Trees and Ensembles

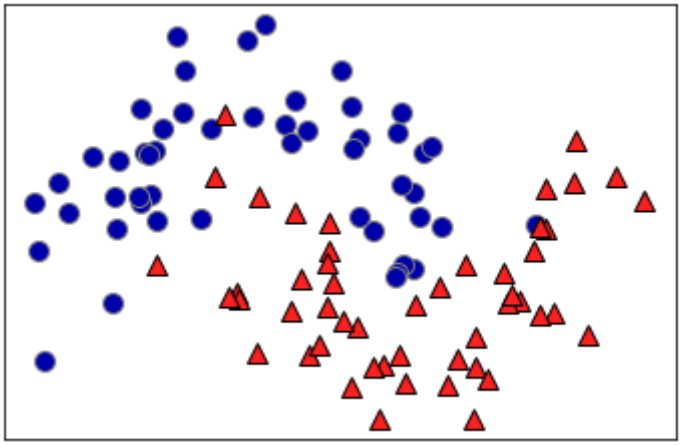
Trees

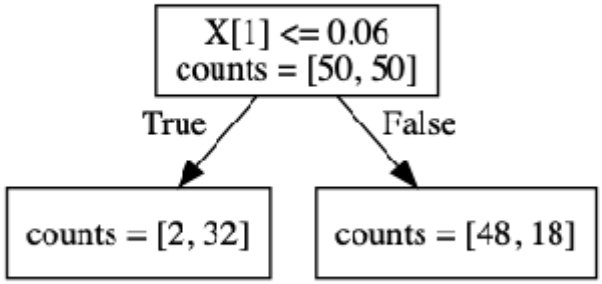
Decision Trees

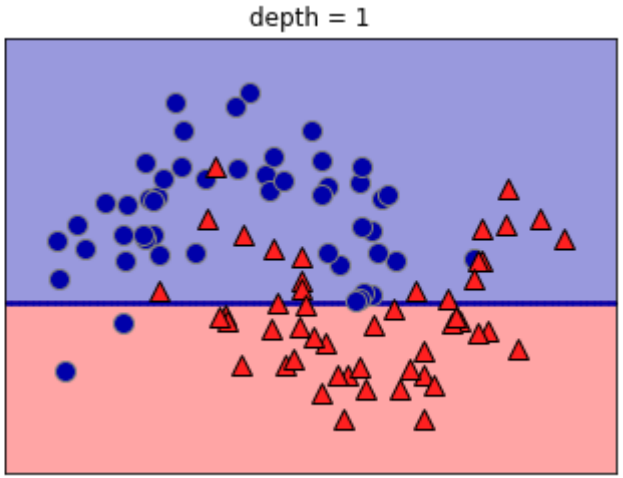
- Representation: Assume we can represent the concept we want to learn with a *decision tree*
    - Repeatedly split the data based on one feature at a time
    - Note: *Oblique trees* can split on combinations of features
- Evaluation (loss): One tree is better that another tree according to some heuristic
    - Classification: Instances in a leaf are all of the same class (`pure` leafs)
    - Regression: Instances in a leaf have values close to each other
- Optimization: Recursive, heuristic greedy search (Hunt's algorithm)
    - Make first split based on the heuristic
    - In each branch, repeat splitting in the same way

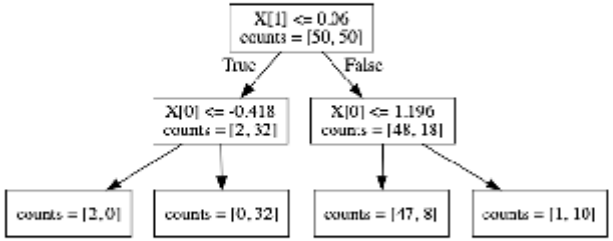
Decision Tree classification

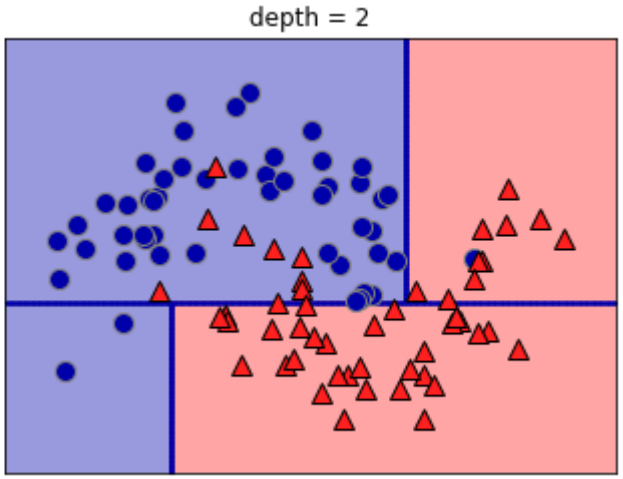
Where would you make the first split?

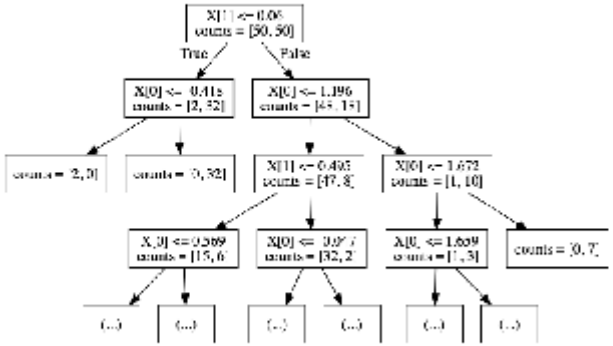


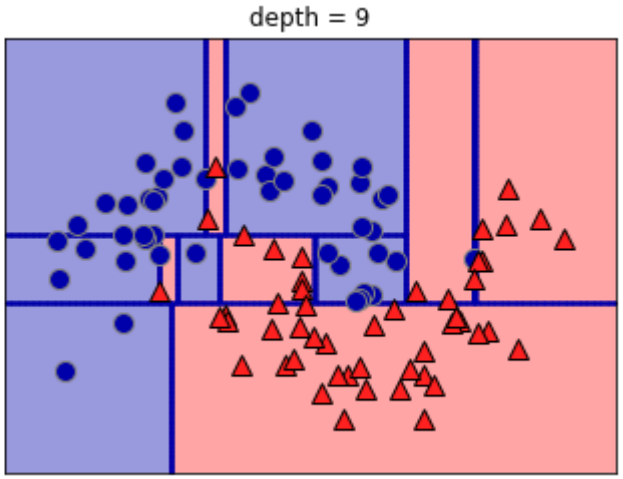

X[1] <= 0.06
counts = [50, 50]
True
False
counts = [2, 32]
counts = [48, 18]


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] <= 0.411
counts = [2, 52]
X[0] <= 1.196
counts = [48, 48]
counts = [2, 0]
counts = [0, 52]
X[1] <= 0.493
counts = [47, 8]
X[0] <= 1.672
counts = [1, 10]
X[0] <= 0.569
counts = [15, 6]
X[0] <= 0.97
counts = [32, 2]
X[0] <= 1.659
counts = [1, 3]
counts = [0, 7]
(...)
(...)
(...)
(...)
(...)
(...)


depth = 9

Heuristics

- We start from a dataset of n points $D = \{(x_i, y_i)\}_{i=1}^{n}$ where y_i is one of k classes
- Consider splits between adjacent data point of different class, for every variable
- After splitting, each leaf will have $\hat{p}_k$ = the relative frequency of class k

We can define several *impurity measures*:

- Misclassification Error (leads to larger trees): $1 - \underset{k}{\text{argmax}}\ \hat{p}_k$
- Gini-Index: $\sum_{k \neq k'} \hat{p}_k \hat{p}_{k'} = \sum_{k=1}^{K} \hat{p}_k (1 - \hat{p}_k)$
- Sum up the heuristics per leaf, weighted by the number of examples in each leaf

$$\sum_{l=1}^{L} \frac{|X_{i=l}|}{|X_i|} Gini(X_{i=l})$$

Visualization: the plots on the right show the class distribution of the 'top' and 'bottom' leaf, respectively.

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

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

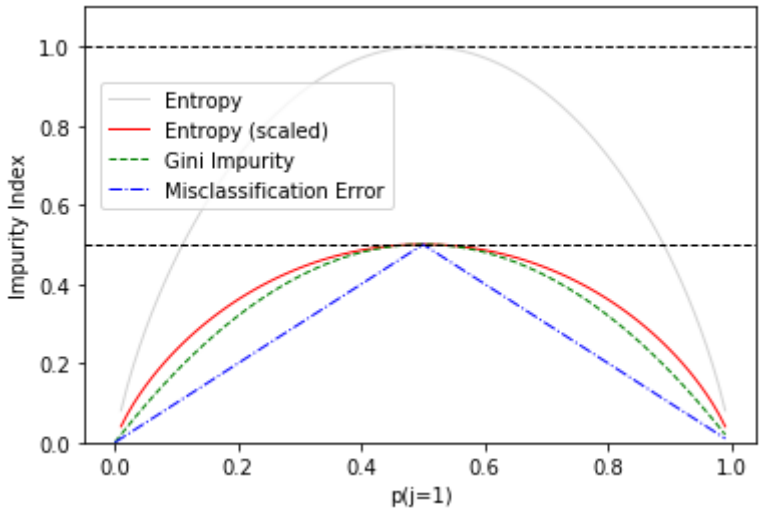
- Information Gain (a.k.a. Kullback–Leibler divergence) measures how much entropy has reduced by splitting on attribute X_i:

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

with X = the training set, l a specific leaf after splitting on feature X_i, $X_{i=l}$ is the set of examples in leaf l: $\{x \in X | X_i \in l\}$

Heuristics visualized (binary class)

- Note that `gini != entropy/2`



Example

Let's compute information gain for a dataset with categorical features:

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	+	+	-	+	-	-

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

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

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

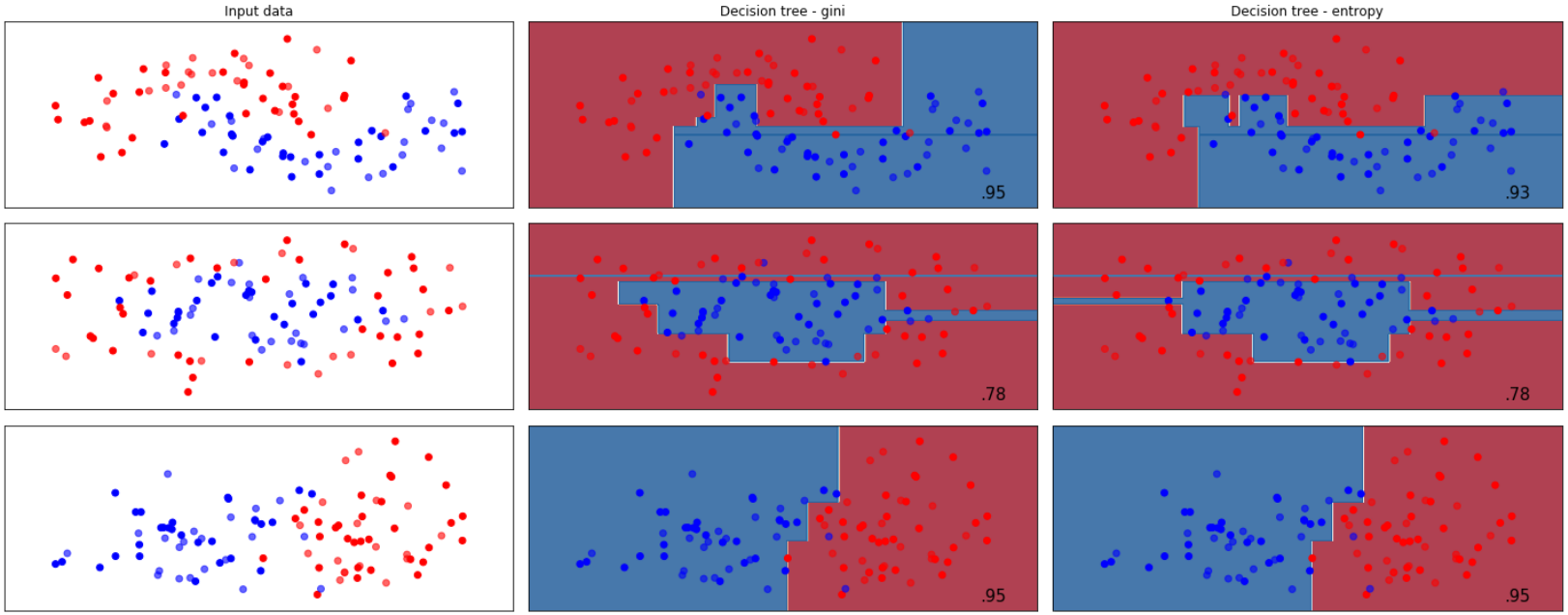
hence we split on a1

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
.95
.93
.78
.78
.95
.95

Handling many-valued features

What happens when a categorical 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.

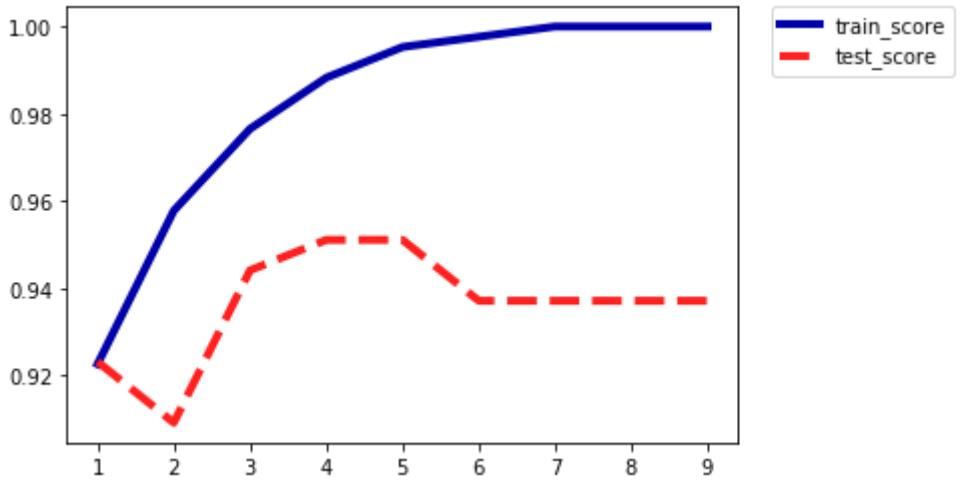
Overfitting: Controlling complexity of Decision Trees

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) to allow a split
- 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)

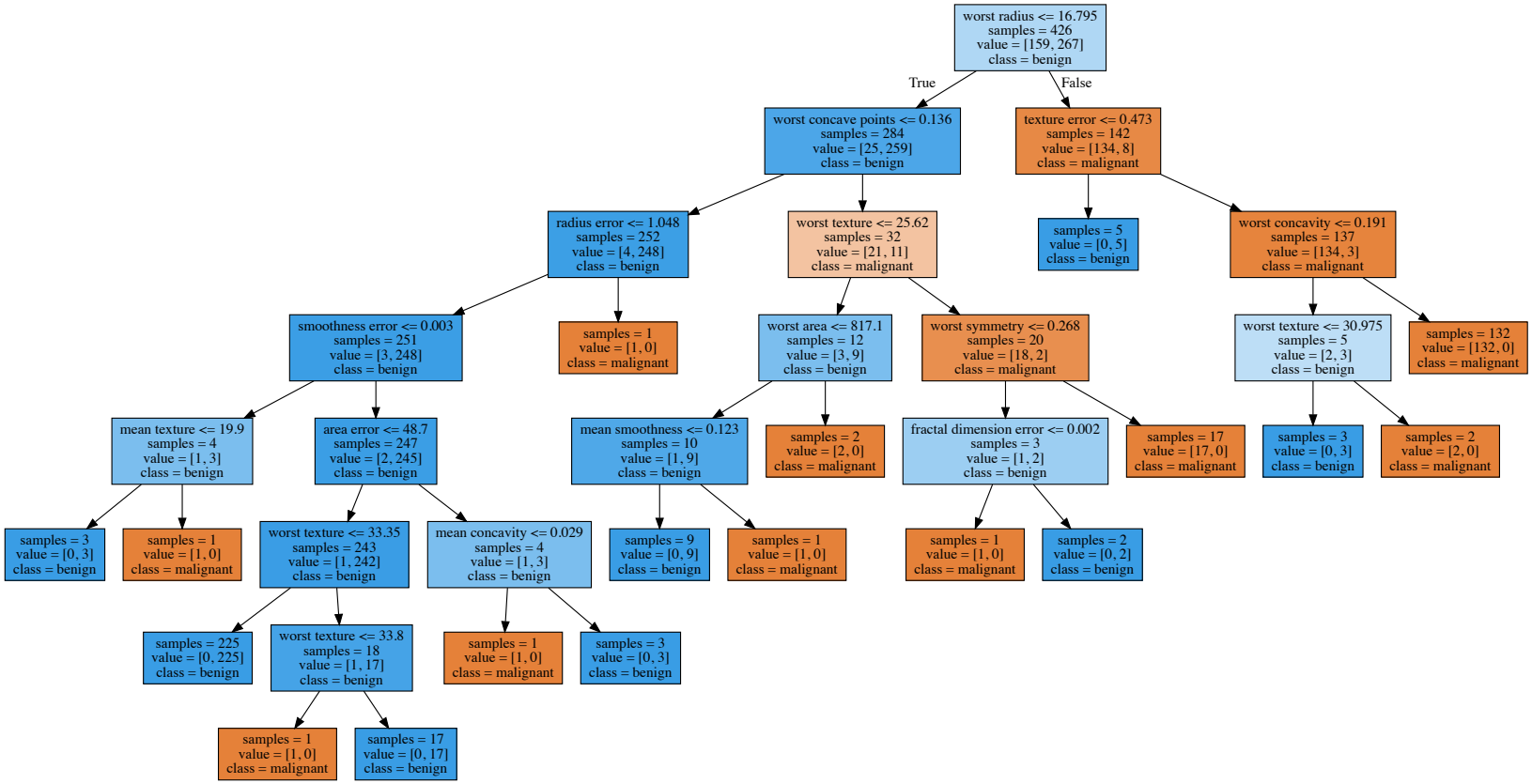
Effect of pre-pruning:

- Shallow trees tend to underfit (high bias)
- Deep trees tend to overfit (high variance)



Decision Trees are easy to interpet

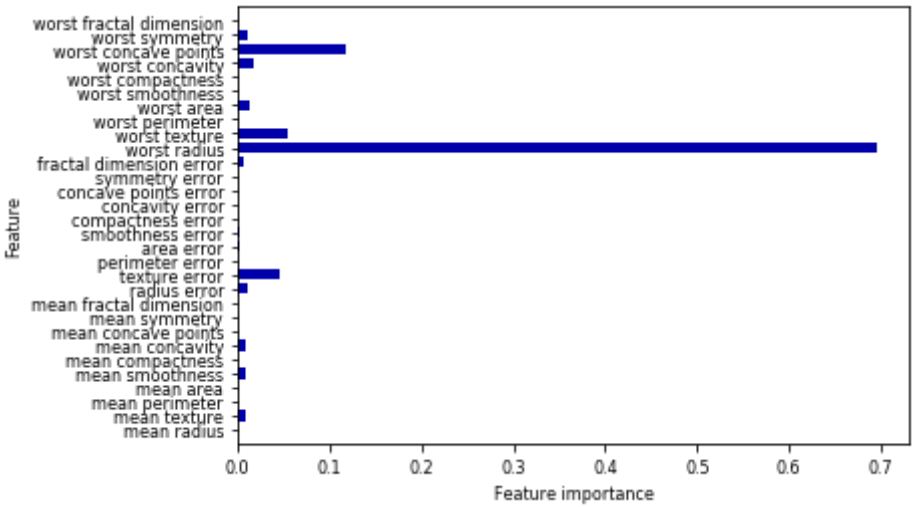
- Visualize and find the path that most data takes



`DecisionTreeClassifier` also returns *feature importances*

`tree.feature_importances_`

- In [0,1], sum up to 1
- High values for features selected early (near the root)



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 (mean of all values in the leaf) and actual value y_i:

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

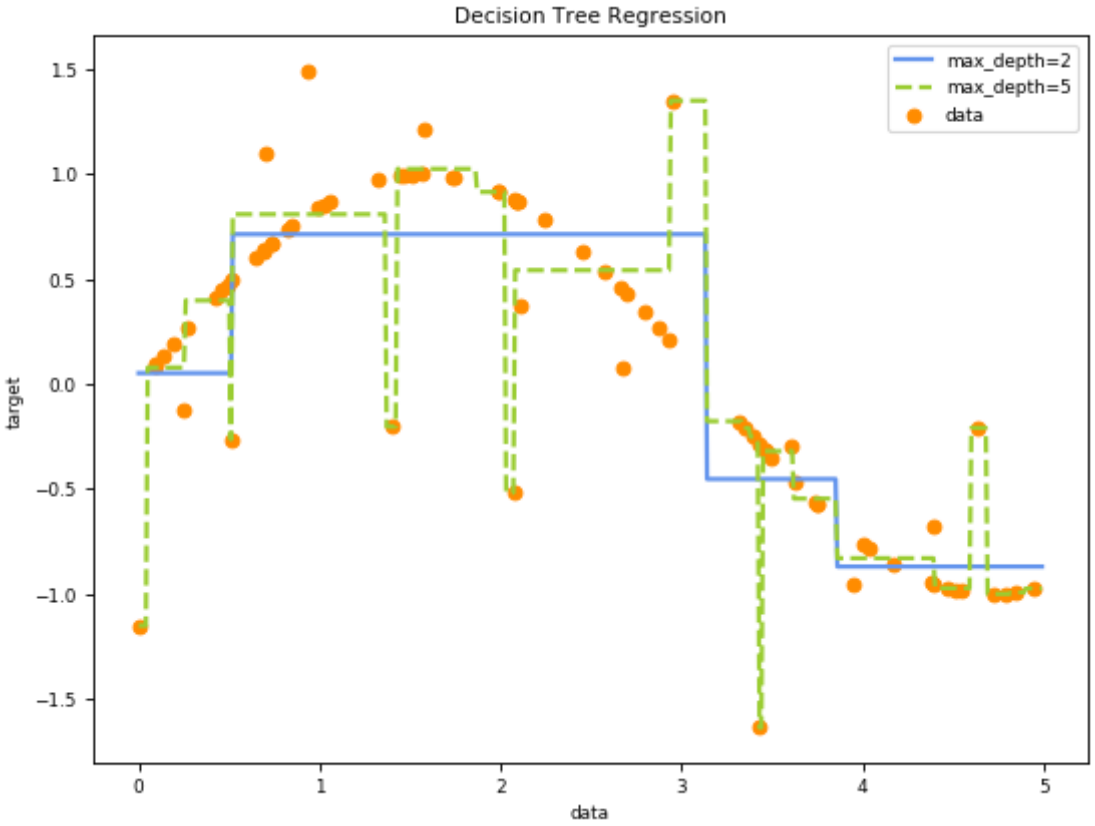
- Assuming that the tree predicts y_i as the average of all x_i in the leaf:

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

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

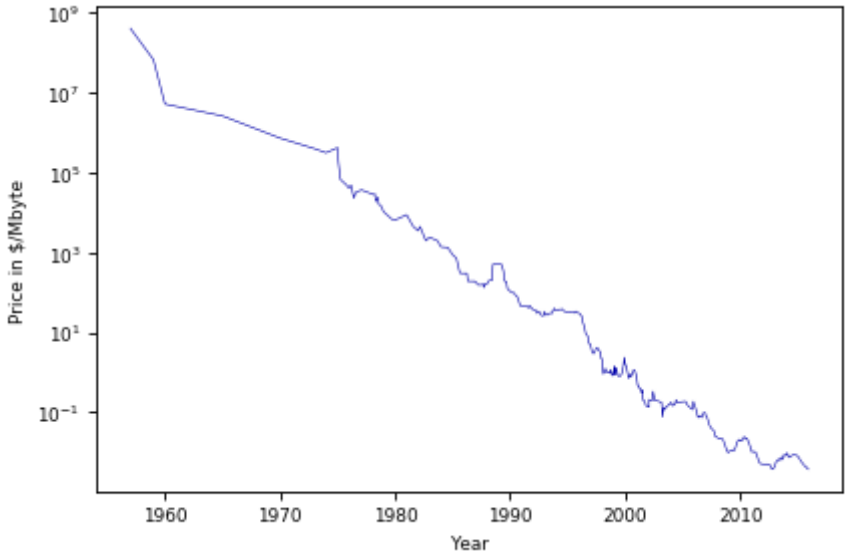
In scikit-learn

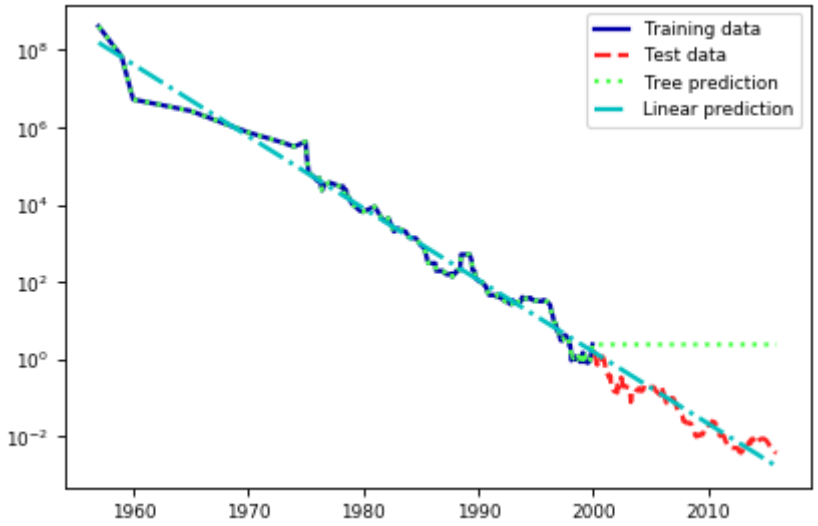
Regression is done with `DecisionTreeRegressor`



Note that decision trees do not extrapolate well.

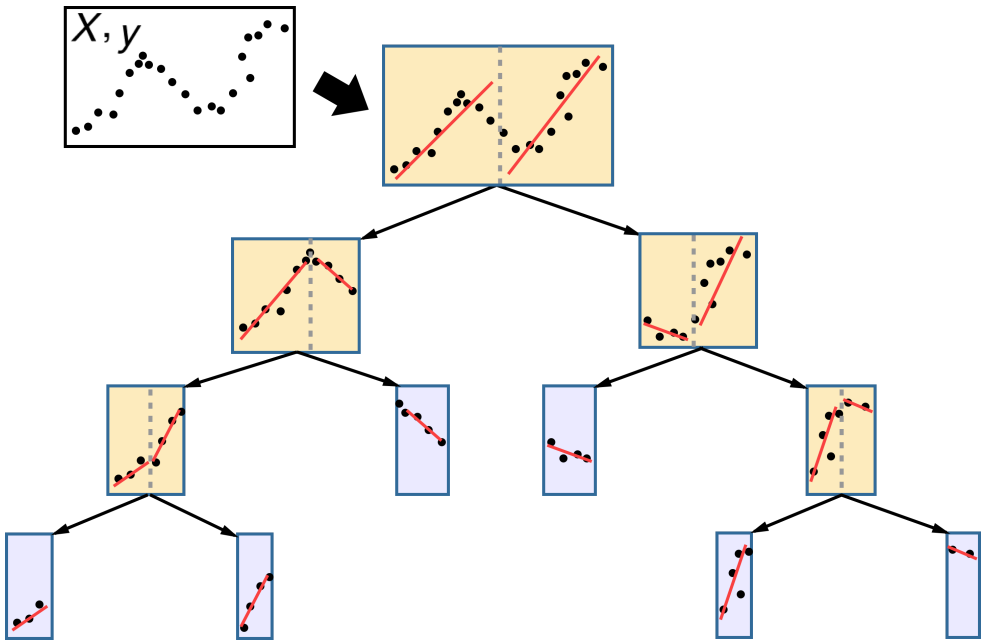
- The leafs return the same *mean* value no matter how far the new data point lies from the training examples.
- Example on the `ram_price` forecasting dataset




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

Model trees

- Instead of predicting a single value per leaf (e.g. mean value for regression), you can build a model on all the points remaining in a leaf
    - E.g. a linear regression model
- Can learn more complex concepts, extrapolates better. Overfits easily.



Strengths, weaknesses and parameters

Decision trees:

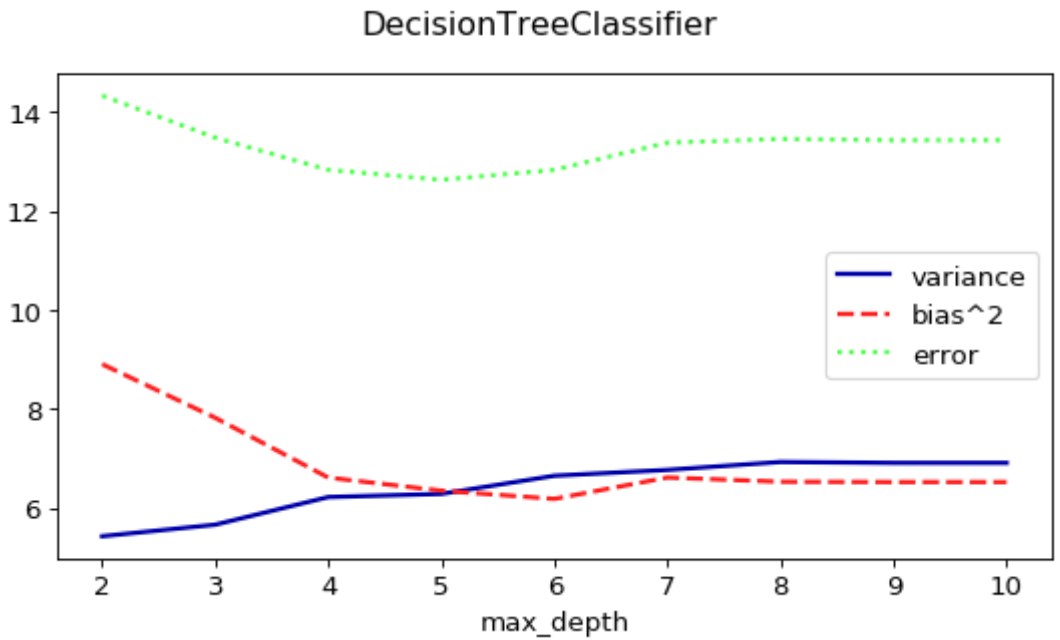
- Work well with features on completely different scales, or a mix of binary and continuous features
    - Does not require normalization
- Interpretable, easily visualized
- Tend to overfit easily.

Pre-pruning: regularize by:

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

On under- and overfitting

- Let's study which types of errors are made by decision trees
- Deep trees have high variance but low bias
  - What if we built many deep trees and average them out to reduce variance?
- Shallow trees have high bias but very low variance
  - What if we could correct the systematic mistakes to reduce bias?



Ensemble learning

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

- **Bagging**: Reduce variance: Build many trees on random samples and do a vote over the predictions
    - **RandomForests**: Build randomized trees on random bootstraps of the data
- **Boosting**: Reduce bias: Build trees iteratively, each correcting the mistakes of the previous trees
    - **Adaboost**: Ensemble of weighted trees, increasing importance of misclassified points
    - **Gradient boosting machines**: Gradually update importance of hard points until ensemble is correct
    - **XGBoost**: Faster implemenation of gradient boosting machines
- **Stacking**: Build group of base models, and train a meta-model to learn how to combine the base model predicions

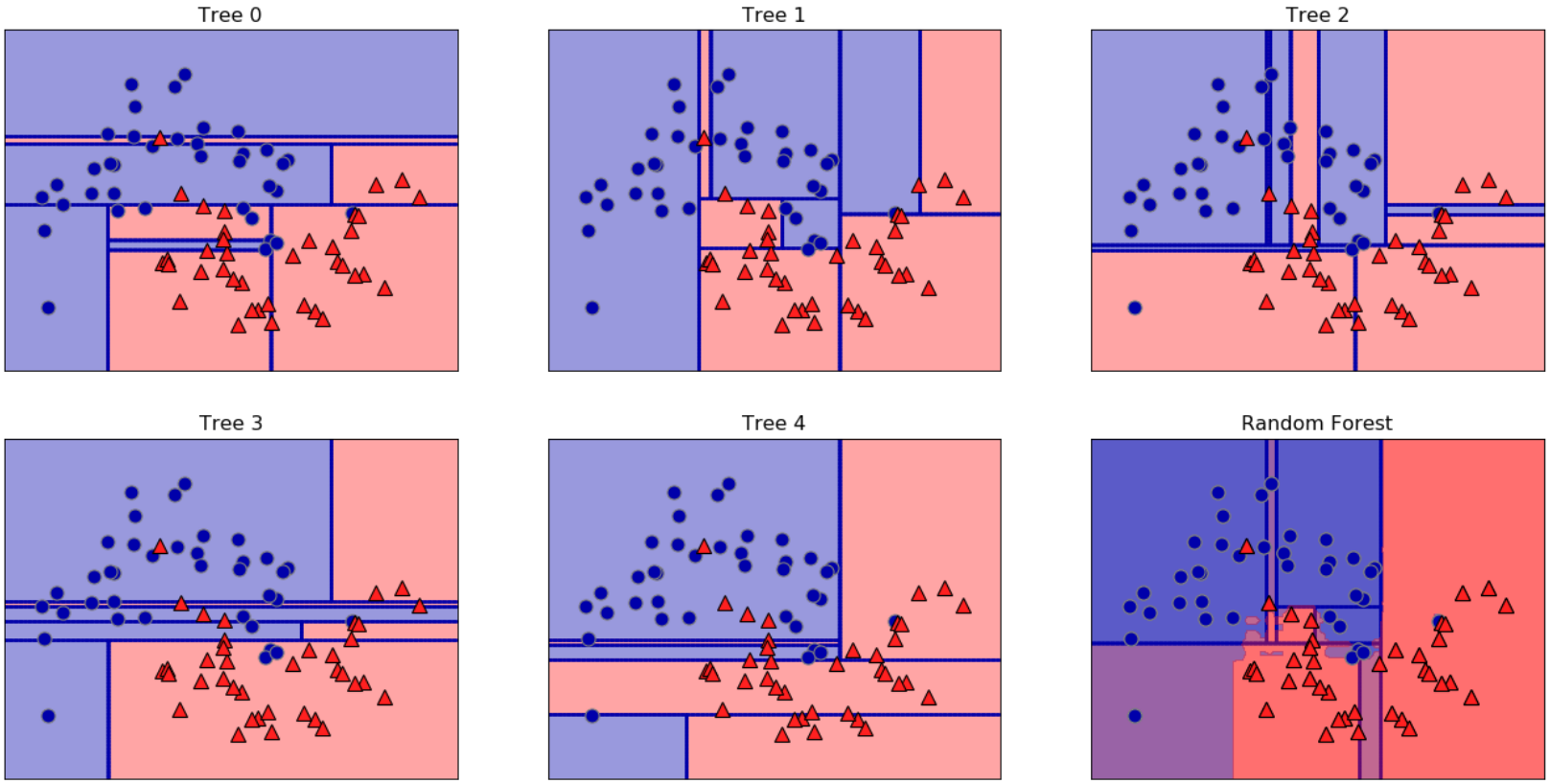
Bagging (Bootstrap Aggregating)

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 aggreting the individual tree predictions
- Can be done with any model (but usually trees)
    - Since Bagging only reduces variance (not bias), it makes sense to use models that are high variance, low bias
- RandomForest: Randomize trees by considering only a random subset of features of size `max_features` *in each node*
    - Higher variance, lower bias than normal trees
    - Small `max_features` yields more different trees, more smoothing
    - Default: $sqrt(n\_features)$ for classification, $log2(n\_features)$ for regression

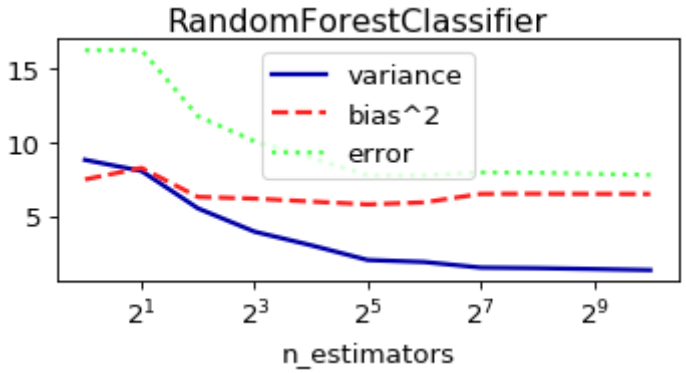
Making predictions:

- 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
- Each base model gets the same weight in the final prediction


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

Effect on bias and variance

- Increasing the number of estimators decreases variance
- Bias is mostly unaffected, but will increase if the forest becomes too large (oversmoothing)



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)

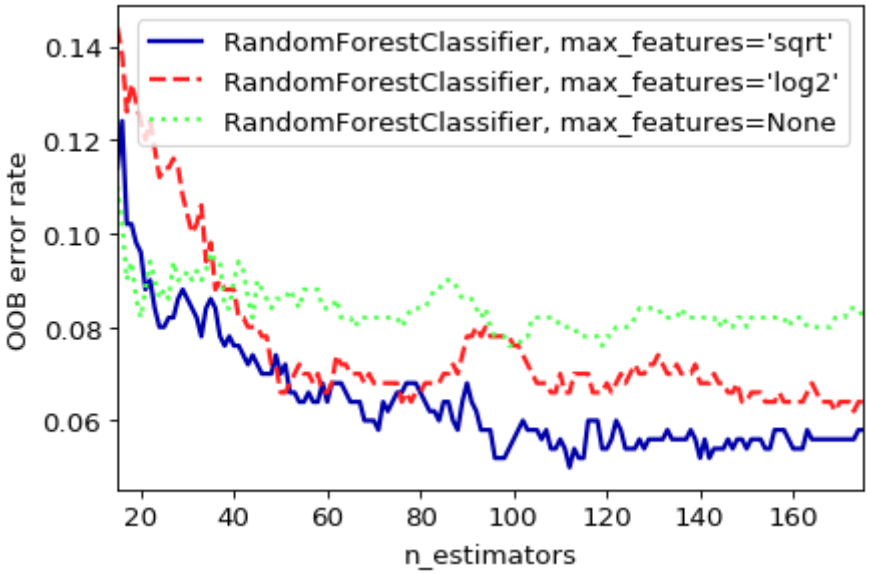
`n_jobs` sets the number of parallel cores to run
`random_state` should be fixed for reproducibility

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

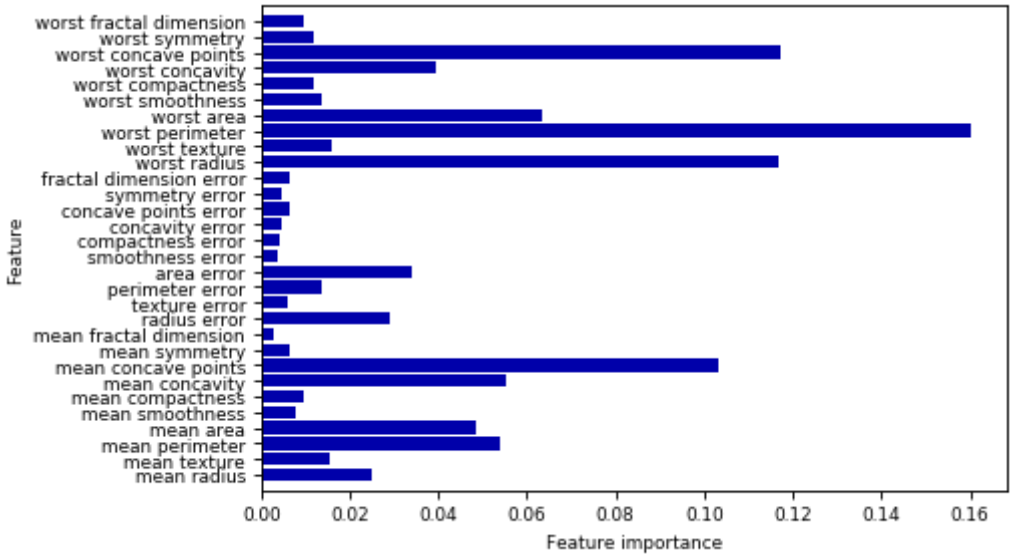
In scikit-learn OOB error are returned as follows:

```
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)

Adaptive Boosting (AdaBoost)

- Builds an ensemble of *weighted* weak learners
    - Typically shallow trees or stumps
- Each base model tries to correct the mistakes of the previous ones
    - Sequential, not parallel (like RandomForest)
    - We give misclassified samples more weight
- Force next model to get these points right by:
    - Passing on the weight to the loss (e.g. weighted Gini index)
    - Sample data with probability = sample weights
        - Important samples sampled multiple times
        - Weights are
- Do weighted vote over all models

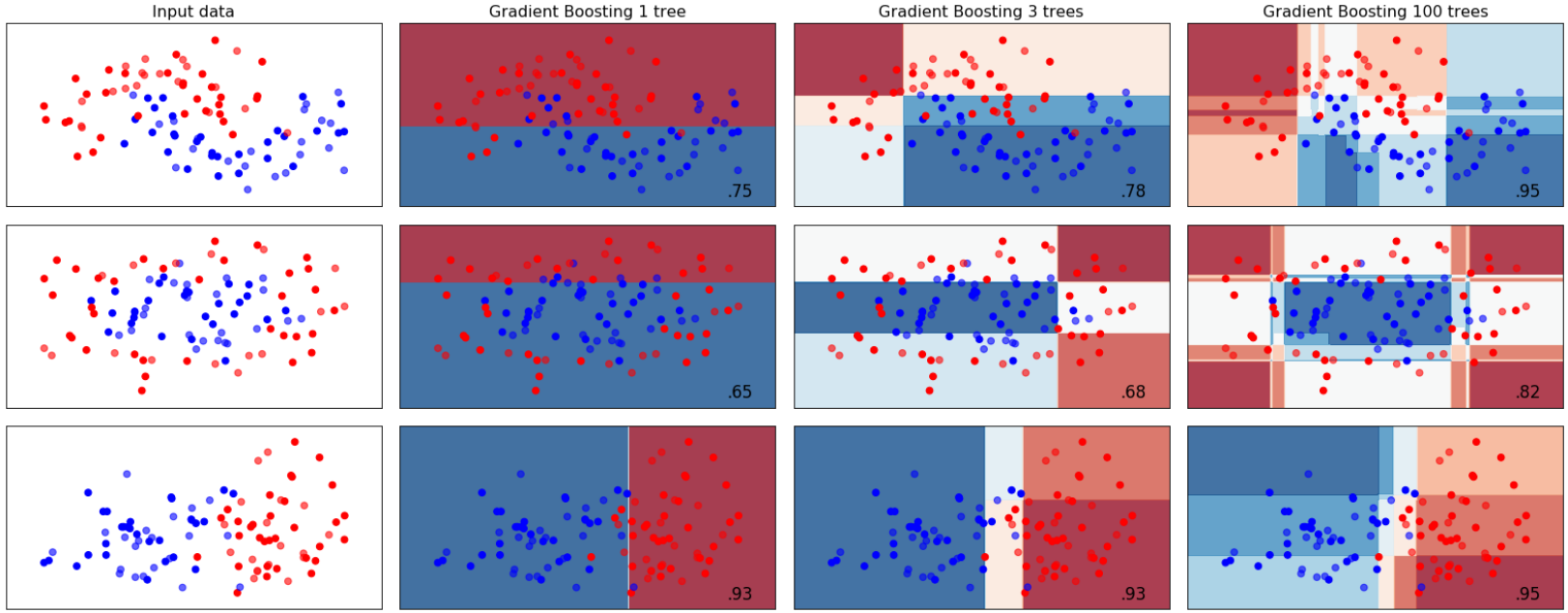
AdaBoost algorithm

- Reset sample weights to $\frac{1}{N}$
- Build a model, using it's own algorithm (e.g. trees with gini index)
- Give it a weight related to its error E

$$w_i = \frac{1}{2}\log(\frac{1-E}{E})$$

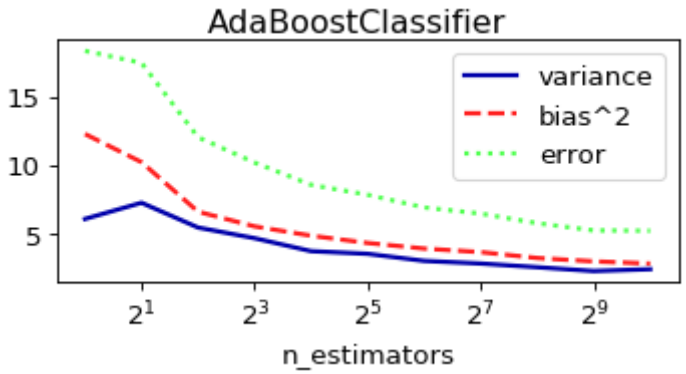
  - Logic function maps error from [0,Inf] to [-1,1]
  - Add small minimum error to E to avoid infinities
- Update the sample weights
  - Increase weight of incorrectly predicted samples: $s_{n,i+1} = s_{n,i}e^{w_i}$
  - Decrease weight of correctly predicted samples: $s_{n,i+1} = s_{n,i}e^{-w_i}$
  - Normalize weights to add up to 1
- Sample new points according to $s_{n,i+1}$
- Repeat for I rounds

Visualization


Input data
Gradient Boosting 1 tree
Gradient Boosting 3 trees
Gradient Boosting 100 trees
.75
.78
.95
.65
.68
.82
.93
.93
.95

AdaBoost reduces bias (and a little variance)

- Boosting too much will eventually increase variance



AdaBoost Recap

- Representation: weighted ensemble of base models
    - Base models can be built by any algorithm
- Loss function: weighted loss function of base models
- Optimization: Greedy search

Gradient Boosted Regression Trees (Gradient Boosting Machines)

Several differences to AdaBoost:

- Start with initial guess (e.g. 1 leaf, average value of all samples)
- Base-models are shallow trees (depth 2-4, not stumps)
- Models are weighted (scaled) by same amount (learning rate)
- Subsequent models aim to predict the error of the previous model
    - *Additive model*: final prediction is the sum of all base-model predictions
- Iterate until I trees are built (or error converges)

GradientBoosting Intuition

- Do initial prediction M_0 (e.g. average target value)
- Compute the *pseudo-residual* (error) for every sample n: $r_n = y_n - y_n^{(M_i)}$
    - Where $y_n^{(M_i)}$ is the prediction for y_n by model M_i

- Build new model M_1 to predict the pseudo-residual of M_0
- New prediction at step I:

$$y_n = y_n^{(M_{i-1})} + \lambda * y_n^{(M_i)} = y_n^{(M_0)} + \sum_{i=1}^{I} \lambda * y_n^{(M_i)}$$

    - λ is the learning rate (or *shrinkage*)
    - Taking small steps in right direction reduces variance
- Compute new pseudo-residuals, and repeat
    - Each step, the pseudo-residuals get smaller
- Stop after given number of iterations, or when the residuals don't decrease anymore (early stopping)

GradientBoosting Algorithm

- Dataset of n points $D = \{(x_i, y_i)\}_{i=1}^{n}$ where y_i is a numeric target
- Differentiable loss function $\mathcal{L}(y_i, F(x))$
  - Most common: $\mathcal{L} = \frac{1}{2}(y_i - \hat{y}_i)^2$
  - $\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2 * \frac{1}{2}(y_i - \hat{y}_i) * (-1)$
- Initialize model with constant value $F_0(x) = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} \mathcal{L}(y_i, \gamma)$
  - For $\mathcal{L} = \frac{1}{2}(y_i - \hat{y}_i)^2$, this is the average of all observations
- For m=1..M (e.g. M=100 trees):
  - For i=1..n, compute pseudo-residuals $r_{im} = -\left[\frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$
  - Fit a regression model to r_{im}, create terminal regions (a.k.a. leafs) $R_{jm}, j = 1..J_m$
  - For each j, compute $\gamma_{jm} = \underset{\gamma}{\text{argmin}} \sum_{x_i \in R_{ij}} \mathcal{L}(y_i, F_{m-1}(x_i) + \gamma)$
  - Update $F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_m I(x \in R_{jm})$

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: 0.965
```

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

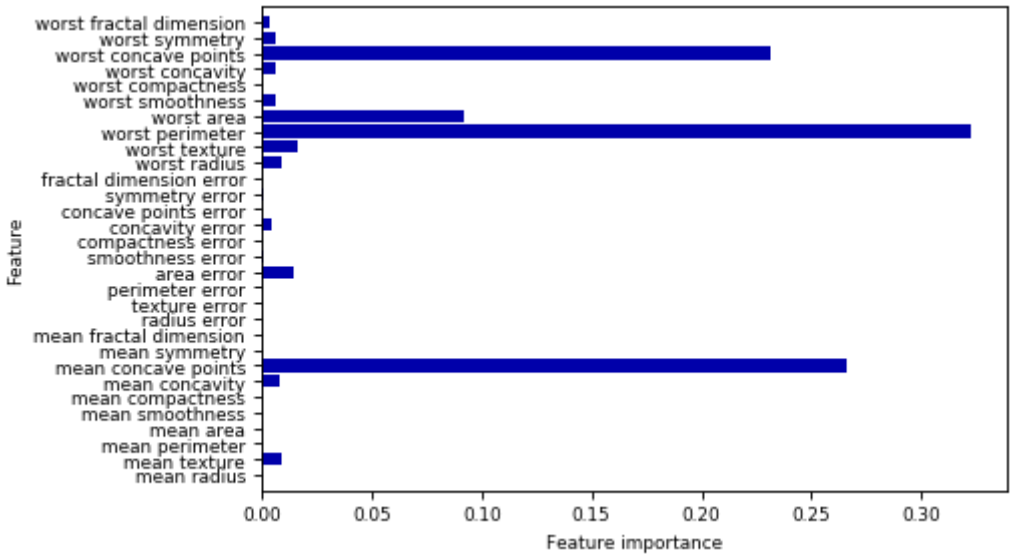
```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

```
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

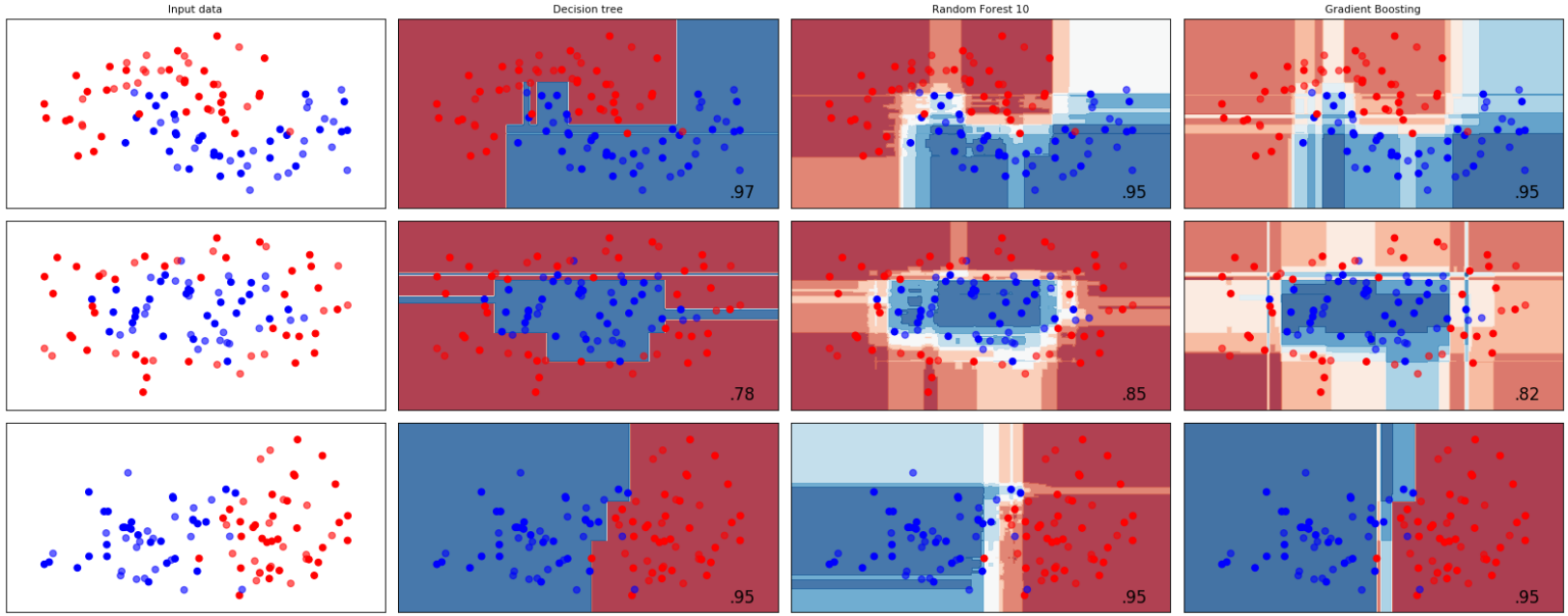
XGBoost

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
- Allows plotting of the learning curve
- Allows to stop and continue later (warm-start)

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
.97
.95
.95
.78
.85
.82
.95
.95
.95

Algorithm overview

Name	Representation	Loss function	Optimization	Regularization
Classification trees	Decision tree	Information Gain (KL div.) / Gini index	Hunt's algorithm	Tree depth,...
Regression trees	Decision tree	Min. quadratic distance	Hunt's algorithm	Tree depth,...
Model trees	Decision tree + other models in leafs	As above + used model's loss	Hunt's algorithm + used model's optimization	Tree depth,...
Bagging	Ensemble of any model	/	/	Number of models,...
RandomForest	Ensemble of random trees	/	/	Number of trees,...
AdaBoost	Ensemble of models (trees)	Weighted loss of base models	Greedy search	Number of trees,...
GradientBoosting	Ensemble of models (trees)	Ensemble loss	Gradient descent	Number of trees,...

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