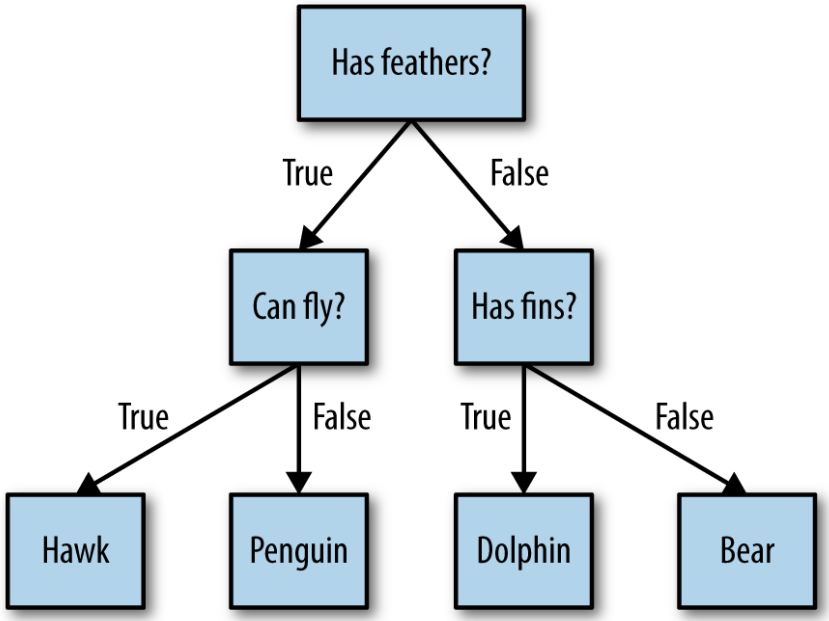
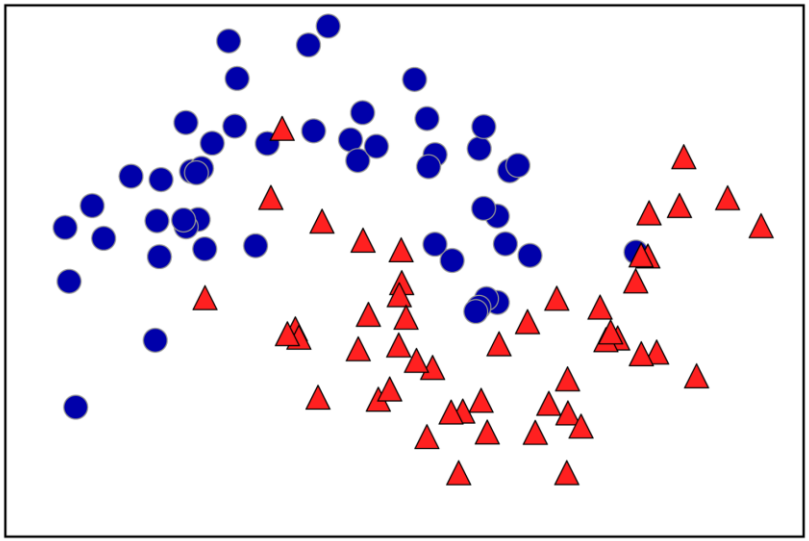
L3 – Supervised Learning II

- Supervised Machine Learning Algorithms
  - Decision Trees
  - Ensembles of Decision Trees
  - Kernel Based Support Vector Machines
  - Neural Networks (Deep Learning)
- Decision Function
- Predicting Probabilities

Decision Trees

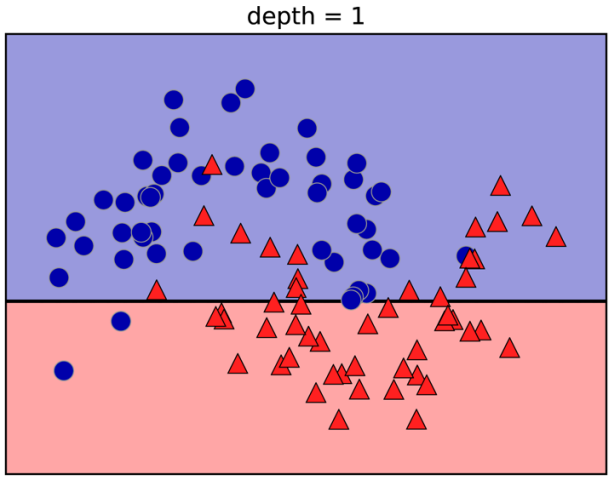
- To get the right answer by asking few if / else questions
  - Can learn these questions from data
  - These questions are called tests
  - Continues form: Is the feature i large than value a?

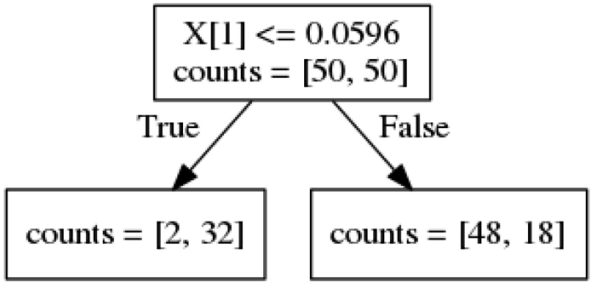


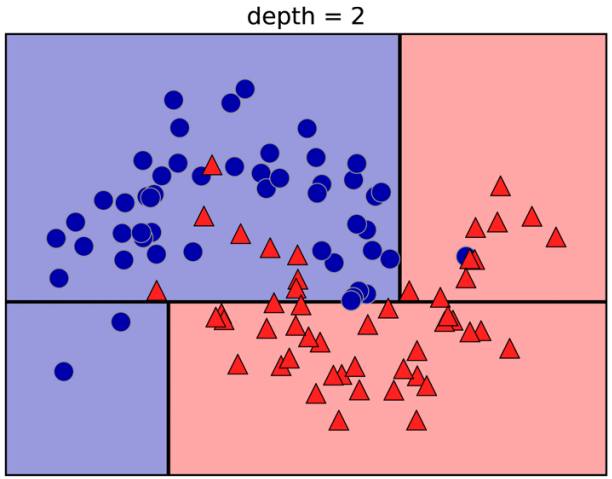


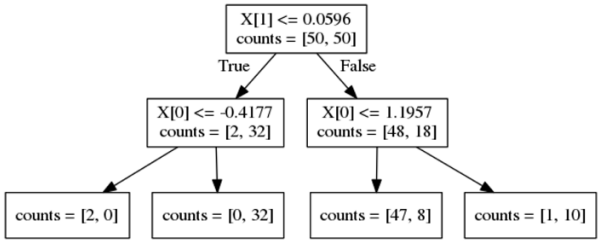
How to build a decision-free for this dataset?

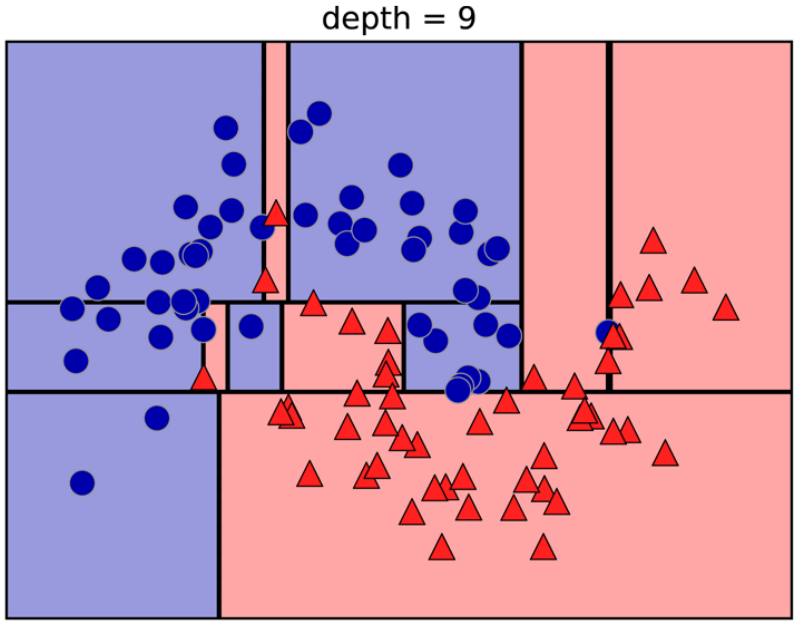
- Building decision trees
  - Recursive partitioning of the data until all data in a region sharing the same target value
  - A leaf of the tree (after construction) is called pure

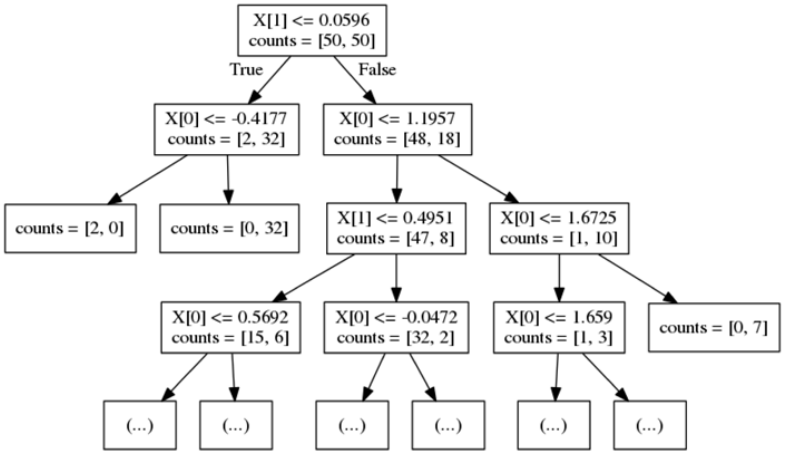












- Actually similar to kD-tree used in kNN-classifier
- Prediction:
  - Checking which region of the feature space the query point lies in
  - Predicting the majority target in that region
- Regression:
  - Similar technique to find the region as above
  - Output is the mean target of the training points in the leaf

- Controlling complexity of decision trees
  - Presence of pure leaves = 100% accurate on the training set

```
from sklearn.tree import DecisionTreeClassifier
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, stratify=cancer.target,
    random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

  - Prevent overfitting (two methods)
    - Stop the creation of the tree earlier (pre-pruning)

      (by setting max_depth)

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

    - Build the tree but then collapsing nodes with little info (post-pruning)

      (not implemented in scikit-learn)

- Analyzing decision trees
  - Visualize the tree using the export_graphviz function

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

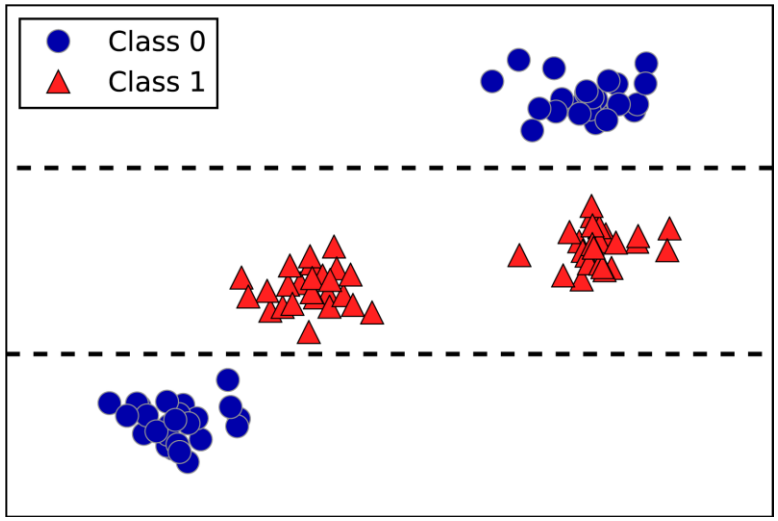
  - Can read this file and visualize it (as a good example of a machine learning algorithm that can be explained to nonexperts)

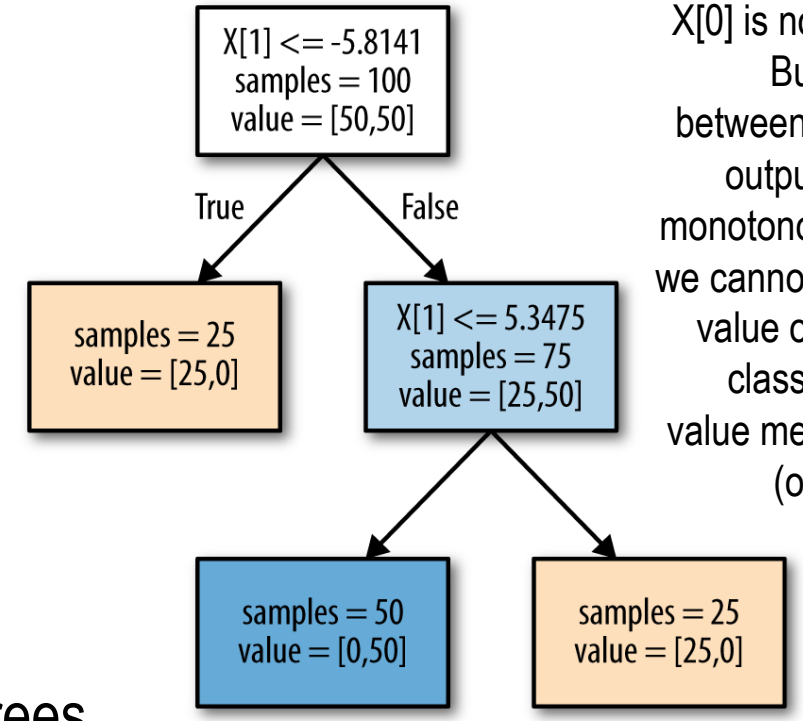
```
import graphviz
with open("tree.dot") as f:
    dot_graph = f.read()
display(graphviz.Source(dot_graph))
```

- Feature importance in tree (0: useless; 1: perfectly contribute)

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")    plt.ylabel("Feature")    plt.ylim(-1, n_features)
plot_feature_importances_cancer(tree)
```

- A feature has a low value in feature_importance_ doesn't mean the feature is uninformative
- It only means that the feature was not picked by the tree (likely because another feature encodes the same information).





X[0] is not used at all. But the relation between X[1] and the output class is not monotonous, meaning we cannot say "a high value of X[1] means class 0, and a low value means class 1" (or vice versa).

- Regression Tree
  - Similar to classification trees
  - Not able to extrapolation (i.e., making predictions outside the range of the training data)

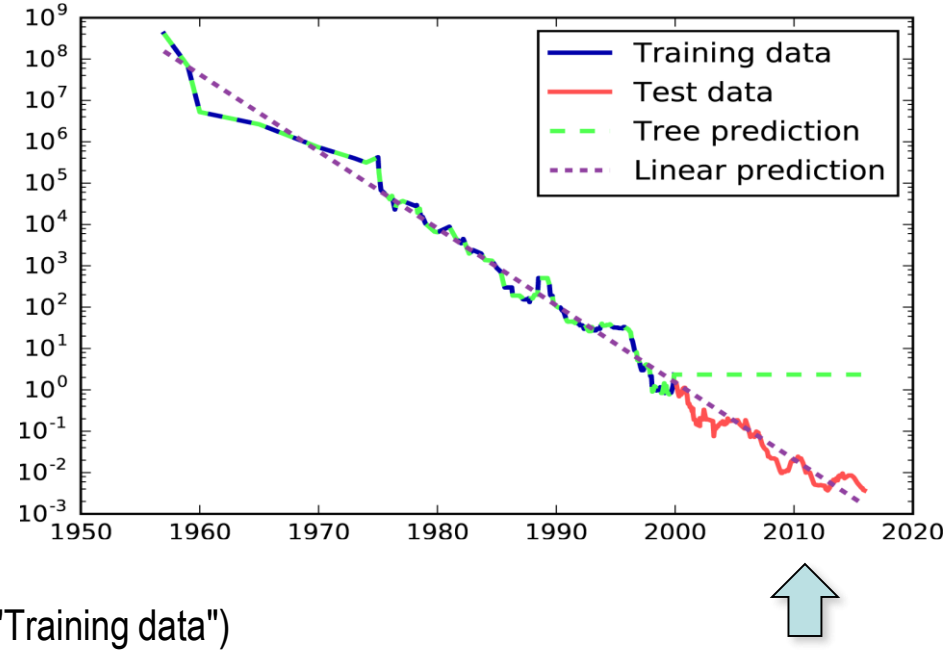
- Using a dataset of historical computer memory (RAM) prices

```
import os
import pandas as pd
ram_prices = pd.read_csv(os.path.join(mglearn.datasets.DATA_PATH, "ram_price.csv"))
plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Year")
plt.ylabel("Price in $/Mbyte")
```

- Training data (before year 2000); Test data (after 2000)

```
from sklearn.tree import DecisionTreeRegressor
# use historical data to forecast prices after the year 2000
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]

# predict prices based on date
X_train = data_train.date[:, np.newaxis]

# we use a log-transform to get a simpler relationship of data to target
y_train = np.log(data_train.price)
tree = DecisionTreeRegressor().fit(X_train, y_train)
linear_reg = LinearRegression().fit(X_train, y_train)
```

```
# predict on all data
X_all = ram_prices.date[:, np.newaxis]
pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)

# undo log-transform
price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```




```
plt.semilogy(data_train.date, data_train.price, label="Training data")
plt.semilogy(data_test.date, data_test.price, label="Test data")
plt.semilogy(ram_prices.date, price_tree, label="Tree prediction")
plt.semilogy(ram_prices.date, price_lr, label="Linear prediction")
plt.legend()
```

Cannot extrapolate the training dataset

- Parameters:
  - max_depth, max_leaf_nodes, or min_samples_leaf
- Strengths: easy to visualize and invariant to scale of data
- Weaknesses: easy to overfit (even after pre-pruning)

Ensembles of Decision Trees

- Ensembles – means combining multiple ML methods, e.g.
    1) Random Forest
    2) Gradient Boosted Decision Trees
- Random Forest
  - A collection of decision-trees, where each is slightly different
  - Idea Behind:
    - Each tree might do a relatively good job in a local region but likely overfit on part of the data
    - We reduce the amount of overfitting by averaging their results
  - Two strategies for realizing random
    - Selecting random data points to build a tree
    - Selecting random features in each split test

- Building Random Forests (both regression / classification)
  - *n_estimators*: the number of trees to build
  - For each tree, generate bootstrap samples of our data
    - From our n_samples data points repeatedly draw an example randomly n_samples times to result a dataset as big as the original dataset
    - Note that the same sample can be picked multiple times
  - When generating each node for a tree
    - Instead of looking for the best test for each node, the algorithm randomly select a subset of features (controlled by the *max_features* parameter)
    - Each node in a tree can make a decision by a different subset of features
- Results of the above randomization:
  - A high *max_features*: the trees in the random forest will be similar
  - A low *max_features*: the trees are quite different but each tree might need to be very deep in order to fit the data well
- For regression: we can average the results from all trees

- For classification, a “soft voting” strategy is adopted
  - Each tree provides a probability for each possible output label.
  - The probabilities predicted by all the trees are averaged, and the class with the highest probability is predicted.
- Try a random forest with 5 trees to the two_moons dataset

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)

fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1], alpha=.4)
axes[-1, -1].set_title("Random Forest")      mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

Can find the decision boundaries learned by the five trees are quite different

- Can have error (by bootstrap sampling)
- Overfit less than any of the trees individually

- Analyze the overfitting on a random forest with 100 trees on breast cancer dataset

```
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

- Similar to decision tree, the random forest also provides feature importance

```
plot_feature_importances_cancer(forest)
```

  - The random forest gives nonzero importance to many more features than a single tree
  - The randomness of a random forest let it capture a much broader picture of the data than a single tree
- Random Forest outperforms in general (unless needs compact rep.)
- Can run on multiple cores (the *n_jobs* parameter); and parameters as
  - *max_features* = sqrt(*n_features*) for classification
  - *max_features* = *n_features* for regression

Gradient Boosted Regression Trees

- Gradient boosting works by building trees in a serial manner, where each tree tries to correct the previous one
  - No randomization in gradient boosted regression trees
  - Idea behind: to combine many simple models (e.g., shallow trees – of depth one to five)
  - Generally a bit more sensitive to parameter than random forest
  - But can be more accurate if parameters are set correctly
- Major parameters:
  - *learning_rate*: a higher value can make stronger correction (i.e., allowing for more complex models)
  - *n_estimators*: the number of trees

- By default, 100 trees of maximum depth 3 and learning rate of 0.1 are used

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

- Stronger pre-pruning can be applied by limiting the maximum depth

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

- Or lowering the learning rate

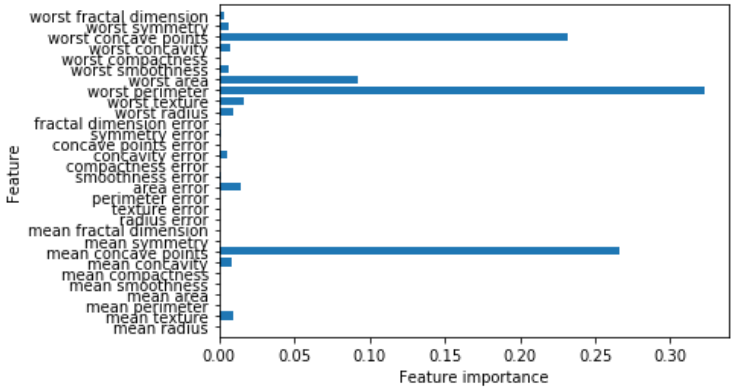
```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Both methods can effectively decrease the model complexity, i.e.,

- Reduced accuracy on the training dataset
- Enhanced accuracy on the test dataset

- Visualize the feature importance

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

plot_feature_importances_cancer(gbrt)
```

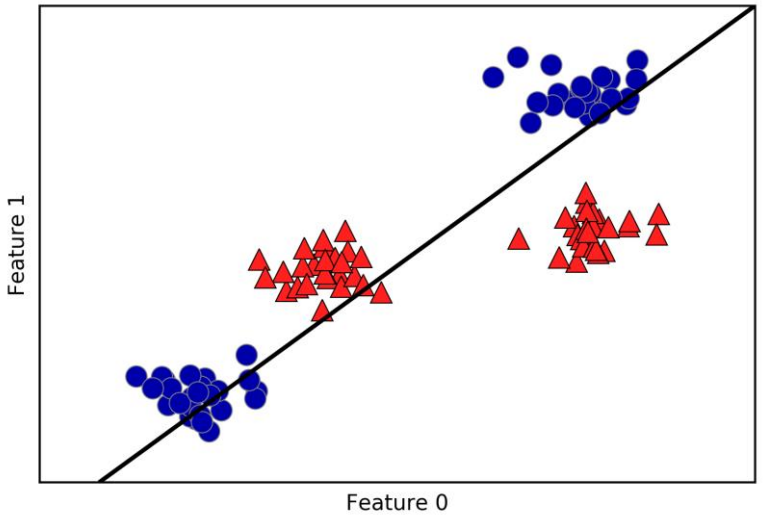


  - The gradient boosting completely ignored some of the features
  - A common strategy: trying random forest first – more robust
- Strengths & Weakness
  - Gradient boosted decision trees are among the most powerful and widely used models for supervised learning
  - But they require careful tuning of the parameters
    - Two parameters n_estimators & learning_rate are highly interconnected
    - Strategy: to fit n_estimators depending on the time and memory budget, then search over different learning_rates
    - Another parameter max_depth: usually set very low for gradient boosted models – often not deeper than five splits.

Kernelized Support Vector Machines

- Comparing to linear support vector machine
  - Allows for more complex models
  - Math (Ch.12 of https://web.stanford.edu/~hastie/ElemStatLearn/)
- Linear models and nonlinear features

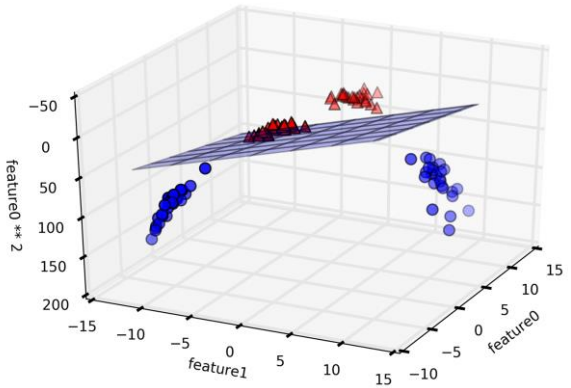
```
from sklearn.svm import LinearSVC
from sklearn.datasets import make_blobs

X, y = make_blobs(centers=4, random_state=8)
y = y % 2

linear_svm = LinearSVC().fit(X, y)
mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```



Decision boundary found by a linear SVM

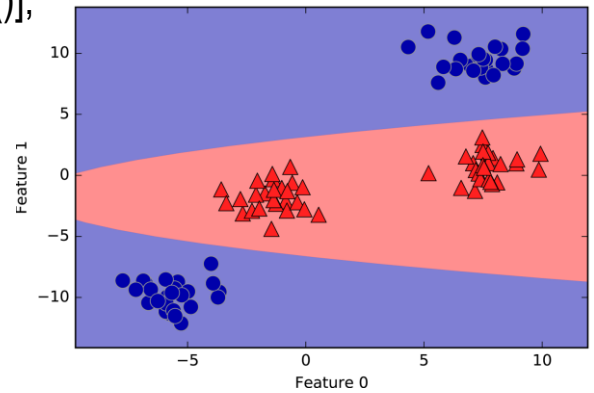
- By adding the square of the second feature
  - 2D points (feature0, feature1) => 3D points (feature0, feature1, feature0**2)

```
X_new = np.hstack([X, X[:, 1:] ** 2])    # add the squared second feature

from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()   # visualize in 3D
ax = Axes3D(figure, elev=-152, azim=-26)
# plot first all the points with y == 0, then all with y == 1
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b', cmap=mglearn.cm2, s=60,
      edgecolor='k')
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^', cmap=mglearn.cm2,
      s=60, edgecolor='k')
ax.set_xlabel("feature0")        ax.set_ylabel("feature1")        ax.set_zlabel("feature1 ** 2")

linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)
XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)   # show linear decision boundary
```



- As a function of the original features by feature extension
  - The linear SVM model is not actually linear anymore
  - Not a line but more of an ellipse

```
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

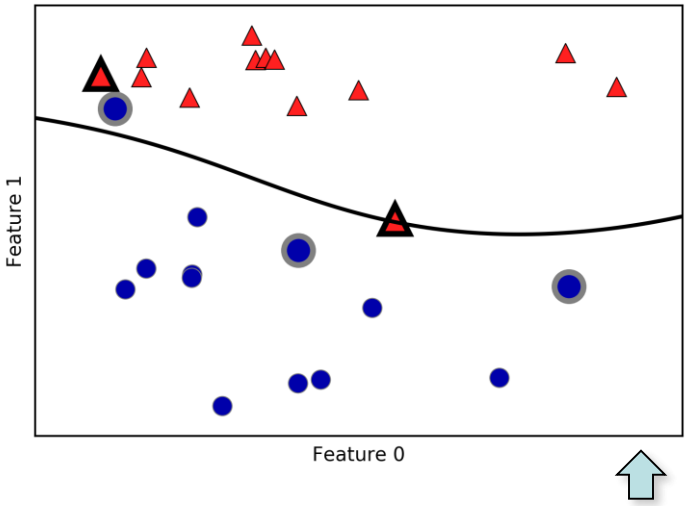


- Kernel trick
  - Lesson learned: Adding nonlinear features to the representation of our data can make linear models much more powerful
  - Mapping your data into a higher-dimensional space:
    - The polynomial kernel (e.g., feature1 ** 2 * feature2 ** 5)
    - The radial basis function (RBF) kernel (also known as Gaussian kernel)

- Kernel-base Support Vector Machine (SVM)
  - Only a subset of the training points matter for defining the decision boundary: the ones lie on the border between classes
  - Distance between data points is measured by Gaussian kernel:

$$k_{\text{rbf}}(x_1, x_2) = \exp(-\gamma \|x_1 - x_2\|^2)$$

```
from sklearn.svm import SVC

X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)

# plot support vectors
sv = svm.support_vectors_
# class labels of support vectors are given by the sign of the dual coefficients
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```



The SVM yields a very smooth and nonlinear (not a straight line) boundary.

- Tuning SVM parameters
  - gamma: corresponds to the inverse of the width of Gaussian
  - C: a regularization parameter (similar to the linear model)

```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))
for ax, C in zip(axes, [-1, 0, 3]):
        for a, gamma in zip(ax, range(-1, 2)):
                mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)
axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"], ncol=4, loc=(.9, 1.2))
```

  - From Left to Right: a low value of gamma means the boundary will vary slowly (a less complex model); high value more complex
  - From Top to Bottom: Increasing C allows the support vectors to have a stronger influence on the model and makes the decision boundary bend to correctly classify them
  - Default value: C=1 and gamma = 1/n_features
  - While SVM often perform quite well, they very sensitive to: 1) the settings of parameters and 2) the scaling of the data

- Default value leads to very poor performance

```
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
svc = SVC(kernel='rbf', C=1, gamma=1/30.0)  # there are 30 features
svc.fit(X_train, y_train)
print("Accuracy on training set: {:.2f}".format(svc.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(svc.score(X_test, y_test)))
```

  - In particular, they require all features to vary on a similar scale
  - Let's check by the following code (displaying the min / max values of all features)

```
plt.boxplot(X_train)
plt.yscale("symlog")
plt.xlabel("Feature index")
plt.ylabel("Feature magnitude")
```

  - This can be somewhat of a problem for other models (like linear)
  - But it can has devastating effects for the kernel SVM
  - Any solution? Re-scaling each feature so that they are all approximately on the same scale

- Preprocessing data for SVMs
  - Re-scaling the training set

```
# compute the minimum value per feature on the training set
min_on_training = X_train.min(axis=0)
# compute the range of each feature (max - min) on the training set
range_on_training = (X_train - min_on_training).max(axis=0)
# subtract the min, and divide by range;    # afterward, min=0 and max=1 for each feature
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum for each feature\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum for each feature\n {}".format(X_train_scaled.max(axis=0)))
```

  - Using the same transformation on the test set and evaluate again

```
X_test_scaled = (X_test - min_on_training) / range_on_training
svc = SVC(kernel='rbf', C=1, gamma=1/30.0)
svc.fit(X_train_scaled, y_train)
print("Accuracy on training set: {:.3f}".format(svc.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

- Then, we can try to increase C or gamma for a better fitting

```
svc = SVC(kernel='rbf', C=1000, gamma=1/30.0)   # training score: 0.988 and test score: 0.972
```

- Summary of kernel-based SVM
  - Work well on both low-dimensional and high-dimensional data
  - Not scale very well with the number of samples (i.e., more samples will take much longer time)
  - Require carefully prepared data-set (i.e., the same scale)

  - Important parameters:
    - The regularization parameter C
    - The choice of kernel
    - The kernel-specific parameters

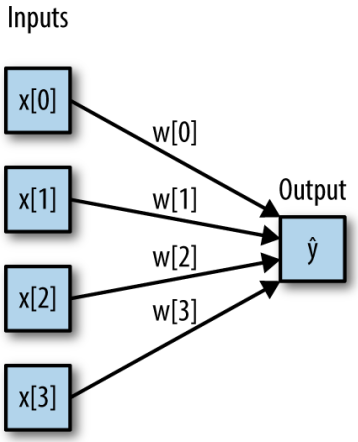
    - The parameters C and $gamma$ needs be adjusted together as being strongly correlated

Neural Networks (Deep Learning)

- Multilayer Perceptrons (MLPs) is mainly discussed here
  - MLPs are also known as feed-forward neural networks
  - Can be viewed as generalization of linear models that perform multiple stages of processing to come to a decision
- Considering the prediction by a linear regressor

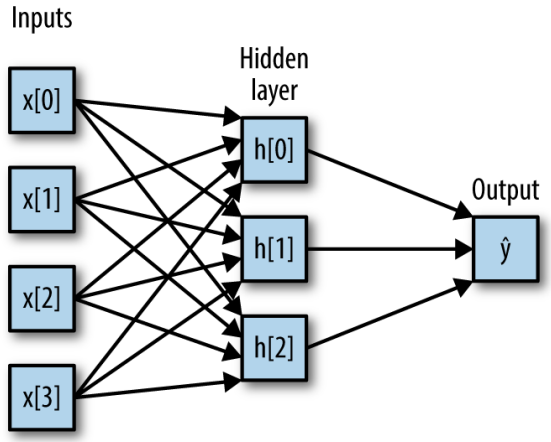
$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

display(mglearn.plots.plot_logistic_regression_graph())

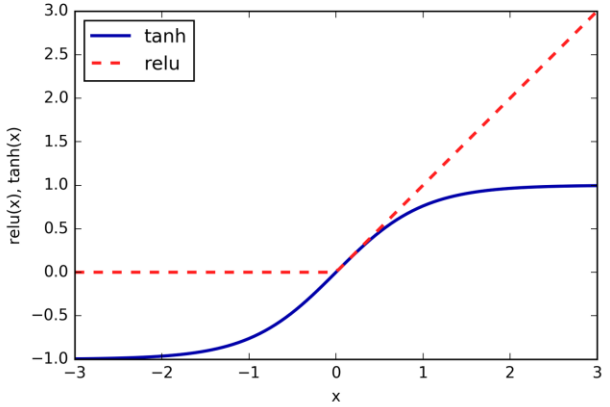


- In an MLP, this process of computing weighted sum is repeated multiple times
  - First computing hidden units (intermediate step)
  - Which are combined using weighted sums to yield final result

- The model has a lot more coefficients
  - One between every input and hidden units
  - One between every unit and the output
  - i.e., every arrow in the right figure

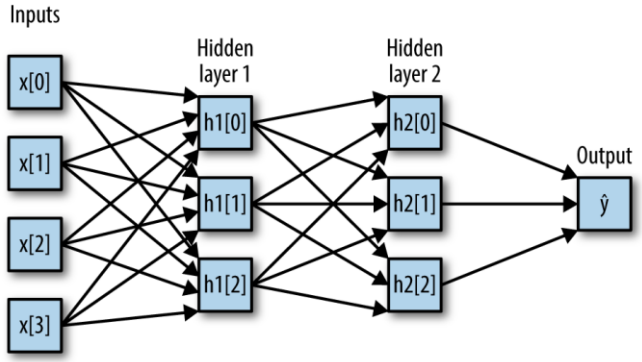


- To make this model truly more powerful
  - Need one extra trick: after computing a weighted sum for each hidden unit, a nonlinear function is applied to the result
    - Rectifying nonlinearity (relu)
    - Tangen hyperpolicus (tanh)
  - The result of this function is then used in the weighted sum to compute output
  - The mathematical formulation as follows



$$h[0] = \tanh(w[0,0] * x[0] + w[1,0] * x[1] + w[2,0] * x[2] + w[3,0] * x[3] + b[0])$$
$$h[1] = \tanh(w[0,1] * x[0] + w[1,1] * x[1] + w[2,1] * x[2] + w[3,1] * x[3] + b[1])$$
$$h[2] = \tanh(w[0,2] * x[0] + w[1,2] * x[1] + w[2,2] * x[2] + w[3,2] * x[3] + b[2])$$
$$\hat{y} = v[0] * h[0] + v[1] * h[1] + v[2] * h[2] + b$$

- To change the complexity of MLPs, important parameters:
  - Number of nodes in the hidden layer
  - Adding additional hidden layers

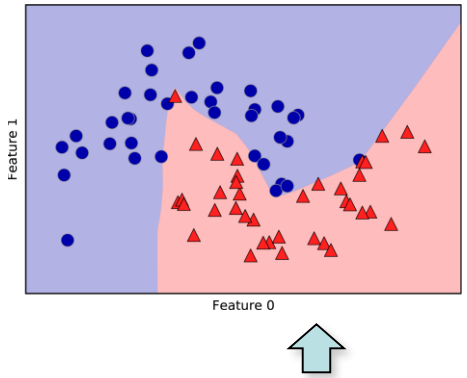



```
mglearn.plots.plot_two_hidden_layer_graph()
```

  - Having large neural networks made up of many of these layers of computation is what inspired the term "deep learning"

- Tuning Neural Networks

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

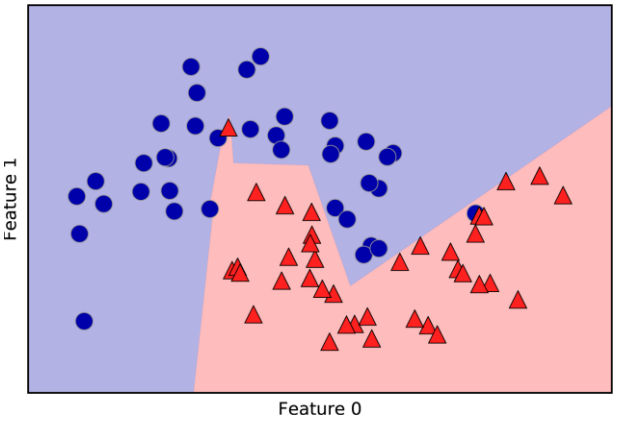


We use solver = 'lbfgs';
By default, use 100 hidden nodes;
Learned a very nonlinear by smooth decision boundary.

- We can use less number of nodes by changing the fitting to

```
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
```

  - Somewhat more ragged
  - The default nonlinearity is relu
  - Decision function: 10 straight segments
  - To obtain a smoother decision boundary
    - Add more hidden units
    - Add a second hidden layer
    - Or use the tanh nonlinearity




```
# using one hidden layers (200 units)
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[200]).fit(X_train, y_train)

# using two hidden layers (10 units for each)
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10, 10]).fit(X_train, y_train)

# using two hidden layers (10 units for each and with tanh nonlinearity)
mlp = MLPClassifier(solver='lbfgs', activation='tanh', random_state=0, hidden_layer_sizes=[10,
    10]).fit(X_train, y_train)
```

- Lastly, we can also control the complexity by using an L2 penalty to shrink the weights toward zero – alpha

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(solver='lbfgs', random_state=0,
              hidden_layer_sizes=[n_hidden_nodes, n_hidden_nodes], alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
        mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(n_hidden_nodes, n_hidden_nodes, alpha))
```

Larger alpha leads to smoother boundary

  - Random initialization also affects the model

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i, hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
```

- We then understand the neural network on real-world data

```
print("Cancer data per-feature maxima:\n{}".format(cancer.data.max(axis=0)))

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)
print("Accuracy on training set: {:.2f}".format(mlp.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_test, y_test)))
```

  - The accuracy is quite good, but not as good as other methods
    - Likely due to scaling of the data; try to re-scale as follows

```
# compute the mean value per feature on the training set
mean_on_train = X_train.mean(axis=0)
# compute the standard deviation of each feature on the training set
std_on_train = X_train.std(axis=0)
# subtract the mean, and scale by inverse standard deviation; afterward, mean=0 and std=1
X_train_scaled = (X_train - mean_on_train) / std_on_train
# use THE SAME transformation (using training mean and std) on the test set
X_test_scaled = (X_test - mean_on_train) / std_on_train
```

  - Then try MLP again as above

```
mlp = MLPClassifier(random_state=42)
mlp.fit(X_train_scaled, y_train)
print("Accuracy on training set: {:.2f}".format(mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_test_scaled, y_test)))
```

- We get a warning about maximum iterations reached but not converged
  - We can increase the number of iteration in adam algorithm for learning the model

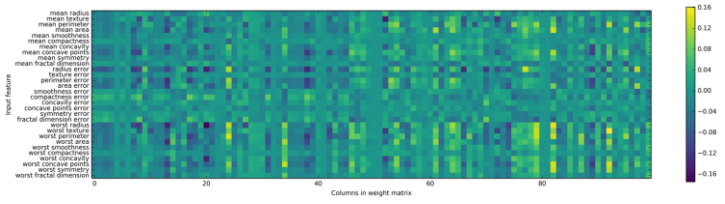
```
mlp = MLPClassifier(max_iter=1000, random_state=42)
```

  - Increase iteration # does not enhance generality, so we need to decrease the model's complexity

```
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=42)
```

- Inspect what we learned to look at the weights in the model

```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Columns in weight matrix")    plt.ylabel("Input feature")
plt.colorbar()
```



- More powerful tools: keras, lasagna & tensor-flow
  - Much more flexible interface and allow the usage of GPU based acceleration
- Advantages
  - Able to capture info in large dataset
  - Build incredibly complex model
- Disadvantage:
  - Long time to train
  - Need to careful preprocessing of the data
  - Require "homogeneous" on the feature type; otherwise, tree-based models might work better
- Strategy for tuning parameters:
  - First making a network large enough to overfit
  - Then improve the generalization
- Learning Algorithms: 'adam' (generally fine but quite sensitive to the scale) and 'lbfgs' (more robust but slower); 'sgd' (advanced user with many parameters to tune)

Uncertain Estimates from Classifiers

- Classifiers are able to provide uncertainty estimates of predictions – by two different functions:
  1. decision_function
  2. predict_proba

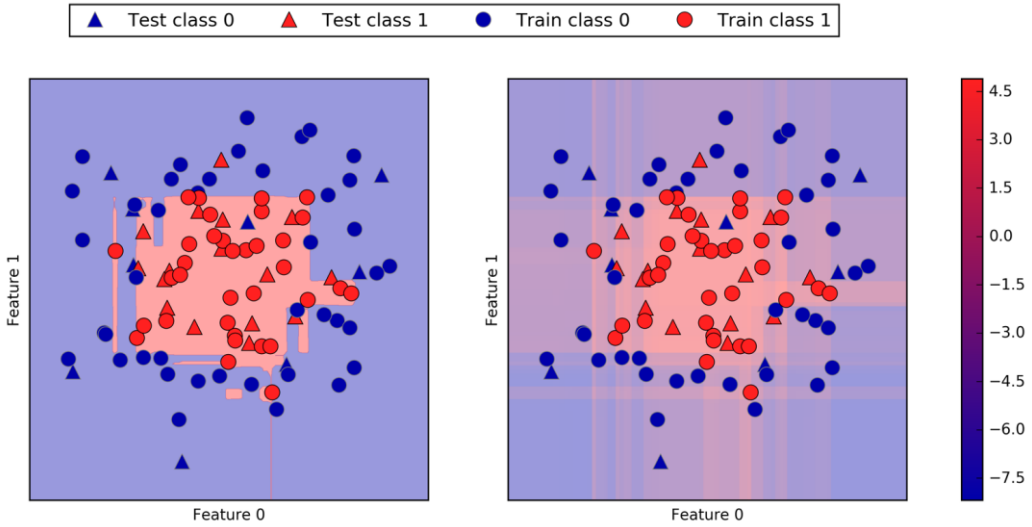
```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)
# we rename the classes "blue" and "red" for illustration purposes
y_named = np.array(["blue", "red"])[y]
# we can call train_test_split with arbitrarily many arrays; all will be split in a consistent manner
X_train, X_test, y_train_named, y_test_named, y_train, y_test = train_test_split(X, y_named, y,
    random_state=0)
# build the gradient boosting model
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

- The Decision function
  - The return value of decision_function is of shape;
  - It returns one floating number for each sample
    - Positive values indicate a preference for the “positive” class
    - Negative values indicate a preference for the “negative” class
  - The range of value can be arbitrary such hard to interpret

```
print("X_test.shape: {}".format(X_test.shape))
print("Decision function shape: {}".format(gbrt.decision_function(X_test).shape))

# show the first few entries of decision_function
print("Decision function:\n{}".format(gbrt.decision_function(X_test)[:6]))

print("Thresholded decision function:\n{}".format(gbrt.decision_function(X_test) > 0))
print("Predictions:\n{}".format(gbrt.predict(X_test)))

decision_function = gbrt.decision_function(X_test)
print("Decision function minimum: {:.2f} maximum: {:.2f}".format(np.min(decision_function),
      np.max(decision_function)))
```

- Plot decision_function for all points in 2D by color coding

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1], alpha=.4, cm=mglearn.ReBl)
for ax in axes:
    # plot training and test points
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test, markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Test class 0", "Test class 1", "Train class 0", "Train class 1"], ncol=4, loc=(.1, 1.1))
```



- Predicting Probabilities – predict_proba function
  - Output is a probability for each class
  - The sum of the entries for both classes is always 1

```
print("Shape of probabilities: {}".format(gbrt.predict_proba(X_test).shape))
# show the first few entries of predict_proba
print("Predicted probabilities:\n{}".format(gbrt.predict_proba(X_test[:6])))

fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl,
    function='predict_proba')
for ax in axes: # plot training and test points
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test, markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Test class 0", "Test class 1", "Train class 0", "Train class 1"], ncol=4, loc=(.1, 1.1))
```

- Uncertainty in Multiclass Classification

```
from sklearn.datasets import load_iris
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, random_state=42)
gbrt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbrt.fit(X_train, y_train)

print("Decision function shape: {}".format(gbrt.decision_function(X_test).shape))
# plot the first few entries of the decision function
print("Decision function:\n{}".format(gbrt.decision_function(X_test)[:6, :]))

print("Argmax of decision function:\n{}".format(np.argmax(gbrt.decision_function(X_test), axis=1)))
print("Predictions:\n{}".format(gbrt.predict(X_test)))

# show the first few entries of predict_proba
print("Predicted probabilities:\n{}".format(gbrt.predict_proba(X_test)[:6]))
# show that sums across rows are one
print("Sums: {}".format(gbrt.predict_proba(X_test)[:6].sum(axis=1)))

print("Argmax of predicted probabilities:\n{}".format(np.argmax(gbrt.predict_proba(X_test), axis=1)))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```