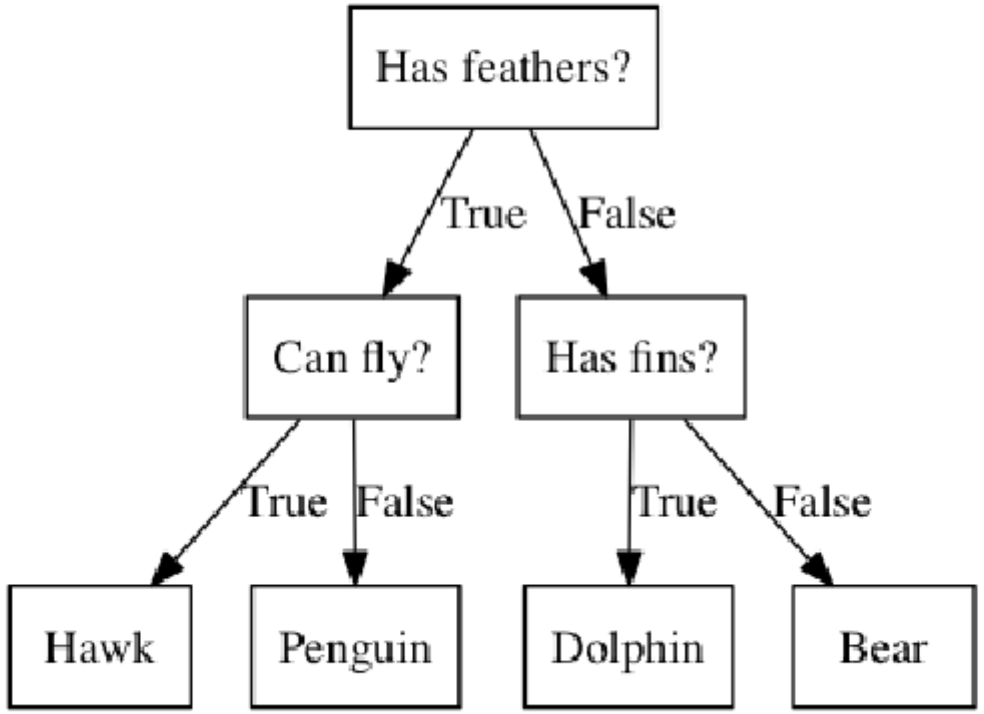
Trees and Ensembles

Trees


Has feathers?
True
False
Can fly?
Has fins?
True
False
True
False
Hawk
Penguin
Dolphin
Bear

Building Decision Trees

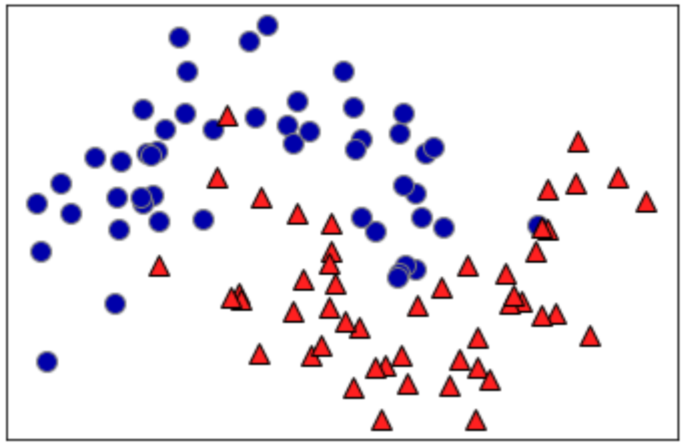
- Split the data in two (or more) parts
- Search over all possible splits and choose the one that is most *informative*
  - Many heuristics
  - E.g. *information gain*: how much does the entropy of the class labels decrease after the split (purer 'leafs')
- Repeat recursive partitioning

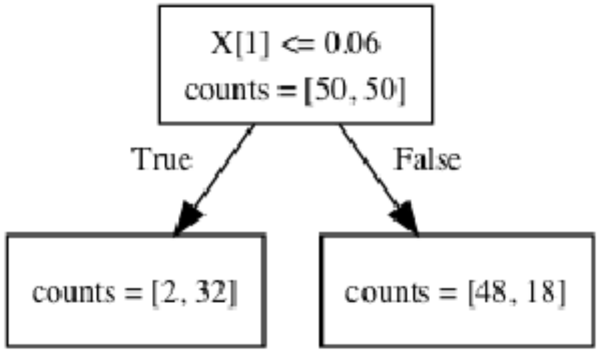
Making predictions:

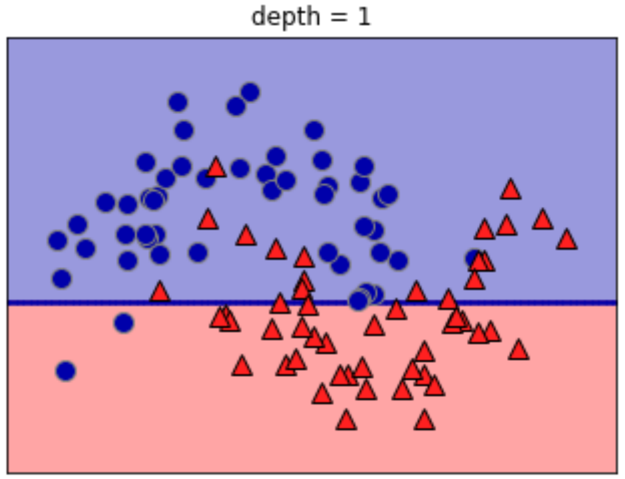
- Classification: find leaf for new data point, predict majority class (or class distribution)
- Regression: idem, but predict the *mean* of all values

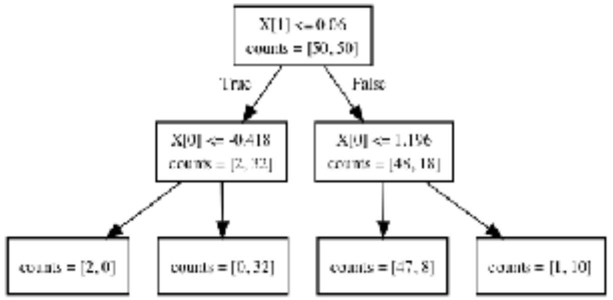
Decision Tree classification

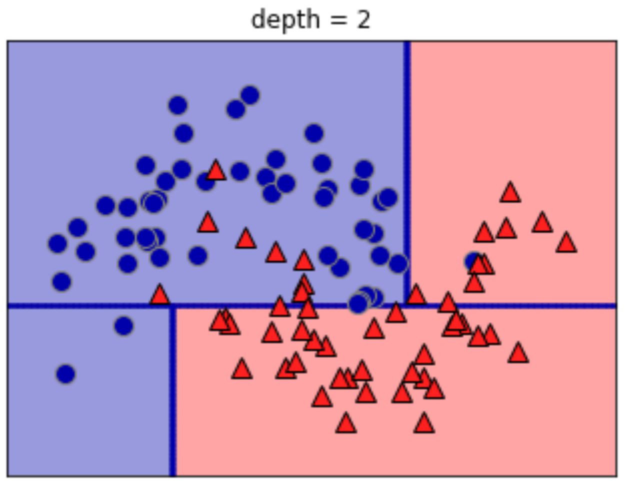
Where woud you make the first split?

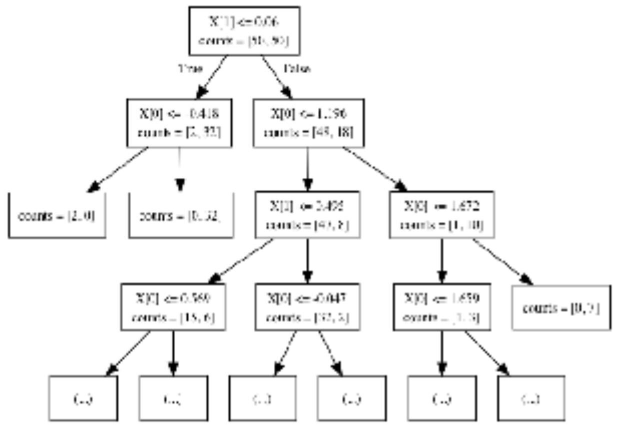


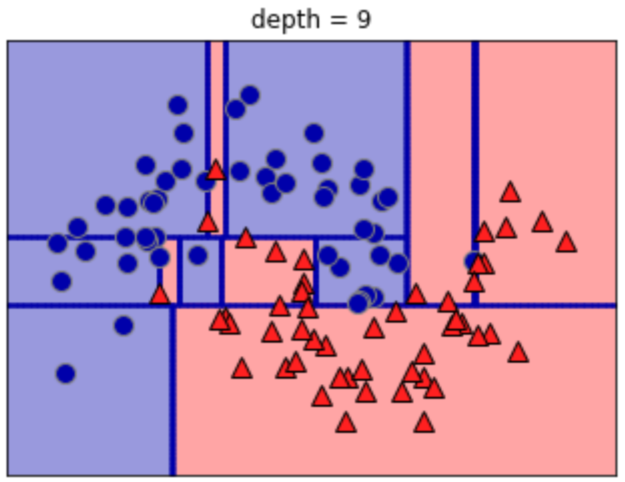

X[1] <= 0.06
counts = [50, 50]
True
False
counts = [2, 32]
counts = [48, 18]


depth = 1


X[1] <= 0.06
counts = [50, 50]
True
False
X[0] <= -0.418
counts = [2, 32]
X[0] <= 1.196
counts = [48, 18]
counts = [2, 0]
counts = [0, 32]
counts = [47, 8]
counts = [1, 10]


depth = 2


X[1] <= 0.06
counts = [50, 50]
True
False
X[0] <= -0.418
counts = [2, 32]
X[0] <= 1.196
counts = [48, 18]
counts = [2, 0]
counts = [0, 32]
X[1] <= 0.496
counts = [47, 8]
X[0] <= 1.672
counts = [1, 10]
X[0] <= 0.569
counts = [15, 6]
X[0] <= -0.047
counts = [32, 2]
X[0] <= 1.659
counts = [1, 3]
counts = [0, 7]
(...)
(...)
(...)
(...)
(...)
(...)


depth = 9

Heuristics

For classification ($X_i \rightarrow class_k$): *Impurity measures*:

- Misclassification Error (leads to larger trees):

$$1 - \operatorname*{argmax}_k \hat{p}_k$$

- Gini-Index (probabilistic predictions):

$$\sum_{k \neq k'} \hat{p}_k \hat{p}_{k'} = \sum_{k=1}^{K} \hat{p}_k (1 - \hat{p}_k)$$

with $\hat{p}_k$ = the relative frequency of class k in the leaf node

- Entropy (of the class attribute) measures *unpredictability* of the data:
    - How likely will random example have class k?

$$E(X) = -\sum_{k=1}^{K} \hat{p}_k \log_2 \hat{p}_k$$

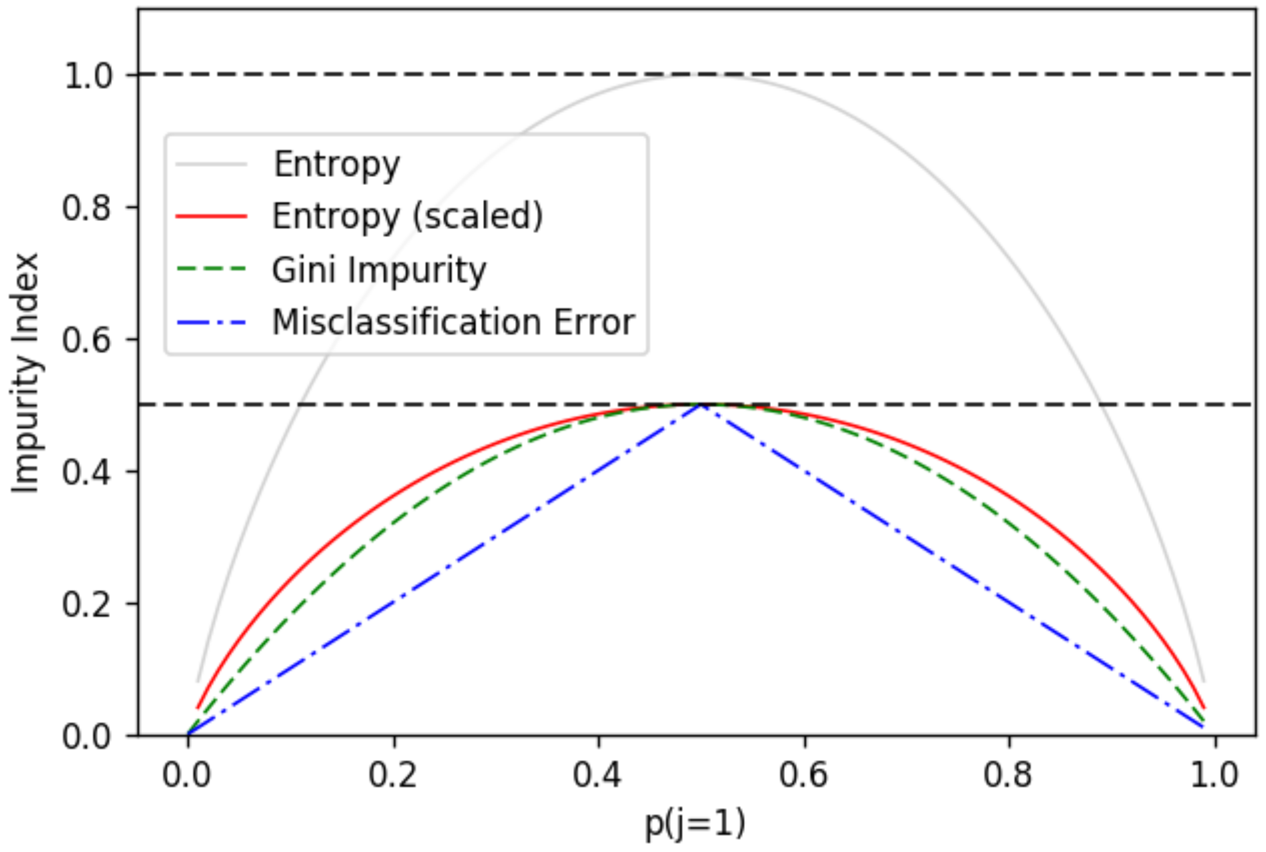
- Information Gain (a.k.a. Kullback–Leibler divergence) for choosing attribute X_i to split the data:

$$G(X, X_i) = E(X) - \sum_{v=1}^{V} \frac{|X_{i=v}|}{|X_i|} E(X_{i=v})$$

with $\hat{p}_k$ = the relative frequency of class k in the leaf node, X = the training set, containing i features (variables) X_i, v a specific value for X_i, $X_{i=v}$ is the set of examples having value v for feature X_i: $\{x \in X | X_i = v\}$

Heuristics visualized (binary class)

- Note that `gini != entropy/2`



Example

Ex.	1	2	3	4	5	6
a1	T	T	T	F	F	F
a2	T	T	F	F	T	T
class	+	+	-	+	-	-

$E(X)$?
$G(X, X_{a2})$?
$G(X, X_{a1})$?

$E(X) = -(\frac{1}{2} * log_2(\frac{1}{2}) + \frac{1}{2} * log_2(\frac{1}{2})) = 1$ (classes have equal probabilities)
$G(X, X_{a2}) = 0$ (after split, classes still have equal probabilities, entropy stays 1)

Ex.	**1**	**2**	**3**	**4**	**5**	**6**
a1	T	T	T	F	F	F
a2	T	T	F	F	T	T
class	+	+	-	+	-	-

$$E(X) = -\sum_{k=1}^{K} \hat{p}_k \log \hat{p}_k \quad , \quad G(X, X_i) = E(X) - \sum_{v=1}^{V} \frac{|X_{i=v}|}{|X_i|} E(X_{i=v})$$

$$E(X_{a1=T}) = -\frac{2}{3}\log_2(\frac{2}{3}) - \frac{1}{3}\log_2(\frac{1}{3}) = 0.9183 \quad (= E(X_{a1=F}))$$

$$G(X, X_{a1}) = 1 - \frac{1}{2}0.9183 - \frac{1}{2}0.9183 = 0.0817$$

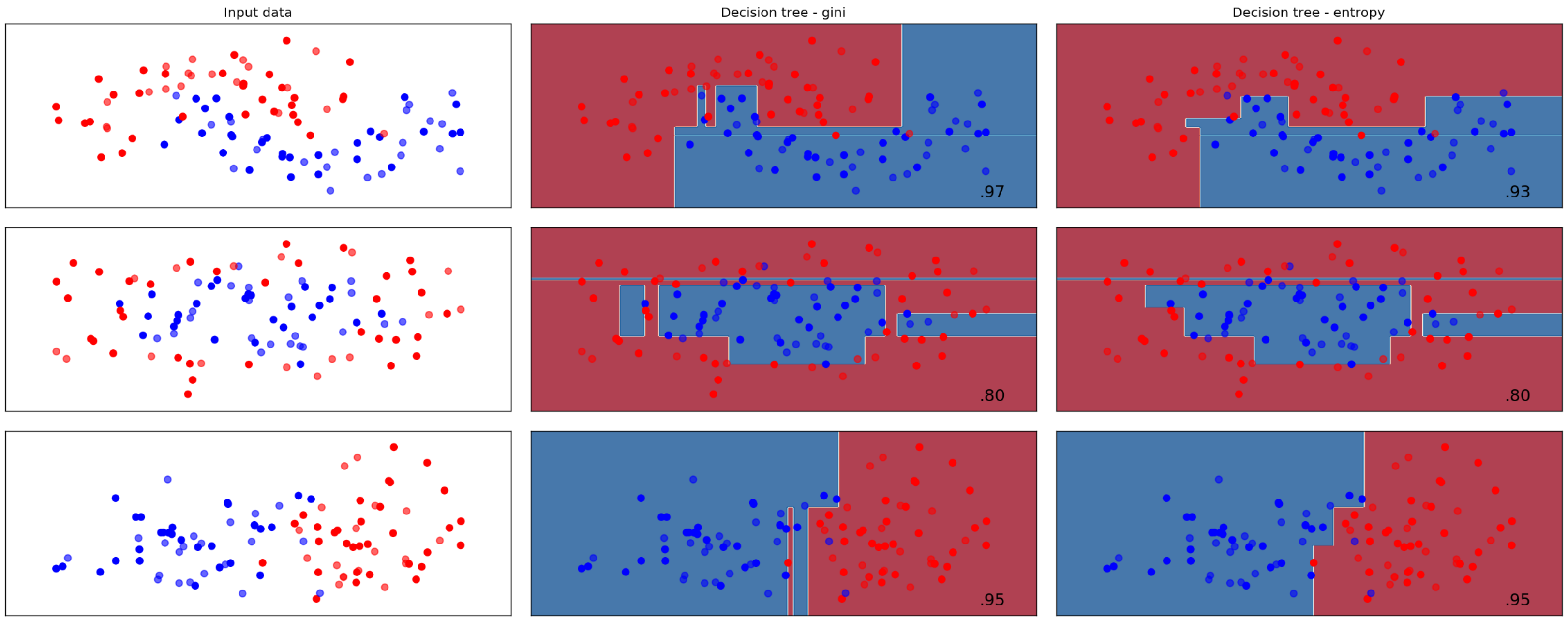
hence we split on a1

Heuristics in scikit-learn

The splitting criterion can be set with the `criterion` option in `DecisionTreeClassifier`

- `gini` (default): gini impurity index
- `entropy`: information gain

Best value depends on dataset, as well as other hyperparameters


Input data
Decision tree - gini
Decision tree - entropy
.97
.93
.80
.80
.95
.95

Handling many-valued features

What happens when a feature has (almost) as many values as examples?

- Information Gain will select it

One approach: use Gain Ratio instead (not available scikit-learn):

$$GainRatio(X, X_i) = \frac{Gain(X, X_i)}{SplitInfo(X, X_i)}$$

$$SplitInfo(X, X_i) = -\sum_{v=1}^{V} \frac{|X_{i=v}|}{|X|} log_2 \frac{|X_{i=v}|}{|X|}$$

where $X_{i=v}$ is the subset of examples for which feature X_i has value v.

SplitInfo will be big if X_i fragments the data into many small subsets, resulting in a smaller Gain Ratio.

Overfitting: Controlling complexity of Decision Trees

Decision trees can very easily overfit the data. Regularization strategies:

- Pre-pruning: stop creation of new leafs at some point
  - Limiting the depth of the tree, or the number of leafs
  - Requiring a minimal leaf size (number of instances)
- Post-pruning: build full tree, then prune (join) leafs
  - Reduced error pruning: evaluate against held-out data
  - Many other strategies exist.
  - scikit-learn supports none of them (yet)

Effect of pre-pruning: default tree overfits, setting `max_depth=5` is better

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

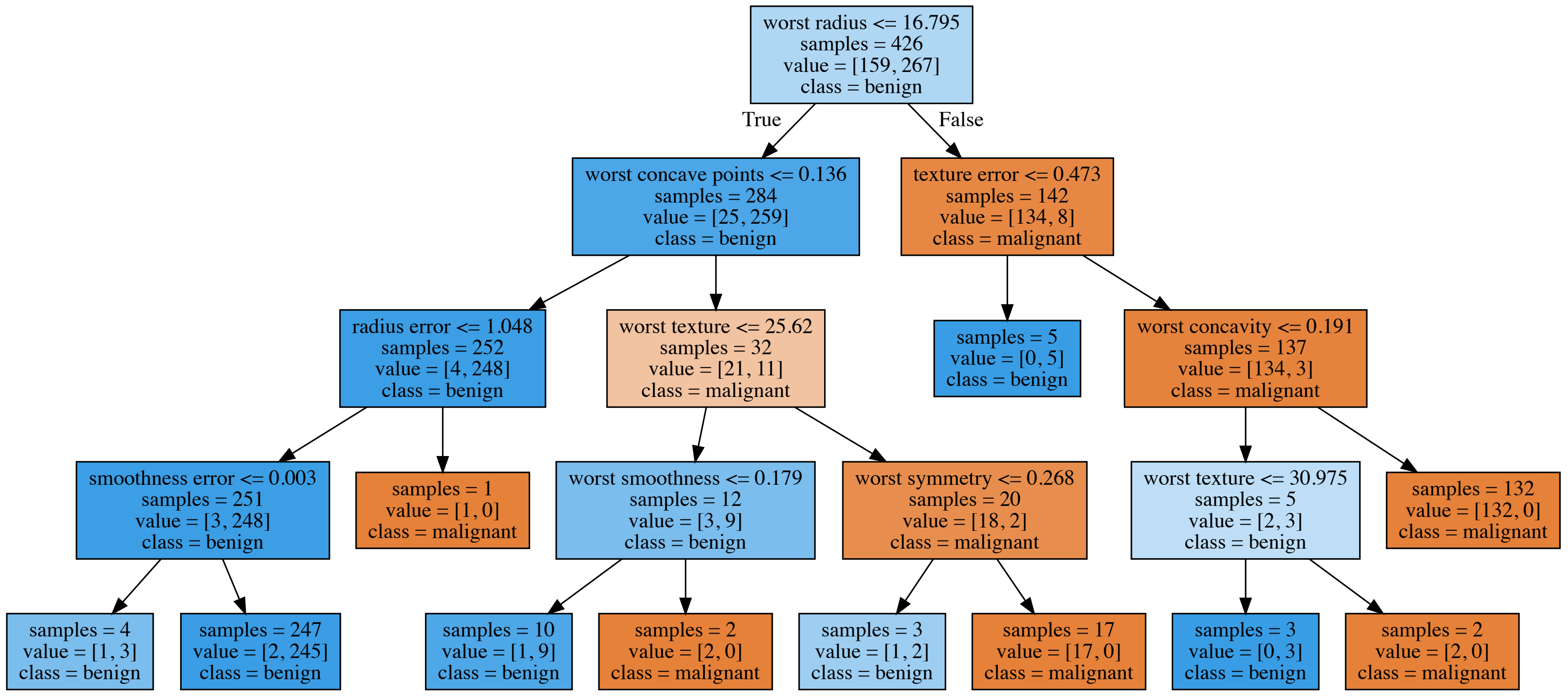
```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

```
tree = DecisionTreeClassifier(max_depth=5,random_state=0)
tree.fit(X_train, y_train)
```

```
Accuracy on training set: 0.995
Accuracy on test set: 0.951
```

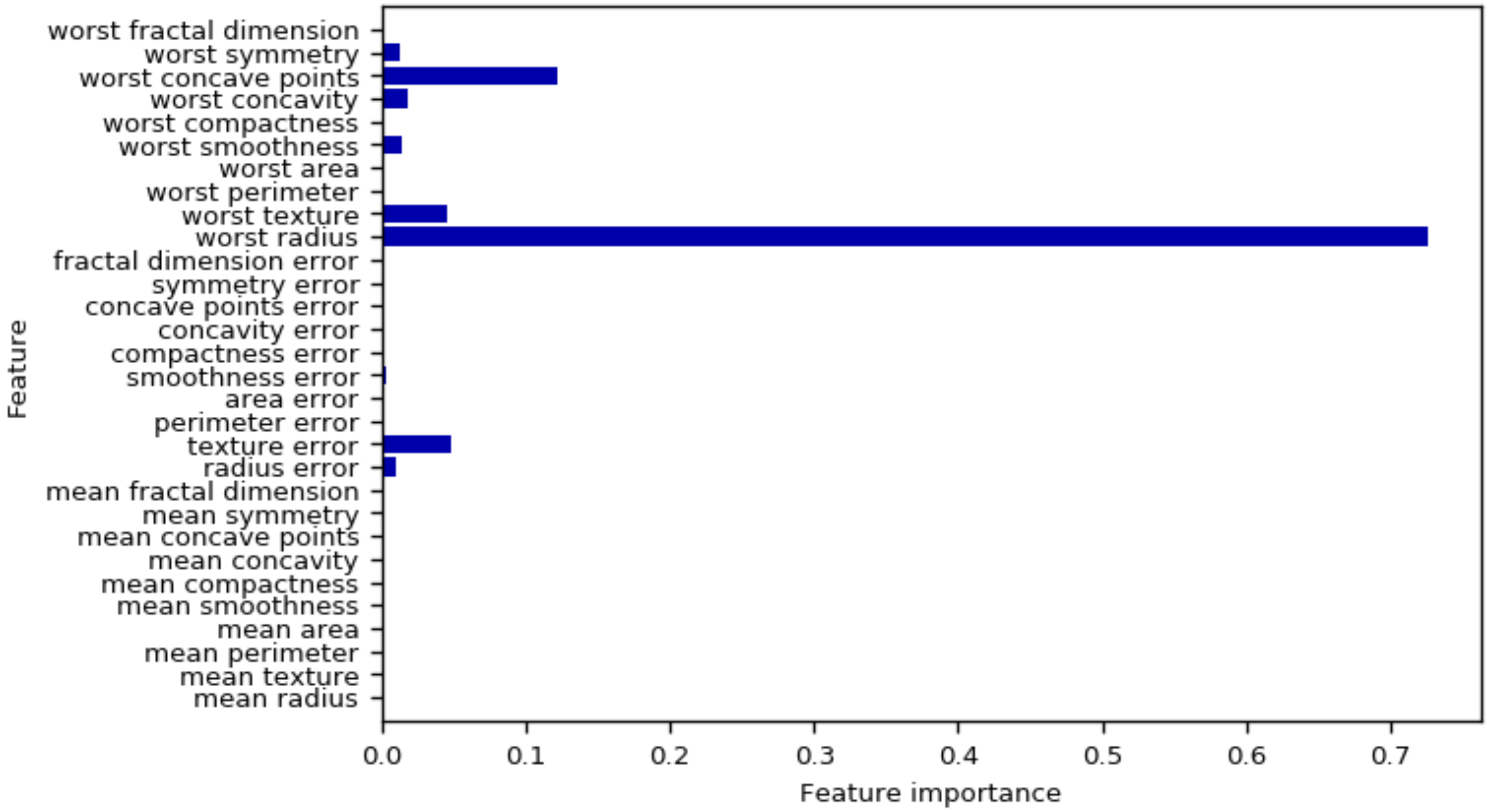
Analyzing Decision Trees manually

- Visualize and find the path that most data takes



`DecisionTreeClassifier` also returns *feature importances*

- In [0,1], sum up to 1
- High values for features selected by the algorithm, others don't contribute new information given the selected features


worst fractal dimension
worst symmetry
worst concave points
worst concavity
worst compactness
worst smoothness
worst area
worst perimeter
worst texture
worst radius
fractal dimension error
symmetry error
concave points error
concavity error
compactness error
smoothness error
area error
perimeter error
texture error
radius error
mean fractal dimension
mean symmetry
mean concave points
mean concavity
mean compactness
mean smoothness
mean area
mean perimeter
mean texture
mean radius
Feature
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
Feature importance

Decision tree regression

- Heuristic: *Minimal quadratic distance*
- Consider splits at every data point for every variable (or halfway between)
- Dividing the data on X_j at splitpoint s leads to the following half-spaces:

$$R_1(j,s) = X : X_j \leq s \quad \textit{and} \quad R_2(j,s) = X : X_j > s$$

- The best split, with predicted value c_i and actual value y_i:

$$\min_{j,s}\left(\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2\right)$$

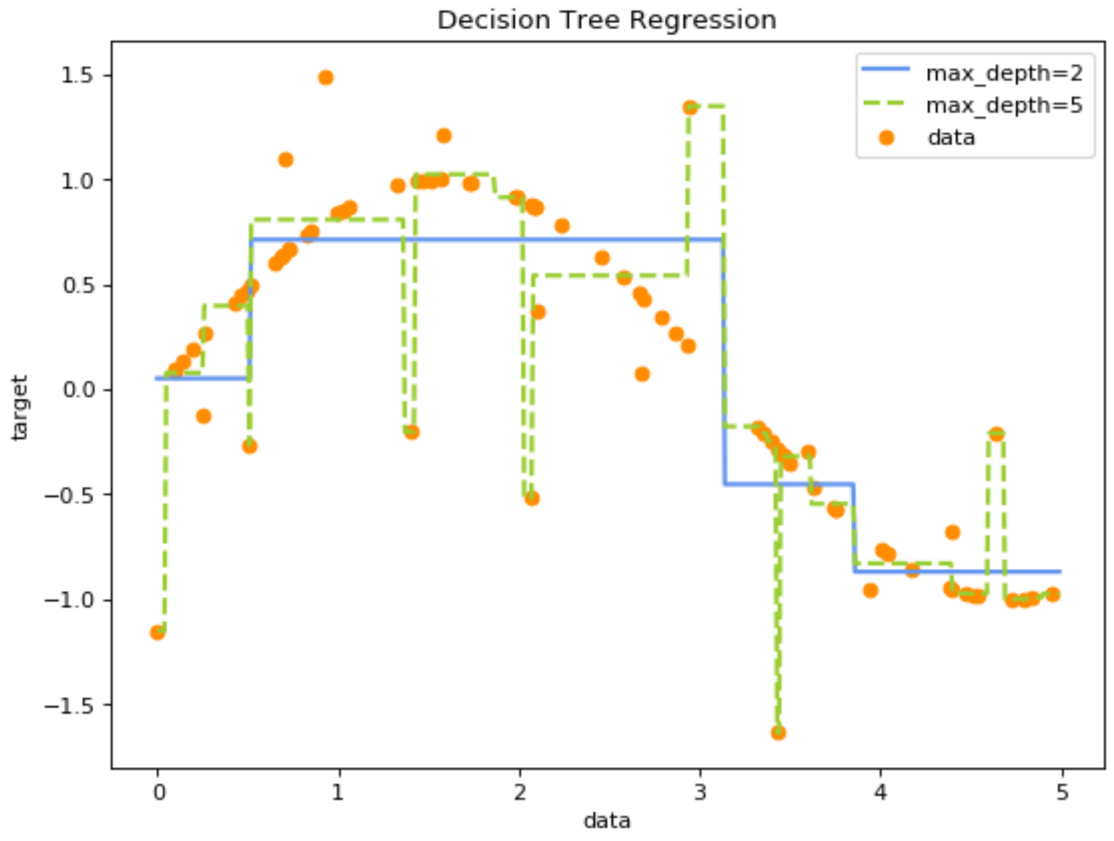
- Assuming that the tree predicts y_i as the average of all x_i in the leaf:

$$\hat{c}_1 = \text{avg}(y_i | x_i \in R_1(j,s)) \quad \textit{and} \quad \hat{c}_2 = \text{avg}(y_i | x_i \in R_2(j,s))$$

with x_i being the i-th example in the data, with target value y_i

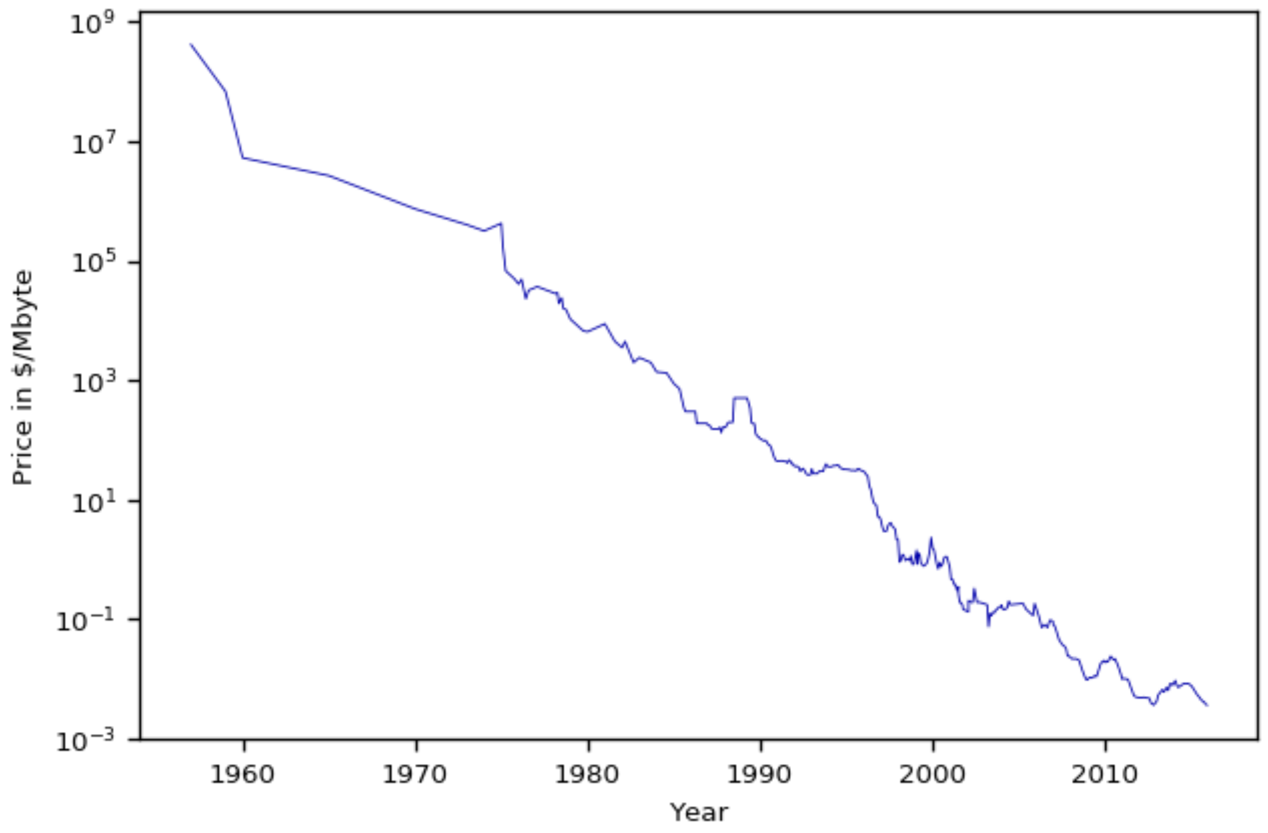
In scikit-learn

Regression is done with `DecisionTreeRegressor`

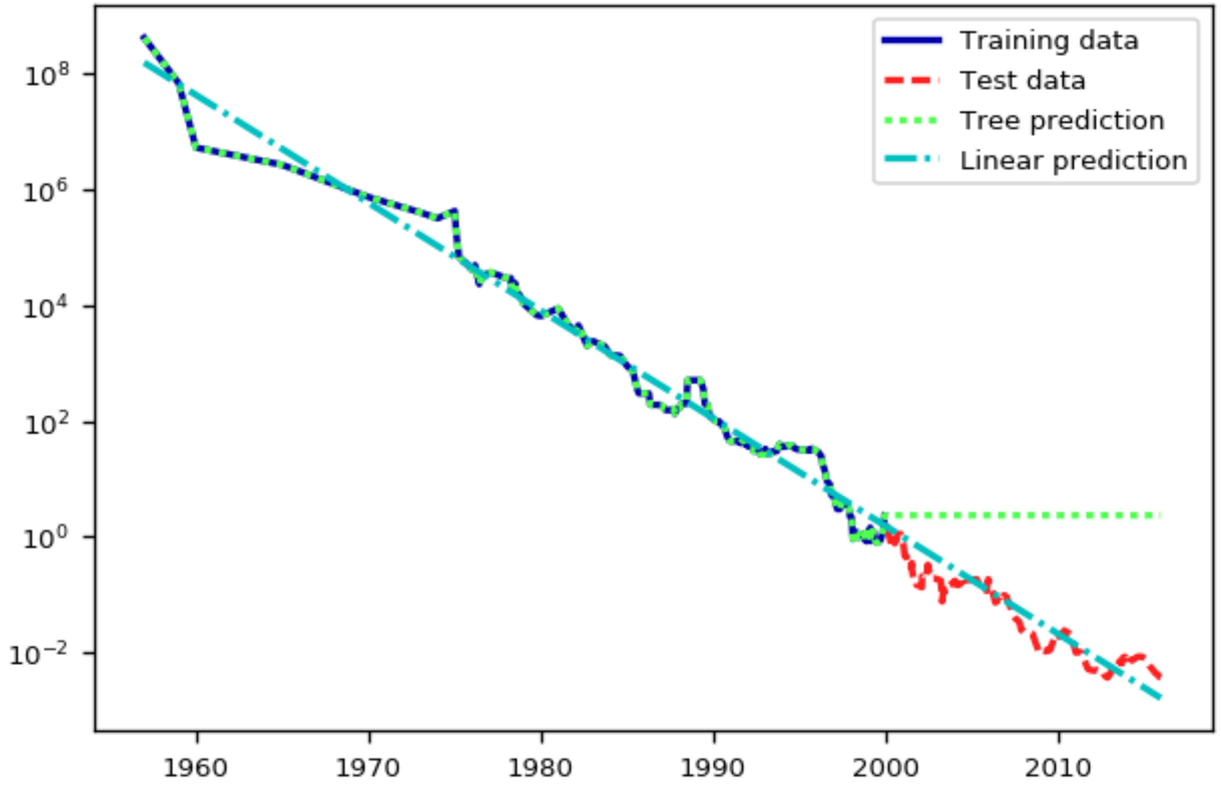


Note that decision trees do not extrapolate well.

- The leafs return the same *mean* value no matter how far the new data point lies from the training examples.
- Example on the `ram_price` forecasting dataset



10^9
10^7
10^5
10^3
10^1
10^{-1}
10^{-3}
Price in \$/Mbyte
1960
1970
1980
1990
2000
2010
Year



10^8
10^6
10^4
10^2
10^0
10^{-2}
1960
1970
1980
1990
2000
2010
Training data
Test data
Tree prediction
Linear prediction

Strengths, weaknesses and parameters

Pre-pruning: regularize by:

- Setting a low `max_depth`, `max_leaf_nodes`
- Setting a higher `min_samples_leaf` (default=1)

Decision trees:

- Work well with features on completely different scales, or a mix of binary and continuous features
    - Does not require normalization
- Interpretable, easily visualized
- Still tend to overfit easily. Use ensembles of trees.

Ensemble learning

Ensembles are methods that combine multiple machine learning models to create more powerful models. Most popular are:

- **RandomForests**: Build randomized trees on random samples of the data
- **Gradient boosting machines**: Build trees iteratively, giving higher weights to the points misclassified by previous trees

In both cases, predictions are made by doing a vote over the members of the example.
Stacking is another technique that builds a (meta)model over the predictions of each member.

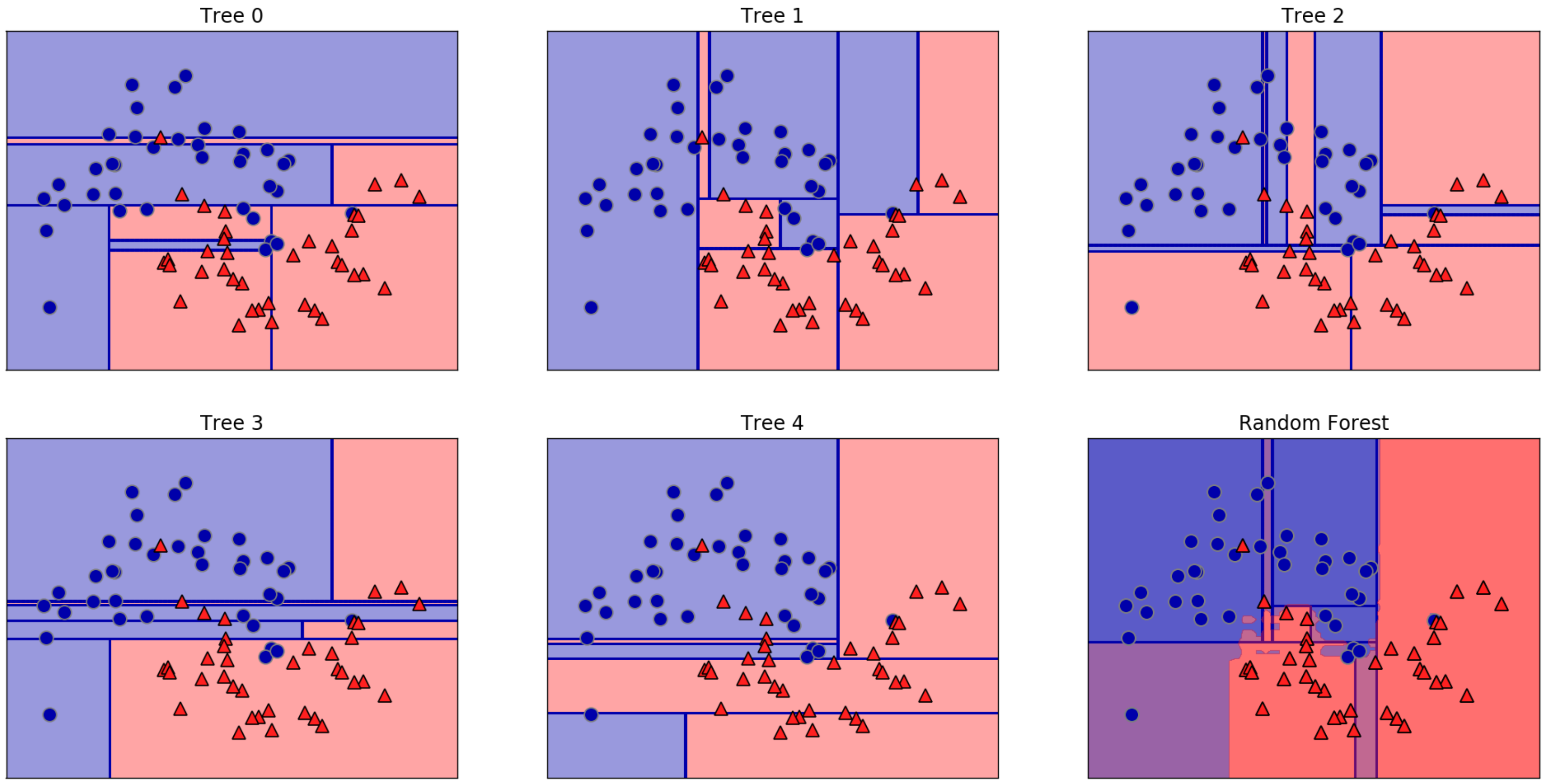
RandomForests

Reduce overfitting by averaging out individual predictions (variance reduction)

- Take a *bootstrap sample* of your data
    - Randomly sample with replacement
    - Build a tree on each bootstrap
- Repeat `n_estimators` times
    - Higher values: more trees, more smoothing
    - Make prediction by aggrating the individual tree predictions
        - a.k.a. Bootstrap aggregating (Bagging)
- RandomForest: Randomize trees by considering only a random subset of features of size `max_features` *in each node*
    - Small `max_features` yields more different trees, more smoothing
    - Default: $sqrt(n\_features)$ for classification, $log2(n\_features)$ for regression

Making predictions:

- Classification: soft voting (softmax)
  - Every member returns probability for each class
  - After averaging, the class with highest probability wins
- Regression:
  - Return the *mean* of all predictions


Tree 0
Tree 1
Tree 2
Tree 3
Tree 4
Random Forest

Scikit-learn algorithms:

- `RandomForestClassifier` (or Regressor)
- `ExtraTreesClassifier`: Grows deeper trees, faster

Most important parameters:

- `n_estimators` (higher is better, but diminishing returns)
  - Will start to underfit (bias error component increases slightly)
- `max_features` (default is typically ok)
  - Set smaller to reduce space/time requirements
- parameters of trees, e.g. `max_depth` (less effect)

`n_jobs` sets the number of parallel cores to run
`random_state` should be fixed for reproducibility

```
forest = RandomForestClassifier(n_estimators=100, random_state=0) # Vary
forest.fit(X_train, y_train)
```

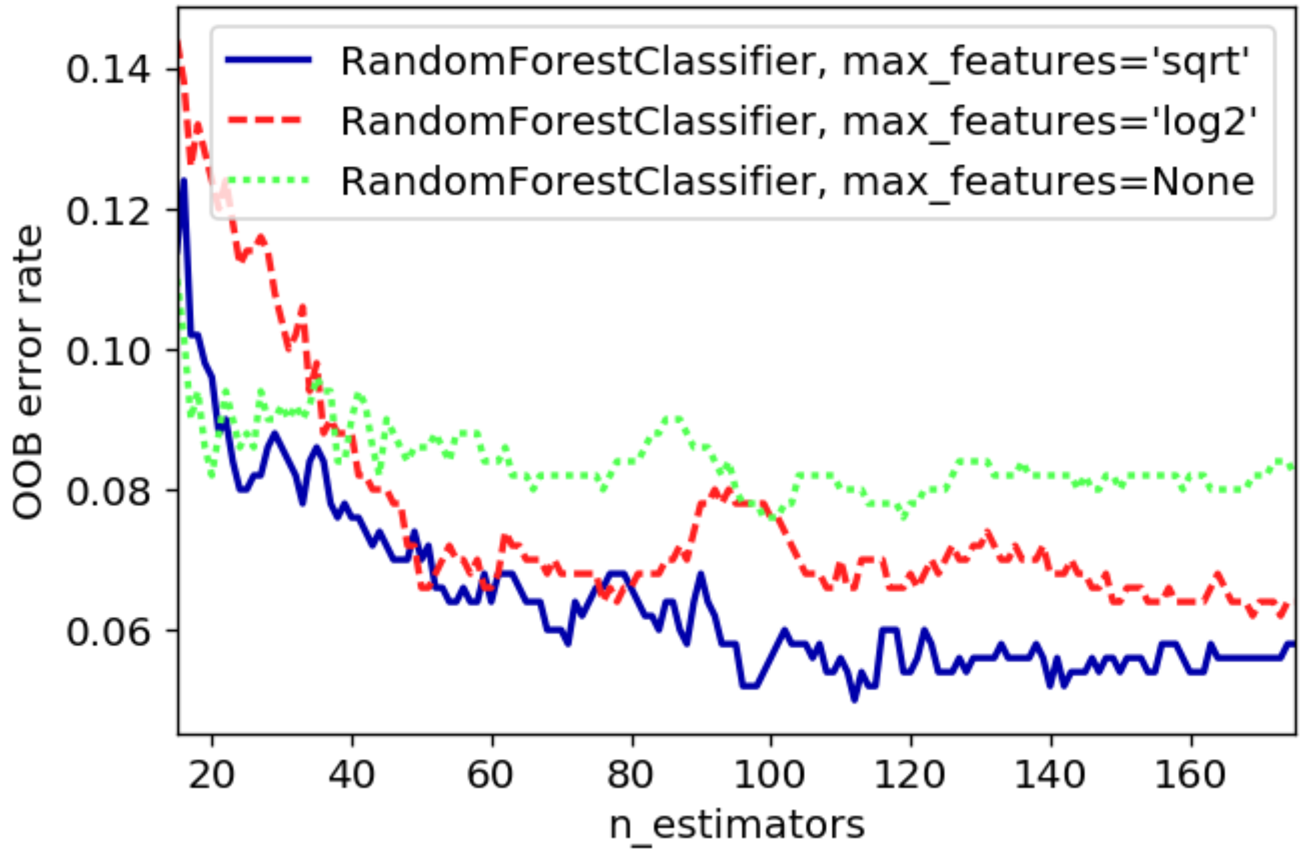
```
Accuracy on training set: 1.000
Accuracy on test set: 0.972
```

RandomForest allow another way to evaluate performance: out-of-bag (OOB) error

- While growing forest, estimate test error from training samples
- For each tree grown, 33-36% of samples are not selected in bootstrap
  - Called the 'out of bootstrap' (OOB) samples
  - Predictions are made as if they were novel test samples
  - Through book-keeping, majority vote is computed for all OOB samples from all trees
- OOB estimated test error is rather accurate in practice
  - As good as CV estimates, but can be computed on the fly (without repeated model fitting)
  - Tends to be slightly pessimistic

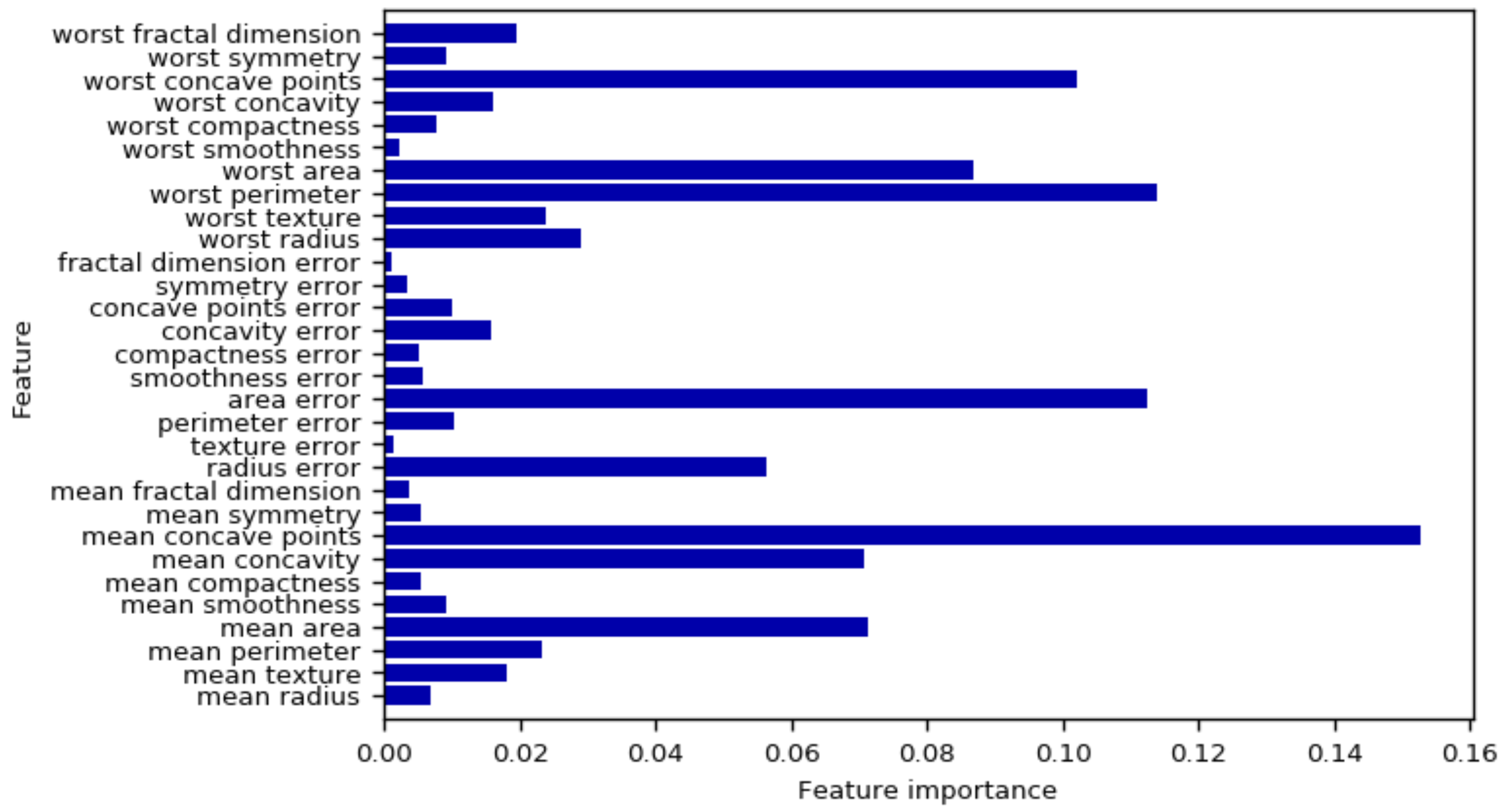
In scikit-learn OOB error are returned as follows:

```
oob_error = 1 - clf.oob_score_
```



Feature importance

RandomForests provide more reliable feature importances, based on many alternative hypotheses (trees)



Strengths, weaknesses and parameters

RandomForest are among most widely used algorithms:

- Don't require a lot of tuning
- Typically very accurate models
- Handles heterogeneous features well
- Implictly selects most relevant features

Downsides:

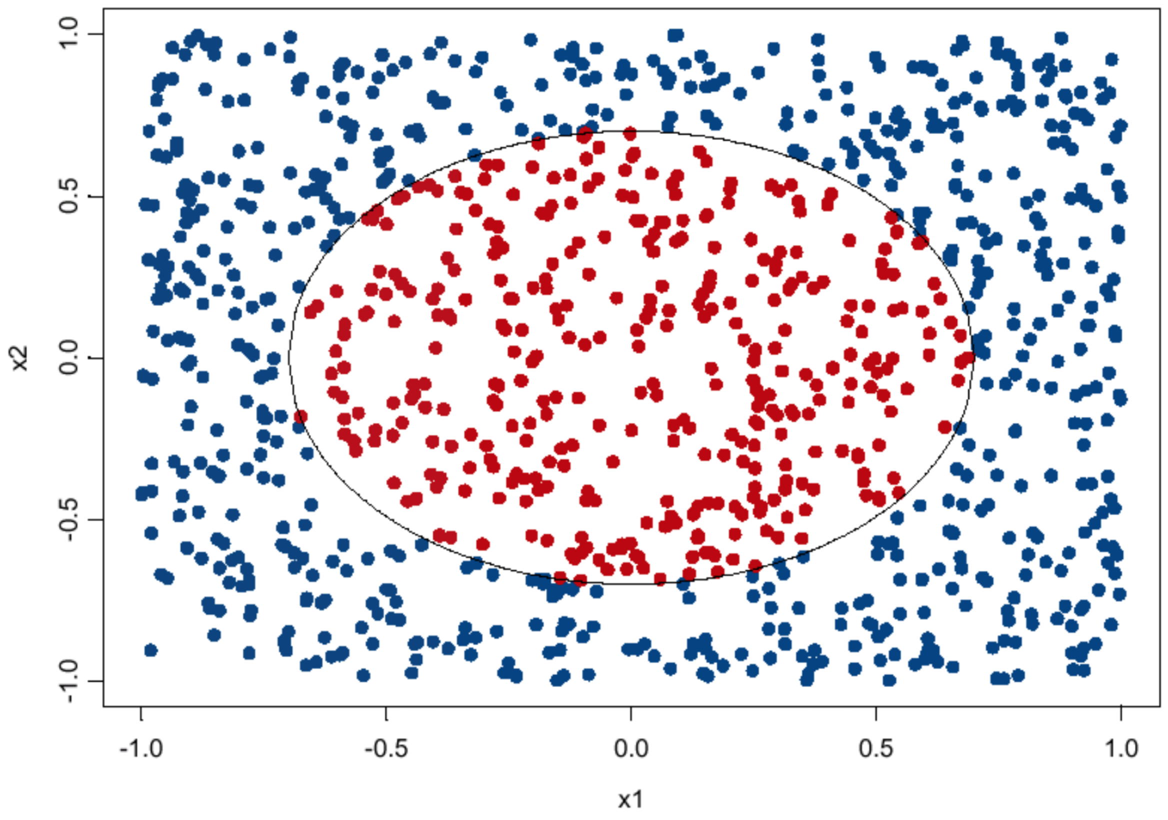
- less interpretable, slower to train (but parallellizable)
- don't work well on high dimensional sparse data (e.g. text)

Gradient Boosted Regression Trees (Gradient Boosting Machines)

Instead of reducing the variance of overfitted models, reduce the bias of underfitted models

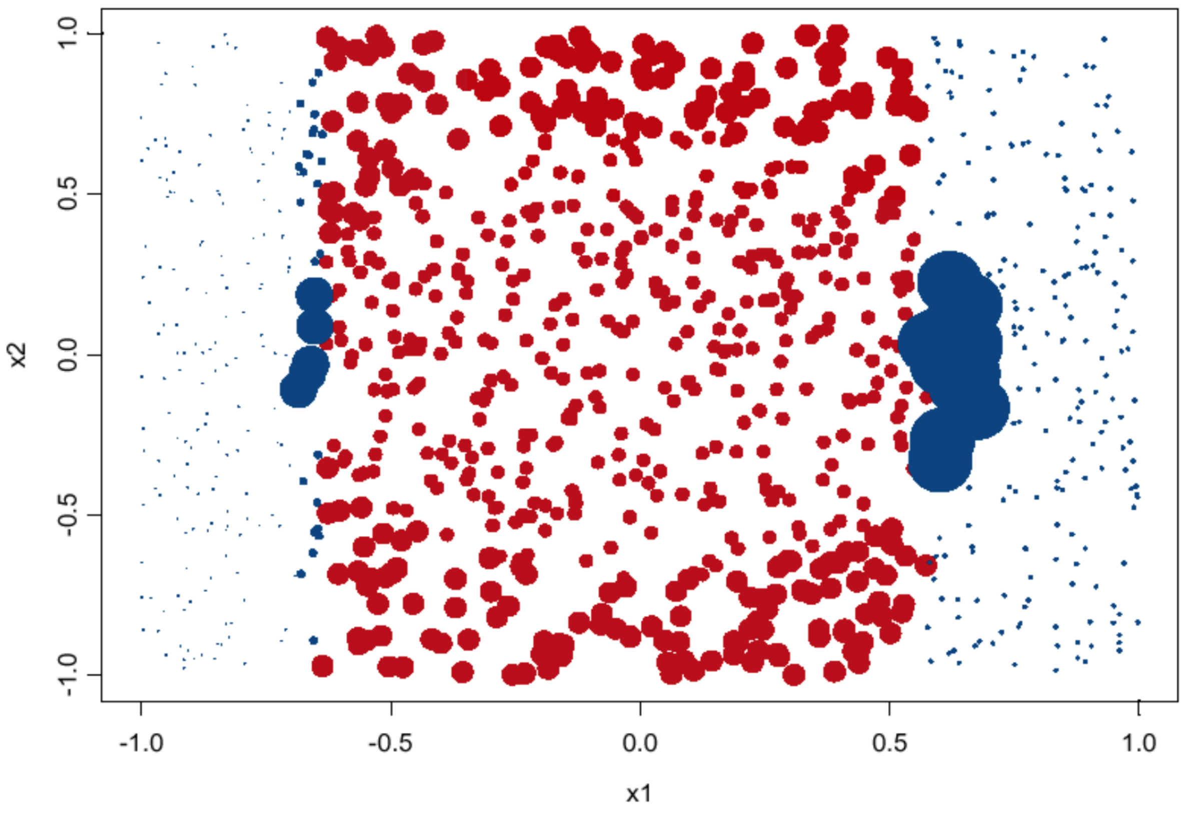
- Use strong pre-pruning to build very shallow trees
  - Default `max_depth`=3
- Iteratively build new trees by increasing weights of points that were badly predicted
- Example of *additive modelling*: each tree depends on the outcome of previous trees
- Optimization: find optimal weights for all data points
  - Gradient descent (covered later) finds optimal set of weights
  - `learning rate` controls how strongly the weights are altered in each iteration (default 0.1)
- Repeat `n_estimators` times (default 100)

Example:

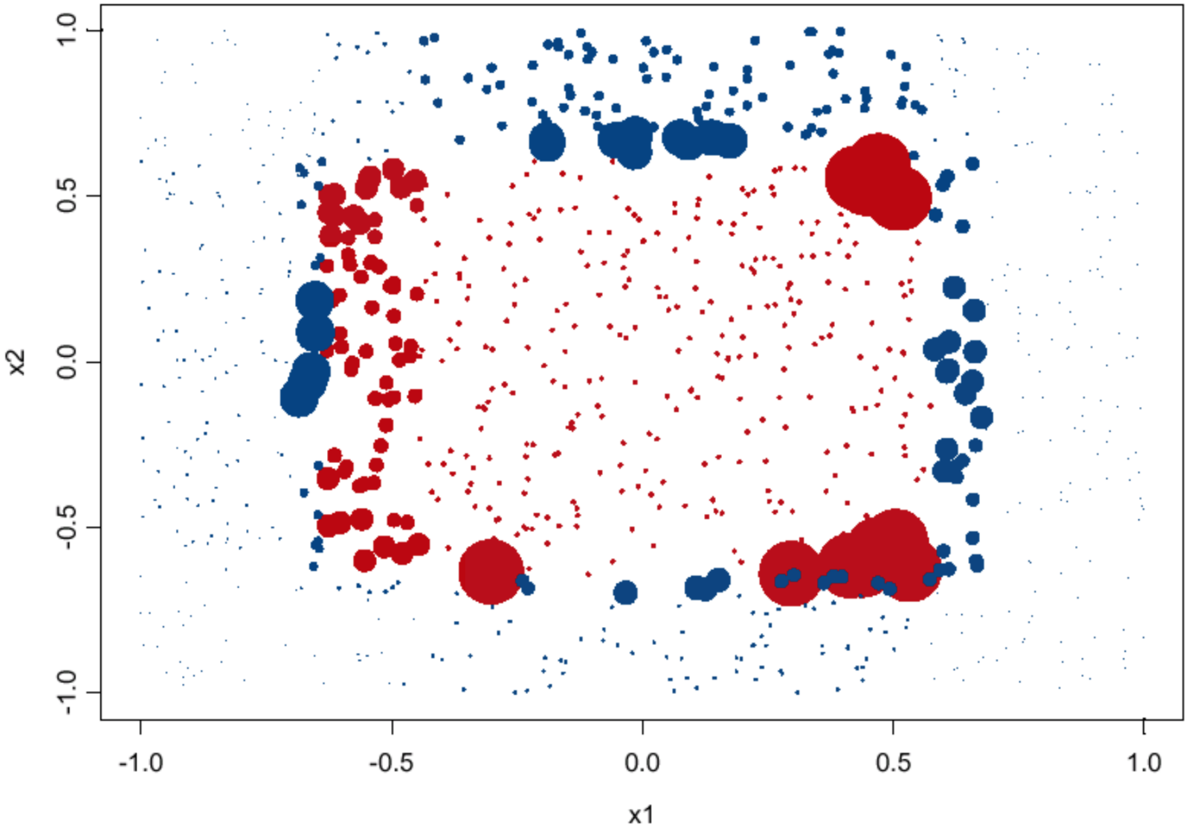


After 1 iteration

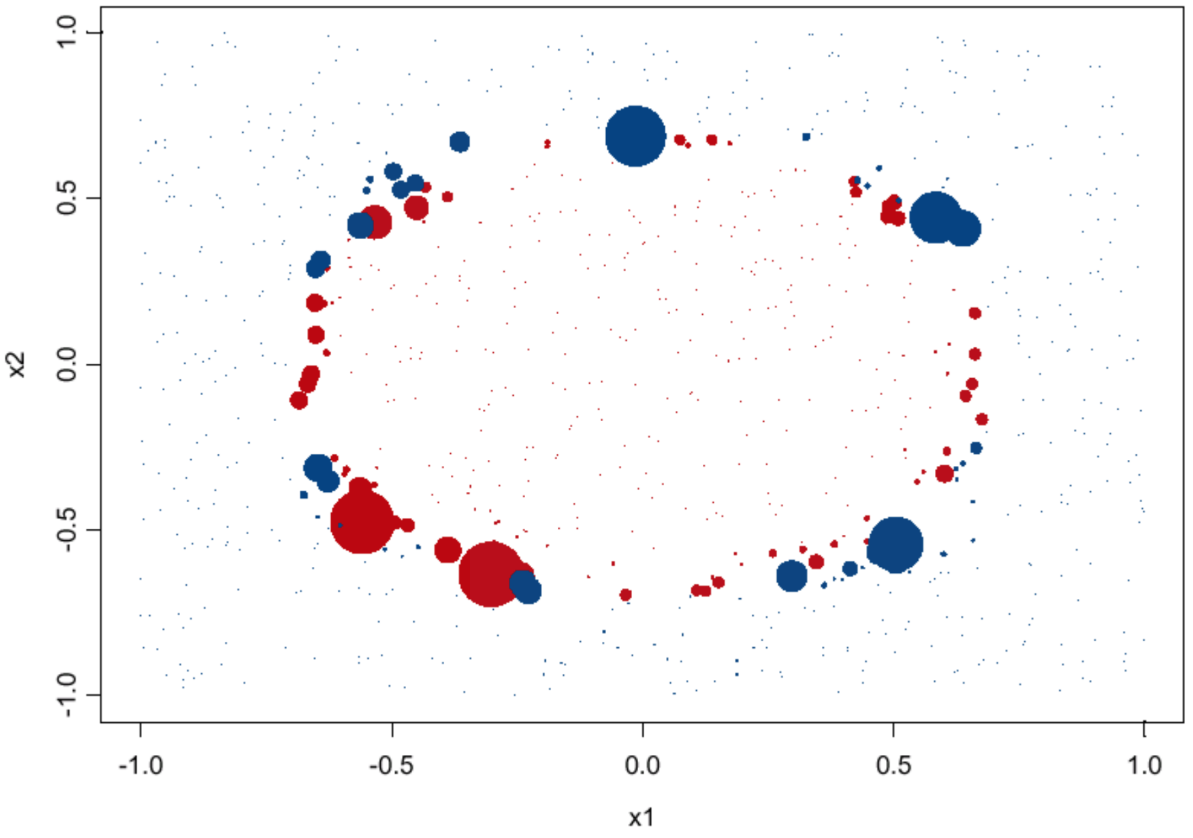
- The simple decision tree divides space
- Misclassified points get higher weight (larger dots)



After 3 iterations

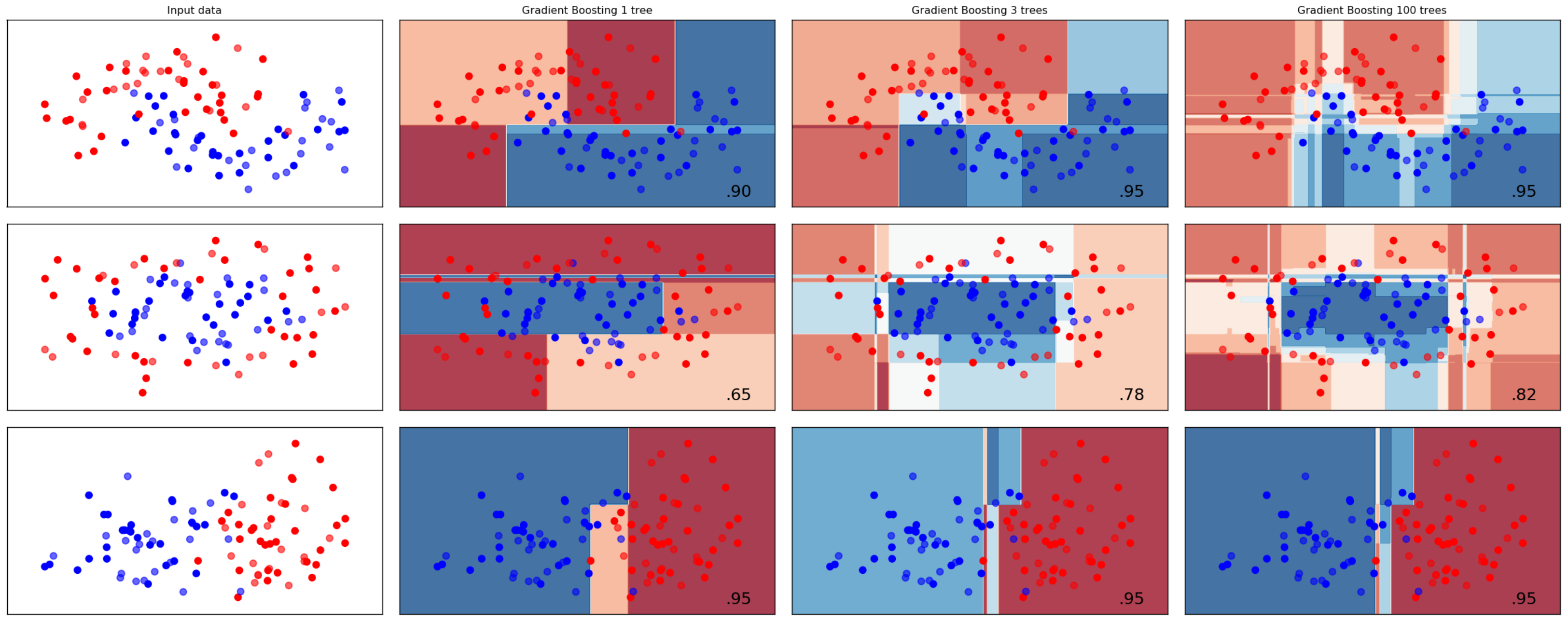


After 20 iterations



Each tree provides good predictions on part of the data, use voting for final prediction

- Soft voting for classification, mean values for regression



Tuning

- n_estimators: Higher is better, but will start to overfit
- learning_rate: Lower rates mean more trees are needed to get more complex models
    - Main regularizer, also known as 'shrinkage'
    - Set n_estimators as high as possible, then tune learning_rate
- max_depth: typically kept low (<5), reduce when overfitting
- loss: Loss function used for gradient descent (defaults OK)
    - Classification:
        - `deviance` (default): log-likelihood loss (as in logistic regression)
        - `exponential`: exponential loss (AdaBoost algorithm)
    - Regression:
        - `ls`: Least squares (typically the best option)

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

```
Accuracy on training set: 1.000
Accuracy on test set: 0.958
```

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

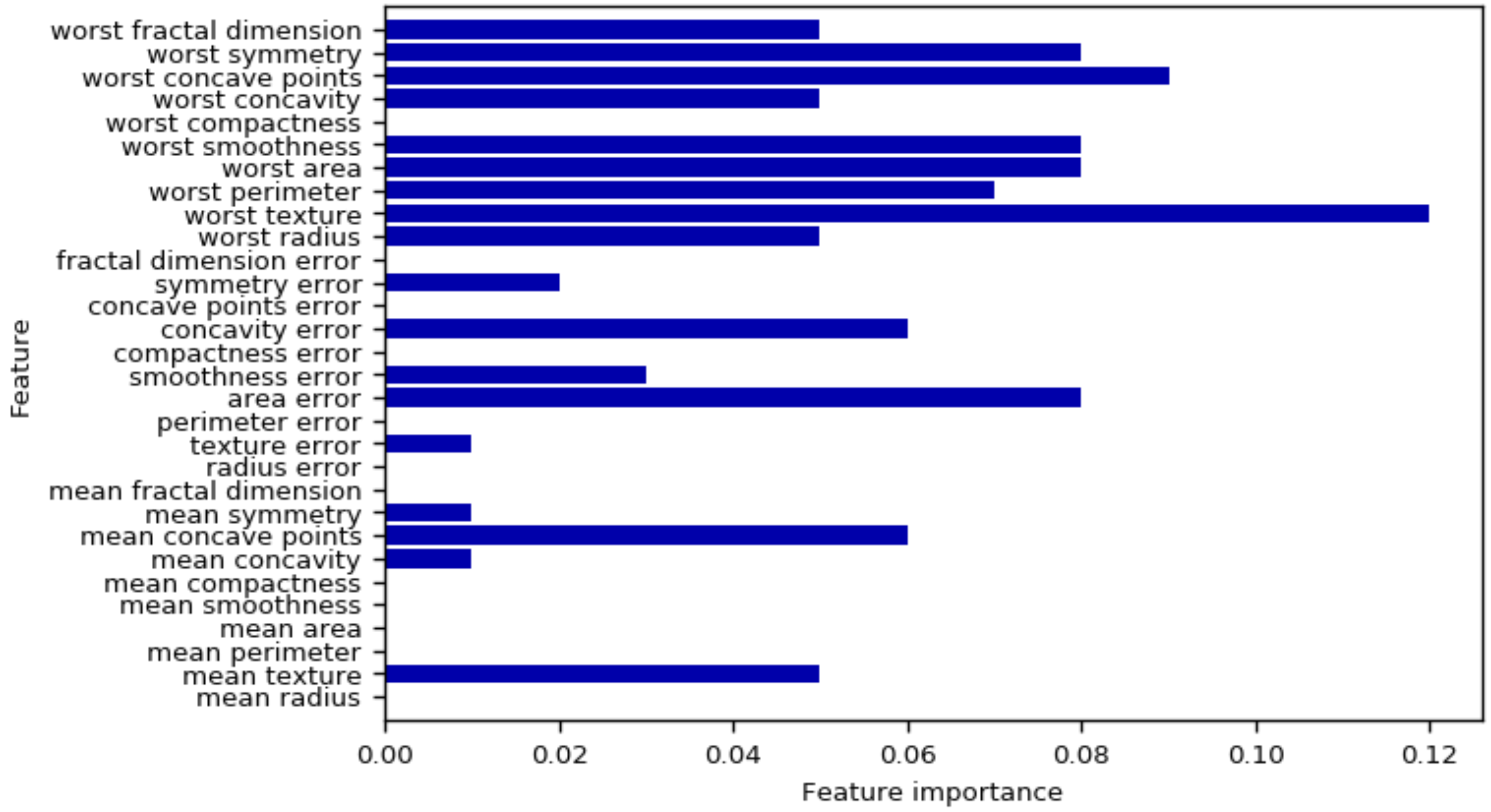
```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

Gradient boosting machines use much simpler trees

- Hence, tends to completely ignore some of the features



Strengths, weaknesses and parameters

- Among the most powerful and widely used models
- Work well on heterogeneous features and different scales
- Require careful tuning, take longer to train.
- Does not work well on high-dimensional sparse data

Main hyperparameters:

- `n_estimators`: Higher is better, but will start to overfit
- `learning_rate`: Lower rates mean more trees are needed to get more complex models
  - Set `n_estimators` as high as possible, then tune `learning_rate`
- `max_depth`: typically kept low (<5), reduce when overfitting

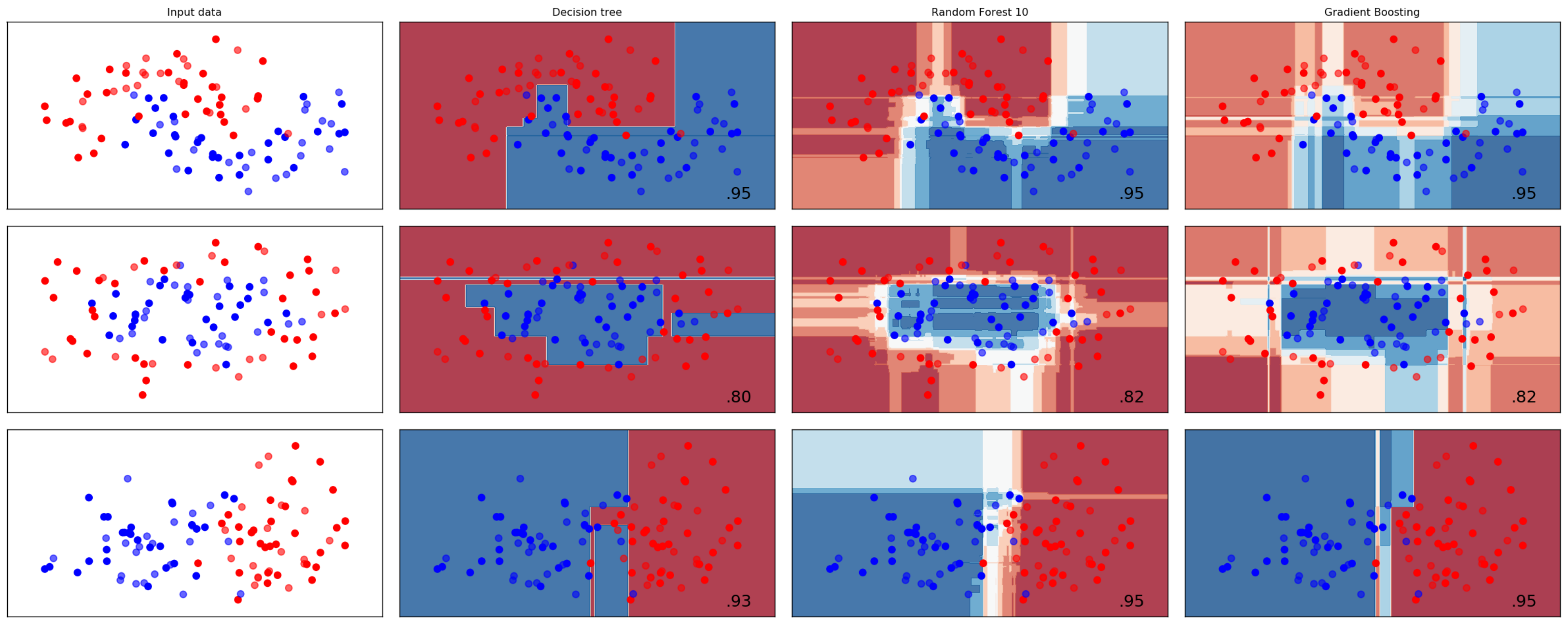
XGBoost

XGBoost is another python library for gradient boosting (install separately).

- The main difference lies the use of approximation techniques to make it faster.
  - Hence, you can do 10x (or 100x) more boosting iterations in same amount of time
- Sketching: Given 10000 possible splits, it will only consider 300 "good enough" splits by default
  - Controlled by the `sketch_eps` parameter (default 0.03)
- Loss function approximation with Taylor Expansion: more efficient way to evaluate splits

Further reading: XGBoost Documentation (https://xgboost.readthedocs.io/en/latest/parameter.html#parameters-for-tree-booster) Paper (http://arxiv.org/abs/1603.02754)

Comparison


Input data
Decision tree
Random Forest 10
Gradient Boosting
.95
.95
.95
.80
.82
.82
.93
.95
.95

Summary

- Bagging / RandomForest is a variance-reduction technique
    - Build many high-variance (overfitting) models
        - Typically deep (randomized) decision trees
        - The more different the models, the better
    - Aggregation (soft voting or averaging) reduces variance
    - Parallellizes easily
- Boosting is a bias-reduction technique
    - Build many high-bias (underfitting) models
        - Typically shallow decision trees
        - Sample weights are updated to create different trees
    - Aggregation (soft voting or averaging) reduces bias
    - Doesn't parallelize easily
- You can build ensembles with other models as well
    - Especially if they show high variance or bias
- It is also possible to build *heterogeneous* ensembles
    - Models from different algorithms
    - Often a meta-classifier is trained on the predictions: Stacking