Outline

1. Nonlinear classifiers
2. Kernel trick and kernel SVM
3. **Ensemble Methods - Boosting, Random Forests**
4. Classification Summary

Ensemble Classifiers

- *Why trust only one expert?*
  - In real life, we may consult several experts, or go with the "wisdom of the crowd"
  - In machine learning, *why trust only one classifier?*

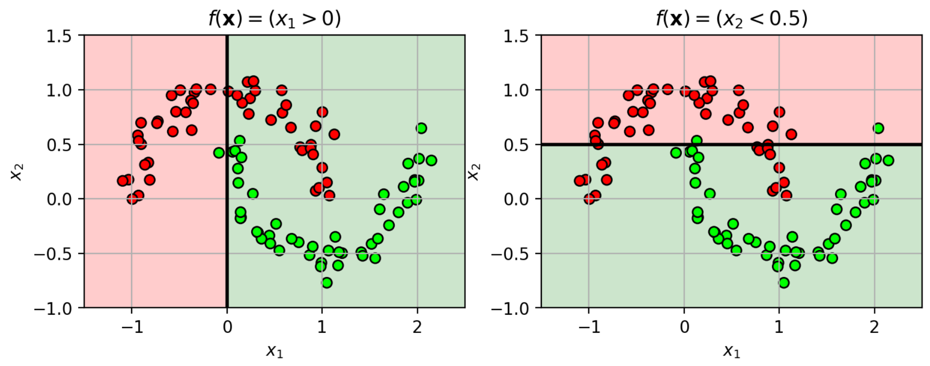
- Ensemble methods aim to combine multiple classifiers together to form a better classifier.
- Examples:
  - **boosting** - training multiple classifiers, each focusing on errors made by previous classifiers.
  - **bagging** - training multiple classifiers from random selection of training data

AdaBoost - Adaptive Boosting

- Base classifier is a "weak learner"
  - A simple classifier that can be slightly better than random chance (>50%)
  - Example: *decision stump classifier*
    - check if feature value is above (or below) a threshold.
    - $y = h(x) = \begin{cases} +1, & x_j \geq T \\ -1, & x_j < T \end{cases}$

In [4]:
```
wlfig
```

Out[4]:



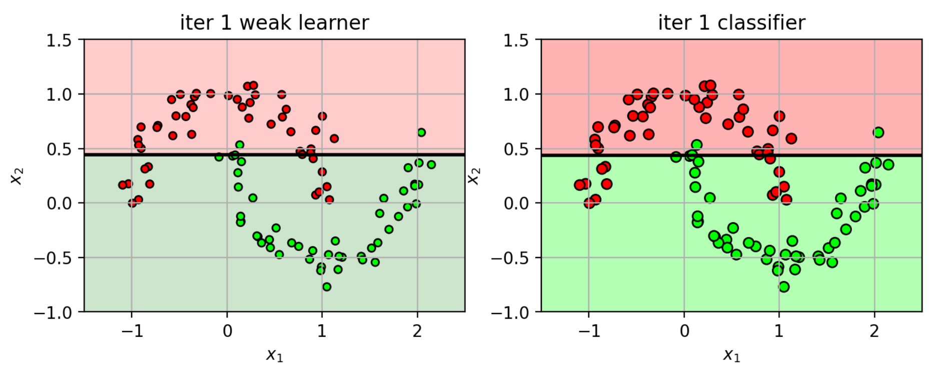
- **Idea:** train weak classifiers sequentially
- In each iteration,
  - Pick a weak learner $h_t(\mathbf{x})$ that best carves out the input space.
  - The weak learner should focus on data that is misclassified.
    - Apply weights to each sample in the training data.
    - Higher weights give more priority to difficult samples.
  - Combine all the weak learners into a strong classifier: $f_t(\mathbf{x}) = f_{t-1}(\mathbf{x}) + \alpha_t h_t(\mathbf{x})$
    - α_t is a weight for each weak learner.

Iteration 1

- Initially, weights for all training samples are equal: $w_i = 1/N$
    - Pick best weak learner $h_1(x)$

In [7]:
```
plts[1]
```

Out[7]:

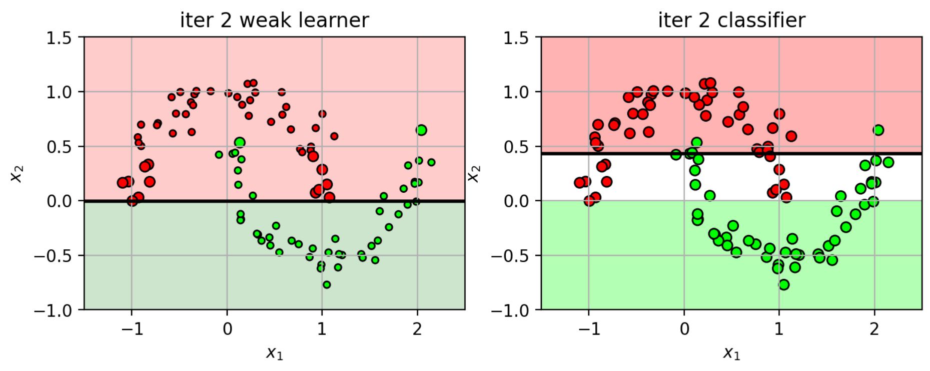


Iteration 2 (part 1)

- points are re-weighted based on the previous errors:
    - increase weights for misclassified samples: $w_i = w_i e^{\alpha}$
    - decrease weights for correctly classified samples: $w_i = w_i e^{-\alpha}$
    - $\alpha = 0.5 \log \frac{1-err}{err}$ is based on the weighted error of the previous weak learner.
    - (larger circles indicates higher weight)

In [8]:
```
plts[2]
```

Out[8]:

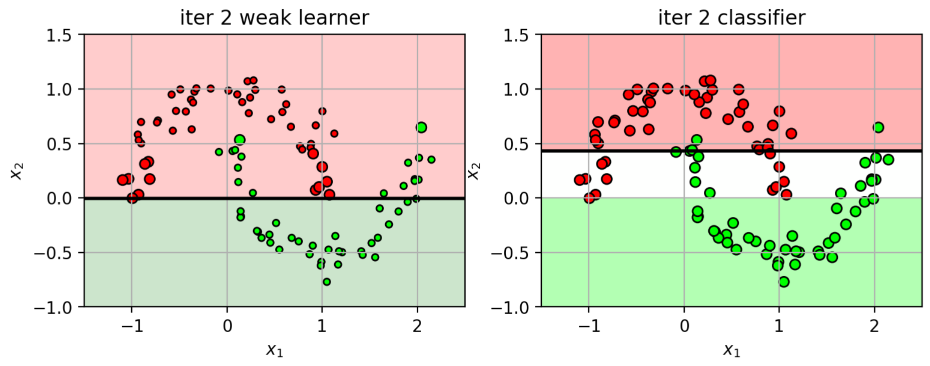


Iteration 2 (part 2)

- using the weighted data, train another weak learner $h_2(\mathbf{x})$.
- the classifier function is the weighted sum of weak learners
    - $f_2(\mathbf{x}) = f_1(\mathbf{x}) + \alpha_2 h_2(\mathbf{x})$

In [9]:
```
plts[2]
```

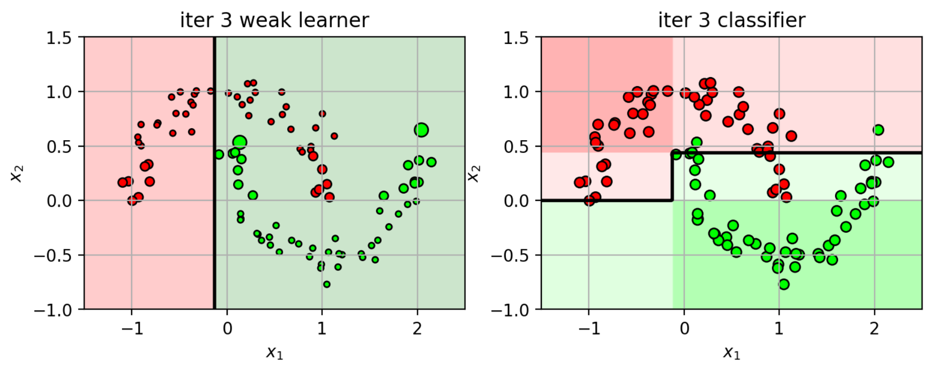
Out[9]:



Keep iterating...

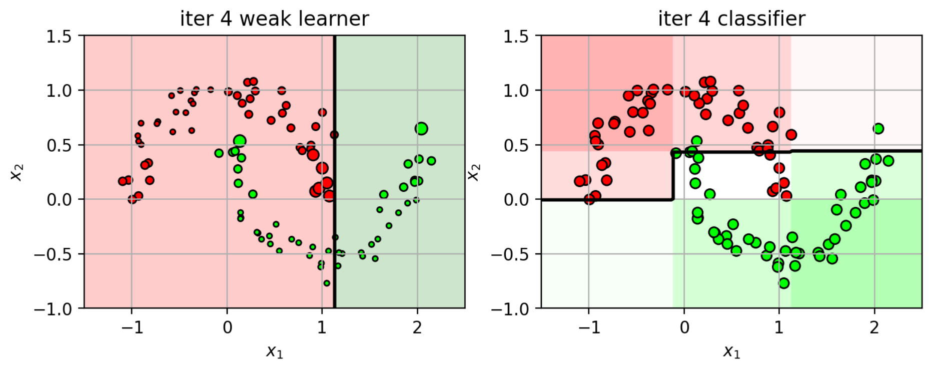
In [10]:
```
plts[3]
```

Out[10]:



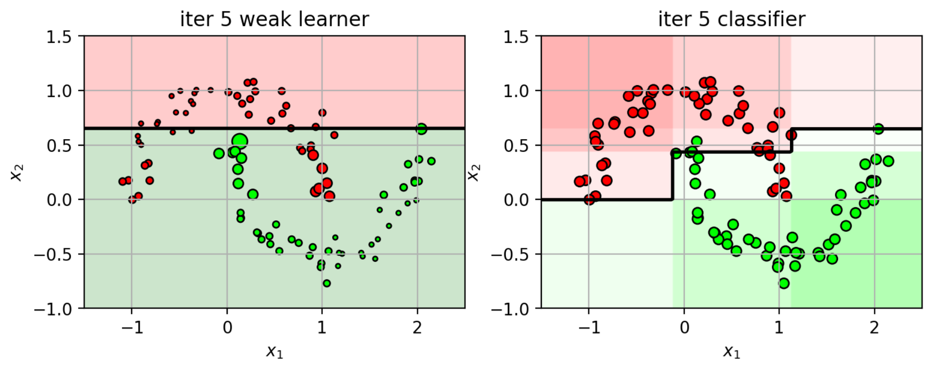
In [11]:
```
plts[4]
```

Out[11]:



In [12]:
```
plts[5]
```

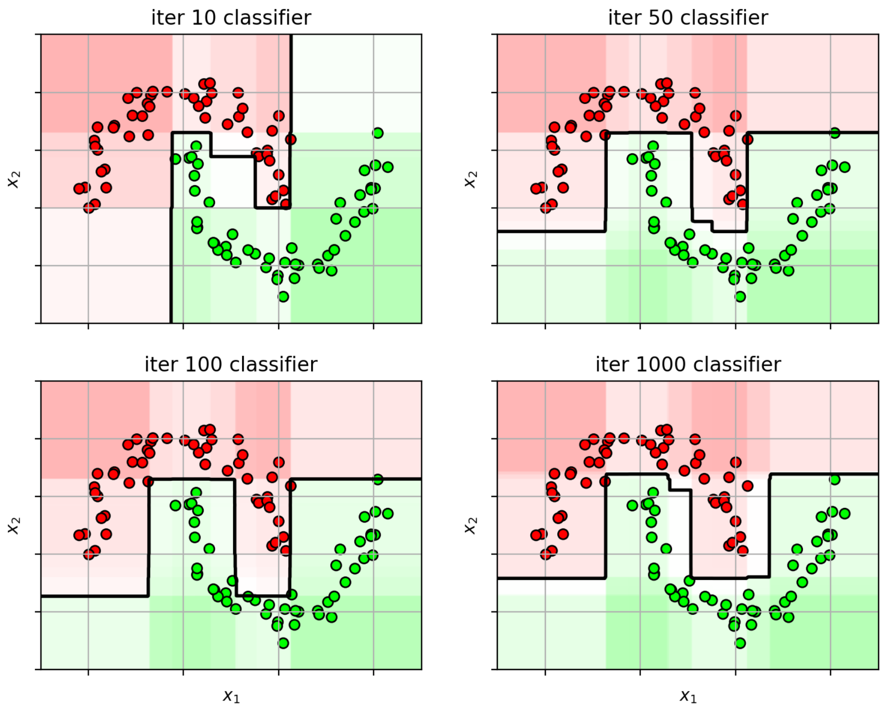
Out[12]:



- After many iterations...

In [14]:
```
adafig
```

Out[14]:



Adaboost Algorithm

- Given data $\{(\mathbf{x}_i, y_i)\}$.
- Initialize data weights, $w_i = 1/N, \forall i$.
- For t = 1 to T,
  - choose weak learner $h_t(\mathbf{x})$
    - minimize the weighted classification error: $\epsilon_t = \sum_{i=1}^N w_i 1(h_t(\mathbf{x}_i) \neq y_i)$.
  - Set the weak learner weight: $\alpha_t = \frac{1}{2}\log(\frac{1-\epsilon_t}{\epsilon_t})$
  - Add to ensemble: $f_t(\mathbf{x}) = f_{t-1}(\mathbf{x}) + \alpha_t h_t(\mathbf{x})$.
  - Update data weights:
    - for all $\mathbf{x}_i$ misclassified, increase weight: $w_i \leftarrow w_i e^{\alpha_t}$.
    - for all $\mathbf{x}_i$ correctly classified, decrease weight: $w_i \leftarrow w_i e^{-\alpha_t}$.
    - normalize weights, so that $\sum_i w_i = 1$.

Adaboost loss function

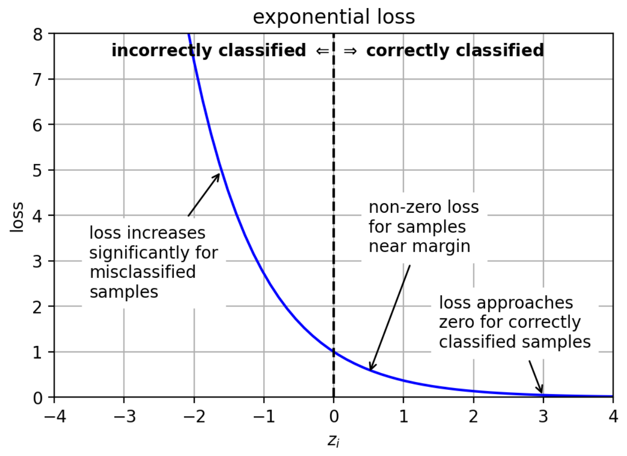
- It can be shown that Adaboost is minimizing:

$$\min_f \sum_i e^{-y_i f(\mathbf{x}_i)}$$

- Thus, it is an exponential loss function
  - $L(z_i) = e^{-z_i}$
    - $z_i = y_i f(\mathbf{x}_i)$
  - very sensitive to misclassified outliers.

In [16]:
```
lossfig
```

Out[16]:

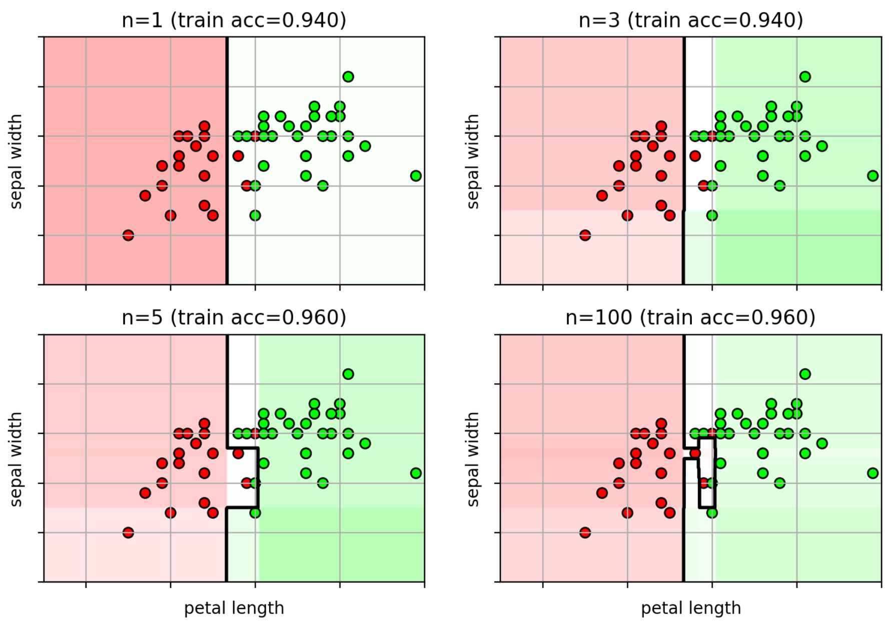


Example on Iris data

- Too many weak-learners and AdaBoost carves out space for the outliers.

In [20]:
```
irisfig
```

Out[20]:



- Note: sklearn uses Real AdaBoost by default
  - uses class probabilities instead of binary class predictions (usually converges faster and has lower error)
- Two hyperparameters regularize boosting:
  1. the number of weak learners - how many boosting iterations
  2. the "learning rate" - a factor for scaling the contribution of each weak-learner
  - usually between 0 and 1.
  - Also called "shrinkage".
- Smaller learning rates require more weak learners.
- Estimate with cross-validation.

In [22]:
```
# setup the list of parameters to try
paramgrid = {'learning_rate': logspace(-6,0,20),
             'n_estimators': array([1, 2, 3, 5, 10, 15, 20, 25, 50, 100, 200, 500, 1000])
            }
print(paramgrid)

# setup the cross-validation object
# (NOTE: using parallelization in GridSearchCV, not in AdaBoost)
adacv = model_selection.GridSearchCV(ensemble.AdaBoostClassifier(random_state=4487),
                                     paramgrid, cv=5, n_jobs=-1)
```

```
# run cross-validation (train for each split)
adacv.fit(trainX, trainY);

print("best params:", adacv.best_params_)
```

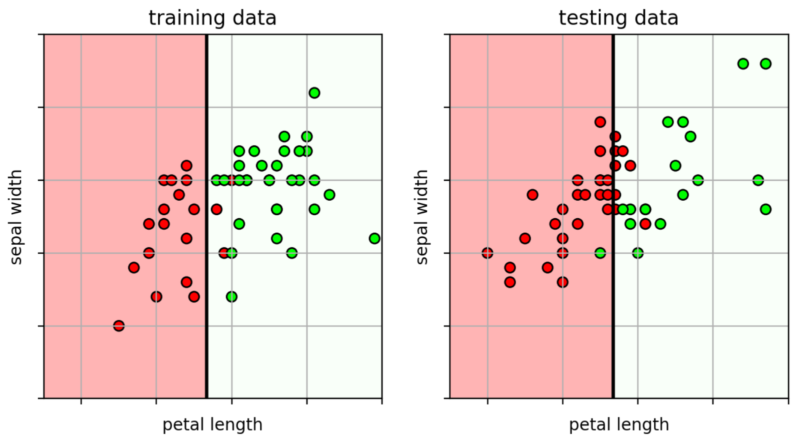
```
{'learning_rate': array([1.00000000e-06, 2.06913808e-06, 4.28133240e-06, 8.8586679
0e-06,
       1.83298071e-05, 3.79269019e-05, 7.84759970e-05, 1.62377674e-04,
       3.35981829e-04, 6.95192796e-04, 1.43844989e-03, 2.97635144e-03,
       6.15848211e-03, 1.27427499e-02, 2.63665090e-02, 5.45559478e-02,
       1.12883789e-01, 2.33572147e-01, 4.83293024e-01, 1.00000000e+00]), 'n_estima
tors': array([   1,    2,    3,    5,   10,   15,   20,   25,   50,  100,  200,
        500, 1000])}
best params: {'learning_rate': 1e-06, 'n_estimators': 1}
```

In [24]:
```
# predict from the model
predY = adacv.predict(testX)

# calculate accuracy
acc       = metrics.accuracy_score(testY, predY)
print("test accuracy =", acc)
```

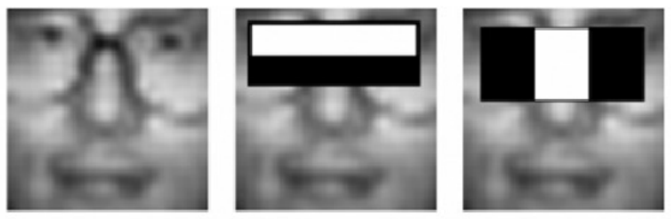
```
test accuracy = 0.82
```

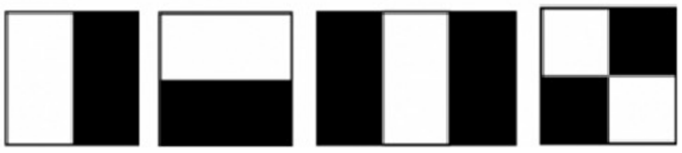
In [26]:
```
ifig2
```

Out[26]:



- Boosting can do feature selection
  - each decision stump classifier looks at one feature
- One of the original face detection methods (Viola-Jones) used Boosting.
  - extract a lot of image features from the face
  - during training, Boosting learns which ones are the most useful.



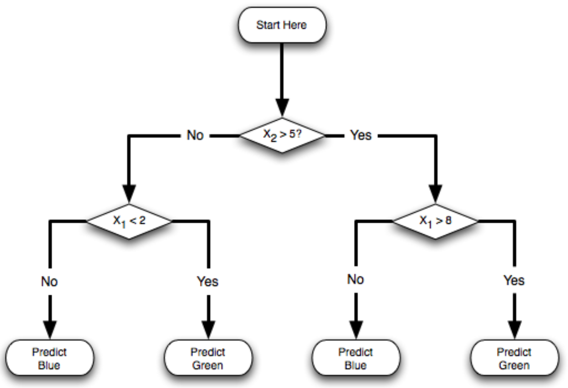


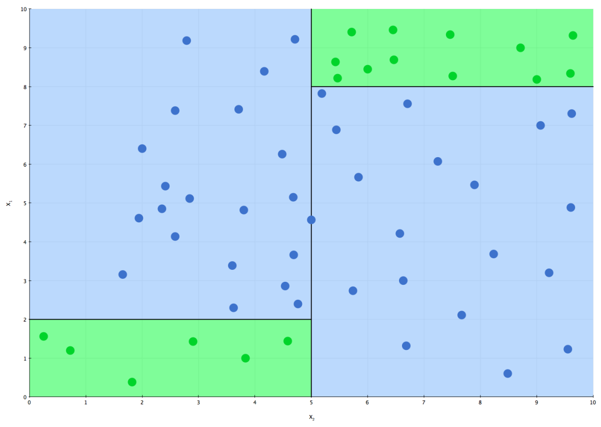
Gradient Boosting

- Variant of boosting
  - each iteration fits the residual between the current predictions and the true labels.

    - the residual is computed as the gradient of the loss function.
- It's a gradient descent algorithm
    - in each iteration, the weak learner fits the gradient of the loss
        - $h_t(\mathbf{x}) \approx \frac{dL}{d\mathbf{f}}$
- and adds it to the function:
    - $f_t(\mathbf{x}) = f_{t-1}(\mathbf{x}) - \alpha_t h_t(\mathbf{x}) \approx f_{t-1}(\mathbf{x}) - \alpha_t \frac{dL}{d\mathbf{f}_{t-1}}$
- Generalizes boosting to other loss functions

- Typically uses decision trees for the weak learner:
    - At each node, move down the tree based on that node's criteria.
    - leaf node contains the prediction



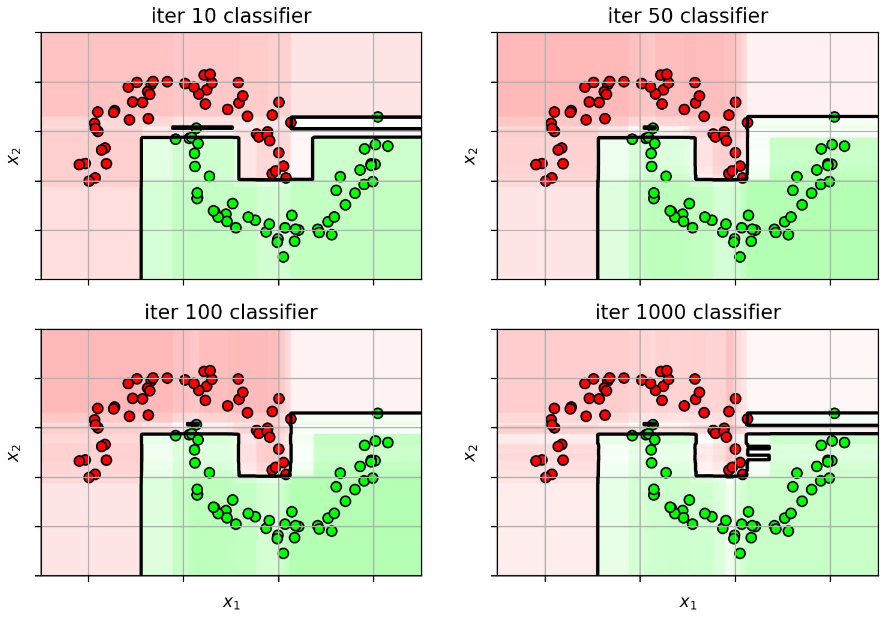


Example

- more iterations tends to overfit severely
    - because the "weak" classifier is actually strong (decision tree).

In [28]:
```
xgbfig
```

Out[28]:



Cross-validation

- select the best hyperparameters
    - number of estimators
    - learning rate (shrinkage term)

In [29]:
```
# use the XGBoost package, compatible with sklearn
import xgboost as xgb

# use "multi:softprob" for multi-class classification
xclf = xgb.XGBClassifier(objective="binary:logistic", eval_metric='logloss',
                         random_state=4487, use_label_encoder=False)

# setup the list of parameters to try
paramgrid = {'learning_rate': logspace(-6,0,20),
             'n_estimators': array([1, 2, 3, 5, 10, 15, 20, 25, 50, 100, 200, 500, 1000])
            }
print(paramgrid)

# setup the cross-validation object
xgbcv = model_selection.GridSearchCV(xclf, paramgrid, cv=5, n_jobs=-1)

# run cross-validation (train for each split)
xgbcv.fit(X3, Y3);

print("best params:", xgbcv.best_params_)
```

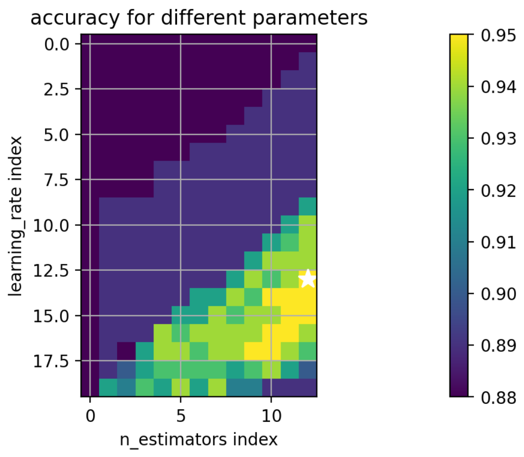
```
{'learning_rate': array([1.00000000e-06, 2.06913808e-06, 4.28133240e-06, 8.8586679
0e-06,
       1.83298071e-05, 3.79269019e-05, 7.84759970e-05, 1.62377674e-04,
       3.35981829e-04, 6.95192796e-04, 1.43844989e-03, 2.97635144e-03,
       6.15848211e-03, 1.27427499e-02, 2.63665090e-02, 5.45559478e-02,
       1.12883789e-01, 2.33572147e-01, 4.83293024e-01, 1.00000000e+00]), 'n_estima
tors': array([   1,    2,    3,    5,   10,   15,   20,   25,   50,  100,  200,
        500, 1000])}
```

```
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
```

```
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
/Users/abc/miniforge3/envs/py39np/lib/python3.9/site-packages/xgboost/compat.py:3
6: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas
in a future version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
```

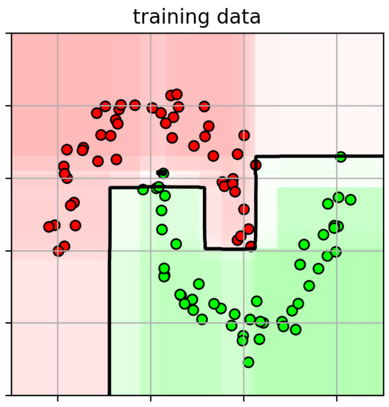
```
best params: {'learning_rate': 0.012742749857031322, 'n_estimators': 1000}
```

In [30]:
```python
(avgscores, pnames, bestind) = extract_grid_scores(xgbcv, paramgrid)
paramfig = plt.figure()
plt.imshow(avgscores, interpolation='nearest')
plt.plot(bestind[1], bestind[0], '*w', markersize=12)
plt.ylabel(pnames[0] + ' index'); plt.xlabel(pnames[1] + ' index')
plt.grid(True)
plt.title('accuracy for different parameters')
plt.colorbar()
plt.axis('image');
```



In [32]:
```python
ifig2
```

Out[32]:



- Since decision trees are used, there are a lot of hyperparameters to tune for the decision tree.
  - `max_depth` : maximum depth of the tree
  - `gamma` : minimum loss reduction in order to split a leaf.
  - `colsample_bytree` : fraction of features to randomly subsample when building a tree.
  - `subsample` : fraction of training data to subsample during each boosting iteration (for each tree).

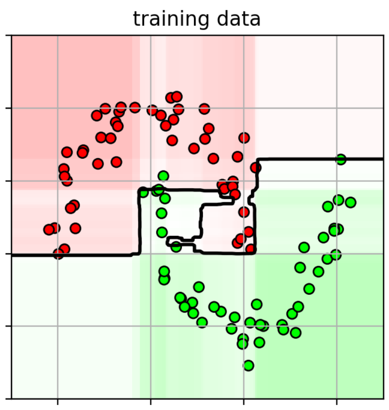
- **Problem:** Too many parameters to use grid-search!
- **Solution:** use randomized search
  - specify probability distributions for the parameters to try
    - `stats.uniform(a, b)` = uniform distribution between [a, a+b]
    - `stats.randint(a,b)` = random integer between [a, b]

In [33]:
```python
# setup dictionary of distributions for each parameter
paramsampler = {
    "colsample_bytree": stats.uniform(0.7, 0.3),   # default=1
    "gamma":            stats.uniform(0, 0.5),    # default=0
    "max_depth":        stats.randint(2, 6),      # default=6
    "subsample":        stats.uniform(0.6, 0.4),  # default=1
    "learning_rate":    stats.uniform(.001,1),    # default=1 (could also use loguniform)
    "n_estimators":     stats.randint(10, 1000),
}

xclf = xgb.XGBClassifier(objective="binary:logistic", eval_metric='logloss',
                         random_state=4487, use_label_encoder=False)

# cross-validation via random search
# n_iter = number of parameter combinations to try
xgbrcv = model_selection.RandomizedSearchCV(xclf, param_distributions=paramsampler,
                                            random_state=4487, n_iter=200, cv=5,
                                            verbose=1, n_jobs=-1)

xgbrcv.fit(X3, Y3)
print("best params:", xgbrcv.best_params_)
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
best params: {'colsample_bytree': 0.9682214619643752, 'gamma': 0.4341101816965796
7, 'learning_rate': 0.014847933781299671, 'max_depth': 4, 'n_estimators': 152, 'su
bsample': 0.6743715045033899}
```

In [35]:
```
ifig2
```

Out[35]:



Boosting Summary

- **Ensemble Classifier:**
  - Combine the outputs of many "weak" classifiers to make a "strong" classifier
- **Training:**
  - In each iteration,
    - training data is re-weighted based on whether it is correctly classified or not.
    - weak classifier focuses on misclassified data from previous iterations.
  - Use cross-validation to pick number of weak learners.

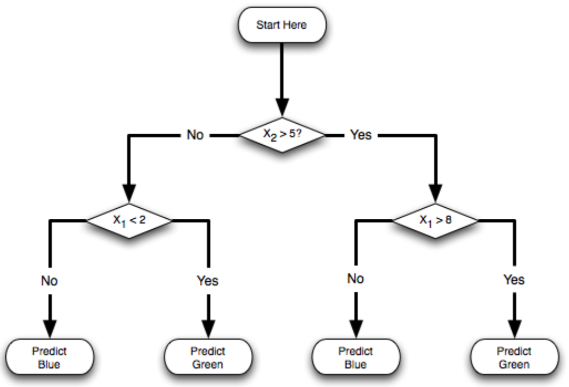
- **Advantages:**
  - Good generalization performance
  - Built-in features selection - decision stump selects one feature at a time.
- **Disadvantages:**
  - Sensitive to outliers.

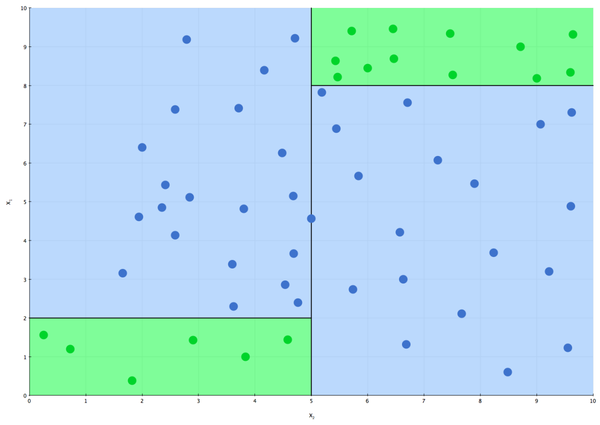
Outline

1. Nonlinear classifiers
2. Kernel trick and kernel SVM
3. **Ensemble Methods - Boosting, Random Forests**
4. Classification Summary

Decision Tree

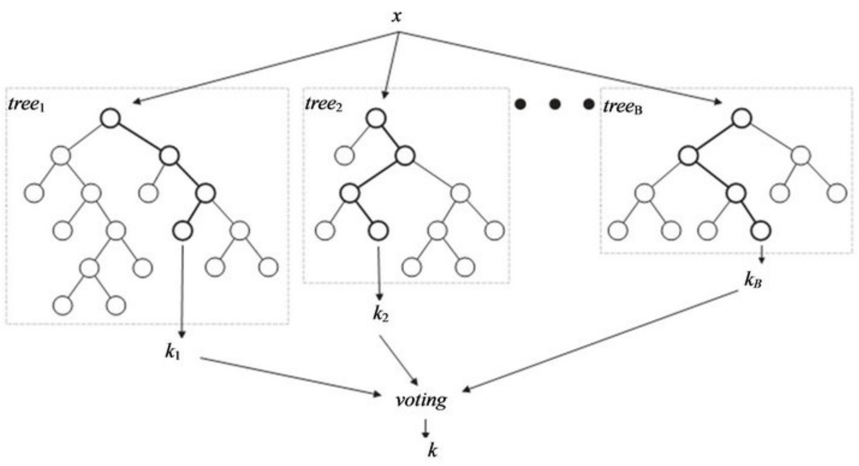
- Simple "Rule-based" classifier
  - At each node, move down the tree based on that node's criteria.
  - leaf node contains the prediction
- **Advantage:** can create complex conjunction of rules
- **Disadvantage:** easy to overfit by itself
  - can fix with bagging!





Random Forest Classifier

- Use **bagging** to make an ensemble of Decision Tree Classifiers
  - for each *Decision Tree Classifier*
    - create a new training set by randomly sampling from the training set
    - for each split in a tree, select a random subset of features to use
- for a test sample, the prediction is aggregated over all trees.



In [36]:
```
# learn a RF classifier
# use 4 trees
```

```
clf = ensemble.RandomForestClassifier(n_estimators=4, random_state=4487, n_jobs=-1)
clf.fit(X3, Y3)
```

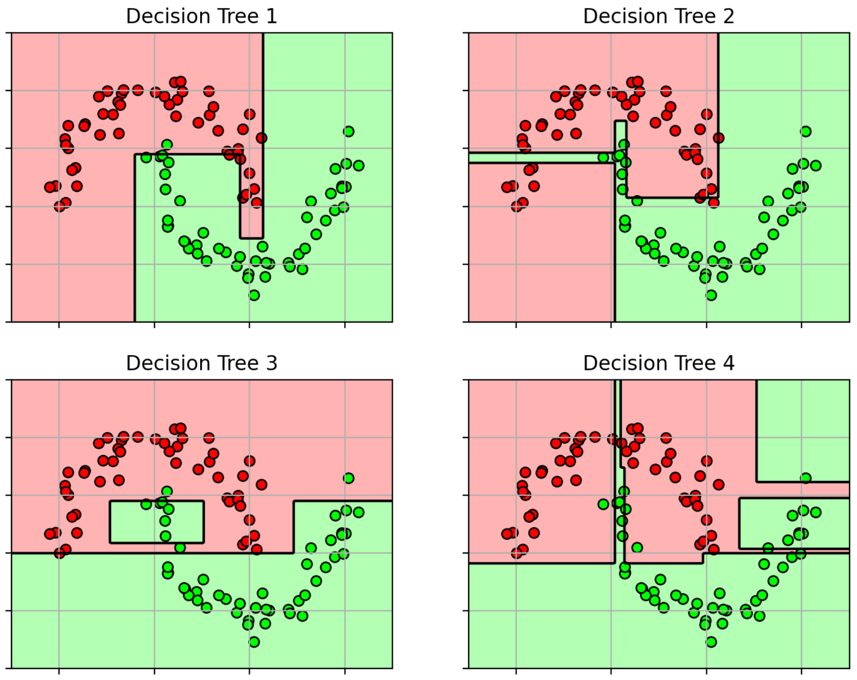
Out[36]:

```
▾ RandomForestClassifier
RandomForestClassifier(n_estimators=4, n_jobs=-1, random_state=4487)
```

- Here are the 4 decision trees
  - each uses a different random sampling of original training set

In [39]:
```
dtfig
```

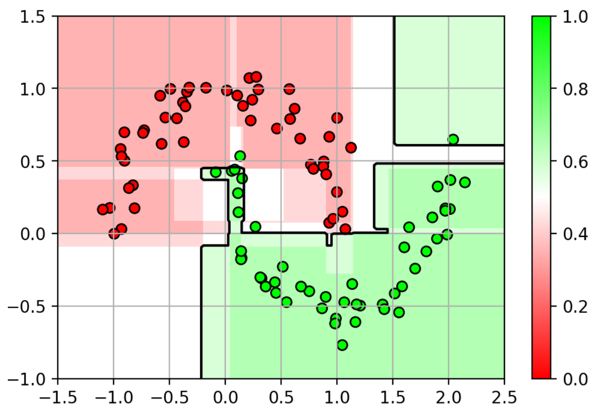
Out[39]:



- and the aggregated classifier

In [40]:
```
rffig
```

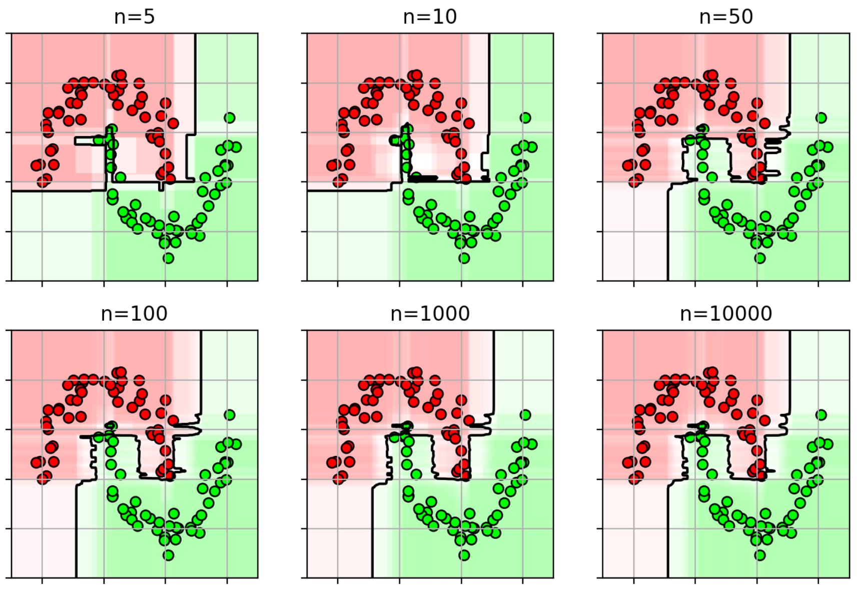
Out[40]:



- Using more trees

In [42]:
```
rffig
```

Out[42]:

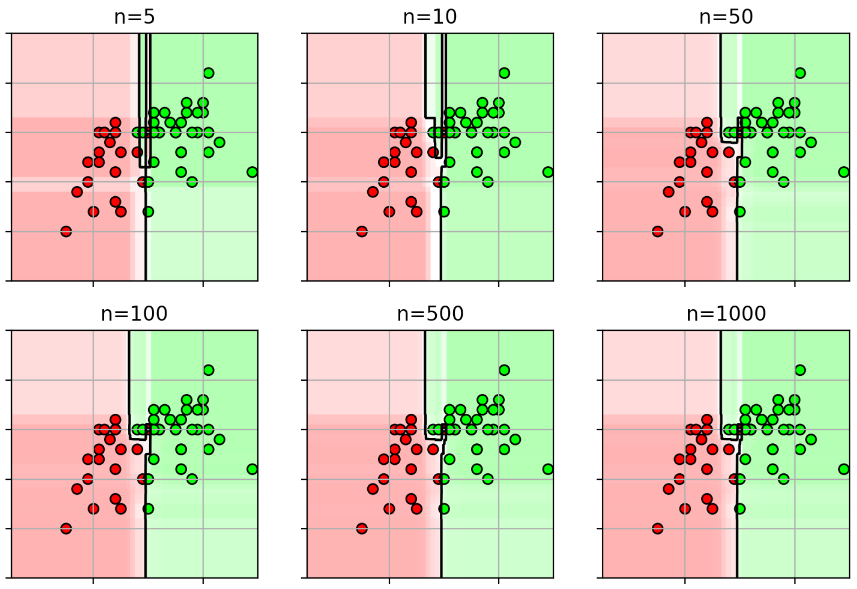


- Try on the iris data

In [43]:
```
# learn RF classifiers for different n_estimators
clfs = {}
for i,n in enumerate([5, 10, 50, 100, 500, 1000]):
    clfs[n] = ensemble.RandomForestClassifier(n_estimators=n, random_state=4487, n_jobs=-1)
    clfs[n].fit(trainX, trainY)
```

In [45]:
```
rfnfig
```

Out[45]:

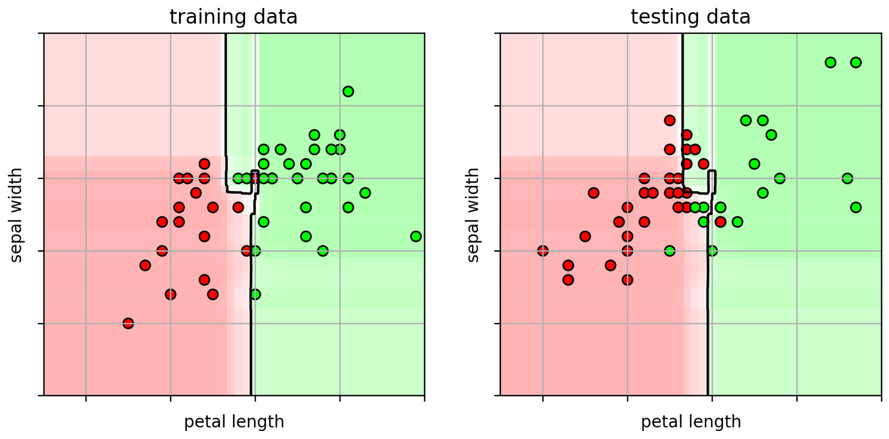


In [46]:
```
# predict from the model
predY = clfs[1000].predict(testX)

# calculate accuracy
acc      = metrics.accuracy_score(testY, predY)
print("test accuracy =", acc)
```

```
test accuracy = 0.78
```

In [48]:
```
# classifier boundary w/ training and test data
ifig3
```

Out[48]:



- Important parameters for cross-validation
  - `max_features` - maximum number of features used for each split
  - `max_depth` - maximum depth of a decision tree
  - `min_samples_split` - minimum fraction of samples to split a node.
  - `min_samples_leaf` - min fraction of samples in a leaf node.

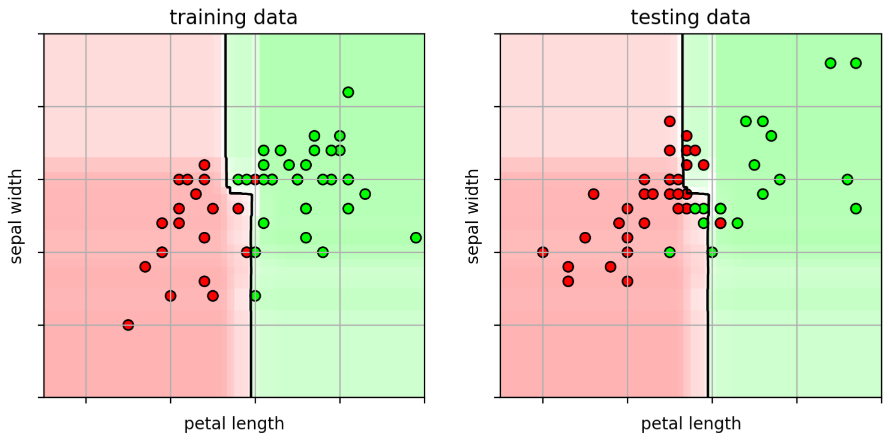
In [49]:
```
# setup the list of parameters to try
paramsampler = {#'max_features': stats.uniform(0,1.0),
                 'max_depth':          stats.randint(1,5),
                 'min_samples_split': stats.uniform(0,0.5),
                 'min_samples_leaf':  stats.uniform(0,0.5),
               }

# setup the cross-validation object
rfrcv = model_selection.RandomizedSearchCV(
                                ensemble.RandomForestClassifier(n_estimators=100, random_state=4
                                param_distributions=paramsampler,
                                random_state=4487, n_iter=1000, cv=5,
                                verbose=1, n_jobs=-1)

# run cross-validation (train for each split)
rfrcv.fit(trainX, trainY);

print("best params:", rfrcv.best_params_)
```

```
Fitting 5 folds for each of 1000 candidates, totalling 5000 fits
best params: {'max_depth': 4, 'min_samples_leaf': 0.013009207368046005, 'min_sampl
es_split': 0.03382188764018978 5}
```

- Result

In [51]:
```
ifig3
```

Out[51]:



In [52]:
```
# predict from the model
predY = rfrcv.predict(testX)
```

```
# calculate accuracy
acc       = metrics.accuracy_score(testY, predY)
print("test accuracy =", acc)
```

```
test accuracy = 0.8
```

Random Forest Summary

- **Ensemble Classifier & Training:**
  - aggregate predictions over several decision trees
  - trained using different subsets of data, and different subsets of features.
- **Advantages**
  - non-linear decision boundary.
  - can do feature selection.
  - good generalization.
  - fast.
- **Disadvantages**
  - can be sensitive to outliers
  - based on trees -- cannot well represent "diagonal" decision boundaries.