Lecture 2b. Ensemble Learning

Tree-based methods

Joaquin Vanschoren, Eindhoven University of Technology

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

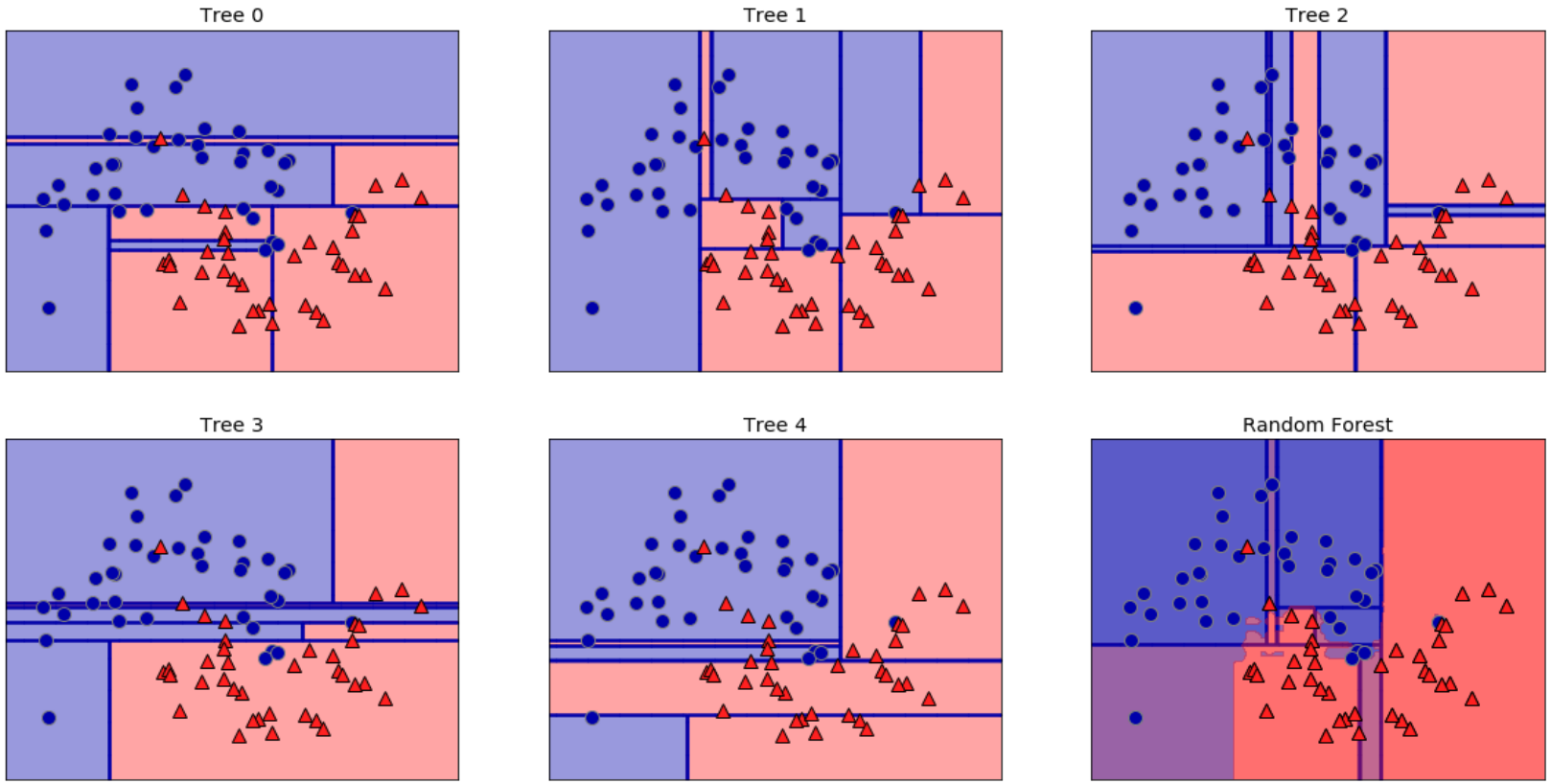
Bagging (Bootstrap Aggregating)

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

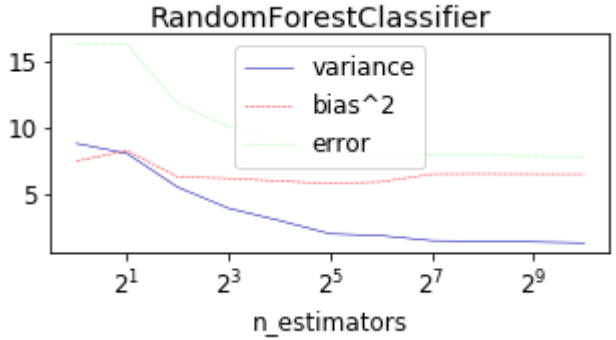
Making predictions:

- 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

- See exact definition of bias and variance in Lecture 3.
- 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)

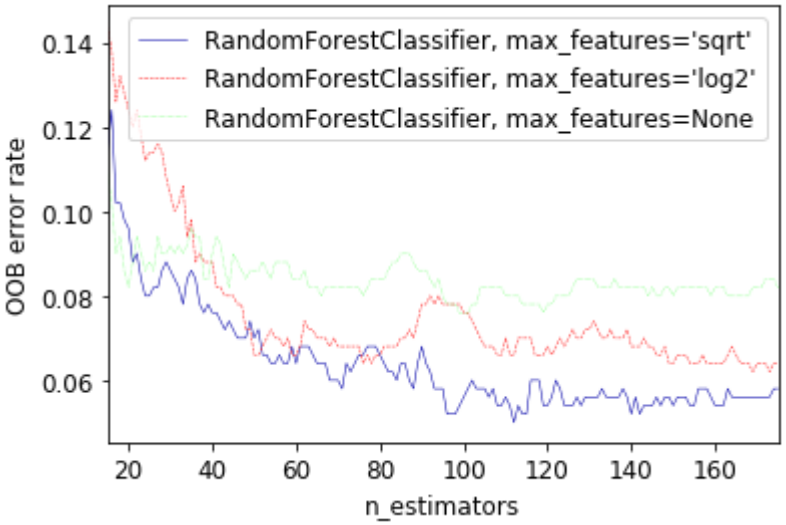
`n_jobs` sets the number of parallel cores to run
`random_state` should be fixed for reproducibility

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

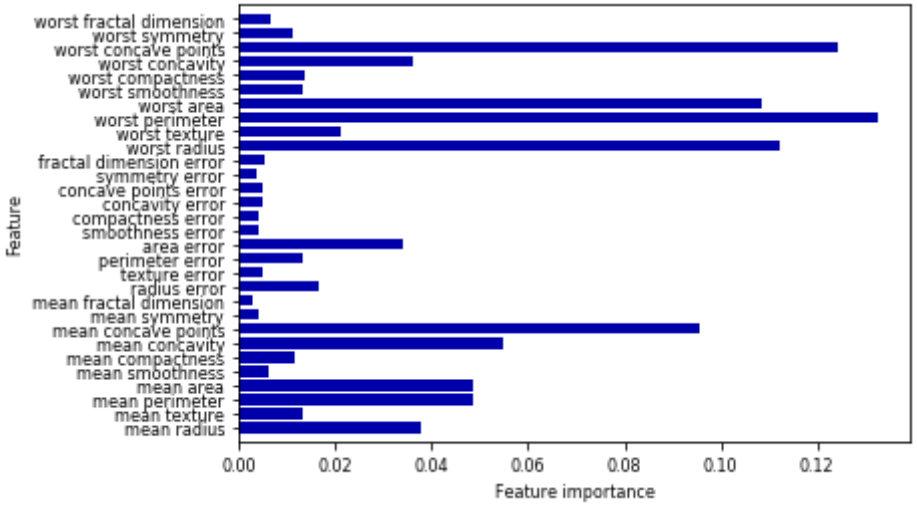
In scikit-learn OOB error are returned as follows:

```
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 either:
    - Passing on the weight to the loss (e.g. weighted Gini index)
    - Sample data with probability = sample weights
        - Misclassified samples are sampled multiple times so they get a higher weight
- Do weighted vote over all models

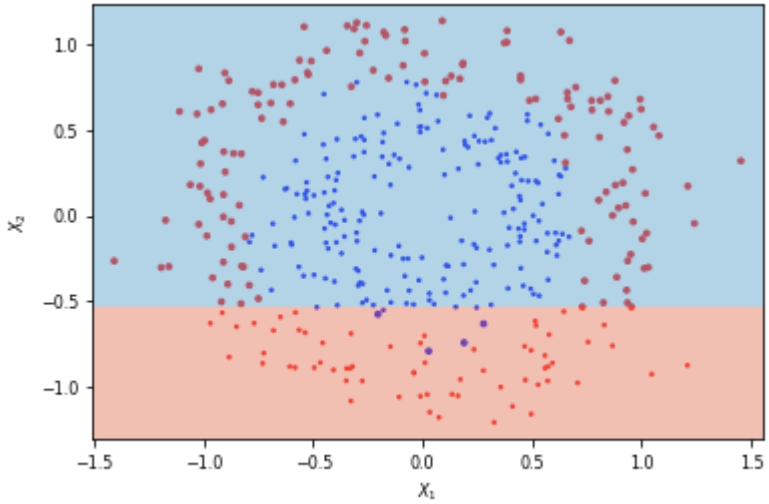
AdaBoost algorithm

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

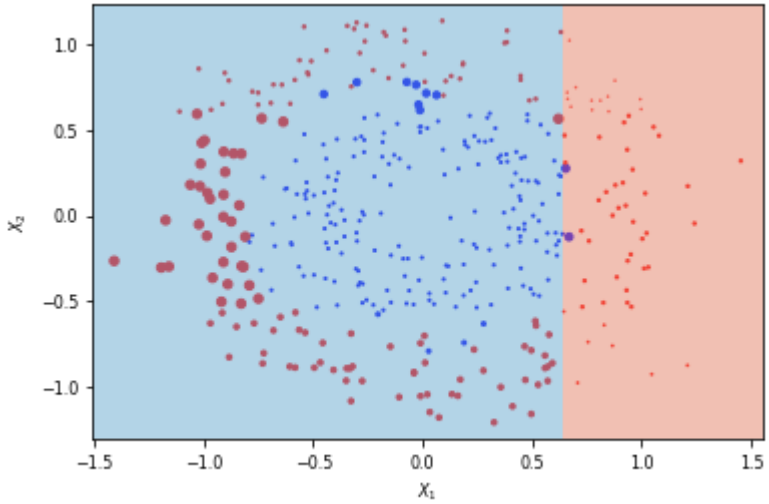
$$w_i = \lambda \log(\frac{1-E}{E})$$

  - Good trees get more weight than bad trees
  - Error is mapped from [0,Inf] to [-1,1], use small minimum error to avoid infinities
  - Learning rate λ (shrinkage) decreases impact of individual classifiers
    - Small updates are often better but requires more iterations
- 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

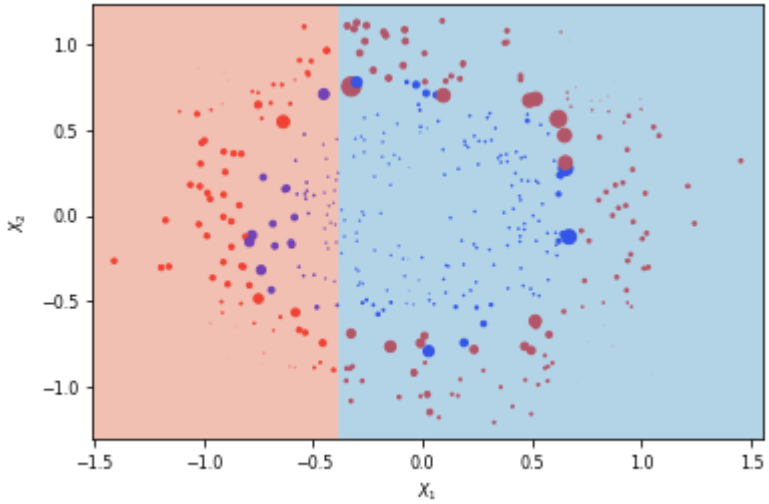
Base model 1, error: 0.35, weight: 0.56



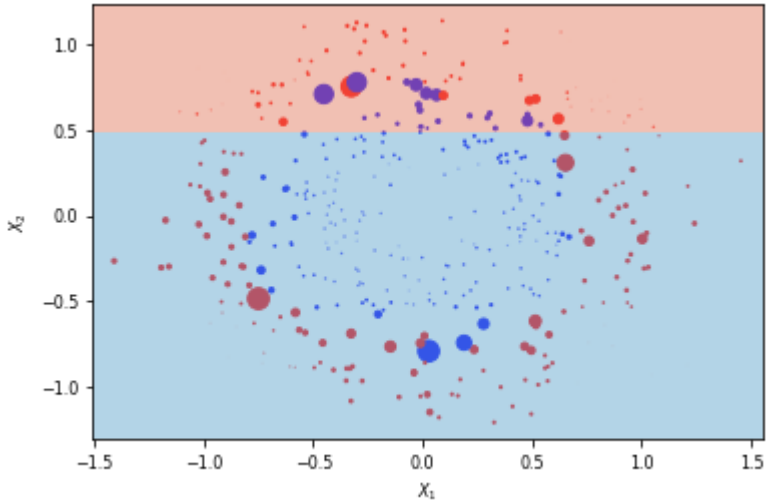
Base model 5, error: 0.21, weight: 1.19

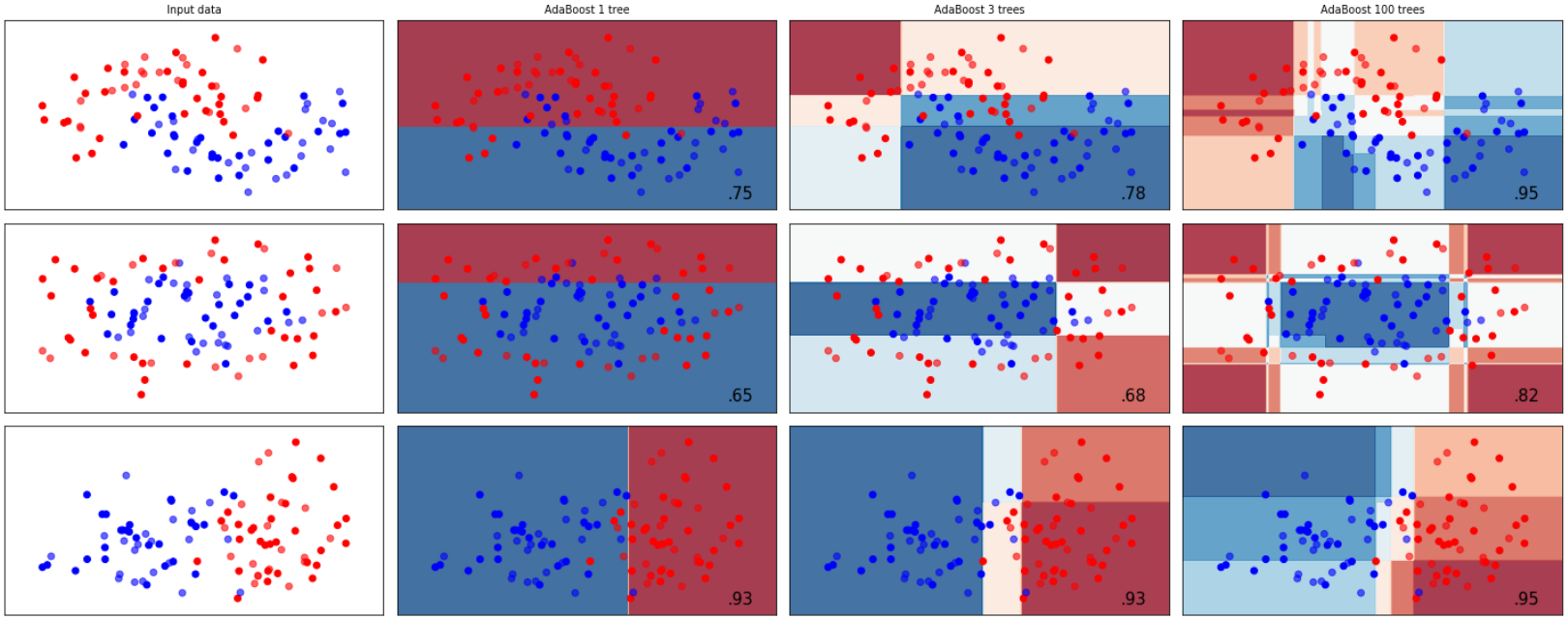


```
Base model 38, error: 0.35, weight: 0.56
```



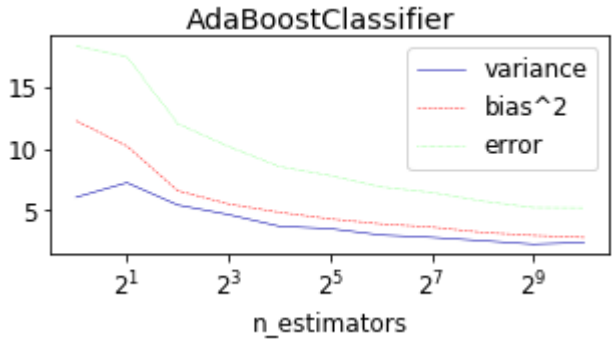

```
Base model 55, error: 0.31, weight: 0.70
```




Input data
AdaBoost 1 tree
AdaBoost 3 trees
AdaBoost 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
  - Classification: weighted vote over all base models
  - Regression:

$$y = \sum_{i=1}^{N} w_i tree_i(X)$$

- 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 (Regression)

- 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 (but requires more iterations)
- 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 (Regression)

- Dataset of n points $D = \{(x_i, y_i)\}_{i=1}^{n}$ where y_i is a numeric target
- Differentiable loss function $\mathcal{L}(y_i, F(x))$. Typically $\mathcal{L} = \frac{1}{2}(y_i - \hat{y}_i)^2$
  - $\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2 * \frac{1}{2}(y_i - \hat{y}_i) * (-1) = \hat{y}_i - y_i$
- Initialize model with constant output value $F_0(x) = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} \mathcal{L}(y_i, \gamma)$
  - Compute $\frac{\partial \mathcal{L}}{\partial \gamma} = 0$. For $\mathcal{L} = \frac{1}{2}(y_i - \hat{y}_i)^2$, $F_0(x)$ is the average of y_i
- 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 leaf j, optimize output $\gamma_{jm} = \underset{\gamma}{\text{argmin}} \sum_{x_i \in R_{ij}} \mathcal{L}(y_i, F_{m-1}(x_i) + \gamma)$
  - For $\mathcal{L} = \frac{1}{2}(y_i - \hat{y}_i)^2$, the optimum is the mean of all values in the leaf.
  - Update $F_m(x) = F_{m-1}(x) + \lambda \sum_{j=1}^{J_m} \gamma_m I(x \in R_{jm}))$

```
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)
```

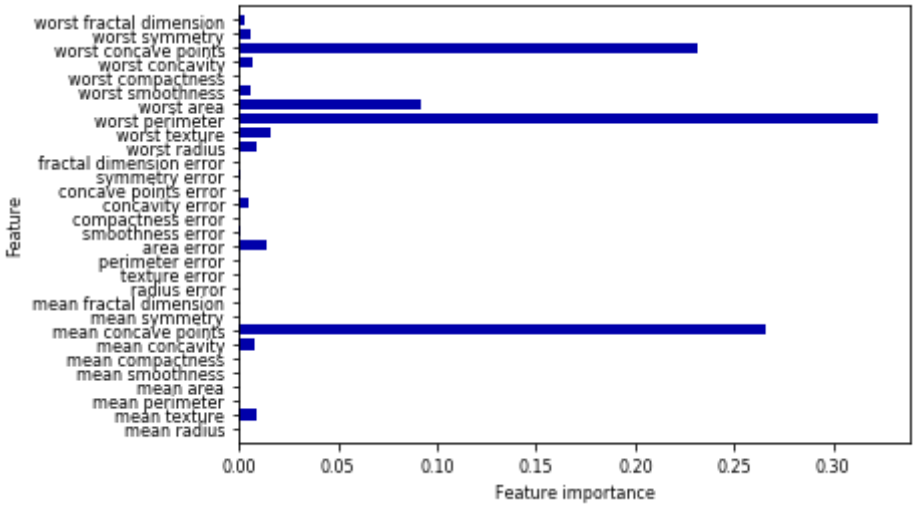
```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

```
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
- `n_iter_no_change`: early stopping: algorithm stops if improvement is less than a certain tolerance `tol` for more than `n_iter_no_change` iterations.

XGBoost

XGBoost is another python library for gradient boosting
Install separately, `conda install -c conda-forge xgboost`

- The main difference lies the use of approximation techniques to make it faster.
    - About 5x faster *per core*. Thus 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)

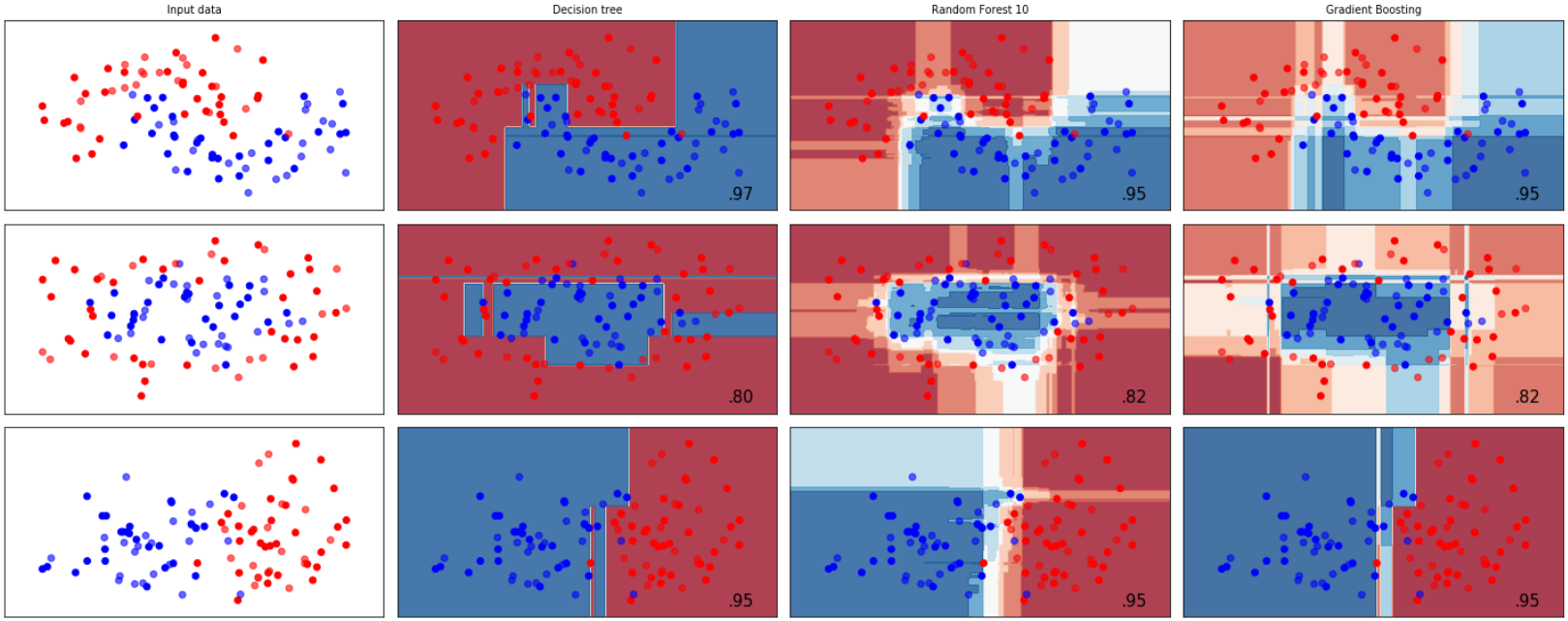
Further reading: XGBoost Documentation (https://xgboost.readthedocs.io/en/latest/parameter.html#parameters-for-tree-booster) Paper (http://arxiv.org/abs/1603.02754)

LightGBM

Another fast boosting technique

- Uses *gradient-based sampling*:
  - use all instances with large gradients (e.g. 10% largest)
  - randomly sample instances with small gradients, ignore the rest
  - intuition: samples with small gradients are already well-trained.
  - requires adapted information gain criterion
- Does smarter encoding of categorical features

Comparison


Input data
Decision tree
Random Forest 10
Gradient Boosting
.97
.95
.95
.80
.82
.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,...
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. Slower to train, much faster to predict.
    - Smaller models, typically more accurate than RandomForests.
- 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