UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

3253 Machine Learning

Module 7 : Decision Trees and Ensemble Learning

Course Plan

Module Titles
Module 1 – Introduction to Machine Learning
Module 2 – End to End Machine Learning Project
Module 3 – Classification
Module 4 – Clustering and Unsupervised Learning
Module 5 – Training Models and Feature Selection
Module 6 – Support Vector Machines
Module 7 – **Current Focus: Decision Trees and Ensemble Learning**
Module 8 – Dimensionality Reduction
Module 9 – Introduction to TensorFlow
Module 10 – Introduction to Deep Learning and Deep Neural Networks
Module 11 – Distributing TensorFlow, CNNs and RNNs
Module 12 – Final Assignment and Presentations (no content)

Learning Outcomes for this Module

- Use Decision Trees for classification and regression
- Select hyper-parameters
- Learn how to avoid overfitting
- Use Random Forests
- Employ various forms of Ensemble Learning for improving models

Topics for this Module

- **7.1** Decision trees
- **7.2** Learning a decision tree
- **7.3** Regularization
- **7.4** Regression trees
- **7.5** Ensembles
- **7.6** Voting
- **7.7** Bagging
- **7.8** Random forests
- **7.9** Boosting
- **7.10** Adaptive boosting
- **7.11** Gradient boosting
- **7.12** Stacking
- **7.13** Resources and Wrap-up

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 7 – Section 1

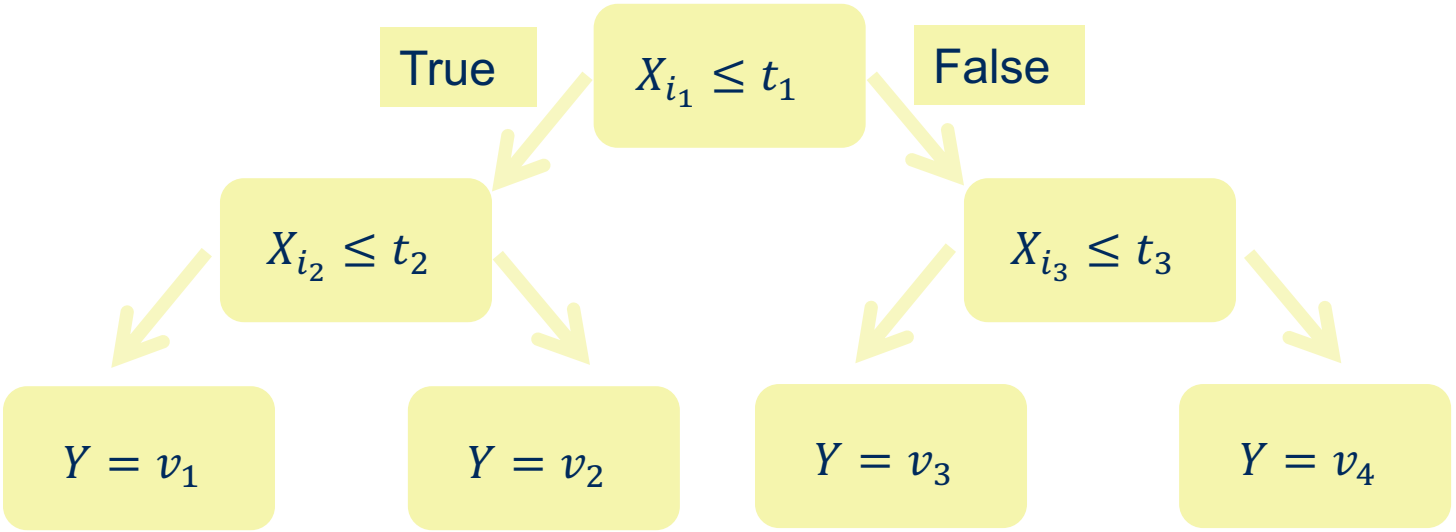
Decision Trees

Decision Trees

- ML algorithms for both regression and classification
- Capable of learning complex patterns
- Could yield high variance models; over-fit
- Have hyper-parameters for regularization
- Decision Trees are the building blocks of Random Forests, and Boosting Trees algorithms

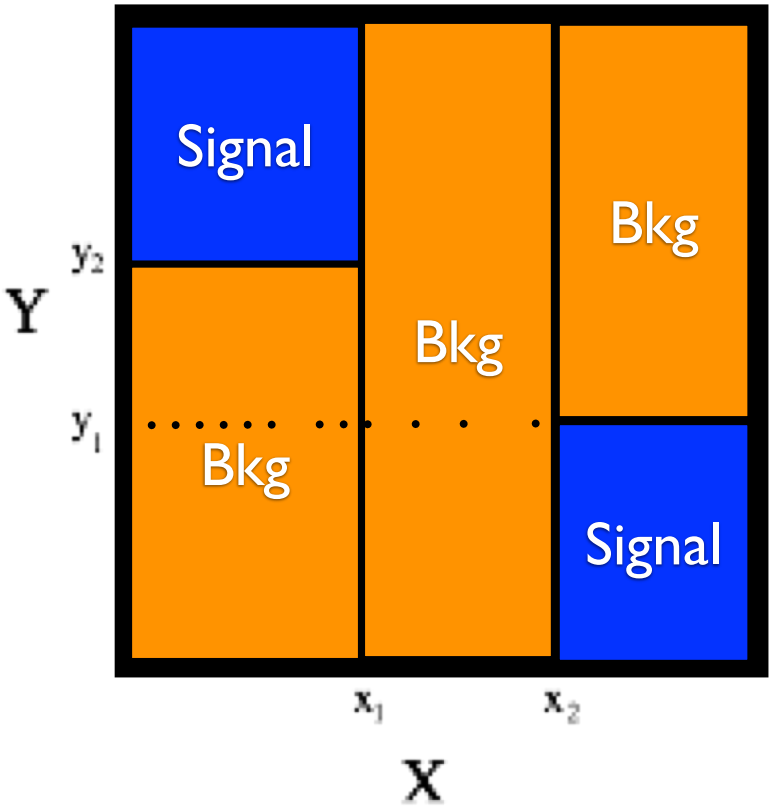
Decision Trees – Classification

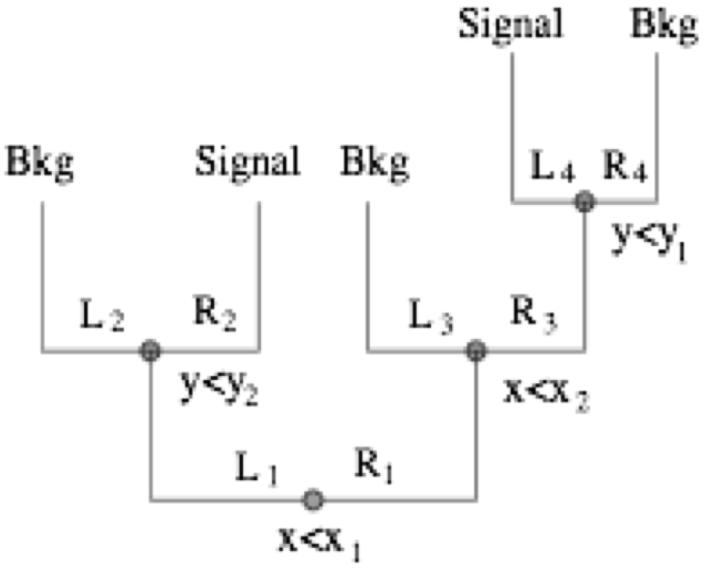
- The predictor is a set of rules (conditional statements) represented by tree



- The choice of X_{i_j} and values t_j, v_j are learned from the data

Toy Decision Tree



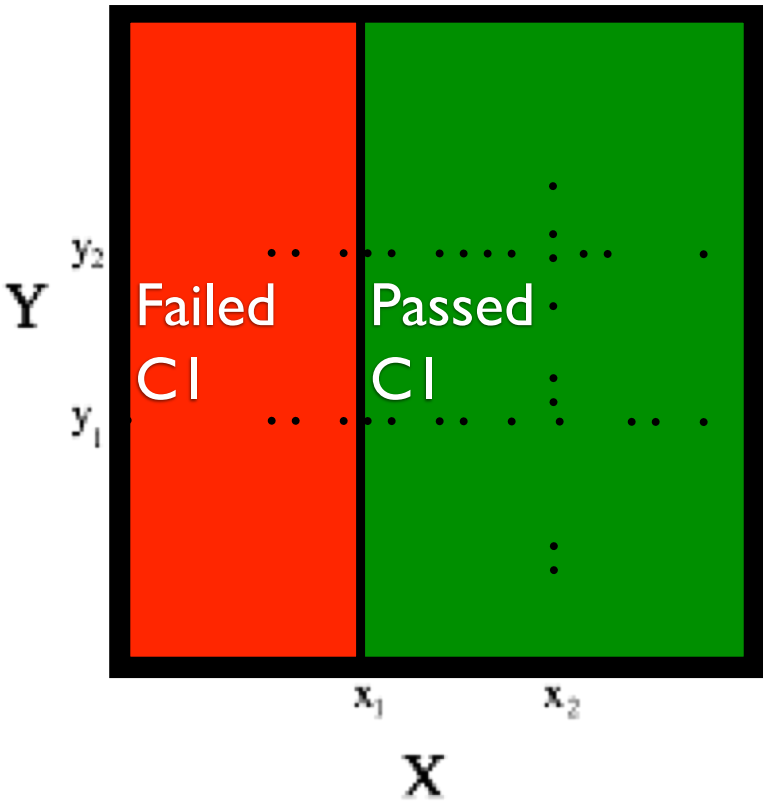


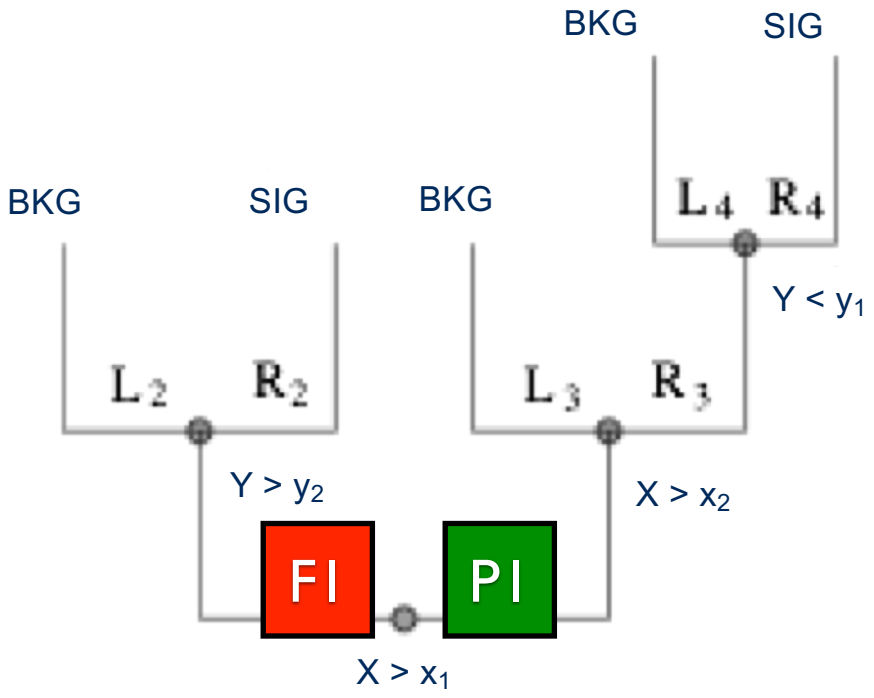
Task: separate signal from background

Issue: A single cut on X or Y does not work!

Solution: Use a series of consecutive cuts, generating a tree structure

Toy Decision Tree (cont'd)

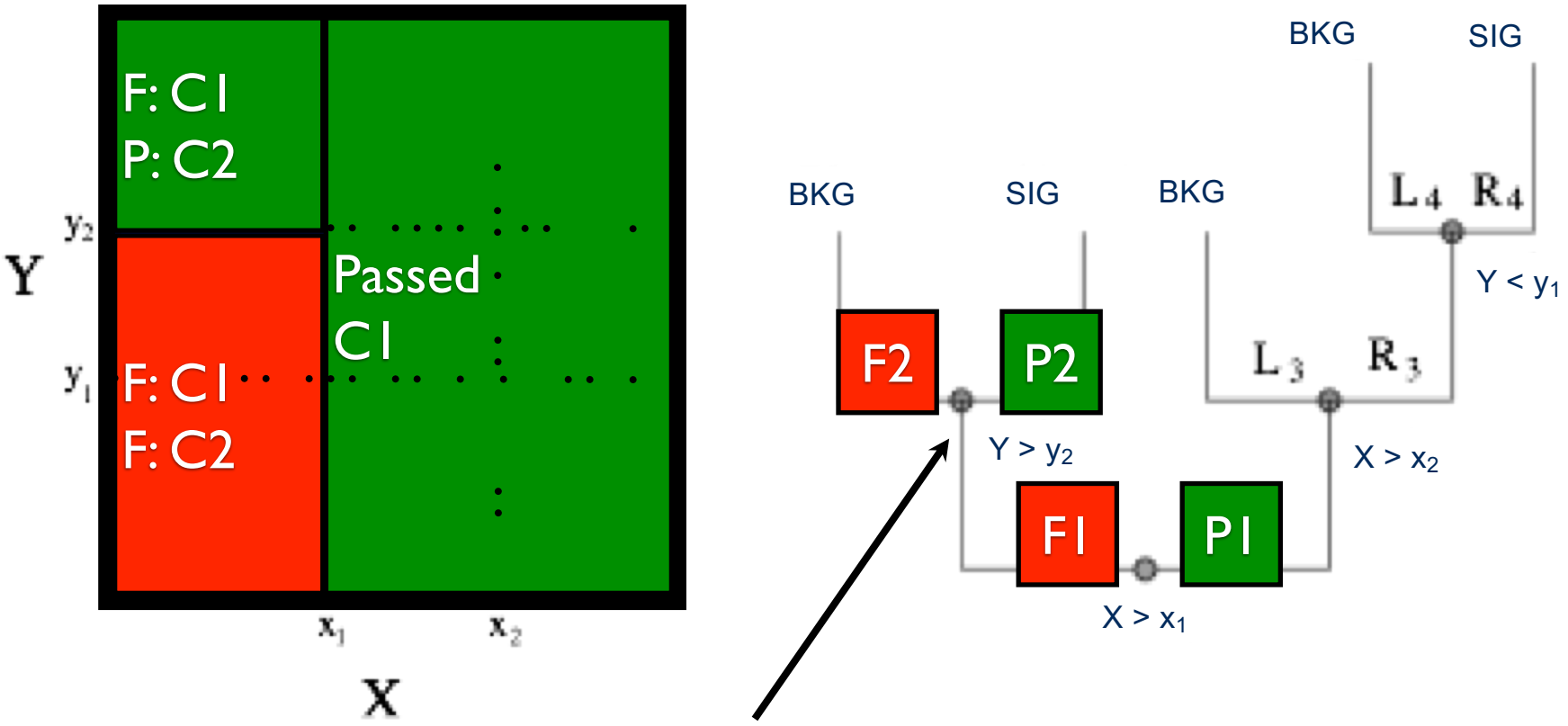




Cut 1: on the X variable

9

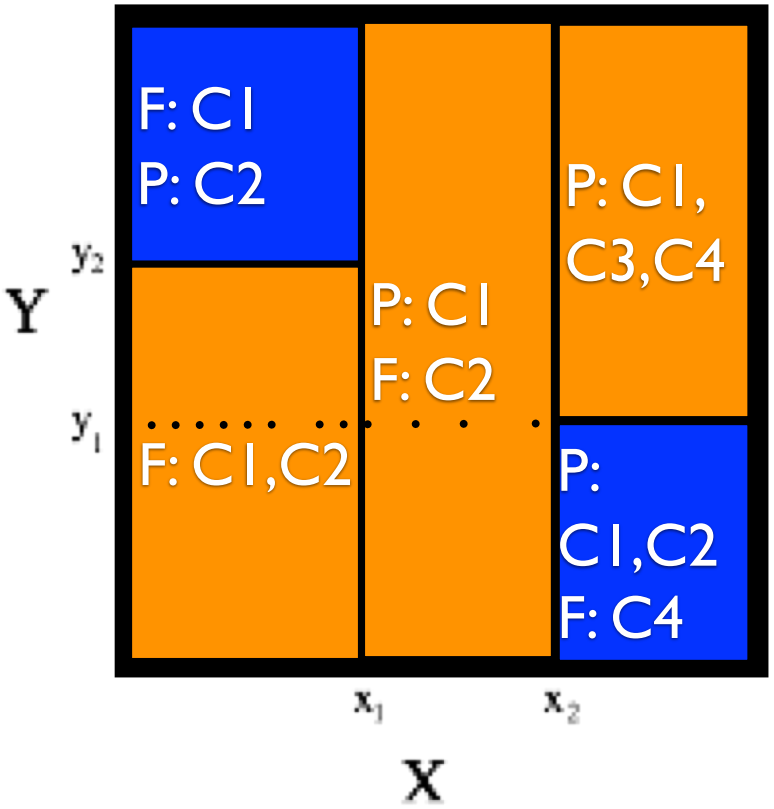
Toy Decision Tree (cont'd)

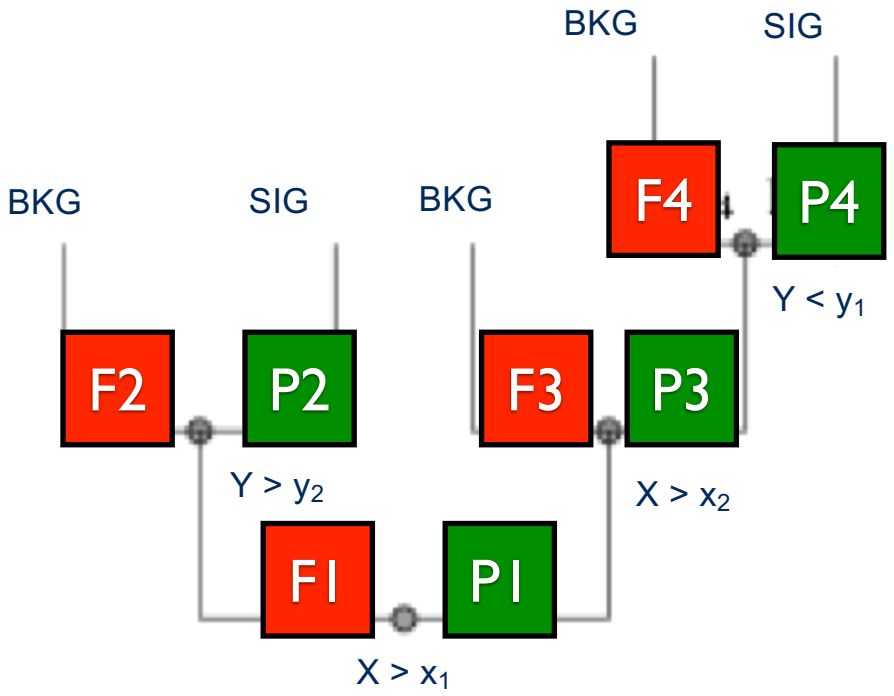


Cut 2: Recovered events that failed the cut 1

Add more cuts...

Toy Decision Tree (cont'd)





After 4 cuts: Signal and Background regions are separated! Done!

Toy model: only 2 variables, easy to determine cut values

Decision Trees – Classification

- The tree shows how the predictions are made
- At each node a feature is compared to a threshold $X_{i_j} \leq t_j$
- The algorithm determines the pair (X_{i_j}, t_j) for each node
- If $X_{i_j} \leq t_j$ is true, and the node is not a leaf, the example is sent to the left child-node, otherwise to the right child-node
- This continues until a leaf is reached

Decision Trees – Classification (cont'd)

Each node in the tree has additional attributes

$X_{i_1} \leq t_1$
Samples: 350
Value: [300,50]
Gini: 0.25

- *samples*: number of training examples that passed through the node
- *value*: number of training examples of each class that passed through the node
- *Gini*: impurity of the node

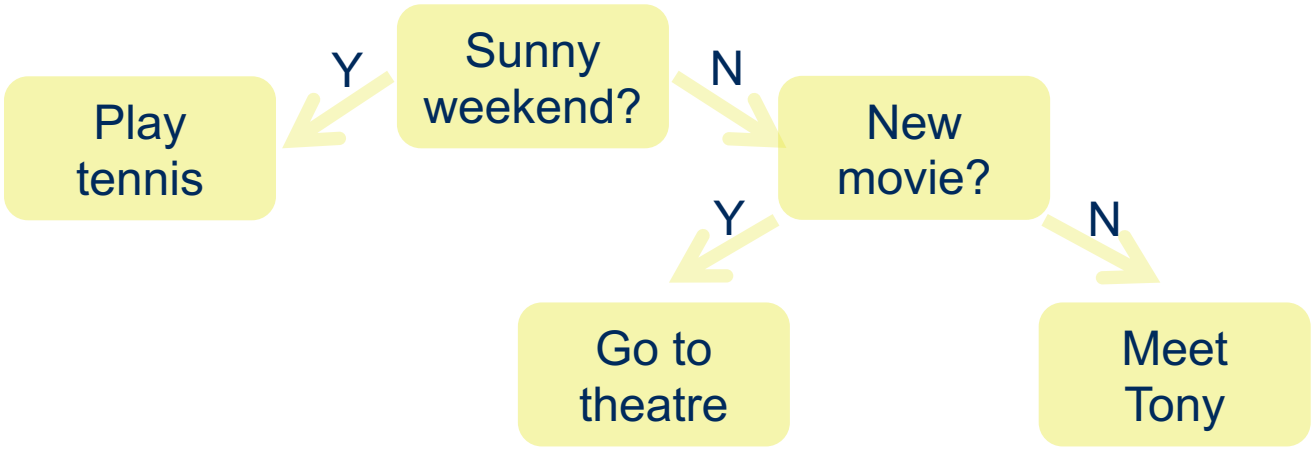
Decision Trees – Classification (cont'd)

- $Gini = 0$, purest, all examples in node are from the same class
- $Gini = 1 - \frac{1}{c}$, most impure, all classes represented equally (in the node)
- $Gini = 1 - \sum_{k=1}^{c} p_k^2$

  Sum over the categories in the labels
  p_k is the fraction of examples in the node and category k

Decision Trees – Classification (cont'd)

- Decision Trees are interpretable
- Each branch, root to leaf, is a conjunction:
  $(X_{i_1} \leq t_1)\&(X_{i_2} > t_2)\& \ldots \&(X_{i_d} \leq t_d)$
- Decision making often looks like this:
  - "*If the weekend is sunny, I'll play tennis, but if not, and there is a good movie, I'll watch it. Otherwise, I'll meet Tony*"



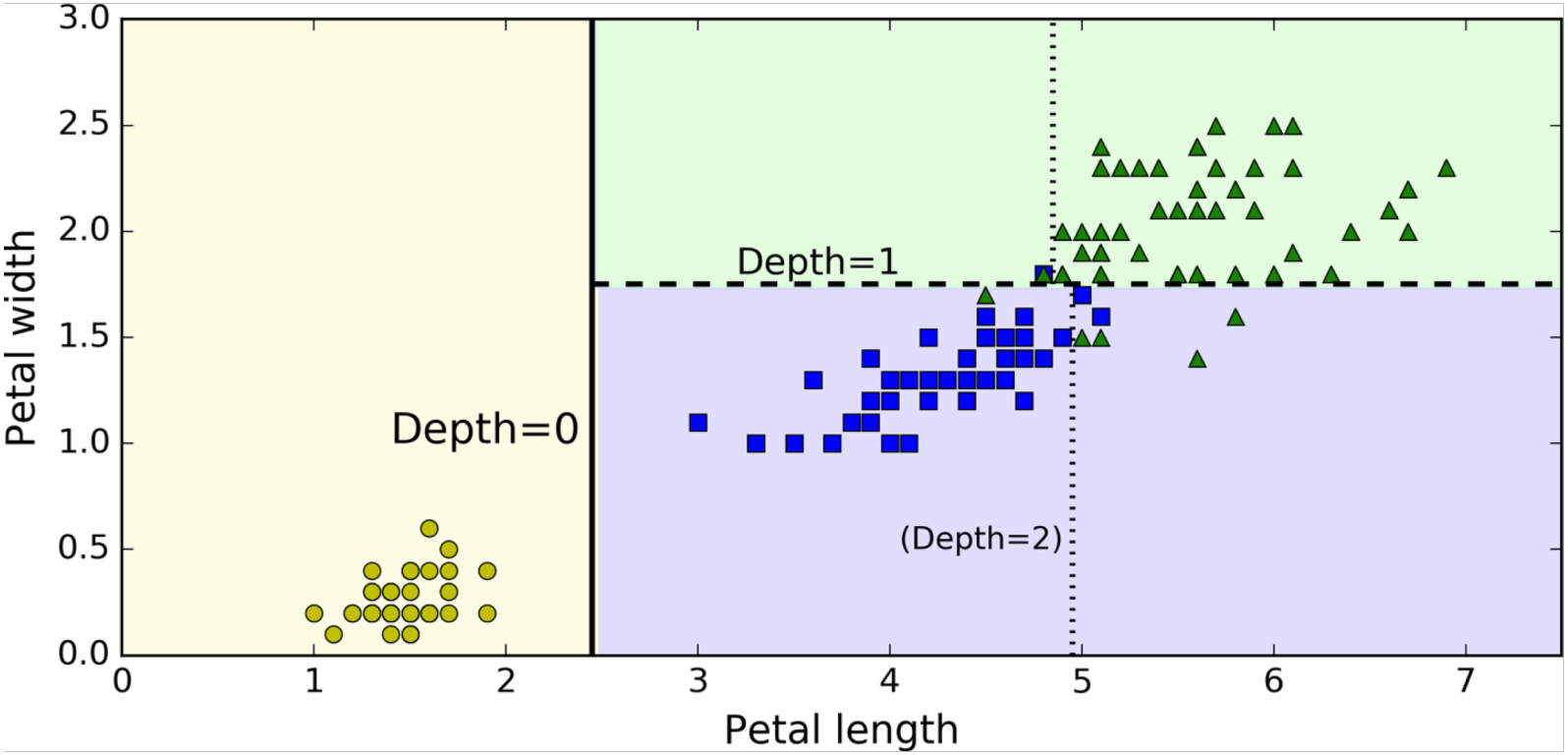
Decision Trees – Classification (cont'd)

- A classification tree estimates probabilities
- If an example x lands on a leaf having $value = [35,90]$, then:

$$P[x\ belongs\ to\ category\ 1] = \frac{35}{125} = 0.28$$

$$P[x\ belongs\ to\ category\ 2] = \frac{90}{125} = 0.72$$

Decision Trees – Classification (cont'd)



Module 7 – Section 2

Learning a Decision Tree

Learning a Decision Tree

- The algorithm that learns the tree is recurrent

- Input: a set S of labelled training examples

- Algorithm:
  - 0) Compute attribute values for root-node
  - 1) If STOP condition not True:
    - Determine optimal split by X_i, t_i
    - Split S:

$$S_{left} = \{x \in S : X_i(x) \leq t_i\}$$

$$S_{right} = \{x \in S : X_i(x) > t_i\}$$

  - 2) Repeat with S_{left} and S_{right}

Learning a Decision Tree (cont'd)

- At a given node in the tree that has S training examples, the optimal split is the one that minimizes the weighted average of the impurity of the children-nodes

- $Cost(k,t) = \frac{m_{left}}{m} Gini_{left} + \frac{m_{right}}{m} Gini_{right}$
- k is the index of the feature, t is the threshold
- $S_{left} = \{x \in S: x_k \leq t\}$, $S_{right} = \{x \in S: x_k > t\}$
- $m_{left} = |S_{left}|$, $m_{right} = |S_{right}|$
- G_{left}: impurity of S_{left} (similarly with G_{right})

Learning a Decision Tree (cont'd)

- Stopping criteria: at every leaf, no split could create a node with even lower impurity

- Stronger stopping conditions:
  - the tree has reached a maximum depth
  - the number of leaves reached a maximum
  - new splits would create a leaf with less than the minimum required number of examples

Learning a Decision Tree (cont'd)

- The learning algorithm is *greedy:* find the best split at each iteration
- Not necessarily gives the optimal tree
- Finding the optimal tree, among all possible trees, is computationally too expensive
- The decision tree found is usually good enough
- The decision tree is found efficiently

Learning a Decision Tree (cont'd)

- Alternative cost function: uses Entropy

- $Cost(k,t) = \frac{m_{left}}{m} H_{left} + \frac{m_{right}}{m} H_{right}$

- H_{left}: entropy on left child (same for H_{right})
- Entropy on a node: $H = -\sum_{k=1}^{c} p_k \log(p_k)$
- p_k: fraction of examples in category k
- H ranges between 0 (all examples in a same category), and $\log(c)$ (examples uniformly distributed across the classes)
- Note: $0 \times \log(0) = 0$

Module 7 – Section 3

Regularization

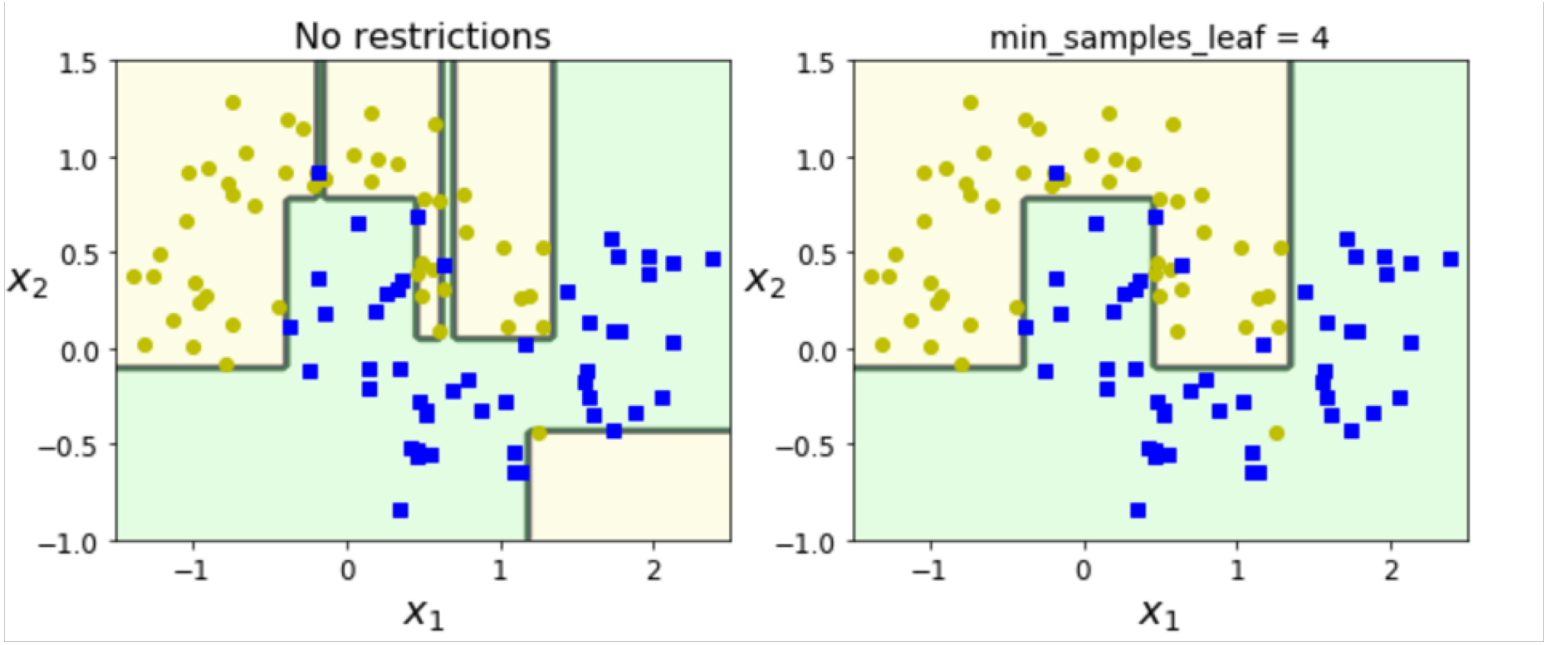
Regularization

- Decision Trees can fit data and its noise well
- To avoid over-fitting, hyper-parameter tuning is necessary (regularization)
- Regularization parameters (scikit-learn)
  - max_depth: upper bound on nodes from root to leaf
  - min_samples_split: minimum required examples in a node to make a new split
  - min_weight_fraction_leaf: same but expressed as a fraction of the total number of examples
  - max_leaf_nodes: maximum number of leaves of the learned tree

These are related, it is not necessary to calibrate all of them

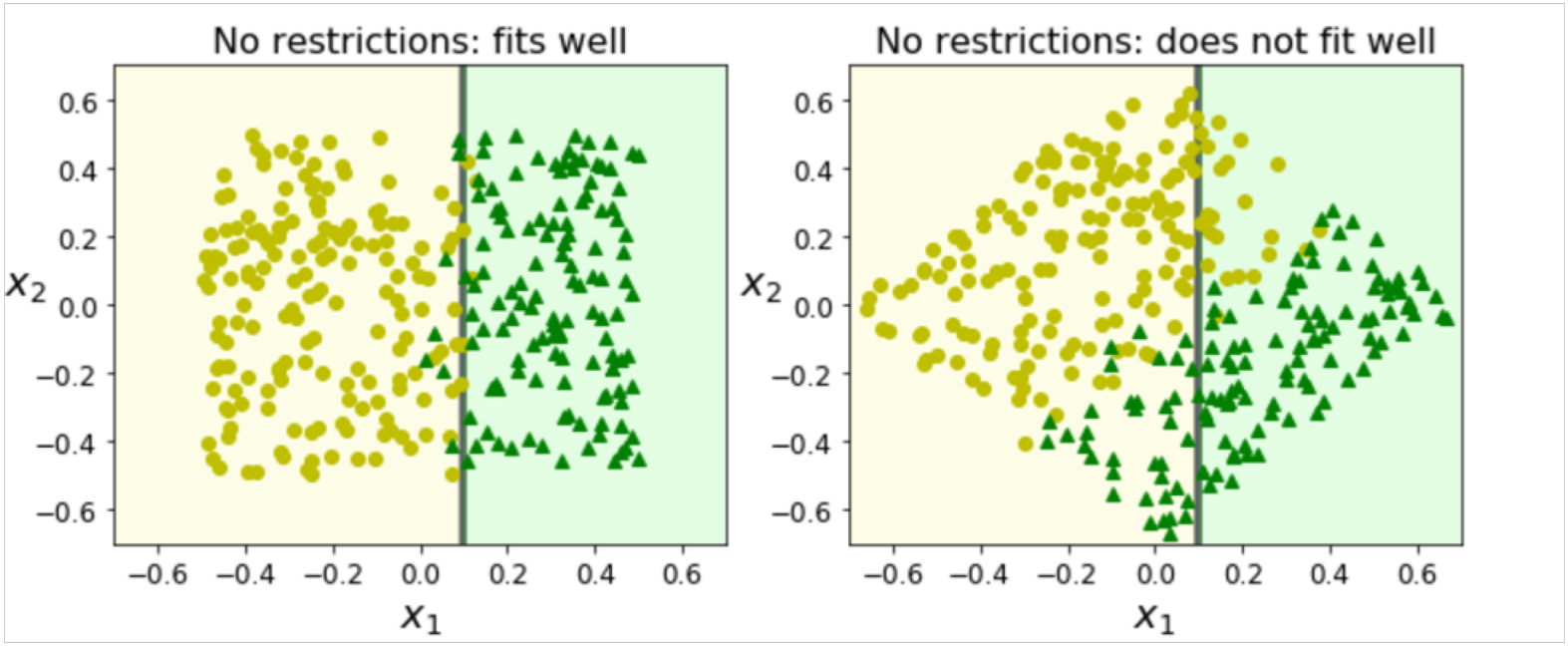
Regularization (cont'd)

- Performance is highly sensitive to hyper-parameter values
- Example: no regularization causes over-fitting



Regularization (cont'd)

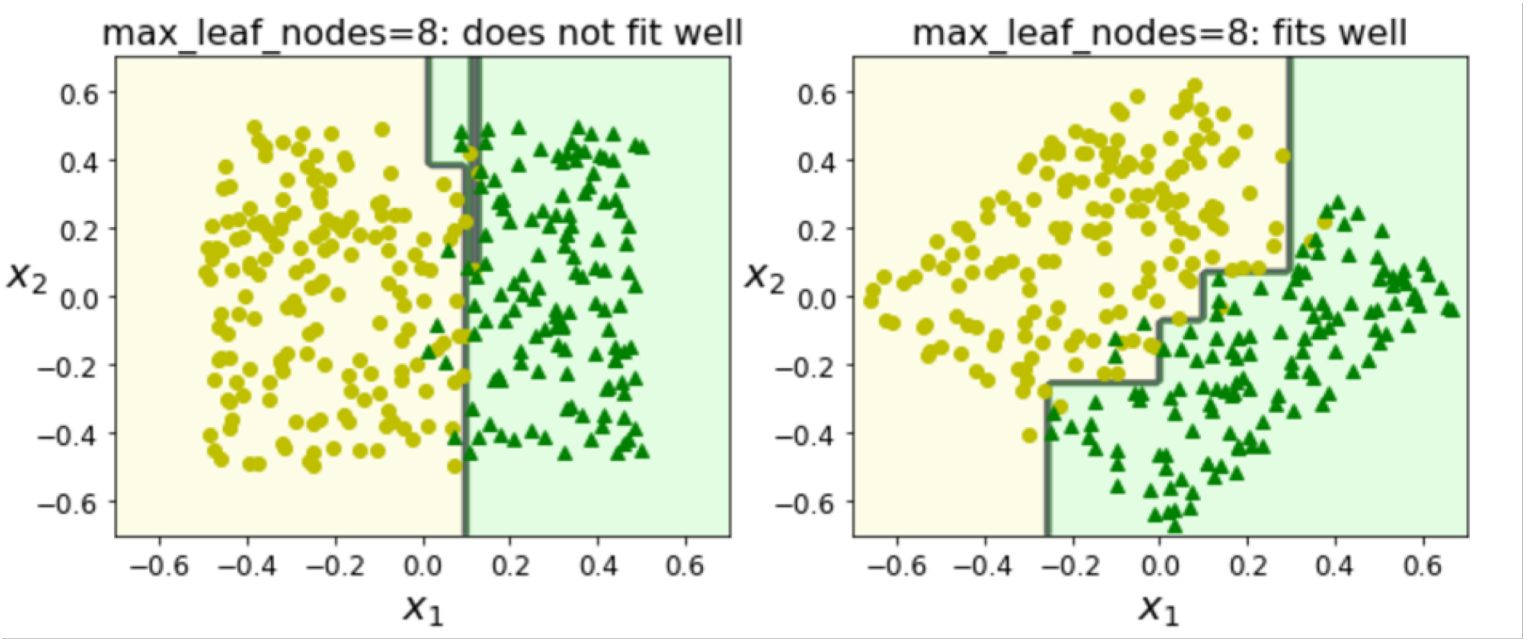
- A linear transformation of the data has impact on the correct value for hyper-parameters



- Same data, rotated 45 degrees

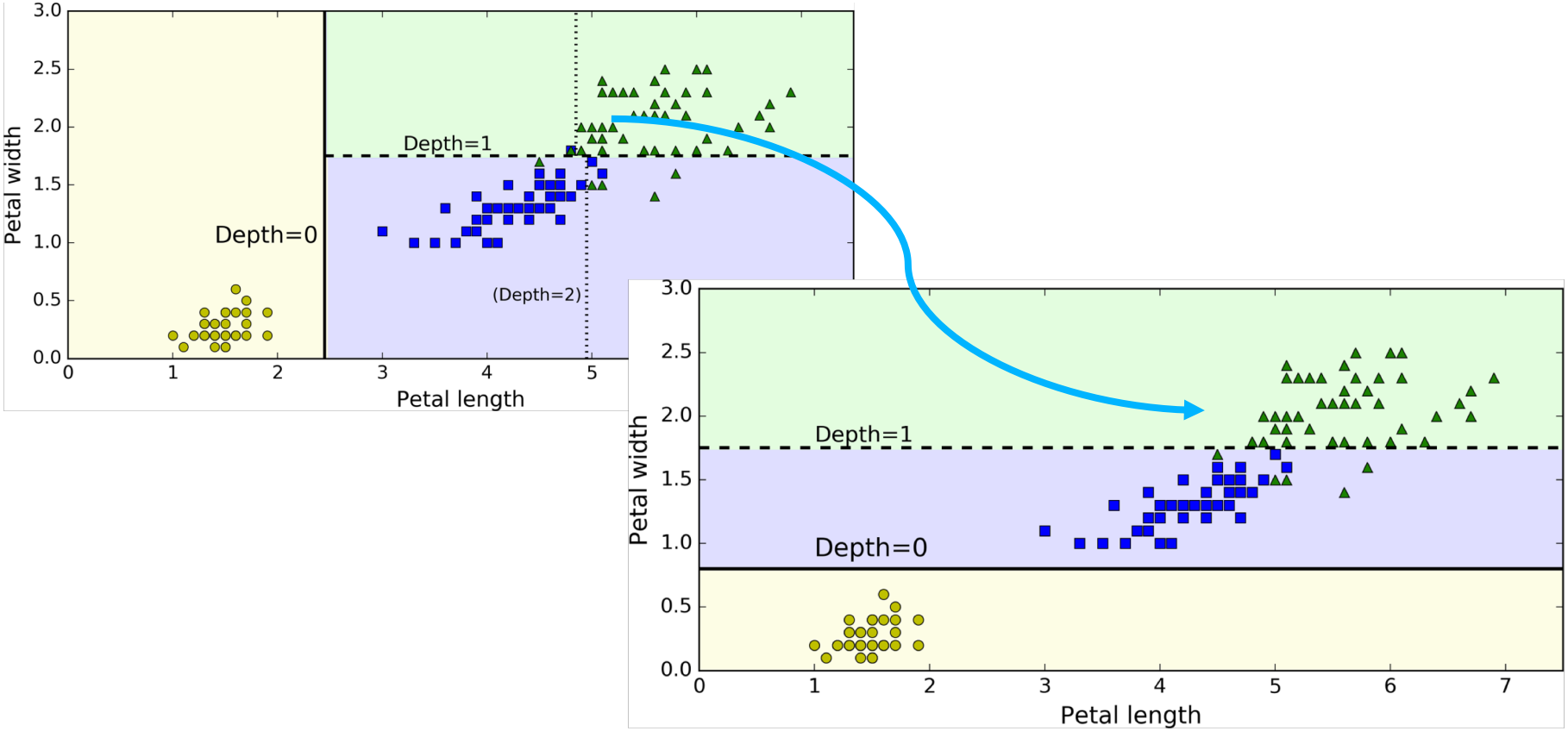
Regularization (cont'd)

- A linear transformation of the data has impact on the correct value for hyper-parameters



Instability

- A small change in the data can lead to completely different boundaries



Module 7 – Section 4

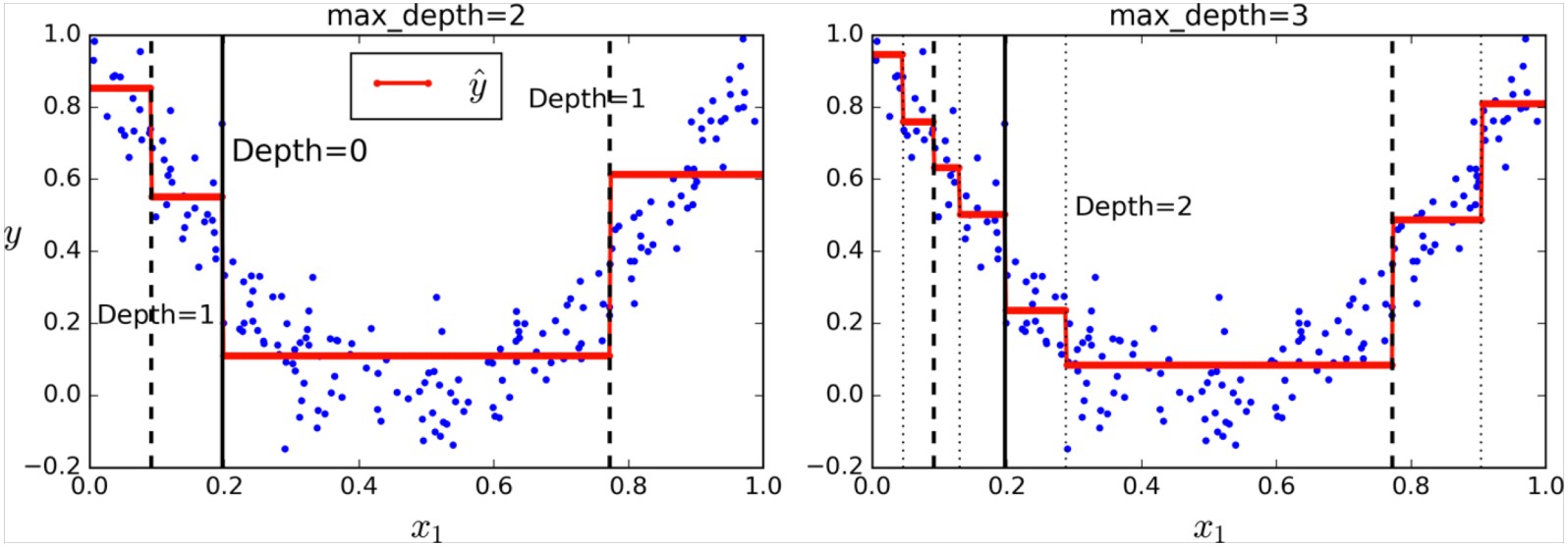
Regression Trees

Decision Tree for Regression

- The data examples are labelled with real values, instead of categories
- Most remains the same, as in classification
- Two things change: *values* (attribute) and cost function
- Now the value is the average value of the examples in the node
- The cost is the mean square error (MSE)

$X_{i_1} \leq t_1$
Samples: 350
Value: 23.4
MSE: 3.2

Decision Tree for Regression (cont'd)



Decision Tree for Regression (cont'd)

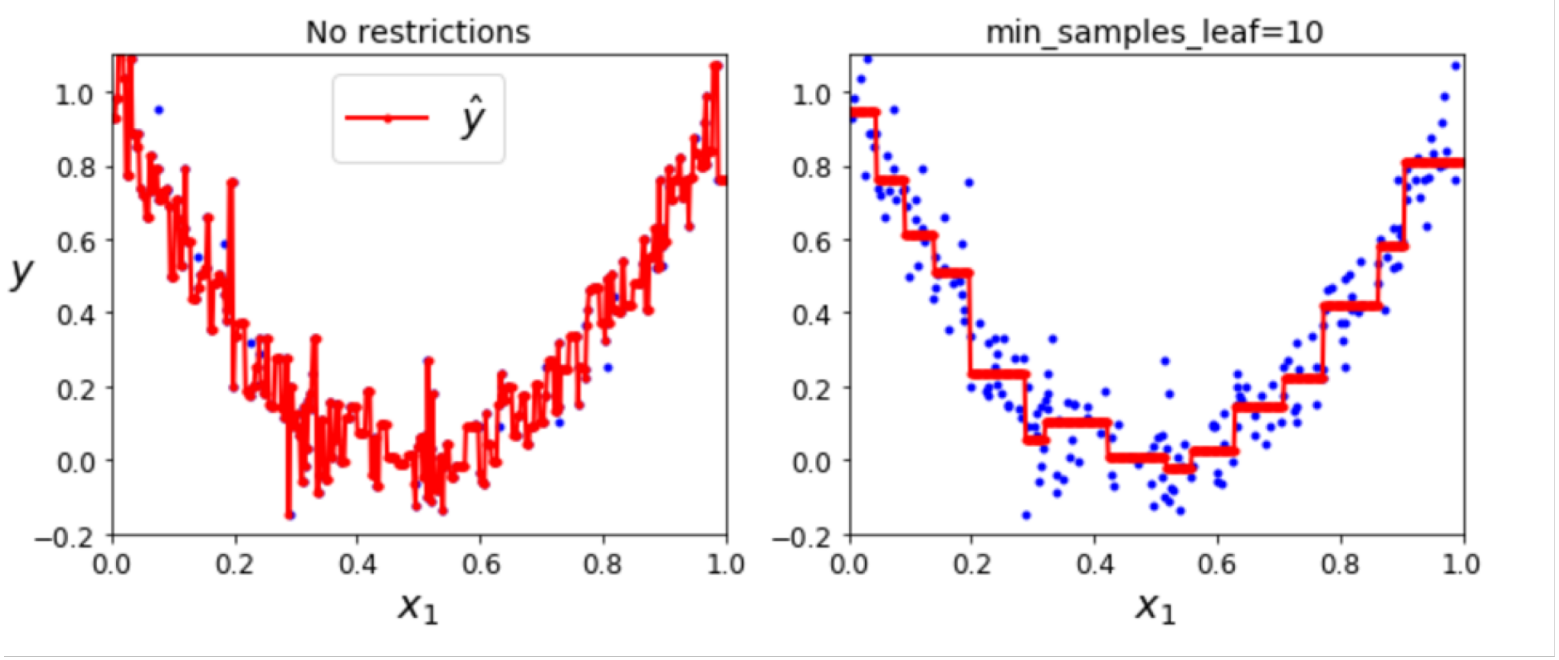
- The learning algorithm works exactly the same, but the optimal split now minimizes the new cost function based on MSE
- $Cost(k,t) = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right}$

$$MSE_{node} = \sum_{i:in\ node} (y_i - \hat{y}_{node})^2$$

where $\hat{y}_{node}$ is the average label on the node

Decision Tree for Regression (cont'd)

- Hyper-parameter tuning is also important
- Without proper values on hyper-parameters over-fitting occurs easily



Module 7 – Section 5

Ensembles

Wisdom of The Crowd

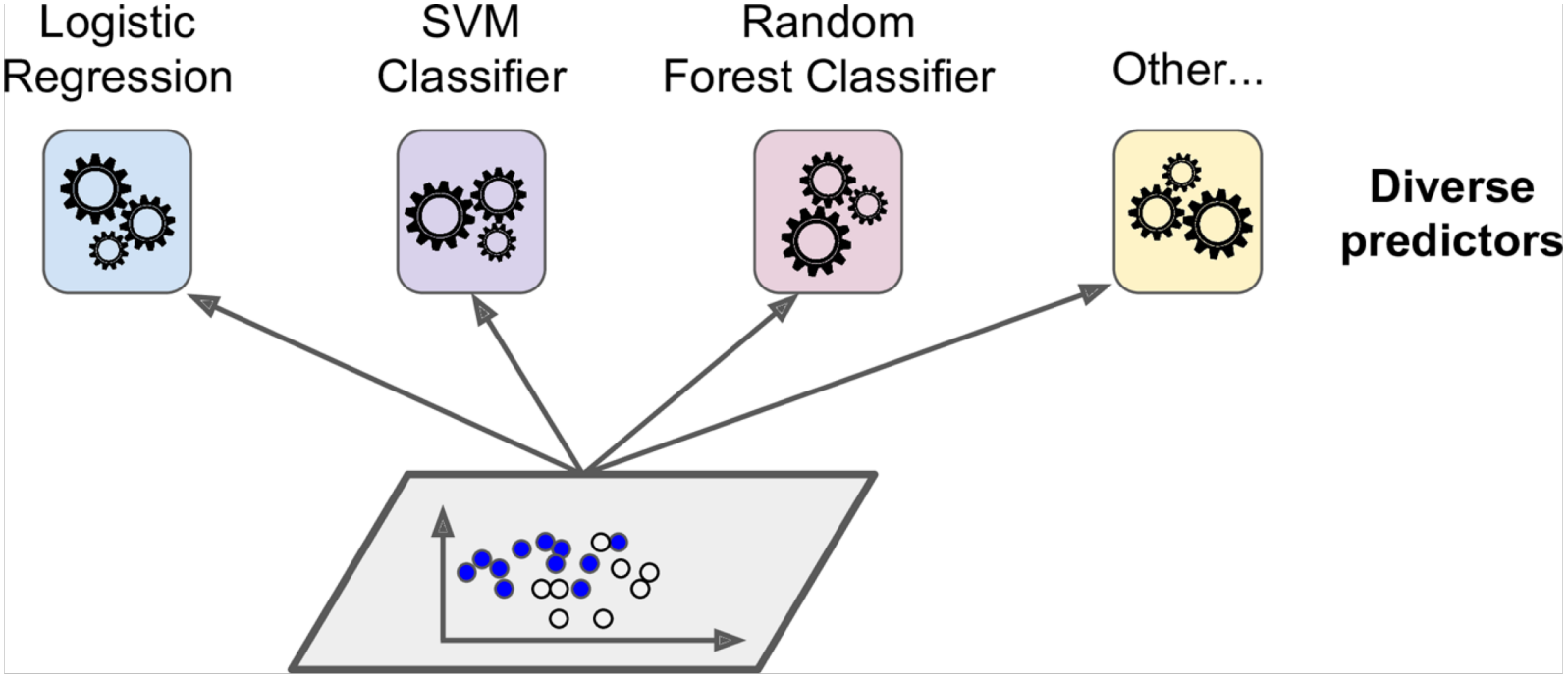
- Suppose you ask a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer.
- A group of predictors is called an ensemble; thus, this technique is called Ensemble Learning, and an Ensemble Learning algorithm is called an Ensemble method.
- A group of Decision Tree classifiers, each on a different random subset of the training set is called a Random Forest, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

Module 7 – Section 6

Voting

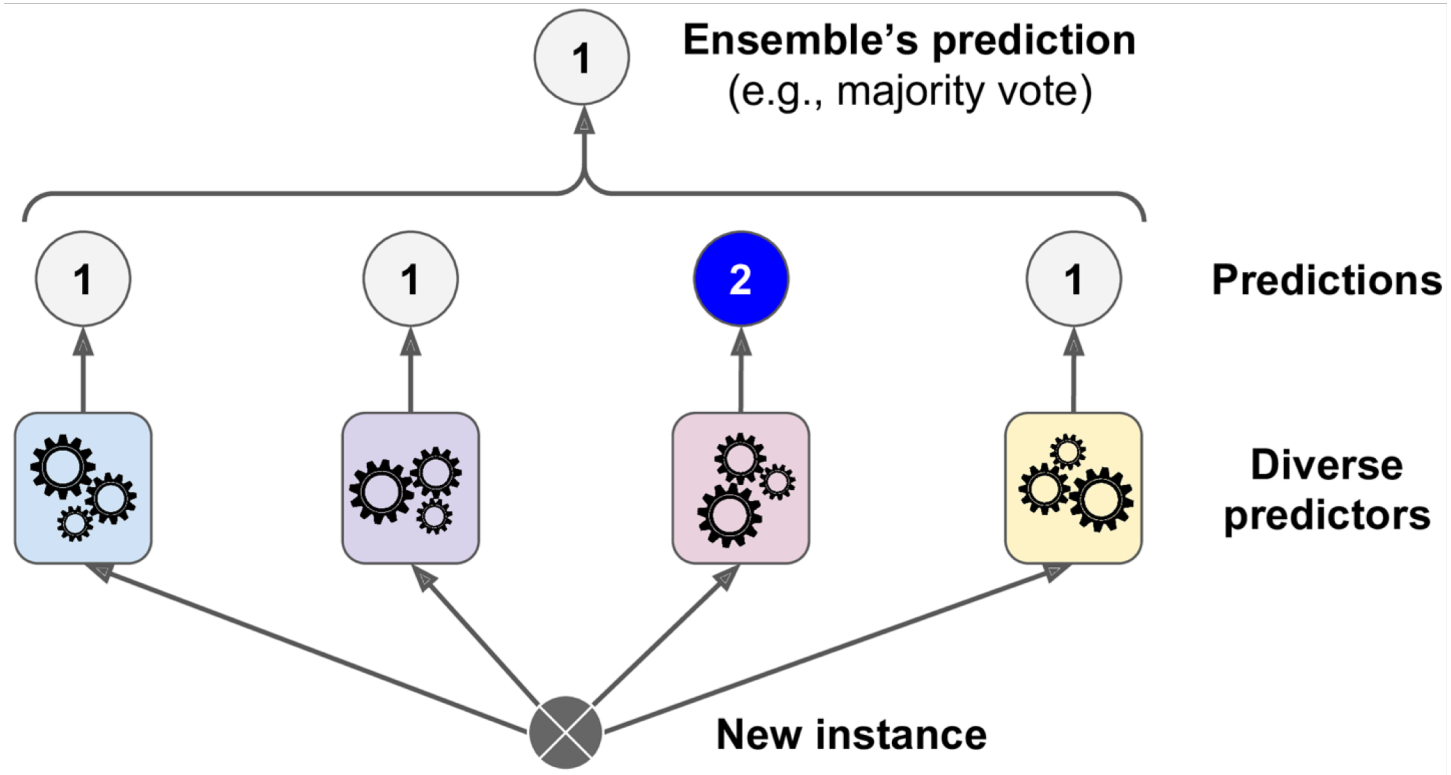
Voting Classifiers

- A way to create an combined classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes.



Voting Classifiers (cont'd)

- Even if each classifier is weak, the ensemble can still be a strong learner



Voting Classifiers (cont'd)

- Ensemble methods work best when the predictors are as independent from one another as possible

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Voting Classifiers

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

- If all classifiers are able to estimate class probabilities, then you can tell Scikit-Learn to predict the class with the highest class probability, This is called soft voting.

Module 7 – Section 7

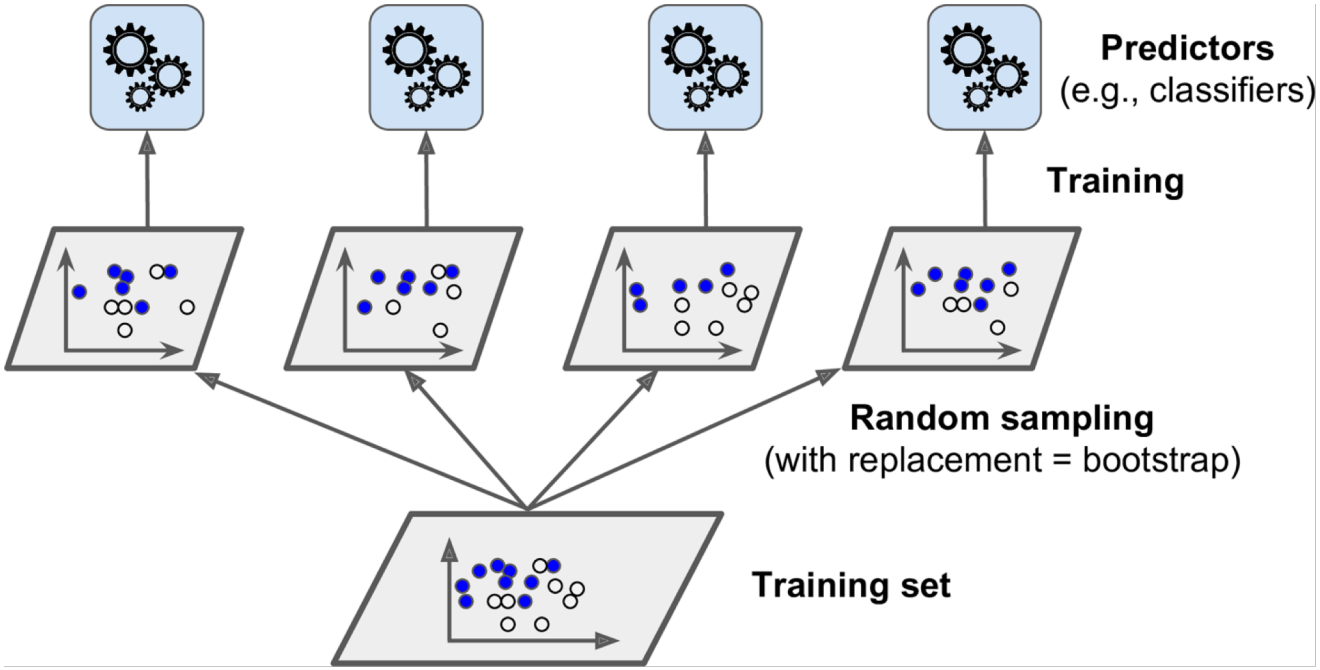
Bagging

Bagging and Pasting

- Another approach to combine classifiers is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set.
- When sampling is performed with replacement, this method is called bagging. When sampling is performed without replacement, it is called pasting.
- The ensemble can make a prediction for a new instance by aggregating the predictions of all predictors:
  - The aggregation function is typically the statistical mode for classification, or the average for regression.
  - Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.

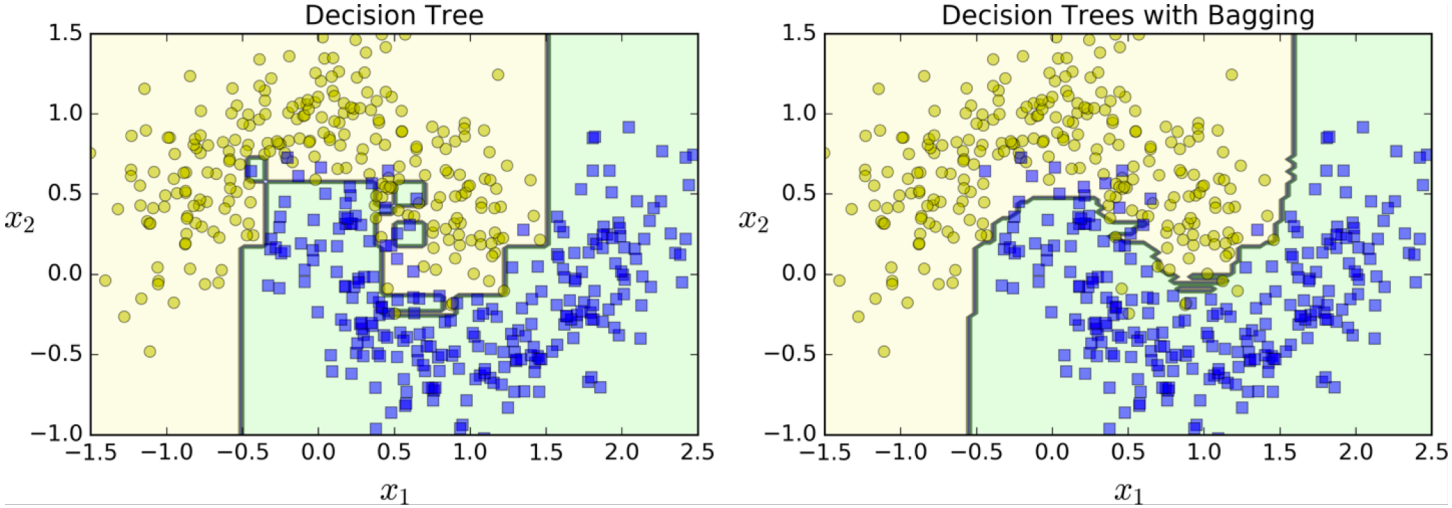
Bagging and Pasting (cont'd)

- Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.
- Predictors can be trained in parallel



Bagging and Pasting (cont'd)

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```



1 Tree vs. 500 Trees

Out-of-Bag Evaluation

- With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all.
  - Only a portion (p) of the training instances are sampled for each predictor.
  - The remaining (1-p) of the training instances that are not sampled are called out-of-bag (oob) instances.
- Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set or cross-validation.
- You can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.

Out-of-Bag Evaluation (cont'd)

```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.90133333333333332
```

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.91200000000000003
```

Module 7 – Section 8

Random Forests

Random Patches and Subspaces

- You can also sample the features used to build the classifer. This is particularly useful when you are dealing with high-dimensional inputs.
  - Sampling both training instances and features is called the Random Patches method.
  - Keeping all training instances (i.e. bootstrap=False and max_samples=1.0) but sampling features (i.e. bootstrap_features=True and/or max_features smaller than 1.0) is called the Random Subspaces method.

- Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forest (RF)

- Random Forest is an ensemble of Decision Trees, generally trained via the bagging method, typically with max_samples set to the training set.
  - A RandomForestClassifier has all the hyperparameters of a DecisionTreeClassifier (to control how trees are grown), plus all the hyperparameters of a BaggingClassifier to control the ensemble itself.e size of the training set.

- The RF algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features.

Random Forest (cont'd)

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

Extra Trees

- It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds.
- A forest of such extremely random trees is called an Extremely Randomized Trees ensemble
- Extra-Trees are much faster to train than regular Random Forests since finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

Feature Importance

- Random Forests can measure the relative importance of each feature.
- Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average:
  - It is a weighted average, where each node's weight is equal to the number of training samples that are associated with it
  - Scikit-Learn computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1.

Feature Importance (cont'd)

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Module 7 – Section 9

Boosting

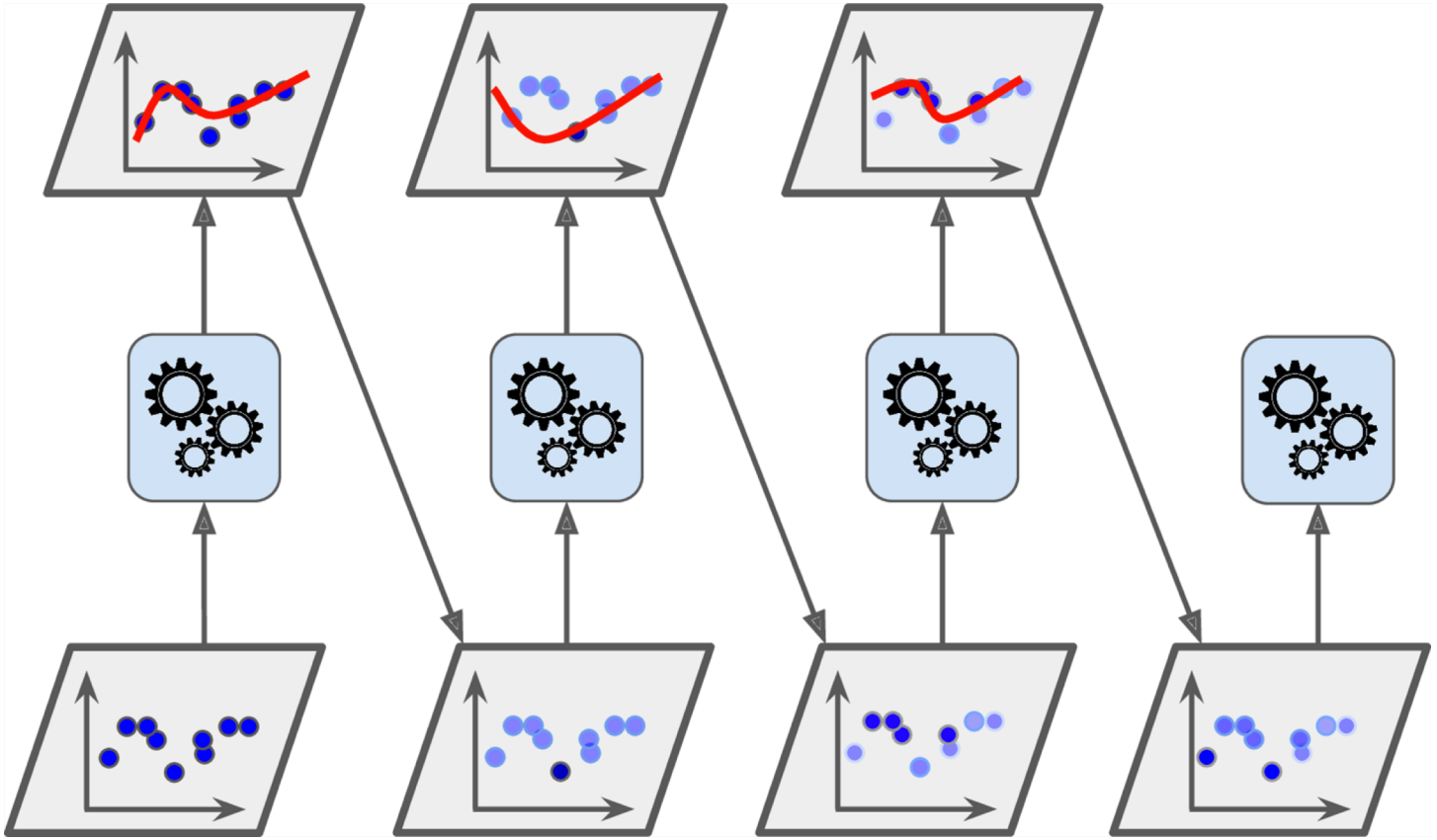
Boosting

- The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor.
  - AdaBoost (short for Adaptive Boosting):
    Pays more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases.
  - Gradient Boosting:
    Tries to fit the new predictor to the residual errors made by the previous predictor.

Module 7 – Section 10

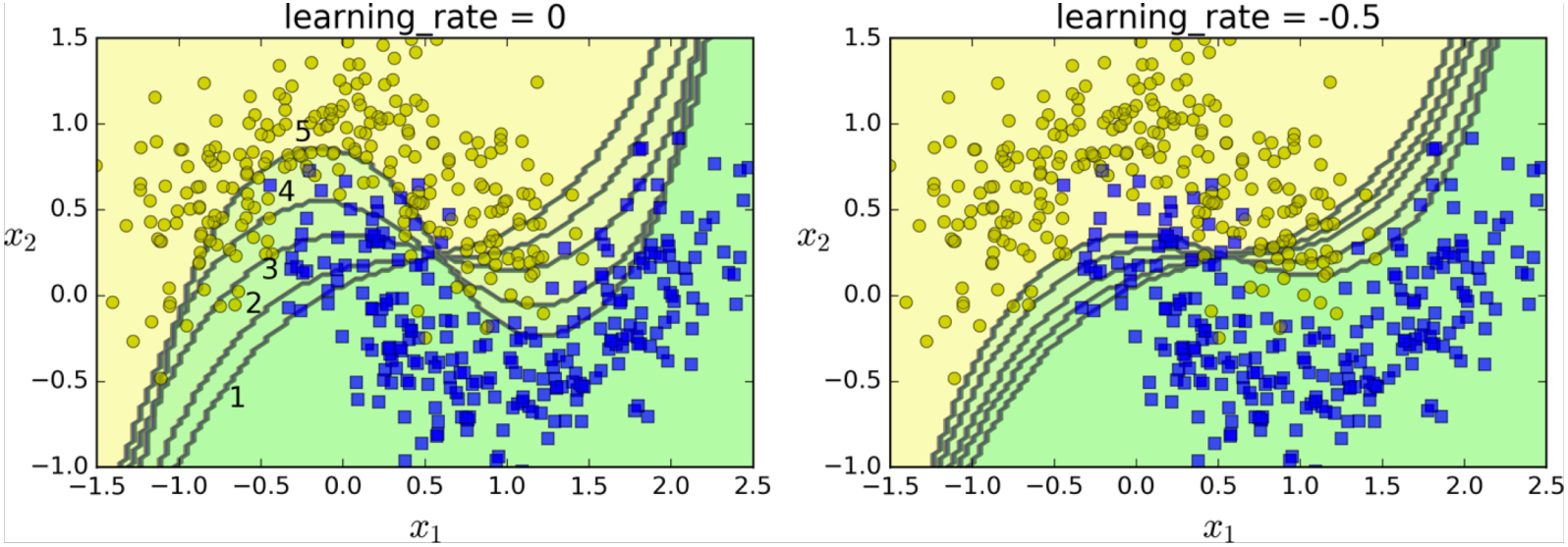
Adaptive Boosting

AdaBoost



AdaBoost

- Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.
- One drawback is that it cannot be parallelized



AdaBoost (cont'd)

- Each instance is weighted to *1/m*
- A first predictor is trained with weighted error rate

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance.}$$

- Predictor weight is computed. η is the learning rate

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

AdaBoost (cont'd)

- Weights are updated

$$\text{for } i = 1, 2, \cdots, m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

- All the instance weights are normalized. Divide by $\sum_{i=1}^{m} w^{(i)}$
- A new predictor is trained using the updated weights, and the whole process is repeated
- The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

AdaBoost (cont'd)

- Compute predictions of all the predictors weighted by α_j. The predicted class is the one that receives the majority of weighted votes

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x})=k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 7 – Section 11

Gradient Boosting

Gradient Boosting (GB)

- Following a serial setup, GB tries to fit the new predictor to the residual errors made by the previous predictor.

- Step 1:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

- Step 2:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

- Step 3:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Gradient Boosting (cont'd)

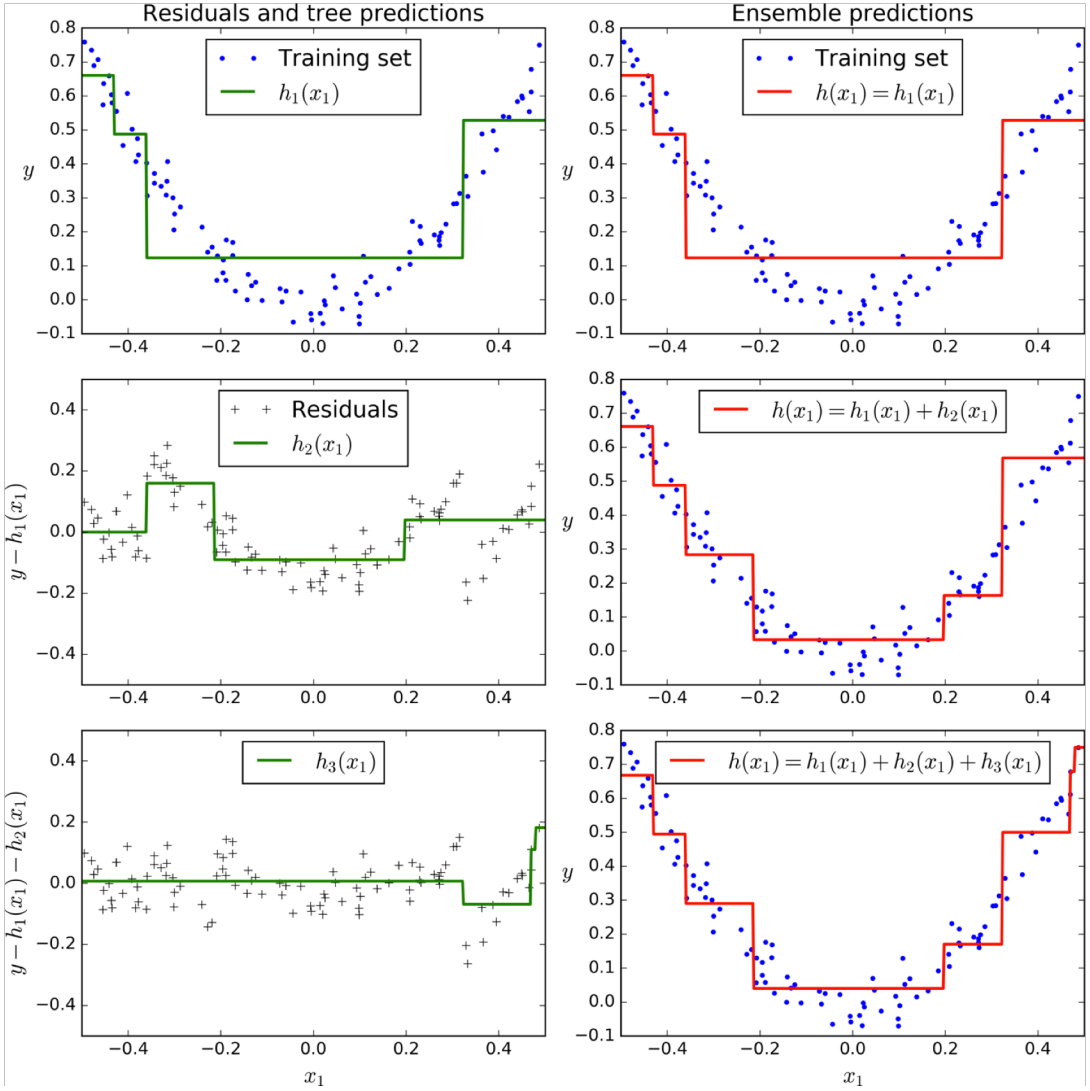
- Predict

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

- Or equivalently:

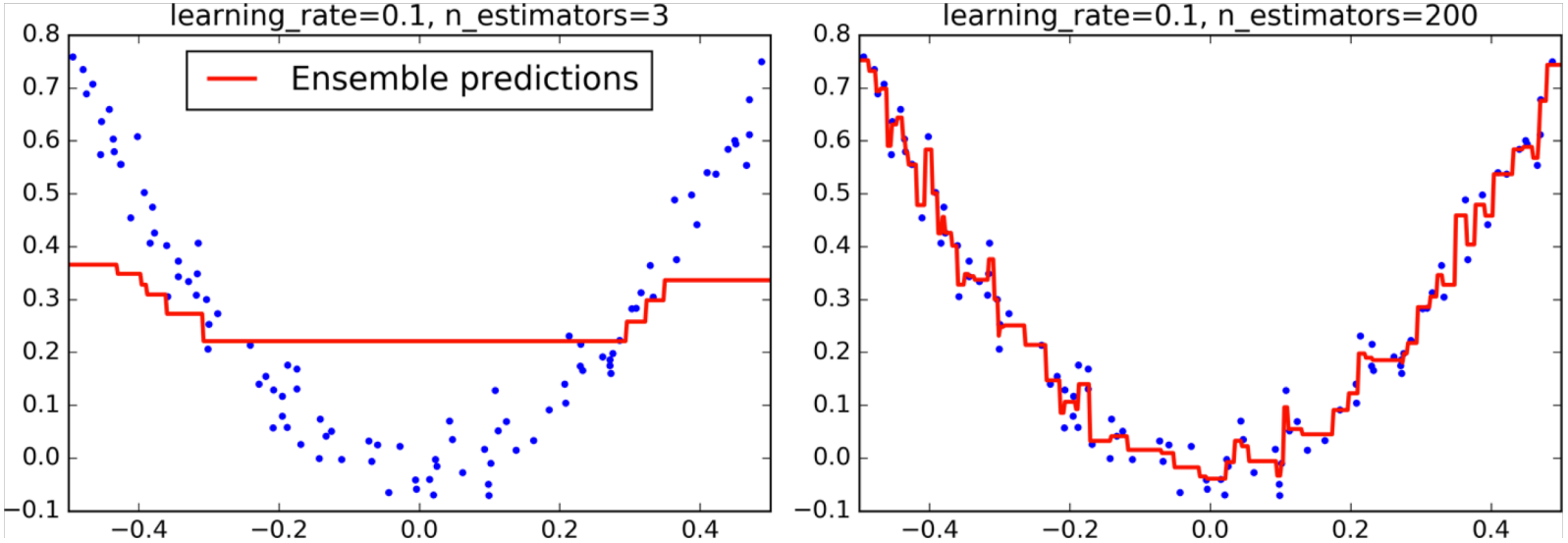
```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

Gradient Boosting (cont'd)



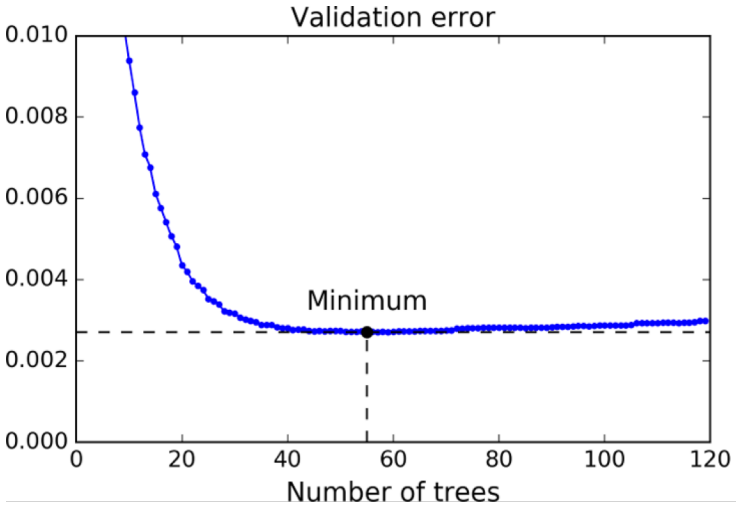
Gradient Boosting (cont'd)

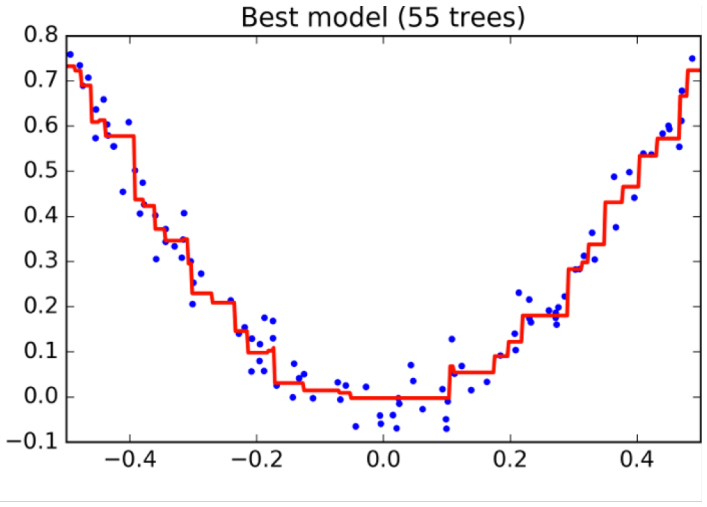
- The learning_rate hyperparameter scales the contribution of each tree.
  - A low value, such as 0.1, more trees are required in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called shrinkage.



GB Early Stopping

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)

gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```



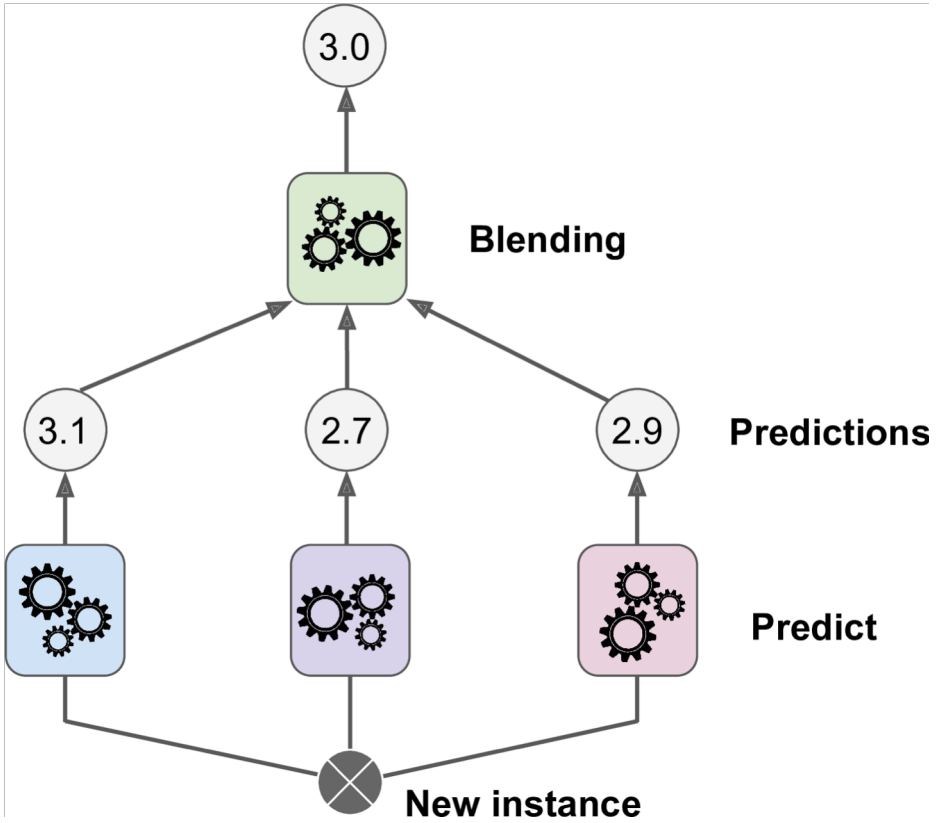


Module 7 – Section 12

Stacking

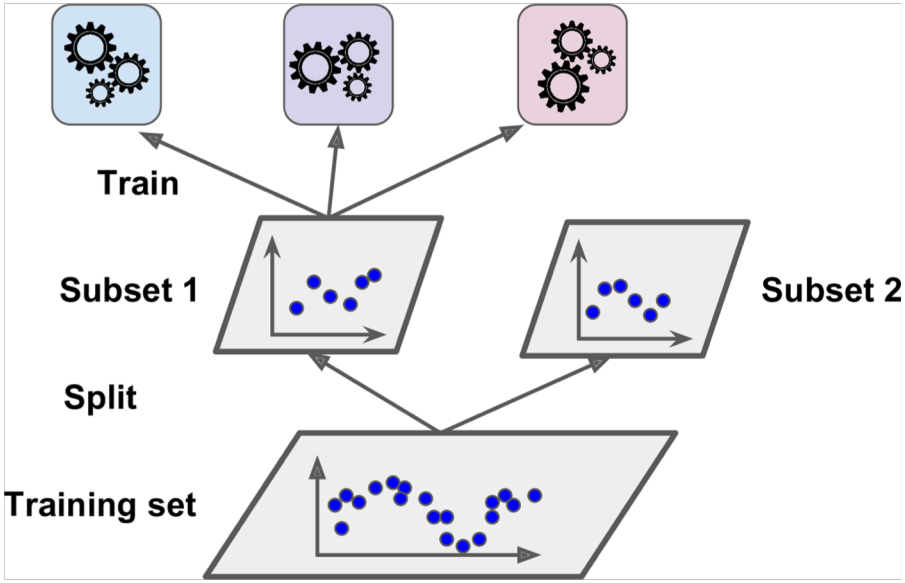
Stacking

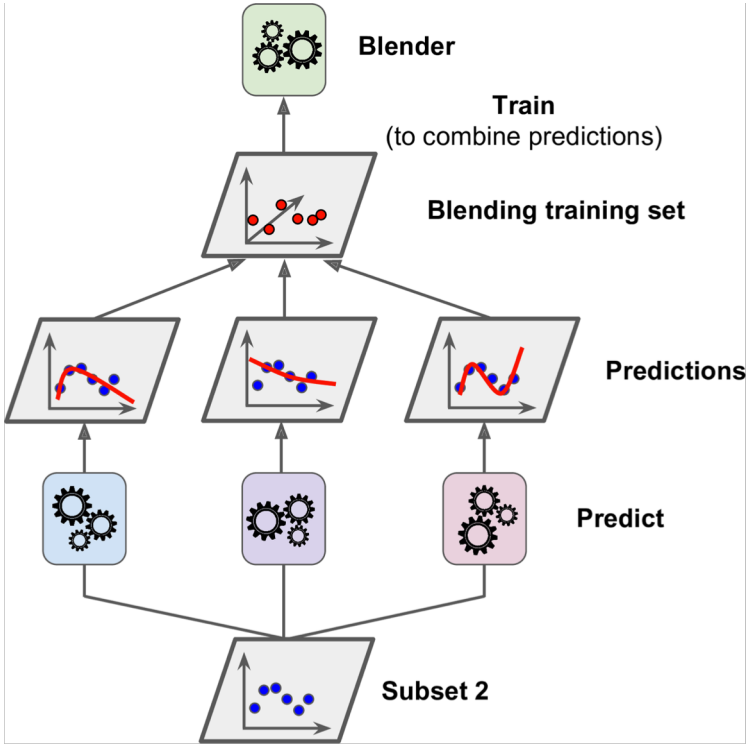
- Replace voting or aggregation, by training another model instead
- The final predictor is called a blender or meta-learner, which takes predictions from other learners and makes the final prediction



Stacking (cont'd)

- Train blender by splitting dataset





UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 7 – Section 13

Resources and Wrap-up

Resources

- Hands-On Machine Learning with Scikit-Learn and Tensorflow:
  - Chapter 6 and 7

Assessment

- See Jupyter Notebook

Next Class

- Dimensionality Reductions

Follow us on social

Join the conversation with us online:

facebook.com/uoftscs

@uoftscs

linkedin.com/company/university-of-toronto-school-of-continuing-studies

@uoftscs

Any questions?

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Thank You

Thank you for choosing the University of Toronto School of Continuing Studies