
Write a few lines of code to:

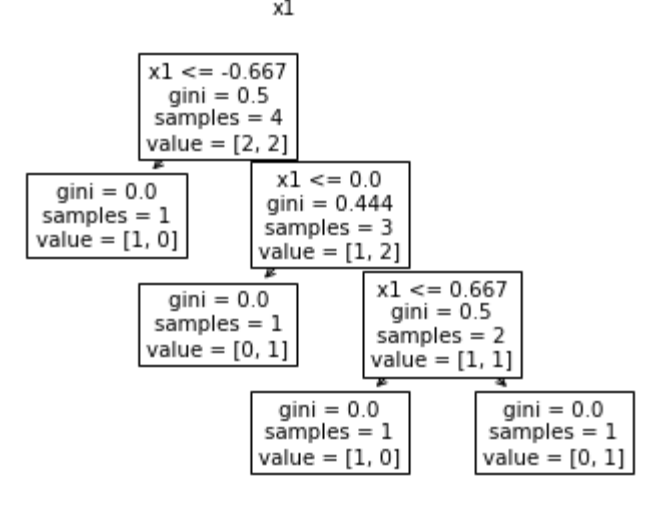
1. Build the small 2D training set below:

$$X = \begin{bmatrix} -1 & 0 \\ -\frac{1}{3} & 0 \\ \frac{1}{3} & 0 \\ 1 & 0 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

1. Train a decision tree classifier on X and y. Use argument *random_state*=0.
2. Plot the decision tree predictions and the data (use *plot_predict* and *plot_data* from preamble).
3. Plot the decision tree in a second figure (use **sklearn.tree.plot_tree**); pass `feature_names=['x1', 'x2']` as an argument.

In [3]:
```
# Your code here. Aim for 8-12 lines.
X = np.array([[-1, 0], [-1/3, 0], [1/3, 0], [1, 0]]);
y = np.array([0, 1, 0, 1]);

decisionTree = sklearn.tree.DecisionTreeClassifier(random_state=0).fit(X,y);
plot_data(X, y);
plot_predict(decisionTree);
plt.figure();
sklearn.tree.plot_tree(decisionTree,feature_names=['x1', 'x2']);
```





Having already completed Lab5, you should understand exactly what you see above, i.e., how the decision tree thresholds, and the leaves, determine the red/blue regions plotted in feature space. You know from lecture that the training accuracy of a decision tree classifier is improved by "splitting" a leaf.

Step 2: Train an *AdaBoostClassifier* on the same data.

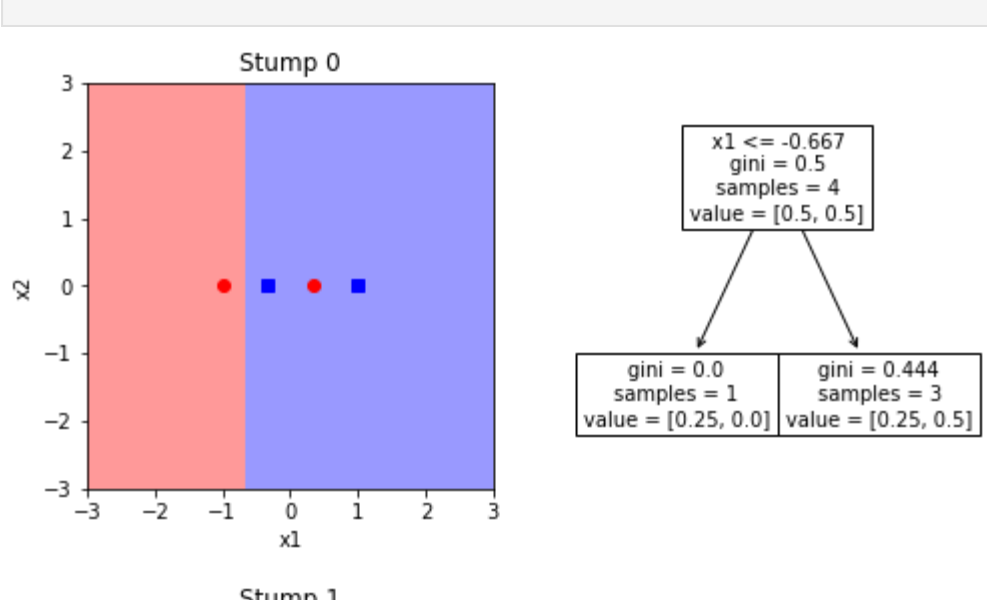
AdaBoost does not try to improve training accuracy by adding complexity to the current tree. In fact, each tree that AdaBoost builds only has one split (*max_depth*=1), so it is hardly a tree at all. Such a 'weak' tree is often called a "decision stump" instead of "decision tree."

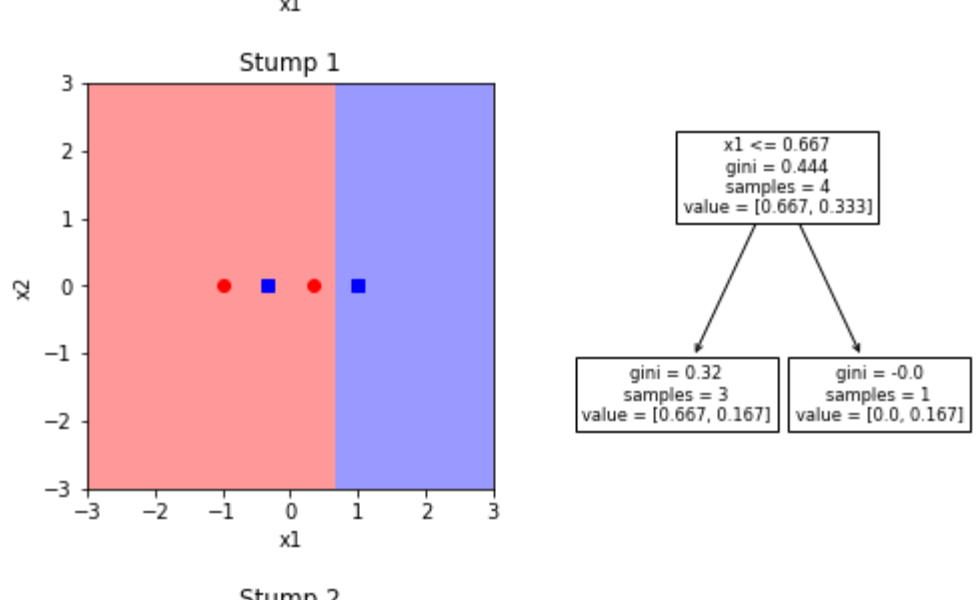
Here you will train an AdaBoost classifier and see how it combines these 'stumps' to give a result comparable to the decision tree you trained. The decision stumps of an AdaBoostClassifier object are available via the *estimators_* attribute, which is a list of *DecisionTreeClassifier* objects.

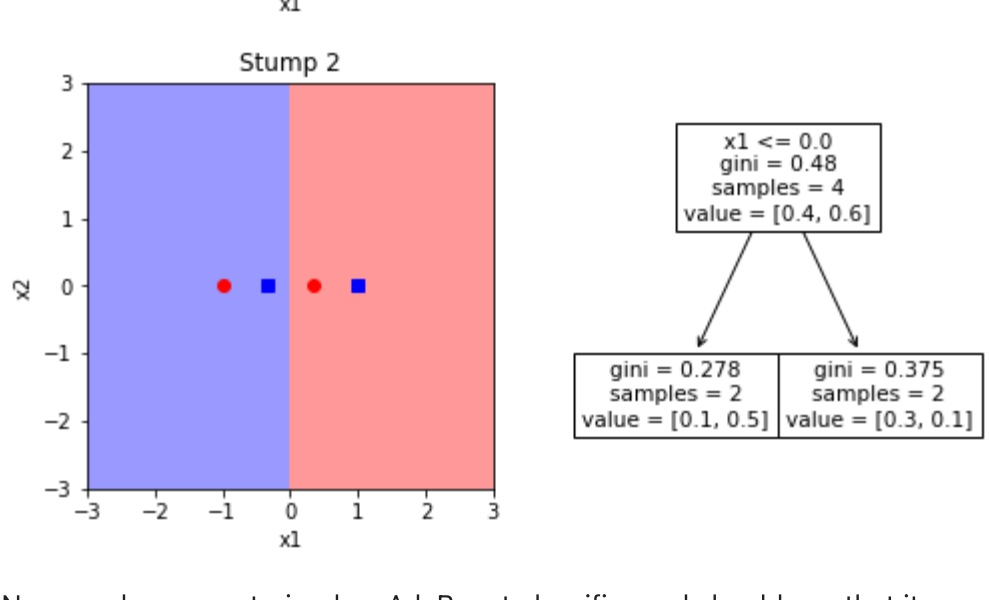
Write a few lines of code to:

1. Train an **AdaBoostClassifier** on the same data set. Use *n_estimators*=3 as an argument when building your classifier in order to keep it simple enough to inspect closely. Specify *algorithm*='SAMME' to use a more classical variant of AdaBoost that more closely aligns with what you will learn in class. Use *random_state*=0 for reproducibility.
2. Plot the decision regions and data like you did for the decision tree.
3. Plot the tree structure of each decision stump in a separate figure. Use *sklearn.tree.plot_tree*.

In [9]:
```
# Your code here. Aim for 4-6 lines.
adaBoostClassifier = sklearn.ensemble.AdaBoostClassifier(n_estimators=3, algorithm='SAMME', random_state=0).fit
for i in range(3):
    plt.figure(figsize=(8 ,4));
    ax = plt.subplot(121);
    ax.set_aspect(1);
    ax.title.set_text(f'Stump {i}');
    plot_data(X, y);
    plot_predict(adaBoostClassifier.estimators_[i]);
    plt.subplot(122);
    sklearn.tree.plot_tree(adaBoostClassifier.estimators_[i],feature_names=['x1', 'x2']);
```







Now you have now trained an AdaBoost classifier and should see that it generates the same predictions as your original decision tree.

Step 3: Plot the individual decision stumps of your boosted ensemble

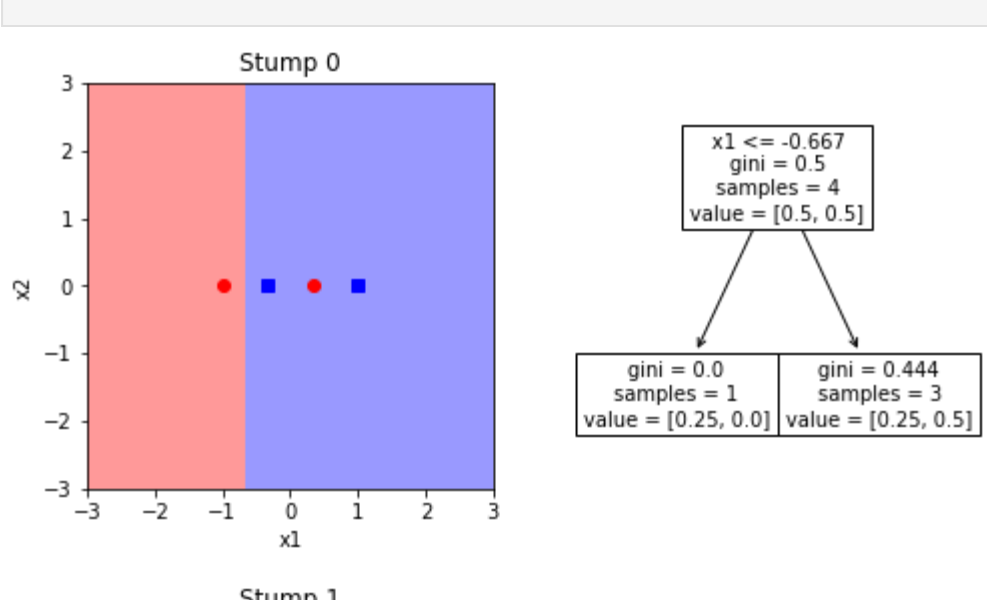
Finally, you should inspect the individual decision 'stumps' that AdaBoost is using to generate its predictions. You should generate three figures (one for each stump, of which there are *n_estimators*). Each figure should have two subplots: on the left plot decision regions of the stump along with the training data, and on the right plot the split that the stump uses to predict class labels. Your first figure should like this:

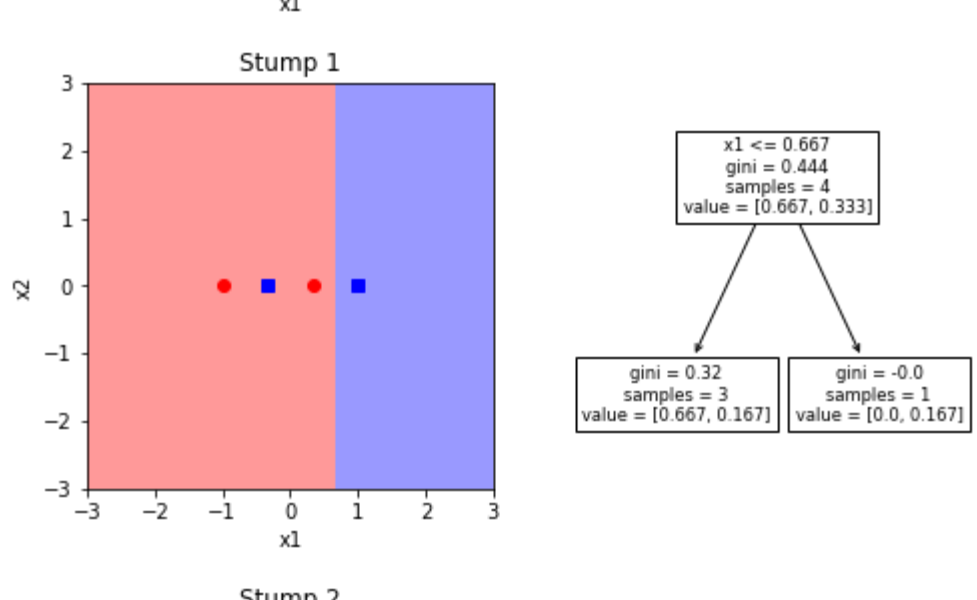
image

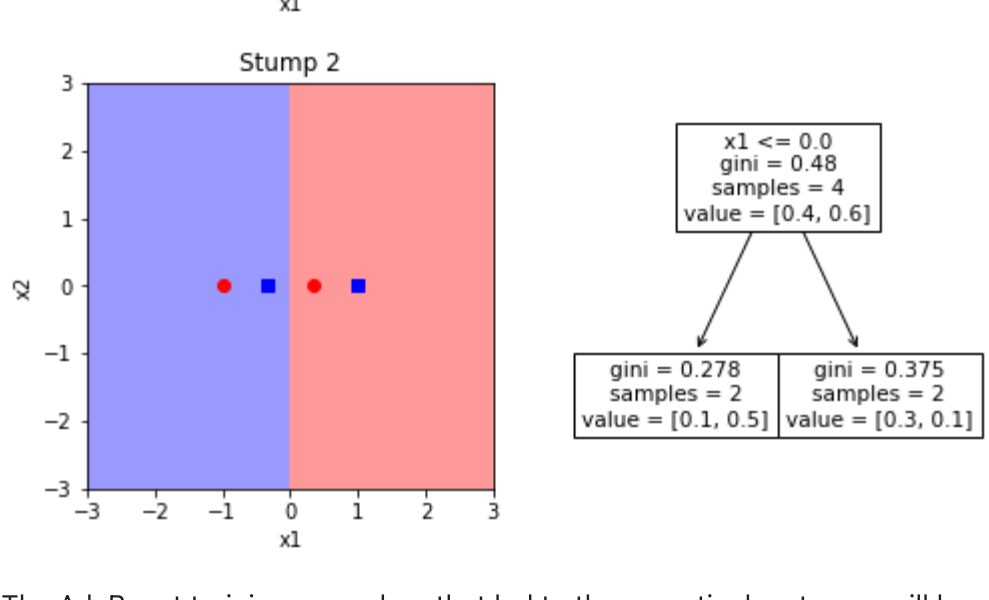
Each decision stump (each estimator) has a weight associated with it. This is the weight the model is given when making predictions. Find the weight using the *estimatorweights* attribute of the AdaBoostClassifier and include the weight as part of the title, as shown above.

Write plotting code below. There should be one plot per estimator in your trained *AdaBoostClassifier* object. The i^{th} plot should have two subplots side-by-side: (1) the decision regions of the i^{th} estimator, and (2) the decision tree (stump) for the i^{th} estimator.

In [10]:
```
# Your code here. Aim for 8-10 lines.
for i in range(3):
    plt.figure(figsize=(8 ,4));
    ax = plt.subplot(121);
    ax.set_aspect(1);
    ax.title.set_text(f'Stump {i}');
    plot_data(X, y);
    plot_predict(adaBoostClassifier.estimators_[i]);
    plt.subplot(122);
    sklearn.tree.plot_tree(adaBoostClassifier.estimators_[i],feature_names=['x1', 'x2']);
```







The AdaBoost training procedure that led to these particular stumps will be made clear in lecture, but is not important for this lab. The rough idea behind AdaBoost training is that the R^{th} decision stump is trained to fix the mistakes (misclassifications) that the previous decision stumps $1, \ldots, R-1$ are currently making on the training set. As long as this new decision stump can fix at least one mistake, we can continue "boosting" by adding more.

Exercise 1.2 — Compute an AdaBoost prediction "by hand"

The point of Exercise 1.1 was to show that complex decision regions can be built not just by deepening a decision tree, but also by (somehow) combining the predictions of decision stumps.

Here you will learn precisely how that 'combining' happens. You will reproduce AdaBoost's predictions by writing your own code to combine the decision stump predictions.

Run the code cell below to define a function that will plot the training data along the x-axis only, using just component x_{i1} of each training point $\mathbf{x}_i = (x_{i1}, x_{i2})$. For this exercise your plots will use the y-axis to represent the probability of class 1 (blue).

In [11]:
```
def plot_data_1d(X, y):
    """
    Plots just the first component of a toy 2D data set.
    Assumes values in range [-3,3] and at most 3 classes.
    """
    plt.plot([-3, 3], [0.5, 0.5], '--k', alpha=0.25)
    plt.plot(X[y==0,0], X[y==0,1]*0+0.5, 'ro', markersize=6)
    plt.plot(X[y==1,0], X[y==1,1]*0+0.5, 'bs', markersize=6)
    plt.plot(X[y==2,0], X[y==2,1]*0+0.5, 'gx', markersize=6, markeredgewidth=2)
    plt.xlim([-3, 3])
    plt.xlabel('x1')
```

Step 1: Plot the prediction of each decision stump, separately.

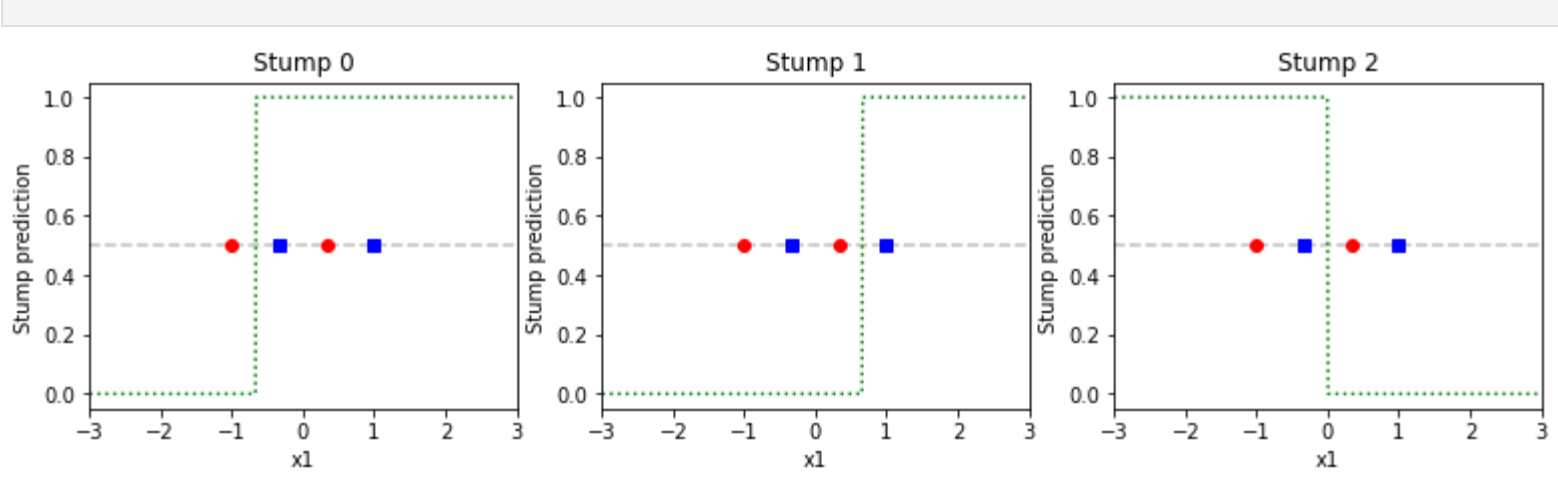
Before combining the decision stumps, plot the predictions of each stump as a 1-dimensional function, so that it will be easier to see how they add up.

Specifically, plot each decision stump's class prediction along the x_1 dimension, using the y-axis to show the class probability. You should generate three figures, one for each decision stump (each estimator). Your first figure should look like: image Note that the green line here is not a decision boundary! We are plotting a 1-dimensional feature space, so the place where the green line crosses the y-axis at 0.5 is the decision boundary! The above plot corresponds to the decision region of the first decision stump you plotted in Exercise 1.1, i.e., red region to the left, blue region to the right.

Write plotting code to show the separate class prediction of each stump.

- Use *plot_data_1d* to plot the four training points.
- For plotting, build an array of the x-axis values. Specifically, use the np.linspace function to create an array of $N = 500$ equally-spaced x values in range $[-3, 3]$.
- To generate y-axis values for your plot, you'll need to call the *predict* function on your already-trained *AdaBoostClassifier* object (from previous exercise). You should generate all N values using a single call to predict. To do this, build an $(N, 2)$ matrix $\mathbf{X}$, where the first column (x_1) is filled with x-axis values and the second column (x_2) is filled with zeros. Then call *predict* once.
- Once you have a single plot working, use a for-loop to generate a separate figure for each decision stump. Use the *n_estimators* attribute of your trained *AdaBoostClassifier*, rather than hard-coding 3 iterations.

In [38]:
```
# Your code here. Aim for 10-12 lines.
fig, ax = plt.subplots(figsize=(13, 3), ncols = 3);
xaxis = np.zeros((500, 2));
xaxis[:, 0] = np.linspace(start=-3, stop = 3, num=500);

for i in range(3):
    yaxis = adaBoostClassifier.estimators_[i].predict(xaxis);
    plt.subplot(131 + i);
    plot_data_1d(X, y);
    ax[i].title.set_text(f'Stump {i}');
    ax[i].set_ylabel('Stump prediction')
    ax[i].plot(xaxis[:, 0], yaxis, color='g', linestyle=':');
```



You should see that the class predictions correspond with the red/blue decision regions for each stump that you observed at the end of Exercise 1.1.

Step 2: Plot a weighted combination of the decision stump predictions.

Here you will plot a *weighted combination* of the individual decision stump predictions. Given an input $\mathbf{x}$, an AdaBoost classifier's decision function $y(\mathbf{x})$ is computed as a weighted combination of decision strump class predictions. Let $f_r(\mathbf{x})$ denote the r^{th} decision stump, and w_r be its weight. A prediction is then:

$$y(\mathbf{x}) = \frac{\sum_{r=1}^{R} w_r f_r(\mathbf{x})}{\sum_{r=1}^{R} w_r}$$

This exercise asks you to implement the above function manually, in Numpy. In effect, you'll implement the "prediction" part of a trained AdaBoost classifier. Since the training data only varies in the first feature (x_1), you'll plot it over the range $x_1 \in [-3, 3]$ and $x_2 = 0$ just like you did in the previous step.

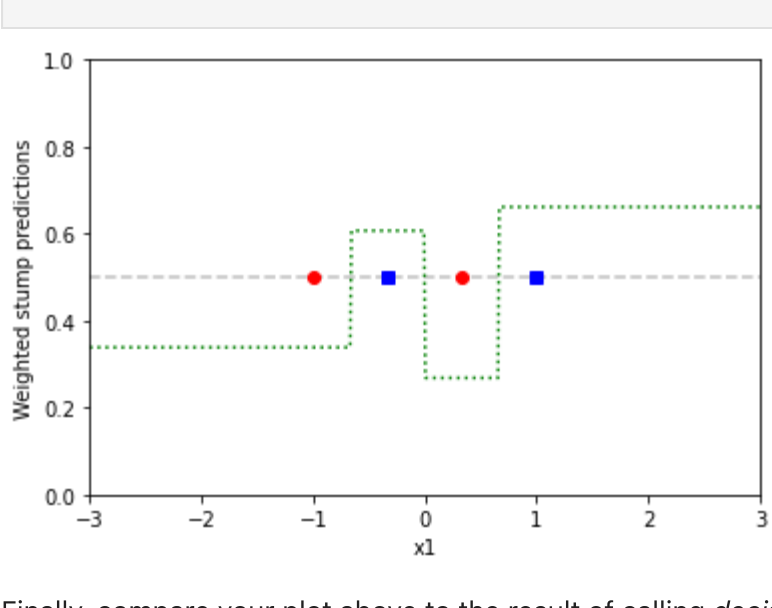
For example, if you made an AdaBoost classifier with $R = 2$ decision stumps, your plot might look something like this:

image

Write code to

- Compute the AdaBoost decision function formula $y(\mathbf{x})$ directly from the *estimators_* and *estimatorweights* attributes of your already-trained AdaBoostClassifier object. You will be plotting the predictions over $x_1 \in [-3, 3]$, so you can just pass your matrix $\mathbf{X}$ from the previous step.
- Plot the $y(\mathbf{x})$ predictions.
- Plot the training data as before (*plot_1d_data*)

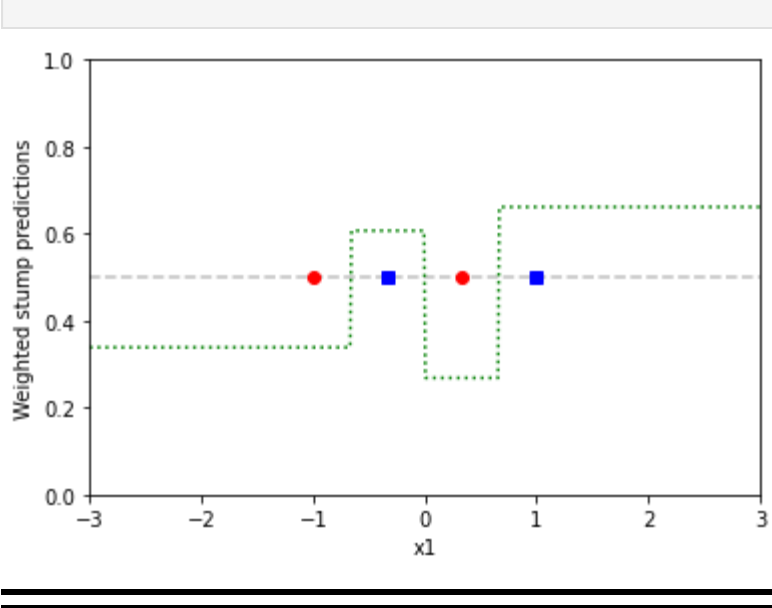
In [162…
```
# Your code for computing y(x) values from the individual estimators.
# Aim for 2-8 lines, and vectorize where possible.
weightedPreds = np.zeros((500,));
for i in range(3):
    predictions = adaBoostClassifier.estimator_weights_[i] * adaBoostClassifier.estimators_[i].predict(xaxis);
    weightedPreds += predictions;
weightedPreds = weightedPreds / adaBoostClassifier.estimator_weights_.sum();

# Your plotting code here. Aim for 5-6 lines.
plot_data_1d(X, y);
plt.ylabel('Weighted stump predictions');
plt.plot(xaxis[:, 0], weightedPreds, color='g', linestyle=':');
plt.ylim([0, 1]);
```



Finally, compare your plot above to the result of calling *decision_function* on your trained *AdaBoostClassifier* object. However, the scikit-learn *AdaBoostClassifier* implementation treats binary classification as special: it generates class predictions in range $[-1, 1]$ rather than in range $[0, 1]$. So, to get the exact same results, you may have to scale (by factor of $\frac{1}{2}$) and shift (by $+\frac{1}{2}$) the *AdaBoostClassifier*'s decision function.

Write a few lines of code to call *decision_function* to generate predictions over the range $x_1 \in [-3, 3]$ just as you did already. (You can re-use variables that you already defined from earlier code cells.) The plot you generate using scikit-learn should be identical to the one you generated by hand, and it should correctly classify all red and blue points in the training set.

In [164…
```
# Your code here. Aim for 6-7 lines.
predictions = adaBoostClassifier.decision_function(xaxis);
predictions = (predictions * 0.5) + 0.5;
plot_data_1d(X, y);
plt.ylabel('Weighted stump predictions');
plt.plot(xaxis[:, 0], predictions, color='g', linestyle=':');
plt.ylim([0, 1]);
```



2. Plotting how AdaBoost increases the training accuracy

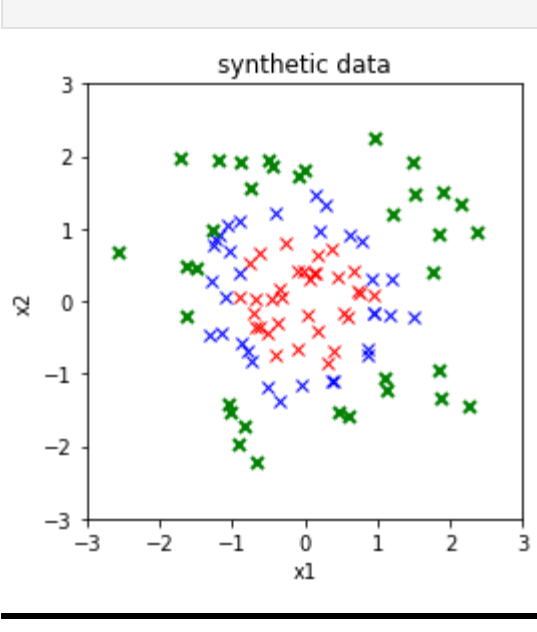
Exercises 2.1–2.2 ask you to train and inspect an AdaBoost classifier on a real data set, to see how each successive 'weak' classifier increases the overall training accuracy when weighted with the previous weak classifiers.

Exercise 2.1 — Create a classification dataset using scikit-learn

Here you'll create a synthetic dataset using one of scikit-learn's dataset utilities. Use the **sklearn.datasets.make_gaussian_quantiles** function to generate a 2-dimensional synthetic dataset with three classes (the default).

Write a few lines of code to generate the dataset and plot it. Use *random_state*=0. Your plot should look exactly like this: image

In [173…
```
# Your code here. Aim for 6-8 lines.
X, y = sklearn.datasets.make_gaussian_quantiles(random_state=0);
plt.plot(X[y==0,0], X[y==0,1], 'rx', markersize=6)
plt.plot(X[y==1,0], X[y==1,1], 'bx', 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');
plt.title('synthetic data');
```

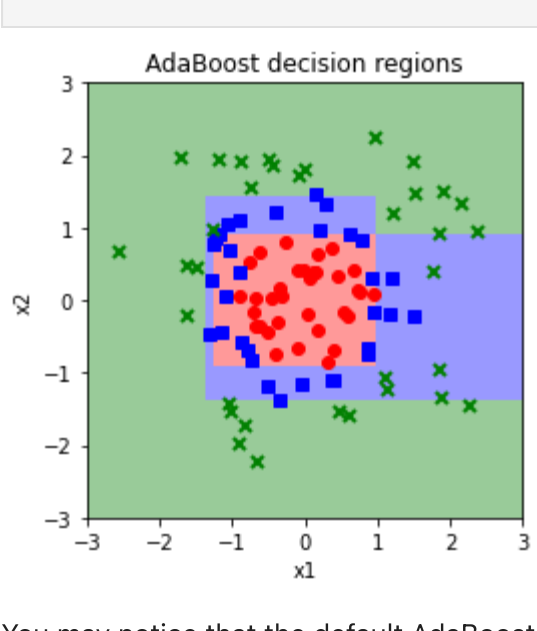


Exercise 2.2 — Train an AdaBoost classifier on the Iris dataset and plot the decision regions

You must train an AdaBoost classifier on the synthetic training set from Exercise 2.1. Your plot should look something like this:
image

Write a few lines of code train the classifier and plot the resulting decision regions. Use the *plot_predict* and *plot_data* functions defined at the start of the lab. Use *random_state*=0 and *algorithm*='SAMME' as before.

In [193…
```
# Your code here. Aim for 4-6 lines.
adaBoostClassifier = sklearn.ensemble.AdaBoostClassifier(algorithm='SAMME.R', random_state=0).fit(X, y);
plot_data(X, y);
plot_predict(adaBoostClassifier);
plt.title('AdaBoost decision regions');
```



You may notice that the default AdaBoost hyperparameters struggle to fit this particular data. You can try playing with the *learning_rate* or *n_estimators* parameters to see how they effect the result.

Exercise 2.3 — Plot the decision regions of successive boosting "rounds"

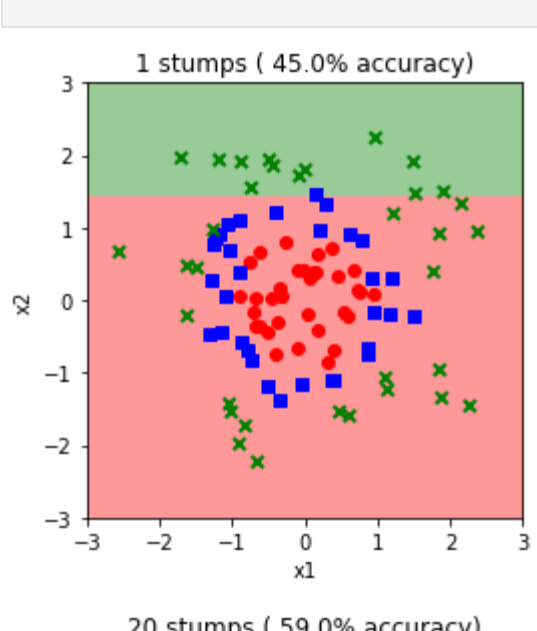
You are asked to plot the decision regions of an AdaBoost classifier with *n_estimators* $\in [1, 20, 40, 60, 80, 100]$. There should be 6 plots and your first plot should look like this:

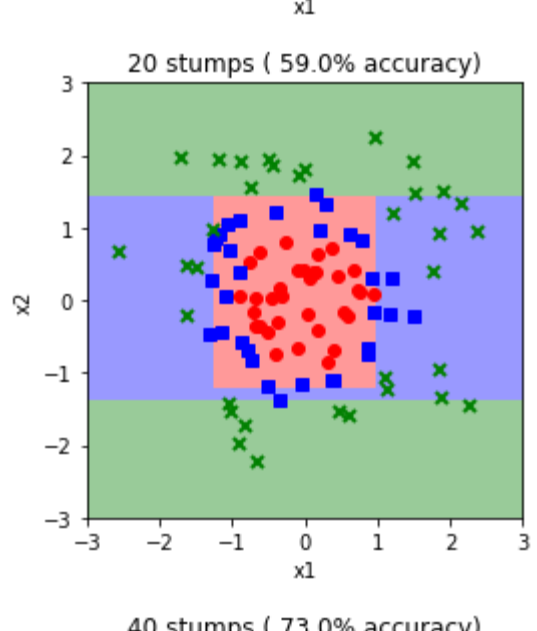
image

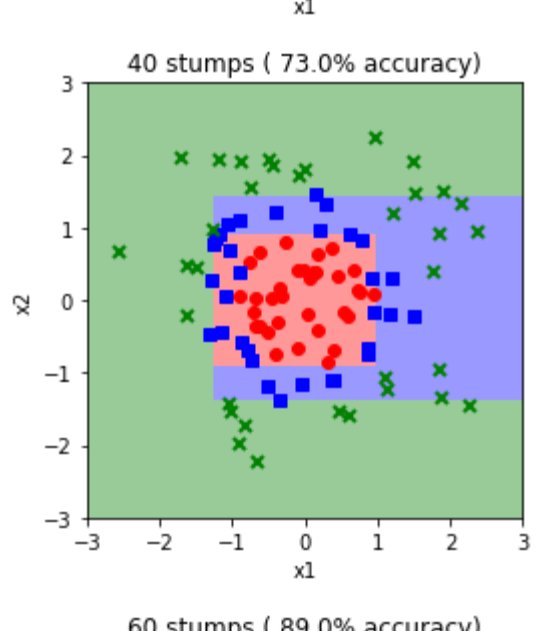
Write a few lines of code to generate the plots.

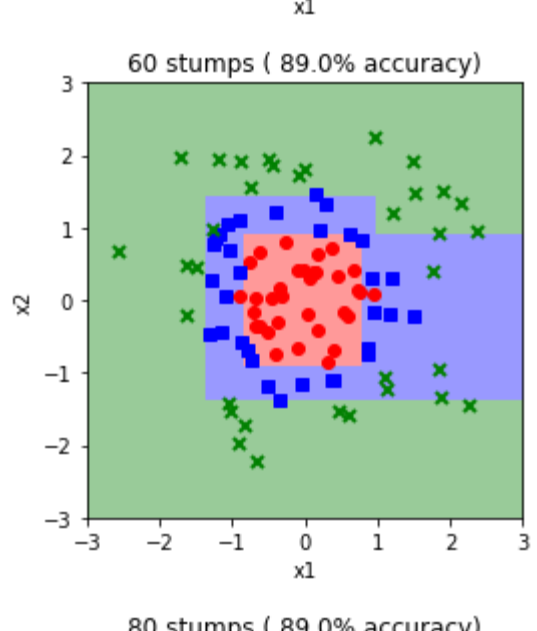
- Use *algorithm*='SAMME' and *random_state*=0 when training, as before.
- Use a for-loop and generate a new plot on each iteration.
- In each plot, display the *accuracy_score* on the training set in the title, as shown above.

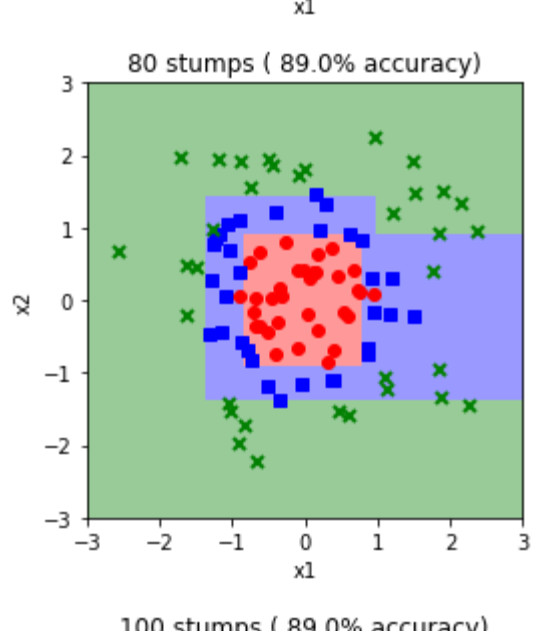
In [194…
```
# Your code here. Aim for 7-10 lines.
for i in (1, 20, 40, 60, 80, 100):
    adaBoostClassifier = sklearn.ensemble.AdaBoostClassifier(algorithm='SAMME.R', n_estimators = i, random_stat
    predictions = adaBoostClassifier.predict(X);
    plot_data(X, y);
    plot_predict(adaBoostClassifier);
    plt.title(f'{i} stumps ({sklearn.metrics.accuracy_score(y, predictions) * 100: .1f}% accuracy)');
    plt.figure();
```

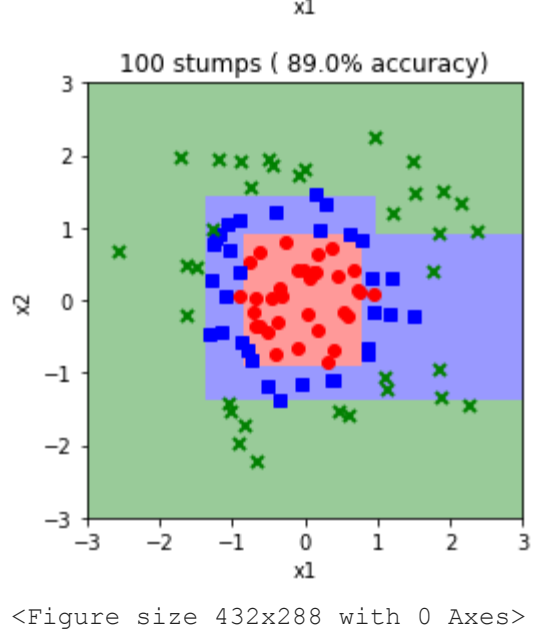













```
<Figure size 432x288 with 0 Axes>
```

Notice that the accuracy fluctuates and is not particularly good even after 100 rounds of boosting.

Exercise 2.4 — Plot the training accuracy of successive boosting "rounds"

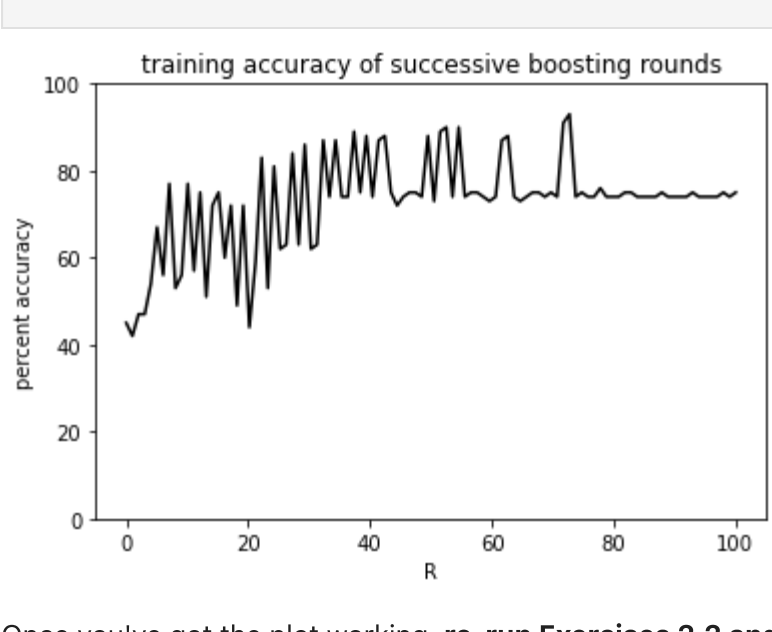
You are asked to plot the accuracy of an AdaBoost classifier with *n_estimators*$= 1, 2, \ldots, 100$ decision stumps (estimators). This is just like Exercise 2.3 except you do not plot the decision regions, and instead keep a record of all the accuracies. Your plot should end up looking like this:

image

Write a few lines of code to generate the plot.

- Use *algorithm*='SAMME' and *random_state*=0 as before.
- Compute a list of 100 accuracies, one for each *n_estimators* setting.
- Plot the accuracies with a single call to Matplotlib's *plot* function.

In [195…
```
# Your code here. Aim for 9-12 lines.
accuracyScores = np.empty((100));
for i in range(1, 101):
    adaBoostClassifier = sklearn.ensemble.AdaBoostClassifier(n_estimators = i, algorithm='SAMME', random_state=
    predictions = adaBoostClassifier.predict(X);
    accuracyScores[i-1] = sklearn.metrics.accuracy_score(y, predictions) * 100;

plt.plot(np.linspace(start=0, stop=100, num=100), accuracyScores, c='black');
plt.ylim([0, 100]);
plt.xlabel('R');
plt.ylabel('percent accuracy');
plt.title('training accuracy of successive boosting rounds');
```



Once you've got the plot working, **re-run Exercises 2.2 and 2.3 using *algorithm*="SAMME.R"** instead. Notice the training accuracy improves. The SAMME.R algorithm (R for "real numbers") is a modification of AdaBoost that builds a weighted combination of the estimators' *decision functions* (a real number), rather than a weighted combination of their *class predictions* (discrete). Scikit-learn uses SAMME.R by default.