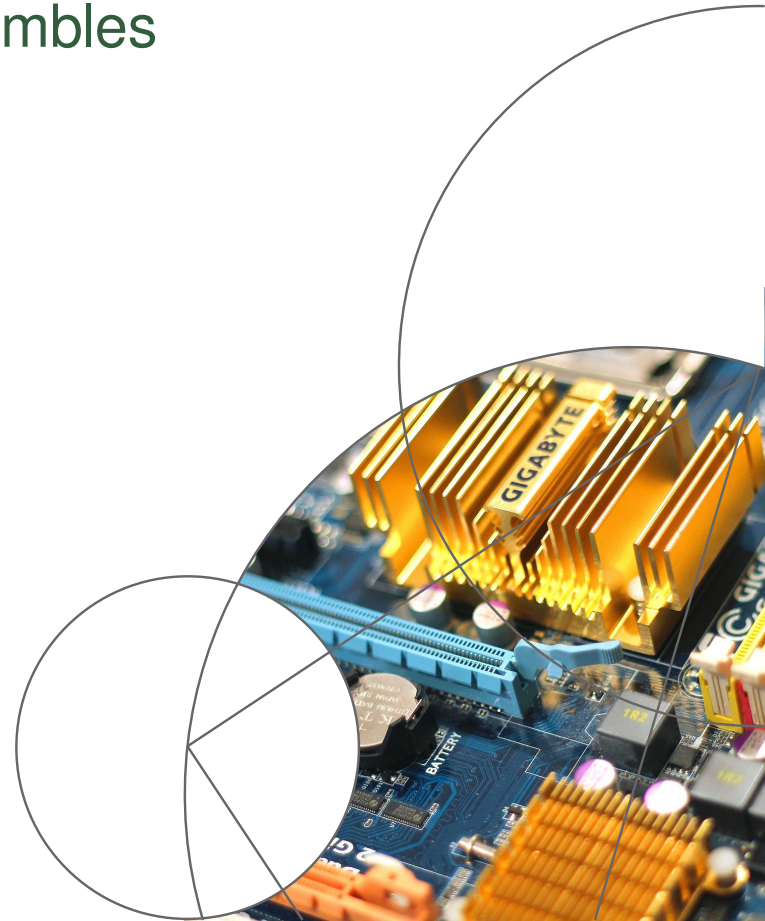
L5 – Large-Scale (Boosted) Tree Ensembles

Large-Scale Data Analysis

Fabian Gieseke

Image Group
Department of Computer Science
University of Copenhagen

Universitetsparken 1, Room 1-1-N110
fabian.gieseke@di.ku.dk



7 May 2019

Today!

http://jmlr.csail.mit.edu/papers/volume15/delgado14a/delgado14a.pdf

Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?

Manuel Fernández-Delgado MANUEL.FERNANDEZ.DELGADO@USC.ES
Eva Cernadas EVA.CERNADAS@USC.ES
Senén Barro SENEN.BARRO@USC.ES
CITIUS: Centro de Investigación en Tecnoloxías da Información da USC
University of Santiago de Compostela
Campus Vida, 15872, Santiago de Compostela, Spain

Dinani Amorim DINANIAMORIM@GMAIL.COM
Departamento de Tecnologia e Ciências Sociais- DTCS
Universidade do Estado da Bahia
Av. Edgard Chastinet S/N - São Geraldo - Juazeiro-BA, CEP: 48.305-680, Brasil

Editor: Russ Greiner

Abstract

We evaluate **179 classifiers** arising from **17 families** (discriminant analysis, Bayesian, neural networks, support vector machines, decision trees, rule-based classifiers, boosting, bagging, stacking, random forests and other ensembles, generalized linear models, nearest-neighbors, partial least squares and principal component regression, logistic and multinomial regression, multiple adaptive regression splines and other methods), implemented in Weka, R (with and without the caret package), C and Matlab, including all the relevant classifiers available today. We use **121 data sets**, which represent **the whole UCI** data base (excluding the large-scale problems) and other own real problems, in order to achieve significant conclusions about the classifier behavior, not dependent on the data set collection. **The classifiers most likely to be the bests are the random forest** (RF) versions, the best of which (implemented in R and accessed via caret) achieves 94.1% of the maximum accuracy overcoming 90% in the 84.3% of the data sets. However, the difference is not statistically significant with the second best, the SVM with Gaussian kernel

Today!

http://jmlr.csail.mit.edu/papers/volume15/delgado14a/delgado14a.pdf

Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?

Manuel Fernández-Delgado MANUEL.FERNANDEZ.DELGADO@USC.ES
Eva Cernadas EVA.CERNADAS@USC.ES
Senén Barro SENEN.BARRO@USC.ES
CITIUS: Centro de Investigación en Tecnoloxías da Información da USC
University of Santiago de Compostela
Campus Vida, 15872, Santiago de Compostela, Spain

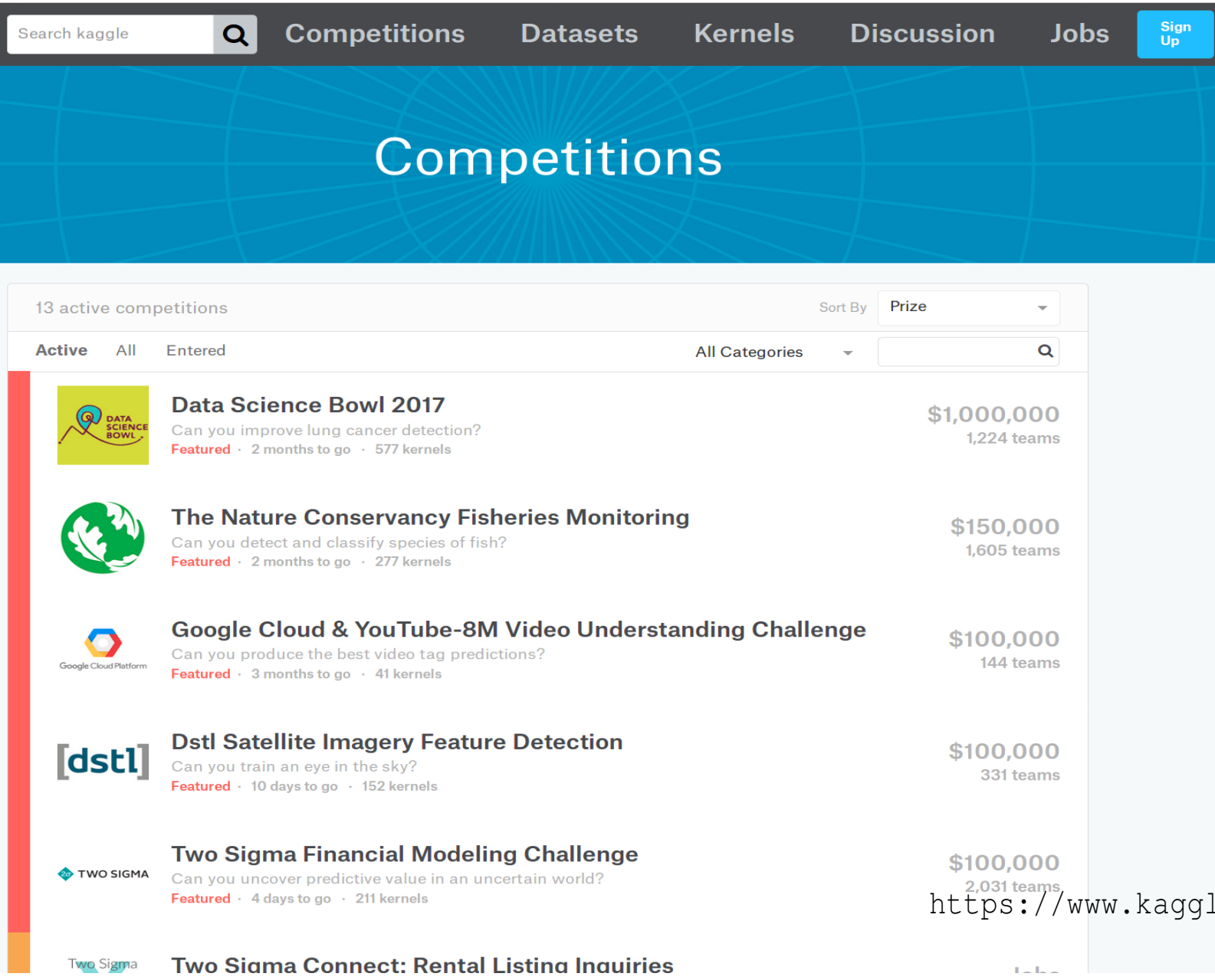
Dinani Amorim DINANIAMORIM@GMAIL.COM
Departamento de Tecnologia e Ciências Sociais- DTCS
Universidade do Estado da Bahia
Av. Edgard Chastinet S/N - São Geraldo - Juazeiro-BA, CEP: 48.305-680, Brasil

Editor: Russ Greiner

Abstract

We evaluate **179 classifiers** arising from **17 families** (discriminant analysis, Bayesian, neural networks, support vector machines, decision trees, rule-based classifiers, boosting, bagging, stacking, random forests and other ensembles, generalized linear models, nearest-neighbors, partial least squares and principal component regression, logistic and multinomial regression, multiple adaptive regression splines and other methods), implemented in Weka, R (with and without the caret package), C and Matlab, including all the relevant classifiers available today. We use **121 data sets**, which represent **the whole UCI** data base (excluding the large-scale problems) and other own real problems, in order to achieve significant conclusions about the classifier behavior, not dependent on the data set collection. **The classifiers most likely to be the bests are the random forest** (RF) versions, the best of which (implemented in R and accessed via caret) achieves 94.1% of the maximum accuracy overcoming 90% in the 84.3% of the data sets. However, the difference is not statistically significant with the second best, the SVM with Gaussian kernel

Data Analysis: Real-World Competitions



https://www.kaggle.com

Today!

https://dnc1994.com/2016/05/rank-10-percent-in-first-kaggle-competition-en/

Wille
最後に辿り着いた場所

Home
About
Archives
Categories
Tags

How to Rank 10% in Your First Kaggle Competition

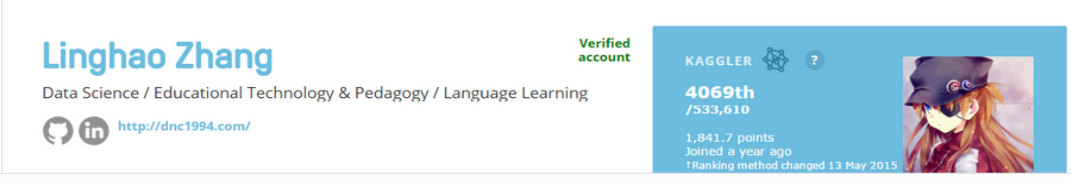
Posted on 2016-05-10 | In Data Science | 22 Comments

Introduction

Kaggle is the best place to learn from other data scientists. Many companies provide data and prize money to set up data science competitions on Kaggle. Recently I had my first shot on Kaggle and **ranked 98th (~ 5%) among 2125 teams**. Being my Kaggle debut, I feel quite satisfied with the result. Since many Kaggle beginners set 10% as their first goal, I want to share my two cents on how to achieve that.

This post is also available in Chinese.

Updated on Oct 28th, 2016: I made many wording changes and added several updates to this post. Note that Kaggle has went through some major changes since I published this post, especially with its ranking system. Therefore some descriptions here might not apply anymore.



Today!

https://dnc1994.com/2016/05/rank-10-percent-in-first-kaggle-competition-en/

Model Selection

When the features are set, we can start training models. Kaggle competitions usually favor **tree-based models**:

- **Gradient Boosted Trees**
- Random Forest
- Extra Randomized Trees

The following models are slightly worse in terms of general performance, but are suitable as base models in ensemble learning (will be discussed later):

- SVM
- Linear Regression
- Logistic Regression
- Neural Networks

Note that this does not apply to computer vision competitions which are pretty much dominated by neural network models.

All these models are implemented in **Sklearn**.

Here I want to emphasize the greatness of **Xgboost**. The outstanding performance of gradient boosted trees and Xgboost's efficient implementation makes it very popular in Kaggle competitions. Nowadays almost every winner uses Xgboost in one way or another.

Updated on Oct 28th, 2016: Recently Microsoft open sourced **LightGBM**, a potentially better

Today!

https://dnc1994.com/2016/05/rank-10-percent-in-first-kaggle-competition-en/

Model Selection

When the features are set, we can start training models. Kaggle competitions usually favor **tree-based models**:

- **Gradient Boosted Trees**
- Random Forest
- Extra Randomized Trees

The following models are slightly worse in terms of general performance, but are suitable as base models in ensemble learning (will be discussed later):

- SVM
- Linear Regression
- Logistic Regression
- Neural Networks

Note that this does not apply to computer vision competitions which are pretty much dominated by neural network models.

All these models are implemented in **Sklearn**.

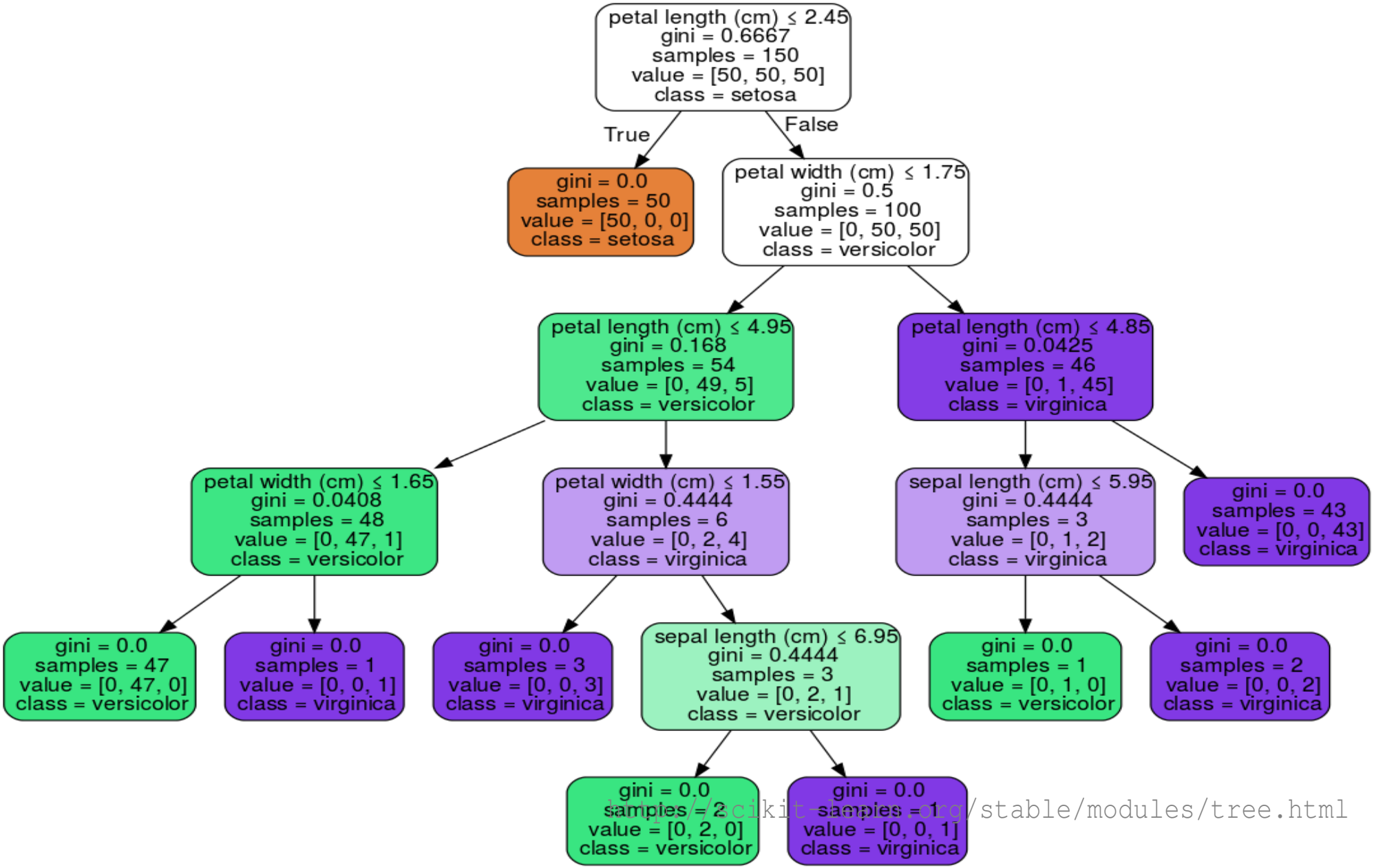
Here I want to emphasize the greatness of **Xgboost**. The outstanding performance of gradient boosted trees and Xgboost's efficient implementation makes it very popular in Kaggle competitions. Nowadays almost every winner uses Xgboost in one way or another.

Updated on Oct 28th, 2016: Recently Microsoft open sourced **LightGBM**, a potentially better

Outline

Outline

Example: Iris Flower Classification Tree

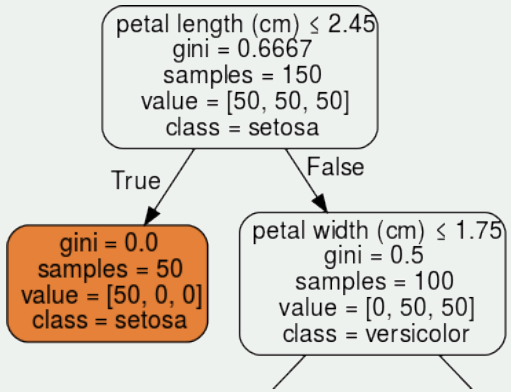


http://scikit-learn.org/stable/modules/tree.html

Classification and Regression Trees (CARTs)

Structure & Training

- Every internal node is associated with one coordinate $i \in \{1, \dots, d\}$ and a threshold θ.
- At each inner node, the training data $S = \{(\mathbf{x}_1, y_1), \dots\}$ at that node are split into



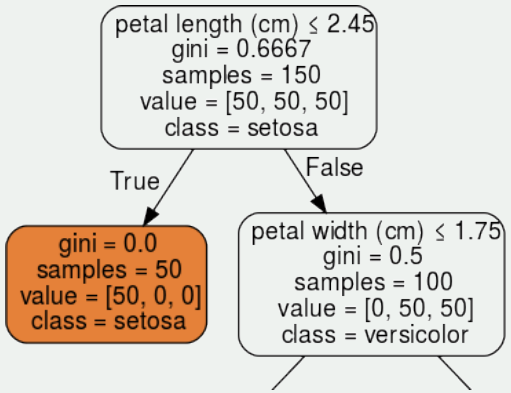
$$L_{i,\theta} = \{(\mathbf{x}, y) \in S \,|\, x_i \leq \theta\} \text{ and } R_{i,\theta} = \{(\mathbf{x}, y) \in S \,|\, x_i > \theta\},$$

  which are passed to the left/right child for further processing.
- Construction: The optimal tree cannot be found efficiently. Therefore, trees are built using a heuristic.
- The leaf nodes, indexed by $\tau = 1, \dots, M$, define regions $\mathcal{R}_\tau \subseteq \mathbb{R}^d$.

Building CARTs

Basic Idea

- Every inner node is associated with one coordinate $i \in \{1, \ldots, d\}$ and a threshold θ.
- At each inner node, the associated data $S = \{(\mathbf{x}_1, y_1), \ldots\}$ are split into $L_{i,\theta}$ and $R_{i,\theta}$ such that the information gain



$$G_{i,\theta}(S) = Q(S) - \frac{|L_{i,\theta}|}{|S|} Q(L_{i,\theta}) - \frac{|R_{i,\theta}|}{|S|} Q(R_{i,\theta})$$

  is maximized, where Q is some impurity measure.
- If the number $|S|$ of points at a node is smaller than a user-defined value or if S is pure (i.e., all point have the same label), the node becomes a leaf (further stopping rules exist!).

Simplified Objective

Note that

$$\underset{i,\theta}{\text{maximize}} \; G_{i,\theta}(S) = Q(S) - \frac{|L_{i,\theta}|}{|S|} Q(L_{i,\theta}) - \frac{|R_{i,\theta}|}{|S|} Q(R_{i,\theta})$$

is equivalent to

$$\underset{i,\theta}{\text{minimize}} \; |L_{i,\theta}| Q(L_{i,\theta}) + |R_{i,\theta}| Q(R_{i,\theta})$$

Let's build a tree!

Recursive Tree Construction

Procedure: BUILDTREE(S,m)

Require: $S = \{(\mathbf{x}_1, y_1), \ldots\}$ and number m.

Ensure: Tree $\mathcal{T}$ built for the input patterns in S.

1: **if** $|S| \leq m$ **then**
2: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
3: **end if**
4: **if** $y_i = y_j$ for all $(\mathbf{x}_i, y_i), (\mathbf{x}_j, y_j) \in S$ **then**
5: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
6: **end if**
7: Find $(i^*, \theta^*) = \text{argmax}_{i,\theta}\, G_{i,\theta}(S)$
8: $\mathcal{T}_l$ = BUILDTREE($L_{i,\theta}$, m)
9: $\mathcal{T}_r$ = BUILDTREE($R_{i,\theta}$, m)
10: Generate node that stores the splitting information (i^*, θ^*) and pointers to its subtrees $\mathcal{T}_l$ and $\mathcal{T}_r$. Let $\mathcal{T}$ denote the resulting tree.
11: **return** $\mathcal{T}$

Regression Trees

Construction: What are "optimal" splits?

- Impurity measure: For regression problems, a common choice is

$$Q(S) = \sum_{(\mathbf{x},y)\in S} (y - \bar{y})^2,$$

  where $\bar{y} = \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} y$

- Find optimal splitting dimension i^* and threshold θ^* w.r.t.

$$\underset{i,\theta}{\text{minimize}} \; |L_{i,\theta}| Q(L_{i,\theta}) + |R_{i,\theta}| Q(R_{i,\theta})$$

Regression Trees

Construction: What are "optimal" splits?

- Impurity measure: For regression problems, a common choice is

$$Q(S) = \sum_{(\mathbf{x},y)\in S} (y - \bar{y})^2,$$

  where $\bar{y} = \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} y$

- Find optimal splitting dimension i^* and threshold θ^* w.r.t.

$$\underset{i,\theta}{\text{minimize}} \; |L_{i,\theta}| Q(L_{i,\theta}) + |R_{i,\theta}| Q(R_{i,\theta})$$

How to compute the optimal dimension and threshold?

Regression Trees

Construction: What are "optimal" splits?

- Impurity measure: For regression problems, a common choice is

$$Q(S) = \sum_{(\mathbf{x},y)\in S} (y - \bar{y})^2,$$

  where $\bar{y} = \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} y$

- Find optimal splitting dimension i^* and threshold θ^* w.r.t.

$$\underset{i,\theta}{\text{minimize}} \; |L_{i,\theta}| Q(L_{i,\theta}) + |R_{i,\theta}| Q(R_{i,\theta})$$

How to compute the optimal dimension and threshold?

1. Naive approach, $\mathcal{O}(d \cdot |S|^2)$ time: For each i, do:
   - Consider all possible thresholds (e.g., $\{\mathbf{x}_i | \mathbf{x} \in S\}$).
   - For each threshold θ: Compute $|L_{i,\theta}| Q(L_{i,\theta})$ and $|R_{i,\theta}| Q(R_{i,\theta})$.

Regression Trees

Construction: What are "optimal" splits?

- Impurity measure: For regression problems, a common choice is

$$Q(S) = \sum_{(\mathbf{x},y)\in S} (y - \bar{y})^2,$$

  where $\bar{y} = \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} y$

- Find optimal splitting dimension i^* and threshold θ^* w.r.t.

$$\underset{i,\theta}{\text{minimize}} \; |L_{i,\theta}| Q(L_{i,\theta}) + |R_{i,\theta}| Q(R_{i,\theta})$$

How to compute the optimal dimension and threshold?

1. Naive approach, $\mathcal{O}(d \cdot |S|^2)$ time: For each i, do:
   - Consider all possible thresholds (e.g., $\{\mathbf{x}_i | \mathbf{x} \in S\}$).
   - For each threshold θ: Compute $|L_{i,\theta}| Q(L_{i,\theta})$ and $|R_{i,\theta}| Q(R_{i,\theta})$.
2. Better approach, $\mathcal{O}(d \cdot |S| \log |S|)$ time: For each i, do:
   - Sort all instances $(\mathbf{x}, y)$ in S according to the patterns i-th dimension.
   - Scan this sorted list and update these values incrementally!

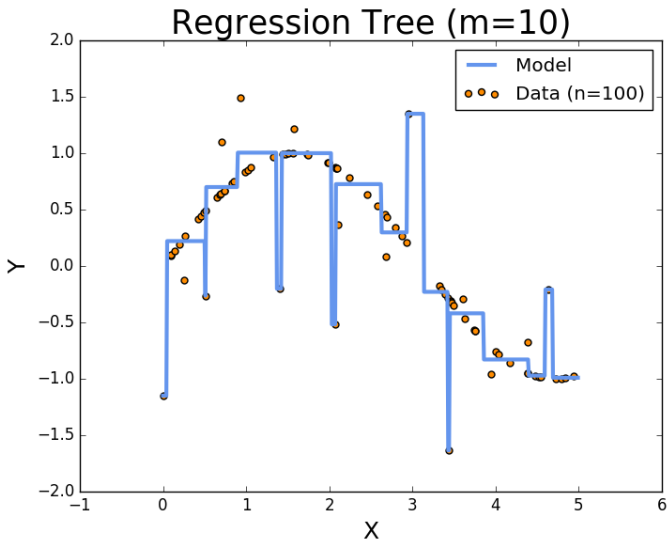
Regression Trees

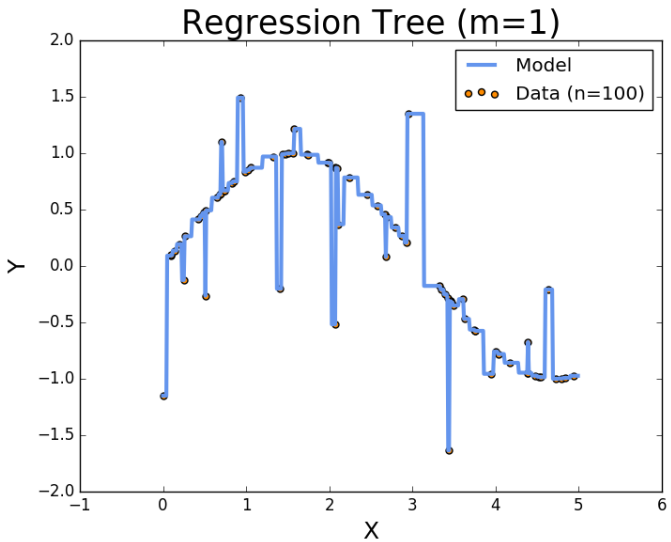
Prediction

- Training data $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$
- Each leaf node $\tau = 1, \ldots, M$ of the tree $\mathcal{T}$ corresponds to a region $R_\tau \subseteq \mathbb{R}^d$
- The output $\hat{y} = \mathcal{T}(\mathbf{x})$ given a new input instance $\mathbf{x}$ is

$$\mathcal{T}(\mathbf{x}) = \sum_{\tau=1}^{M} c_\tau \mathbb{I}\{\mathbf{x} \in \mathcal{R}_\tau\} \ ,$$

where $c_\tau = \frac{1}{|R_\tau \cap S|} \sum_{\mathbf{x}_j \in \mathcal{R}_\tau \cap S} y_j$ (thus: mean of all labels stored in leaf).





Demo Time!

jupyter Regression Trees Last Checkpoint: a few seconds ago (autosaved) Logout

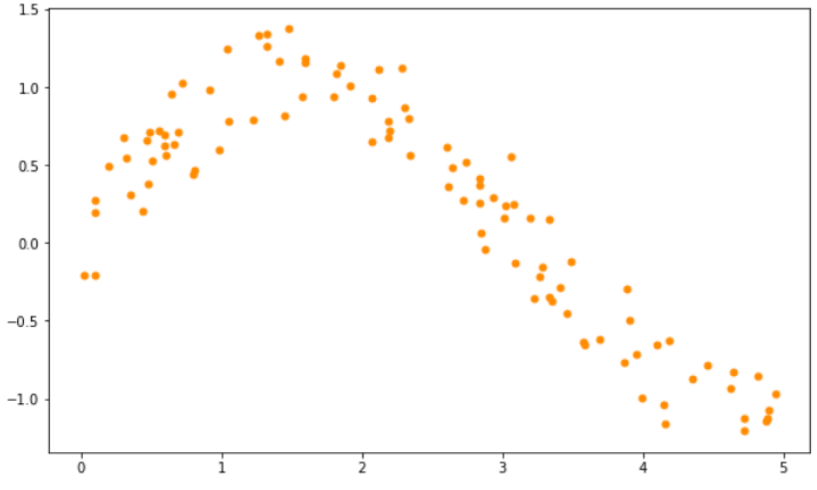
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

In [1]:
```
%matplotlib inline
import numpy as np
np.random.seed(0)
import matplotlib.pyplot as plt
```

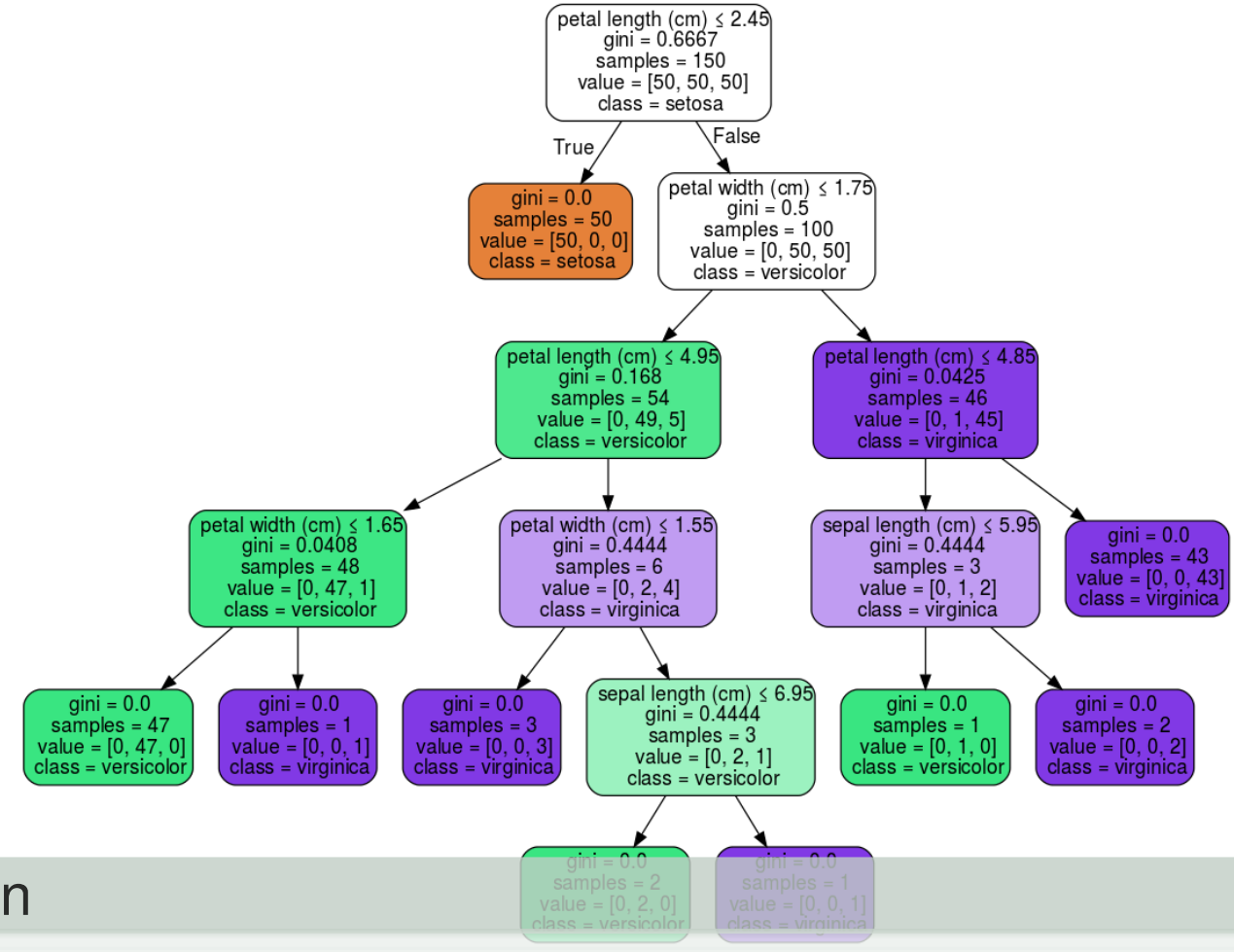
In [2]:
```
# artificial dataset
n_samples = 100
X = np.sort(5 * np.random.rand(n_samples, 1), axis=0).reshape((n_samples,1))
y = np.sin(X) + 0.2*np.random.randn(n_samples).reshape((n_samples,1))
```

In [3]:
```
# plot the data
plt.figure(figsize=(10,6))
plt.scatter(X, y, c="darkorange", label="Data (n=100)", s=25)
plt.show()
```



In [4]:
```
from sklearn.tree import DecisionTreeRegressor

# YOUR TASK: Try out different values for the parameter 'min_samples_split'

# instantiate model and fit it!
model = DecisionTreeRegressor(min_samples_split=20)
model.fit(X,y)
```

Classification Trees



Prediction

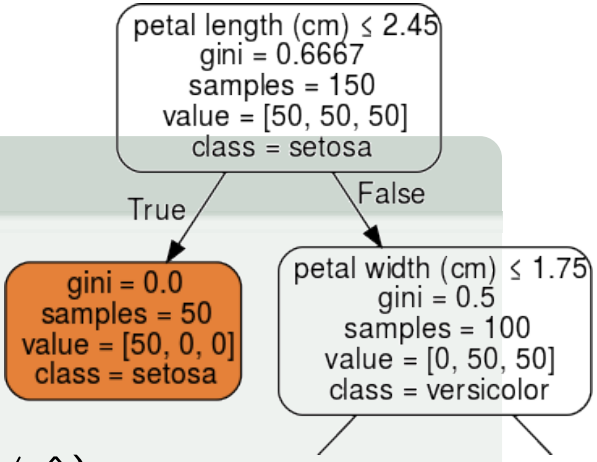
1. Traverse the tree $\mathcal{T}$ to find the leaf that contains the query $\mathbf{x}$.
2. Each leaf node $\tau = 1, \ldots, M$ of the tree $\mathcal{T}$ corresponds to a region $R_\tau \subset \mathbb{R}^d$
3. Use majority class in leaf as prediction $\mathcal{T}(\mathbf{x})$.

Classification Trees

Construction

As before: We need to define an impurity measure $Q(S)$.
Let p_{Sk} be the fraction of points belonging to class k in S.

1. Misclassification error: Let $\hat{y} = \text{argmax}_k\, p_{Sk}$ be the dominant class in S. Then $Q(S) = \frac{1}{|S|} \sum_{(\mathbf{x}_j, y_j) \in S} \mathbb{I}(y_j \neq \hat{y})$
2. Gini index: $Q(S) = \sum_{k=1}^{K} p_{Sk}(1 - p_{Sk})$
3. ...

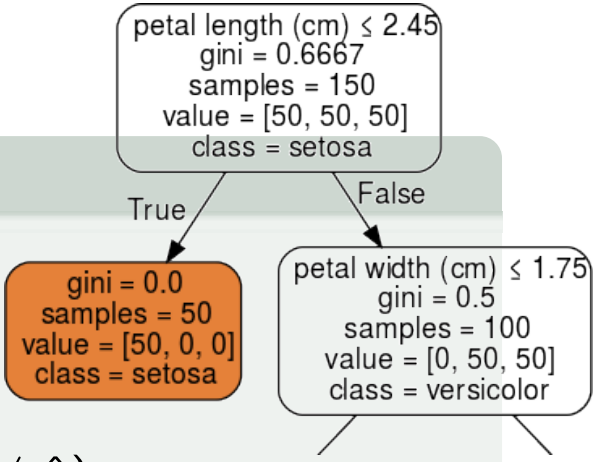


Classification Trees

Construction

As before: We need to define an impurity measure $Q(S)$.
Let p_{Sk} be the fraction of points belonging to class k in S.

1. Misclassification error: Let $\hat{y} = \text{argmax}_k\, p_{Sk}$ be the dominant class in S. Then $Q(S) = \frac{1}{|S|} \sum_{(\mathbf{x}_j, y_j) \in S} \mathbb{I}(y_j \neq \hat{y})$
2. Gini index: $Q(S) = \sum_{k=1}^{K} p_{Sk}(1 - p_{Sk})$
3. ...



Now: Check Gini indices for all three nodes.

Demo Time!

jupyter Classification Trees Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

Simple example showing how to train and visualize a classification tree.

In [1]:
```
%matplotlib inline
import numpy as np
np.random.seed(0)
import matplotlib.pyplot as plt
```

Let us load the Iris dataset (https://archive.ics.uci.edu/ml/datasets/iris). Note that we only consider two features (defined via the 'pair' list).

In [2]:
```
from sklearn.datasets import load_iris

pair = [0,1]
iris = load_iris()

X = iris.data[:, pair]
y = iris.target
```

Let us fit a classification tree using the popular Scikit-Learn library!

In [3]:
```
from sklearn.tree import DecisionTreeClassifier

# instantiate the model
model = DecisionTreeClassifier(max_depth=2)

# train the model
model.fit(X,y)
```

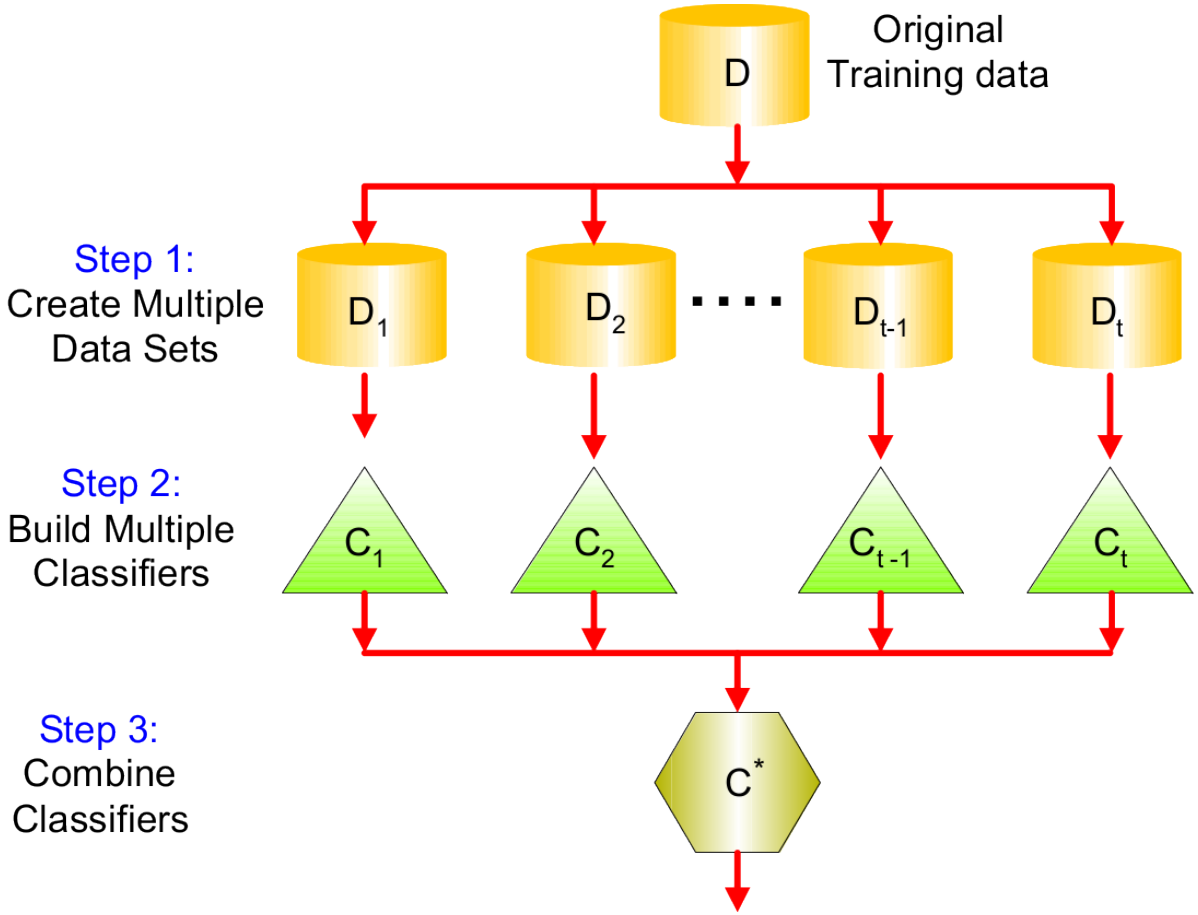
Out[3]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=2,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

Next, let us visualize the final classification surface/boundaries. Since we are only considering two features, we can generate a two-dimensional plot (see http://scikit-learn.org/stable/auto_examples/tree/plot_iris.html).

In [4]:
```
plot_step = 0.02
n_classes = 3
plot_colors = "ryb"

plt.figure()

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
```

Outline

General Idea



"Predict class label of previously unseen records by aggregating predictions made by multiple classifiers. Each of the classifiers is built in a slightly different way!"

Bagging

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Bagging (Round 1)	7	8	10	8	2	5	10	10	5	9
Bagging (Round 2)	1	4	9	1	2	3	2	7	3	2
Bagging (Round 3)	1	8	5	10	5	5	9	6	3	7

Sampling with Replacement I

- Generate multiple training set instances of size n by sampling with replacement. These samples are called bootstrap samples.

Bagging

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Bagging (Round 1)	7	8	10	8	2	5	10	10	5	9
Bagging (Round 2)	1	4	9	1	2	3	2	7	3	2
Bagging (Round 3)	1	8	5	10	5	5	9	6	3	7

Sampling with Replacement I

- Generate multiple training set instances of size n by sampling with replacement. These samples are called bootstrap samples.
- Each bootstrap sample contains n instances of the data set.

Bagging

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Bagging (Round 1)	7	8	10	8	2	5	10	10	5	9
Bagging (Round 2)	1	4	9	1	2	3	2	7	3	2
Bagging (Round 3)	1	8	5	10	5	5	9	6	3	7

Sampling with Replacement I

- Generate multiple training set instances of size n by sampling with replacement. These samples are called bootstrap samples.
- Each bootstrap sample contains n instances of the data set.
- The probability for each sample to be selected at least once is $1 - \left(1 - \frac{1}{n}\right)^n \to 1 - \frac{1}{e} \approx 0.632$. Hence, a bootstrap sample only contains about 63% distinct examples!

Bagging

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Bagging (Round 1)	7	8	10	8	2	5	10	10	5	9
Bagging (Round 2)	1	4	9	1	2	3	2	7	3	2
Bagging (Round 3)	1	8	5	10	5	5	9	6	3	7

Sampling with Replacement I

- Generate multiple training set instances of size n by sampling with replacement. These samples are called bootstrap samples.
- Each bootstrap sample contains n instances of the data set.
- The probability for each sample to be selected at least once is $1 - \left(1 - \frac{1}{n}\right)^n \to 1 - \frac{1}{e} \approx 0.632$. Hence, a bootstrap sample only contains about 63% distinct examples!
- Next: Build one classifier on top of each bootstrap sample.

Random Forests

Construction

Procedure: BUILDRF(S,m,f,B)

Require: $S = \{(\mathbf{x}_1, y_1), \ldots\}$, number m, number f of features to be tested per split, and number B of trees.

Ensure: Trees $\mathcal{T}_1, \ldots, \mathcal{T}_B$ for trees.

1: **for** $b = 1, \ldots, B$ **do**
2: Draw bootstrap sample S' by drawing $|S|$ elements with replacement from S.
3: $\mathcal{T}_b$ = BUILDRFTREE(S', m, f)
4: **end for**
5: **return** $\mathcal{T}_1, \ldots, \mathcal{T}_B$

1. Regression: $f_{\text{RF}}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} \mathcal{T}_b(\mathbf{x})$
2. Classification: $f_{\text{RF}}(\mathbf{x}) =$ majority vote among $\mathcal{T}_1, \ldots, \mathcal{T}_B$

Random Forests: Tree Construction

Recursive RF Tree Construction

Procedure: BUILDRFTREE(S,m,f)

Require: $S = \{(\mathbf{x}_1, y_1), \ldots\}$, number m (usually m=1), and number f of features to be tested per split.

Ensure: Tree $\mathcal{T}$ built for the input patterns in S.

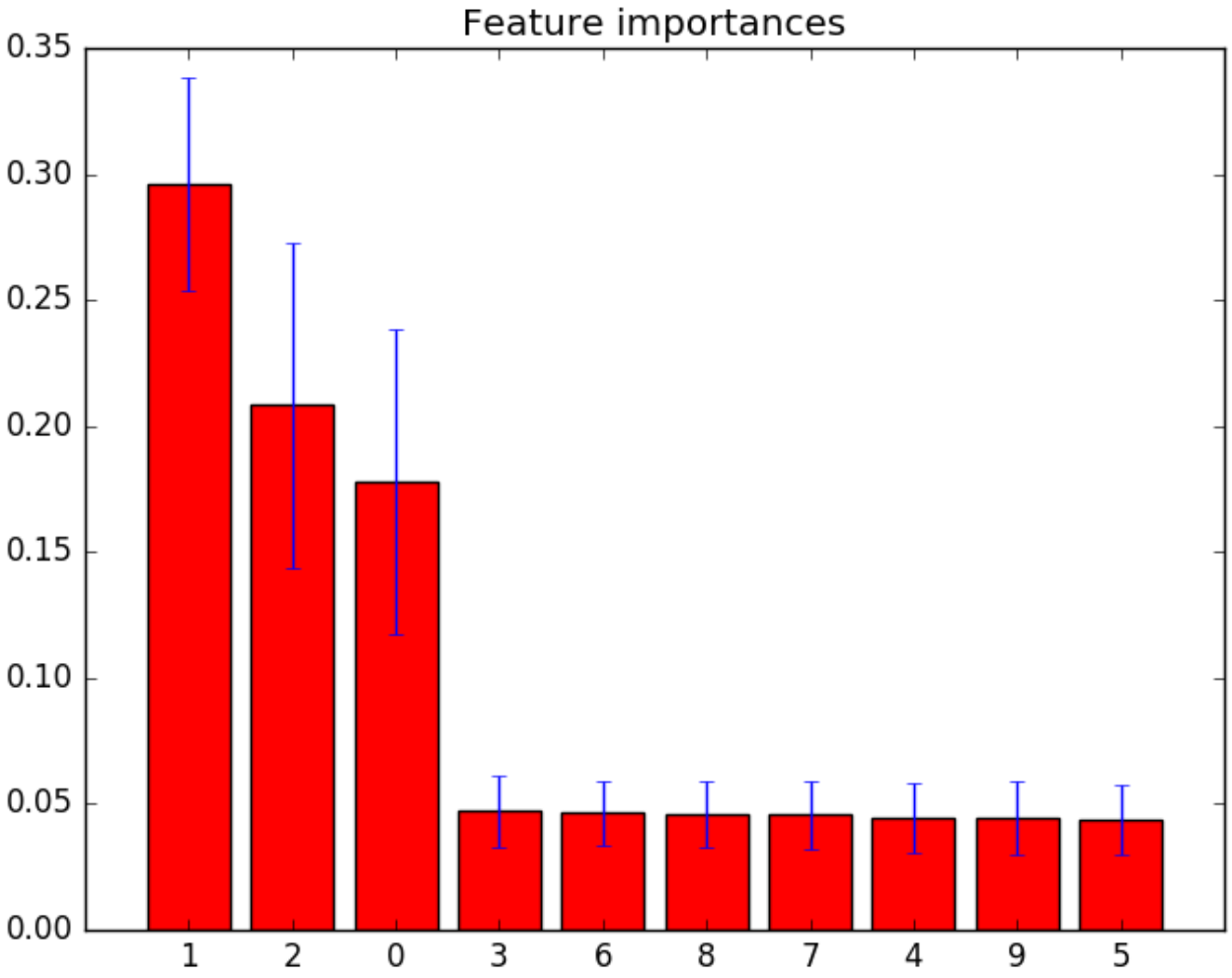
1: **if** $|S| \leq m$ **then**
2: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
3: **end if**
4: **if** $y_i = y_j$ for all $(\mathbf{x}_i, y_i), (\mathbf{x}_j, y_j) \in S$ **then**
5: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
6: **end if**
7: Find $(i^*, \theta^*) = \text{argmax}_{i,\theta} G_{i,\theta}(S)$ for $i \in \{i_1, \ldots, i_f\} \subseteq \{1, \ldots, d\}$ ("only test f random dimensions")
8: $\mathcal{T}_l$ = BUILDRFTREE($L_{i,\theta}$, m, f)
9: $\mathcal{T}_r$ = BUILDRFTREE($R_{i,\theta}$, m, f)
10: Generate node ((i^*, θ^*) and pointers). Let $\mathcal{T}$ be the resulting tree.
11: **return** $\mathcal{T}$

Random Forests: Why Cool?

Random Forests: Why Cool?

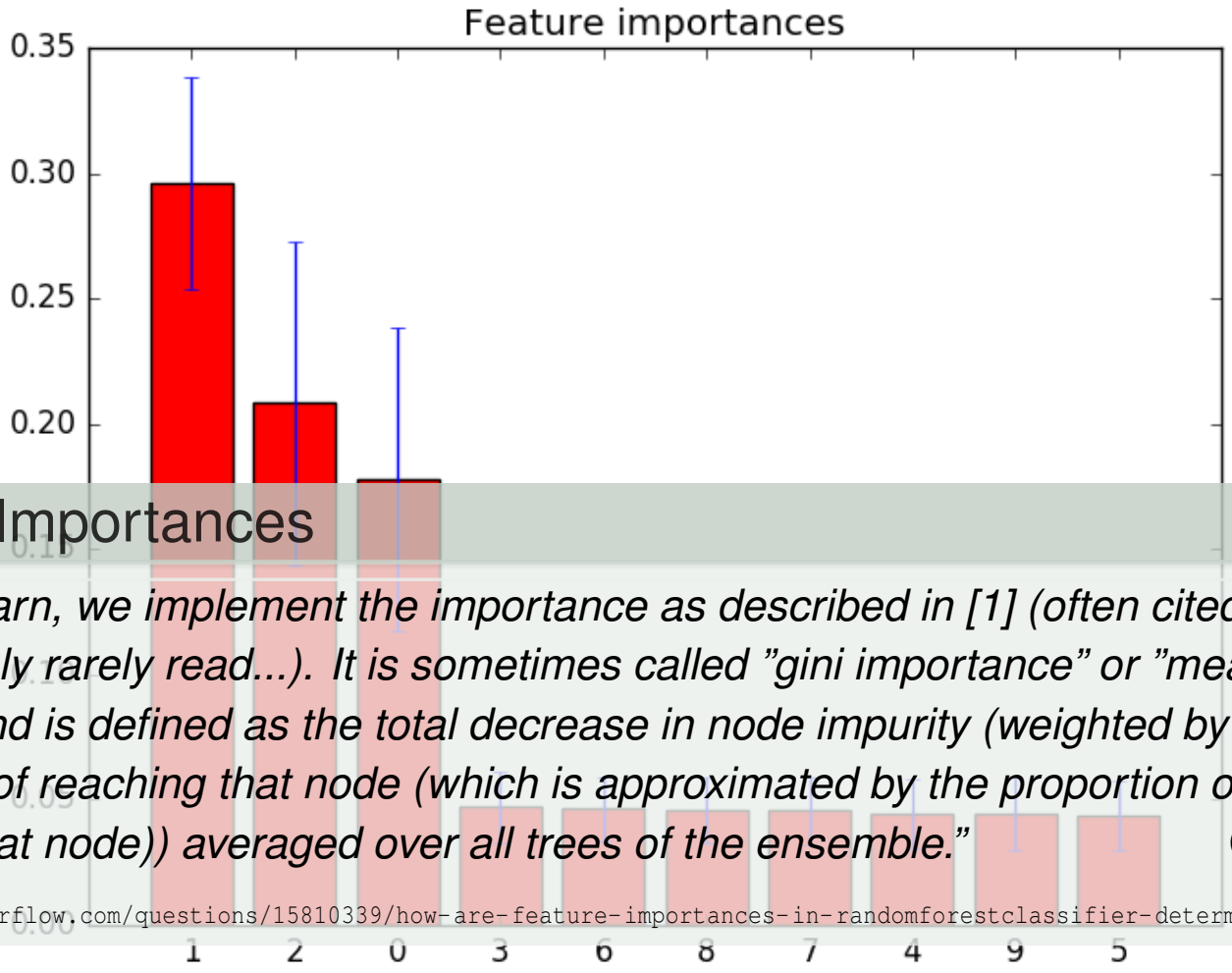
1. The aggregation of multiple different trees leads to a reduction of the random forest's variance.
2. By searching for "good" splitting features at each internal node, irrelevant features are ignored!
3. Each tree induces a hierarchical subdivision of the feature space and the model "adapts" to the induced subregions (e.g., the feature selection takes place at each internal node!).
4. Due to their hierarchical structure, random forests are capable of effectively dealing with (very) unbalanced datasets.
5. . . .

Random Forests: Feature Selection



http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html

Random Forests: Feature Selection



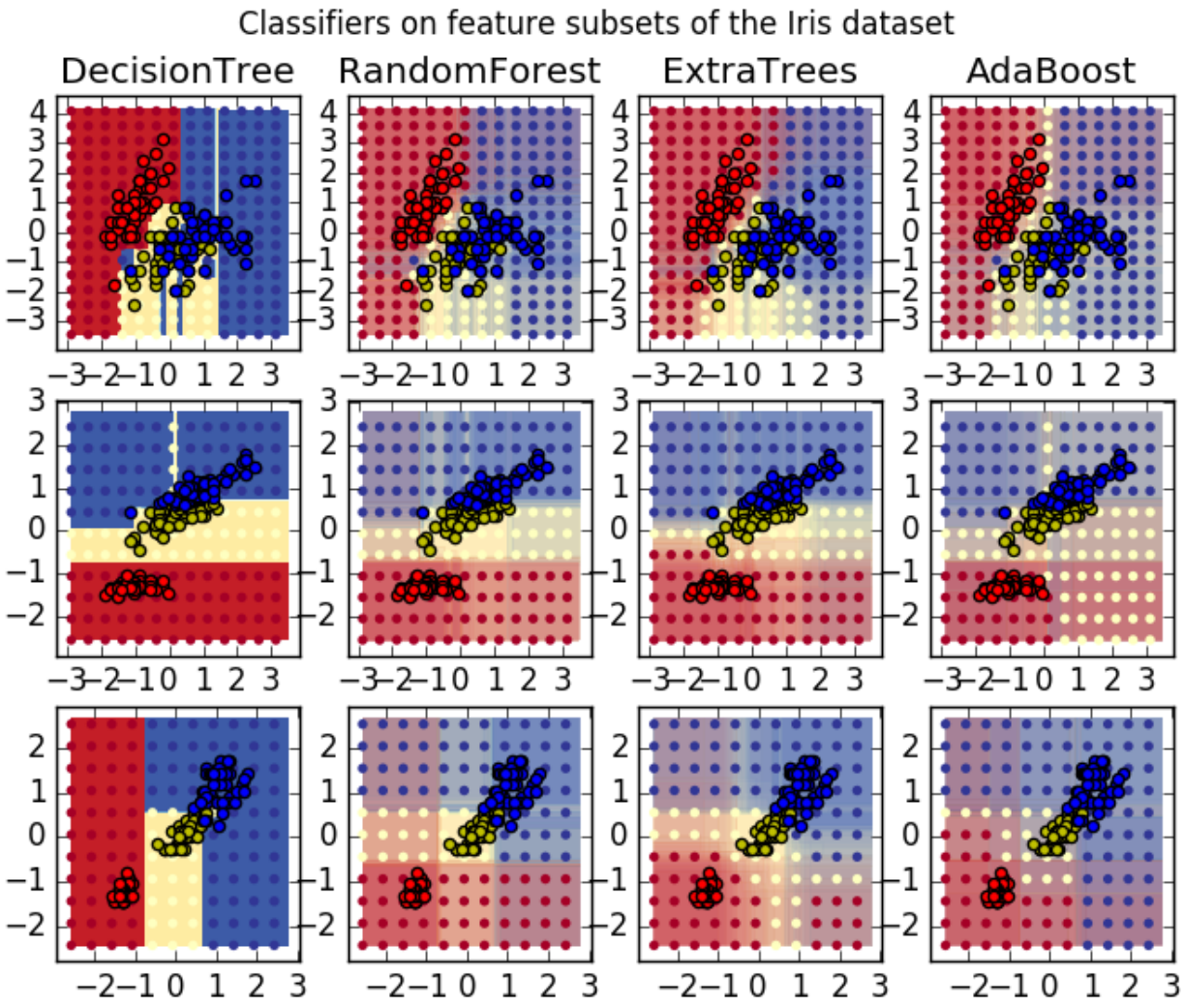
Feature Importances

"In scikit-learn, we implement the importance as described in [1] (often cited, but unfortunately rarely read...). It is sometimes called "gini importance" or "mean decrease impurity" and is defined as the total decrease in node impurity (weighted by the probability of reaching that node (which is approximated by the proportion of samples reaching that node)) averaged over all trees of the ensemble." Gilles Louppe

https://stackoverflow.com/questions/15810339/how-are-feature-importances-in-randomforestclassifier-determined

http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html

Tree Ensembles



http://scikit-learn.org/stable/modules/ensemble.html

Random Forest $\rightarrow$ Extra Trees

Exta Trees

Basically the same as random forests. Two small modifications:

1. Do not draw bootstrap samples.
2. Use random threshold $\hat{\theta} \in [\min_{\mathbf{x} \in S} x_i, \max_{\mathbf{x} \in S} x_i]$ per dimension i.

Why does this still work?! Why does this even work better in practice?

Random Forest $\rightarrow$ Extra Trees

Exta Trees

Basically the same as random forests. Two small modifications:

1. Do not draw bootstrap samples.
2. Use random threshold $\hat{\theta} \in [\min_{\mathbf{x} \in S} x_i, \max_{\mathbf{x} \in S} x_i]$ per dimension i.

Why does this still work?! Why does this even work better in practice?

Answer: We still compute the qualities of the random splits and select the best-performing one. Also, we introduce more randomness, which is usually helpful in practice.

Demo Time!

jupyter Random Forests Boston Last Checkpoint: 6 minutes ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

This exercise is about applying Random Forests to a well-known dataset, the Boston Housing Dataset (https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html). In the first step, you will load and quickly analyze the dataset. In the second step, you will training and apply a random forest regression model.

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt

import numpy as np
np.random.seed(0)
```

Let's load the dataset. Luckily, the Scikit-Learn package provides a very simple way to load the Boston dataset ...

In [2]:
```
from sklearn.datasets import load_boston

boston = load_boston()

print("Number of instances: {}".format(boston.data.shape[0]))
print("Number of features: {}".format(boston.data.shape[1]))

# YOUR TASK: Print details related to the dataset
# print(boston.DESCR)
```

```
Number of instances: 506
Number of features: 13
```

Let's quickly analyze the main characteristics of the data at hand. The Pandas package (https://pandas.pydata.org/) provides some nice out-of-the-box methods to quickly analyze data and to do some preprocessing (if needed).

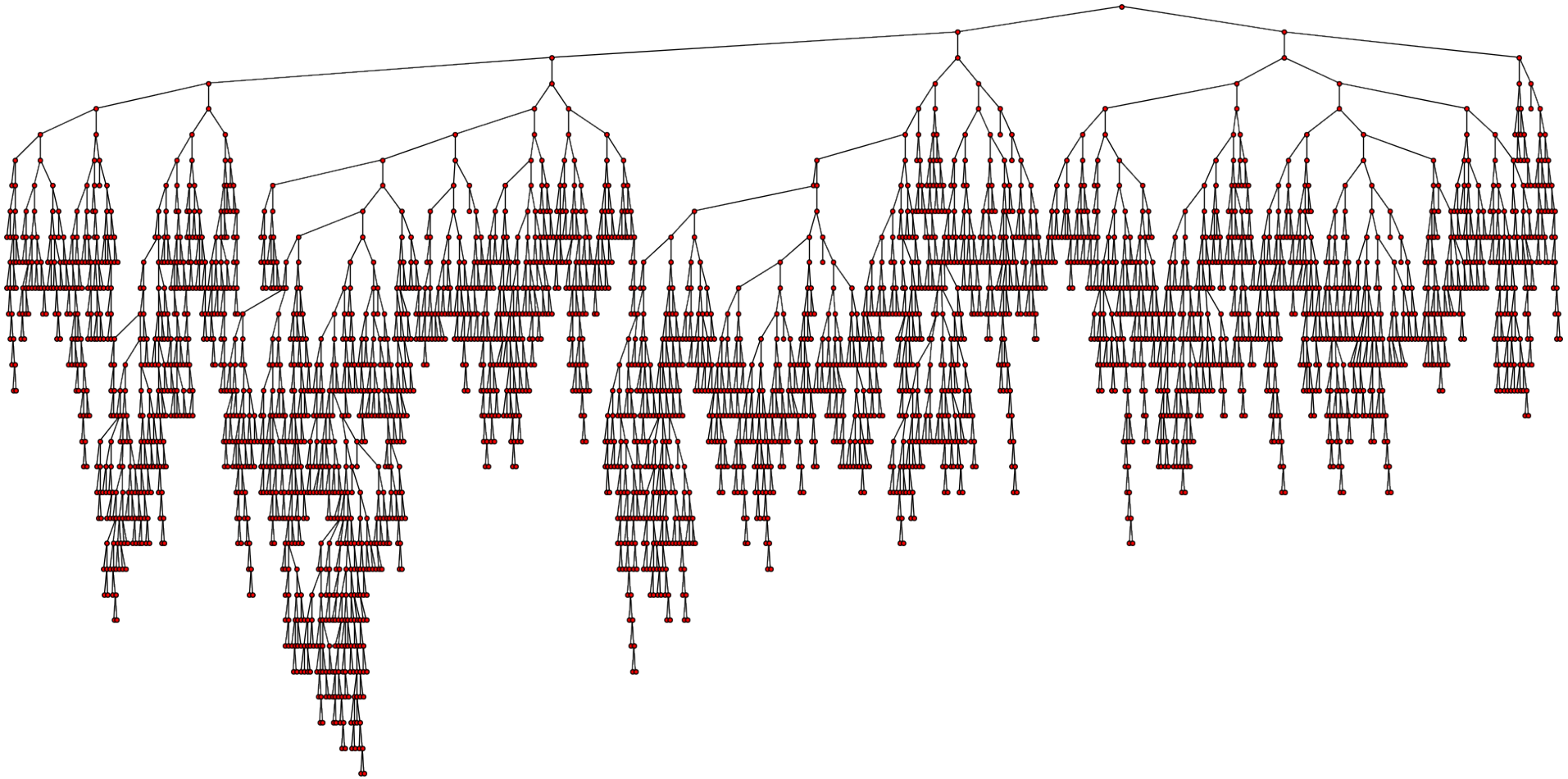
In [3]:
```
import pandas as pd

df = pd.DataFrame(boston.data)
df.columns = boston.feature_names
df['PRICE (REGRESSION TARGET)'] = boston.target
df.head()
```

Out[3]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE (REGRESSION TARGET)
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

In [4]:
```
# YOUR TASK: Make use of the 'describe' function to compute and plot some statistics!
# QUESTION: What is the mean of the target column?
```

Big Trees?



Random Forest, Extra Trees, . . .

Recursive Tree Construction

Procedure: BUILDRFTREE(S,m,f)

Require: $S = \{(\mathbf{x}_1, y_1), \ldots\}$, number m (usually $m = 1$), and number f of features to be tested per split.

Ensure: Tree $\mathcal{T}$ built for the input patterns in S.

1: **if** $|S| \leq m$ or $y_i = y_j$ for all $(\mathbf{x}_i, y_i), (\mathbf{x}_j, y_j) \in S$ **then**
2: **return** leaf node storing the labels $\{y_1, \ldots, y_{|S|}\}$
3: **end if**
4: Find "good" splitting dimension/threshold (i^*, θ^*)
5: $\mathcal{T}_l$ = BUILDRFTREE($L_{i,\theta}$, m, f)
6: $\mathcal{T}_r$ = BUILDRFTREE($R_{i,\theta}$, m, f)
7: Generate node ((i^*, θ^*) and pointers). Let $\mathcal{T}$ be the resulting tree.
8: **return** $\mathcal{T}$

Computational Complexities

Single RF Tree

Let $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ be a training set. We need

$$T(|S|) = C(|S|) + T(|L_{i,\theta}|) + T(|R_{i,\theta}|) \tag{1}$$

time to build a single tree, where $T(N)$ denotes the time to build a tree with N elements and $C(N)$ the time spent at each node wih N elements.
Naturally: $T(m) = c_1$.

1. Random forests: As shown before, one can sort the instances per split to achieve a runtime of $C(N) = \mathcal{O}(f \cdot N \log N)$ per node split.
2. Extra trees: We only need $C(N) = \mathcal{O}(f \cdot N)$ time per node split. Why?

The final structure of a tree, and, hence, its construction time, depends on the particular training data (we also stop in case the current set is pure!).

Computational Complexities

Single RF Tree

Let $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ be a training set. We need

$$T(|S|) = C(|S|) + T(|L_{i,\theta}|) + T(|R_{i,\theta}|) \tag{1}$$

time to build a single tree, where $T(N)$ denotes the time to build a tree with N elements and $C(N)$ the time spent at each node wih N elements.
Naturally: $T(m) = c_1$.

1. Random forests: As shown before, one can sort the instances per split to achieve a runtime of $C(N) = \mathcal{O}(f \cdot N \log N)$ per node split.
2. Extra trees: We only need $C(N) = \mathcal{O}(f \cdot N)$ time per node split. Why?

The final structure of a tree, and, hence, its construction time, depends on the particular training data (we also stop in case the current set is pure!).

Question: *What is $T(|S|)$ in the best case for random forests?*

Computational Complexities

Single RF Tree

Let $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ be a training set. We need

$$T(|S|) = C(|S|) + T(|L_{i,\theta}|) + T(|R_{i,\theta}|) \tag{1}$$

time to build a single tree, where $T(N)$ denotes the time to build a tree with N elements and $C(N)$ the time spent at each node wih N elements.
Naturally: $T(m) = c_1$.

1. Random forests: As shown before, one can sort the instances per split to achieve a runtime of $C(N) = \mathcal{O}(f \cdot N \log N)$ per node split.
2. Extra trees: We only need $C(N) = \mathcal{O}(f \cdot N)$ time per node split. Why?

The final structure of a tree, and, hence, its construction time, depends on the particular training data (we also stop in case the current set is pure!).

Question: What is $T(|S|)$ in the best case for random forests?

Best case: We might be able to stop after having computed the impurity $Q(S)$ for S (root node). Hence: $T(|S|) = C(|S|) = \mathcal{O}(f \cdot |S| \log |S|)$. With additional check (line 1): $T(|S|) = \mathcal{O}(|S|)$.

Computational Complexities: Worst-Case

Single RF Tree: Worst-Case

Let $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ be a training set. In the worst case, we reduce the problem size by only 1 per node split:

$$\begin{aligned} T(|S|) &= C(|S|) + T(1) + T(|S|-1) \\ &= C(|S|) + T(1) + C(|S|-1) + T(1) + T(|S|-2) \\ &= \ldots \\ &= \sum_{j=1}^{|S|} C(j) + |S| \cdot T(1) \end{aligned}$$

Computational Complexities: Worst-Case

Single RF Tree: Worst-Case

Let $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ be a training set. In the worst case, we reduce the problem size by only 1 per node split:

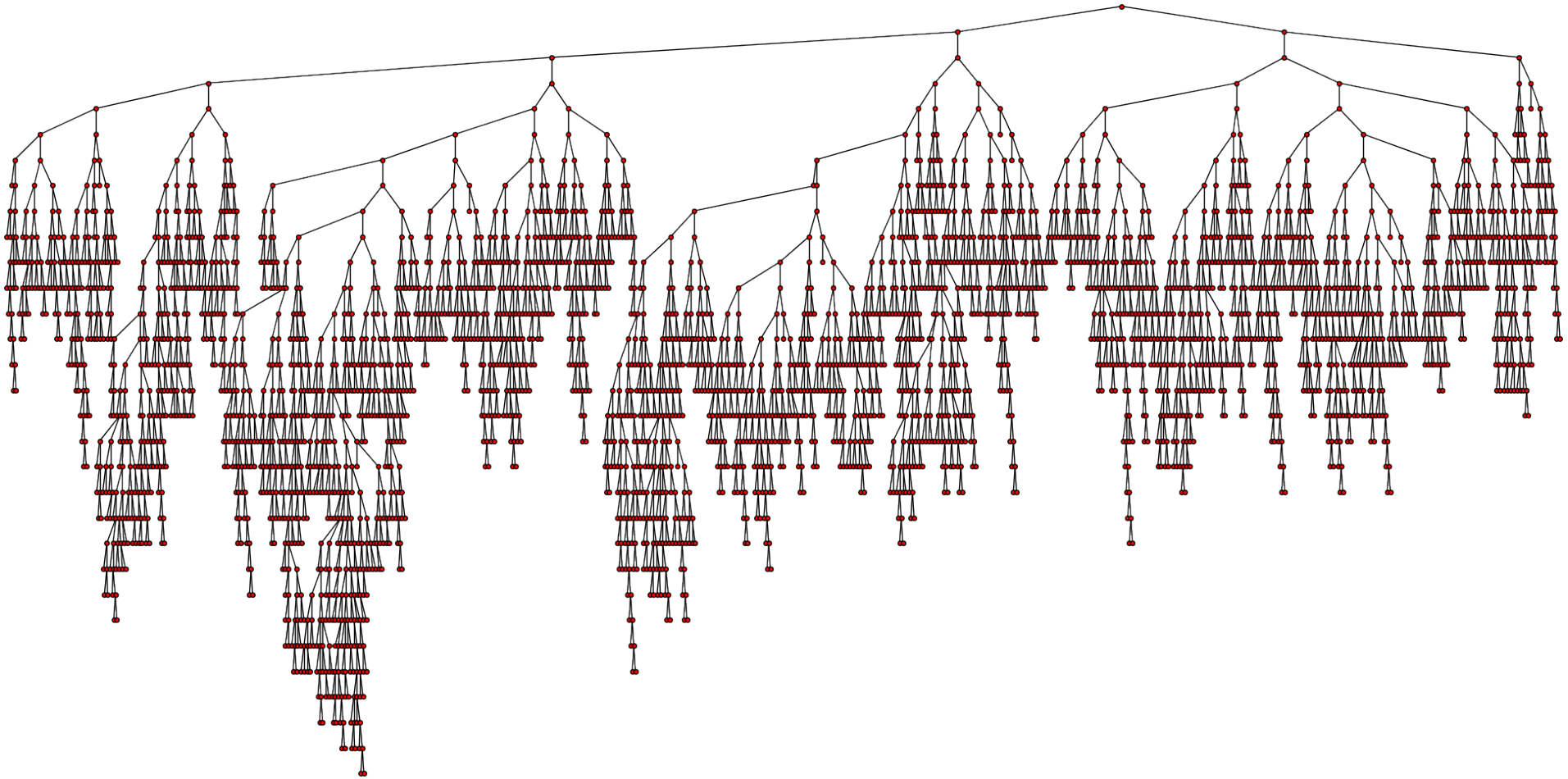
$$\begin{aligned} T(|S|) &= C(|S|) + T(1) + T(|S|-1) \\ &= C(|S|) + T(1) + C(|S|-1) + T(1) + T(|S|-2) \\ &= \ldots \\ &= \sum_{j=1}^{|S|} C(j) + |S| \cdot T(1) \end{aligned}$$

Let us consider Extra Trees: In this case, we have $T(1) \leq c_1$ and $C(j) \leq c_2 \cdot j$ for two constants c_1 and c_2. Therefore:

$$T(|S|) \leq \sum_{j=1}^{|S|} c_2 \cdot j + |S| \cdot c_1 = c_2 \frac{|S| \cdot (|S|+1)}{2} + c_1 |S| \leq c_3 |S|^2$$

for some constant c_3. Hence, one can build a tree in $T(|S|) = O(|S|^2)$ time.

Big Data & Trees

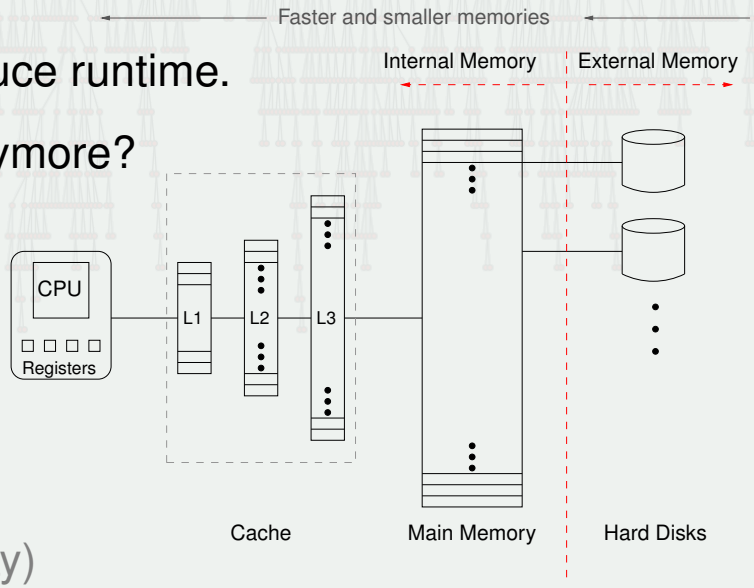


Big Data & Trees

Practical Issues

For many tasks, it is beneficial to train forests with many trees (e.g., 1000 trees) using all the data (e.g., $n = 10^7$). This can become very tricky and is subject of ongoing research.
(Just have a look at some discussion boards on www.kaggle.com!)

1. Training big trees can take a very long time.
2. One cannot easily make use of GPUs to reduce runtime.
3. Often, the data do not fit in main memory anymore?
4. At some point: The trees do not fit in main memory anymore . . .
5. If the data/trees fit in main memory: The movement of the data from main memory to CPU might become the bottleneck!
   (the patterns are accessed in an arbitrary way)
6. Not easy to build forests in a distributed fashion (Hadoop/Spark).
   (building a single tree might already be an issue!)
7. . . .



Scikit-Learn

This documentation is for scikit-learn **version 0.18.1** — Other versions

If you use the software, please consider citing scikit-learn.

3.2.4.3.1. sklearn.ensemble.RandomForestClassifier

class sklearn.ensemble. **RandomForestClassifier** (*n_estimators=10*, *criterion='gini'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_split=1e-07*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*)

[source]

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True (default).

Read more in the User Guide.

Parameters: **n_estimators** : integer, optional (default=10)

The number of trees in the forest.

Scikit-Learn

Implementation Hacks!

The Scikit-Learn implementation is extremely efficient as long as the data/trees fit in main memory. The code is written in Cython (hence: C code). Various "hacks" are used:

1. Trees & Arrays: Each tree is stored in an array (low-level, C-like)
2. Sorting: Optimized sorting algorithm (standard `qsort` $\rightarrow$ factor 2-3 slower)
3. Locally constant features: Keep track of "constant" features during construction.
4. Unique instances: If `bootstrap=True`, only use unique indices/patterns (only 63%) and assign weights to them $\rightarrow$ Speed-up of about 1.5!
5. Consecutive memory access: For evaluating $\mathcal{O}(S)$, prefetch data for dimension i
6. Sparse data: Make use of optimized computations for sparse data.
7. Random numbers: Use manual random number generator ...
8. Parallelization: For random forests, efficient parallelization over trees.
9. ...

In a nutshell: This implementation might easily be a factor of 10-100 faster than a "standard" implementation in C. Speed-up depends on the particular dataset ...

The Secret: Sklearn's Tree Implementation

Branch: master ▾ scikit-learn / sklearn / tree / _tree.pyx

Find file Copy path

NicolasHug MNT Cleaning for fast partial dependence computation (#13738) 4de404d 3 days ago

25 contributors

1233 lines (987 sloc) | 46.2 KB

Raw Blame History

```
1  # cython: cdivision=True
2  # cython: boundscheck=False
3  # cython: wraparound=False
4
5  # Authors: Gilles Louppe <g.louppe@gmail.com>
6  #          Peter Prettenhofer <peter.prettenhofer@gmail.com>
7  #          Brian Holt <bdholt1@gmail.com>
8  #          Noel Dawe <noel@dawe.me>
```

H2O

Random Forest

Random Forest (RF) is a powerful classification tool. When given a set of data, RF generates a forest of classification trees, rather than a single classification tree. Each of these trees generates a classification for a given set of attributes. The classification from each H_2O tree can be thought of as a vote; the most votes determines the classification.

When to use RF

RF is a good choice when your objective is classification.

Defining a Model

Source:

The parsed data set to be used in training a model.

Response:

The dependent variable to be modeled.

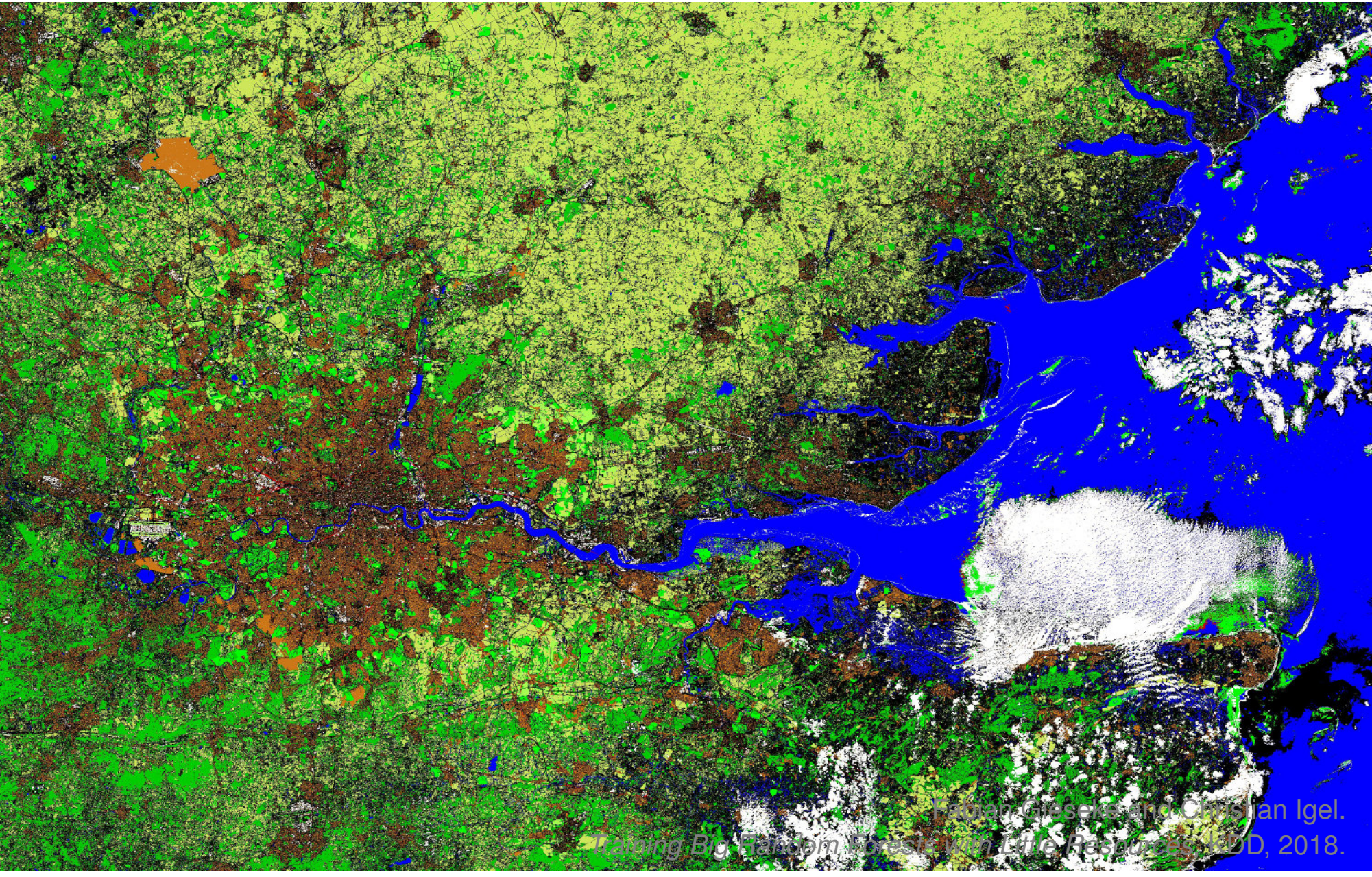
Ignored Columns:

The set of features from the source data set to be omitted from training a model.

Classification:

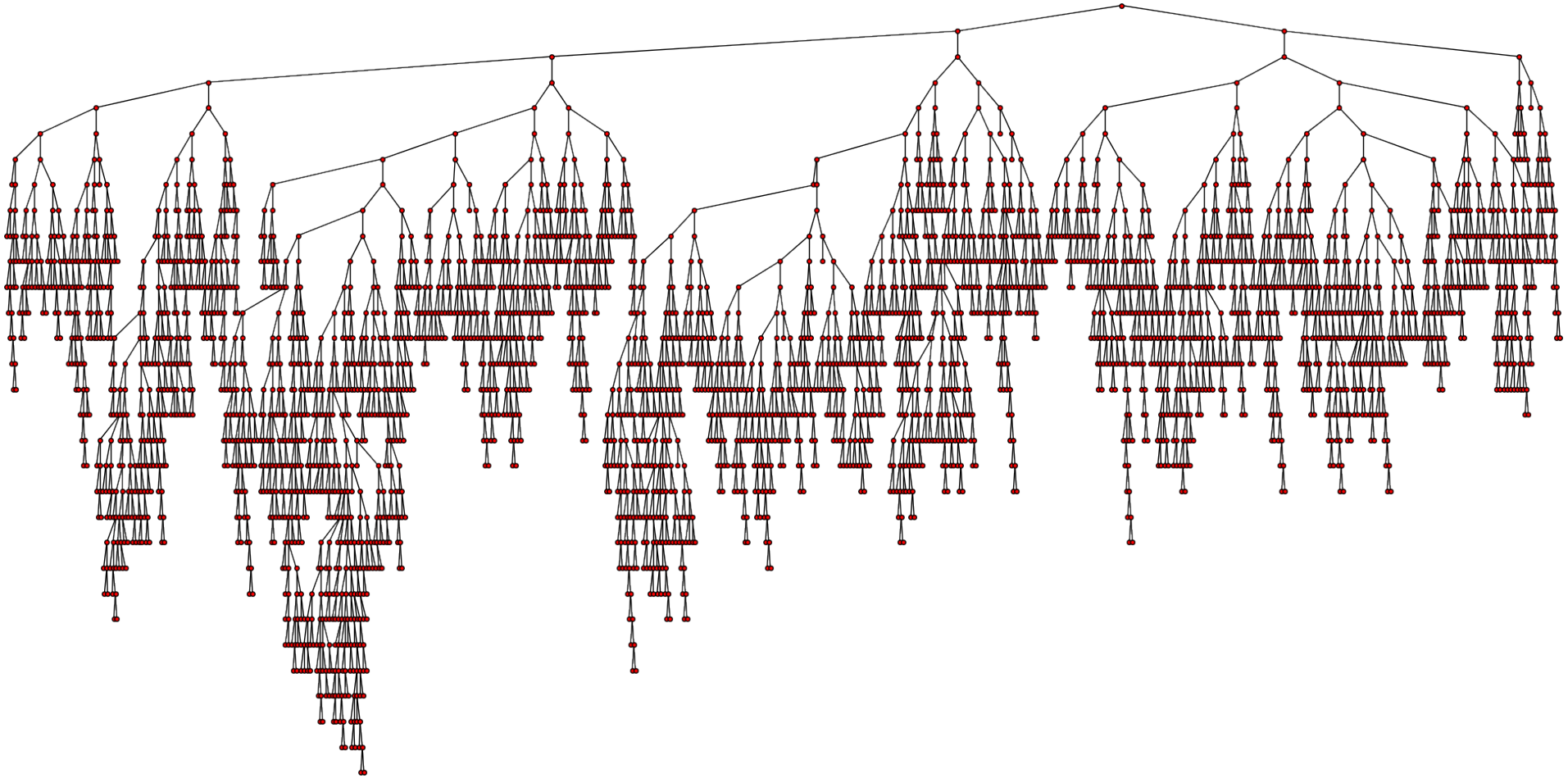
An option that specifies the model to be a classifier when switched on.

Woody: Large-Scale Random Forests



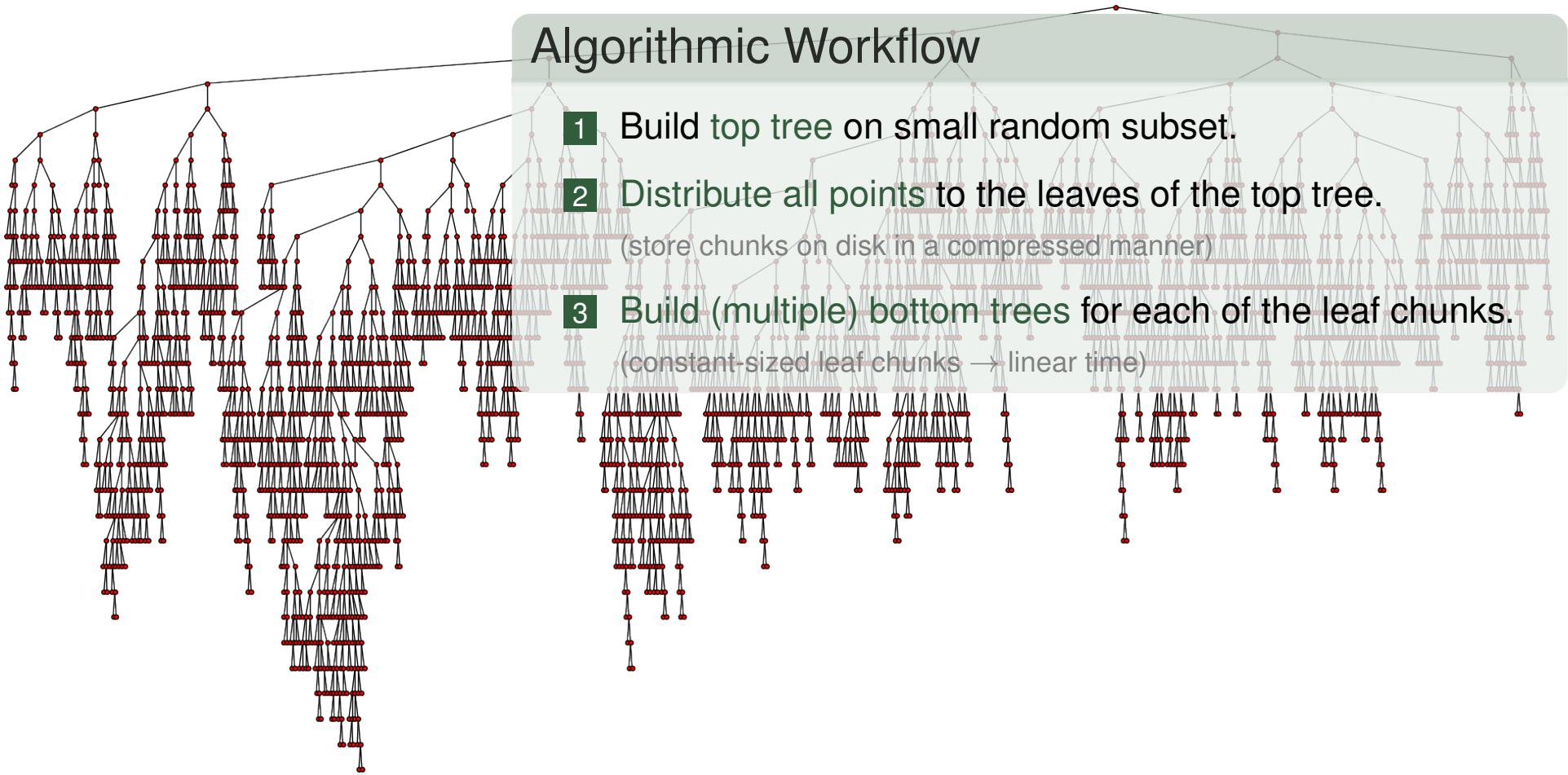
Fabian Gieseke and Christian Igel.
Training Big Random Forests with Little Resources. KDD, 2018.

Woody: Large-Scale Random Forests



Fabian Gieseke and Christian Igel.
Training Big Random Forests with Little Resources, KDD, 2018.

Woody: Large-Scale Random Forests

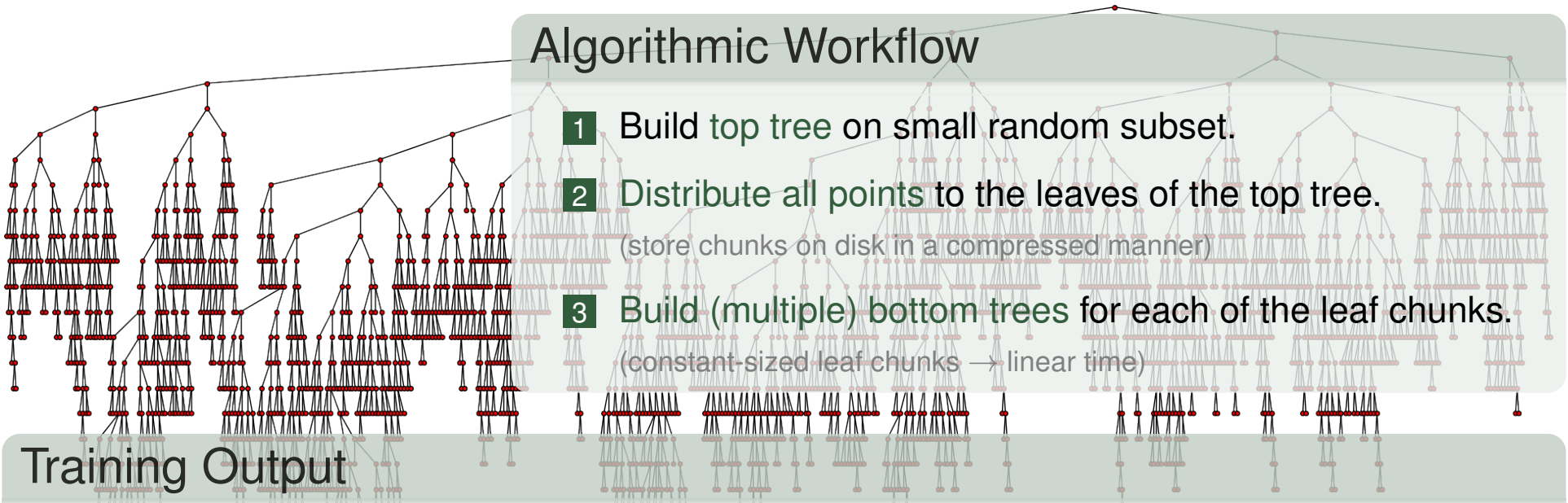


Algorithmic Workflow

1. Build top tree on small random subset.
2. Distribute all points to the leaves of the top tree.
   (store chunks on disk in a compressed manner)
3. Build (multiple) bottom trees for each of the leaf chunks.
   (constant-sized leaf chunks $\rightarrow$ linear time)

Fabian Gieseke and Christian Igel.
Training Big Random Forests with Little Resources, KDD, 2018.

Woody: Large-Scale Random Forests



Algorithmic Workflow

1. Build top tree on small random subset.
2. Distribute all points to the leaves of the top tree.
   (store chunks on disk in a compressed manner)
3. Build (multiple) bottom trees for each of the leaf chunks.
   (constant-sized leaf chunks → linear time)

Training Output

```
13:51:20,588 - Number of training patterns:    113212922
13:51:20,588 - Dimensionality of the data:     11
13:51:20,588 - Fitting forest ...
...
13:59:52,908 - ------------------------------------------------------------------------
13:59:52,908 - (I)   Retrieving subsets:                    53.034 (s)     [10.35  %]
13:59:52,909 - (II)  Top tree constructions:                53.034 (s)     [10.35  %]
13:59:52,909 - (III) Distributing to top tree leaves:       240.208 (s)    [46.89  %]
13:59:52,909 - (IV)  Bottom trees constructions:            219.071 (s)    [42.76  %]
13:59:52,909 -                                              512.312 (s)    [100.00 %]
13:59:52,909 - ------------------------------------------------------------------------
13:51:20,588 - Training time:    512.335904
```


Fabian Gieseke and Christian Igel.
Training Big Random Forests with Little Resources, KDD, 2018.

Outline

Boosting

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Boosting (Round 1)	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2)	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3)	4	4	8	10	4	5	4	6	3	4

Sampling with Replacement II

- An iterative procedure to adaptively change distribution of training data by focusing more on previously misclassified records

Boosting

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Boosting (Round 1)	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2)	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3)	4	4	8	10	4	5	4	6	3	4

Sampling with Replacement II

- An iterative procedure to adaptively change distribution of training data by focusing more on previously misclassified records
- Initially, all *n* records are assigned equal weights

Boosting

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Boosting (Round 1)	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2)	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3)	4	4	8	10	4	5	4	6	3	4

Sampling with Replacement II

- An iterative procedure to adaptively change distribution of training data by focusing more on previously misclassified records
- Initially, all *n* records are assigned equal weights
- Unlike bagging, weights may change at the end of boosting round
  - Records that are wrongly classified will have their weights increased
  - Records that are classified correctly will have their weights decreased

Boosting

http://www-users.cs.umn.edu/~kumar/dmbook

Original Data	1	2	3	4	5	6	7	8	9	10
Boosting (Round 1)	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2)	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3)	4	4	8	10	4	5	4	6	3	4

Sampling with Replacement II

- An iterative procedure to adaptively change distribution of training data by focusing more on previously misclassified records
- Initially, all *n* records are assigned equal weights
- Unlike bagging, weights may change at the end of boosting round
  - Records that are wrongly classified will have their weights increased
  - Records that are classified correctly will have their weights decreased

Example: Record with id 4 is difficult to classify correctly. Therefore, its sampling weight is increased (more likely to be selected).

AdaBoost

AdaBoost Algorithm

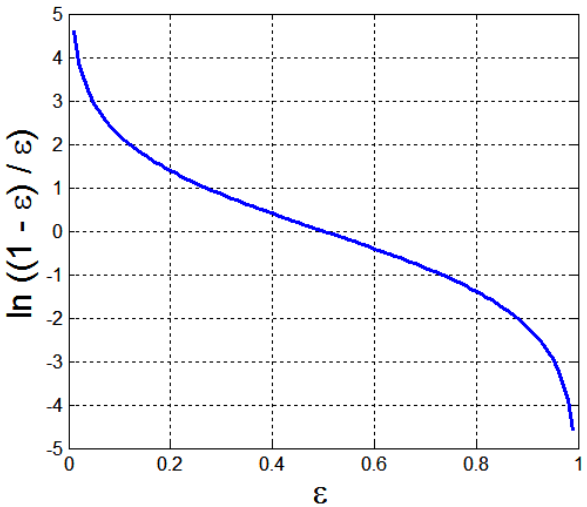
Require: Let $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ be the set of training examples and B the number of boosting rounds.

1: $\mathbf{w} = (\frac{1}{n}, \ldots, \frac{1}{n}) \in \mathbb{R}^n$ [same weight for all examples]
2: **for** $i = 1, \ldots, B$ **do**
3: Create bootstrap sample D_i from D according to $\mathbf{w}$
4: Train a base classifier C_i on D_i
5: Compute error ε_i of C_i based on D [entire original data set!]
6: **if** $\varepsilon_i > 0.5$ **then**
7: Reset weights, i.e., $\mathbf{w} = (\frac{1}{n}, \ldots, \frac{1}{n}) \in \mathbb{R}^n$
8: Go back to Step 3
9: **end if**
10: Determine importance of classifier α_i
11: Update weight vector $\mathbf{w}$
12: **end for**

Final Model: The ensemble is given by $C^*(\mathbf{x}) = \text{argmax}_y \sum_{i=1}^{B} \alpha_i \mathbb{I}(C_i(\mathbf{x}) = y)$, where $\alpha_1, \ldots, \alpha_B$ determine the importances of the individual classifiers.

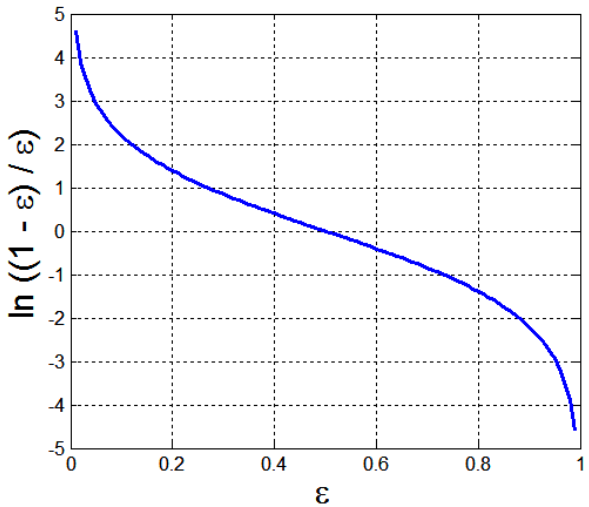
AdaBoost

- The error rate ε_i for a C_i is given by $\varepsilon_i = \sum_{j=1}^{n} w_j \mathbb{I}(C_i(\mathbf{x}_j) \neq y_j)$
- The importance α_i of a classifier is defined as $\alpha_i = \frac{1}{2} \ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$



AdaBoost

- The error rate ε_i for a C_i is given by $\varepsilon_i = \sum_{j=1}^{n} w_j \mathbb{I}(C_i(\mathbf{x}_j) \neq y_j)$
- The importance α_i of a classifier is defined as $\alpha_i = \frac{1}{2} \ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$



- The weights $w_1, \ldots, w_n$ are updated in each round via

$$w_j^{(i+1)} = \frac{w_j^{(i)}}{Z_{i+1}} \cdot \begin{cases} e^{-\alpha_i} & \text{if } C_i(\mathbf{x}_j) = y_j \\ e^{\alpha_i} & \text{if } C_i(\mathbf{x}_j) \neq y_j \end{cases}$$

where Z_{i+1} is a normalization constant to ensure $\sum_{j=1}^{n} w_j^{(i+1)} = 1$.

Demo Time!

jupyter **Adaboost** Last Checkpoint: a few seconds ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

This example shows how to fit boosted trees via the AdaBoost algorithm. See http://scikit-learn.org/stable/auto_examples/ensemble/plot_adaboost_twoclass.html#sphx-glr-auto-examples-ensemble-plot-adaboost-twoclass-py

In [1]:
```
%matplotlib inline
import numpy as np
np.random.seed(0)
import matplotlib.pyplot as plt
```

Let us generate a simple two-dimensional dataset ...

In [2]:
```
from sklearn.datasets import make_gaussian_quantiles

# Construct dataset
X1, y1 = make_gaussian_quantiles(cov=2.,
                                 n_samples=200, n_features=2,
                                 n_classes=2, random_state=1)
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5,
                                 n_samples=300, n_features=2,
                                 n_classes=2, random_state=1)
X = np.concatenate((X1, X2))
y = np.concatenate((y1, - y2 + 1))
```

In [3]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# instantiate the model
model = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
                           algorithm="SAMME",
                           n_estimators=1)

# fit the model
model.fit(X, y)
```

Out[3]:
```
AdaBoostClassifier(algorithm='SAMME',
          base_estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=1,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          learning_rate=1.0, n_estimators=1, random_state=None)
```

In [4]:
```
plot_colors = "br"
plot_step = 0.02
class_names = "AB"

plt.figure(figsize=(8, 8))
```

Multi-Class AdaBoost (SAMME)

- So far: Reset the weights in case $\varepsilon > 0.5$.
  - Problem: For many classes K, this is difficult for "simple" classifiers!
  - Random guessing: Error of about $\frac{1}{K}$ (balanced dataset).

Multi-Class AdaBoost (SAMME)

- So far: Reset the weights in case $\varepsilon > 0.5$.
  - Problem: For many classes K, this is difficult for "simple" classifiers!
  - Random guessing: Error of about $\frac{1}{K}$ (balanced dataset).
- Modification I: Only reset weights if $\varepsilon_i > 1 - \frac{1}{K}$
- Modification II: The importance α_i of a classifier is now defined as

$$\alpha_i = \ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right) + \ln(K-1)$$

Multi-Class AdaBoost (SAMME)

- So far: Reset the weights in case $\varepsilon > 0.5$.
  - Problem: For many classes K, this is difficult for "simple" classifiers!
  - Random guessing: Error of about $\frac{1}{K}$ (balanced dataset).
- Modification I: Only reset weights if $\varepsilon_i > 1 - \frac{1}{K}$
- Modification II: The importance α_i of a classifier is now defined as

$$\alpha_i = \ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right) + \ln(K-1)$$

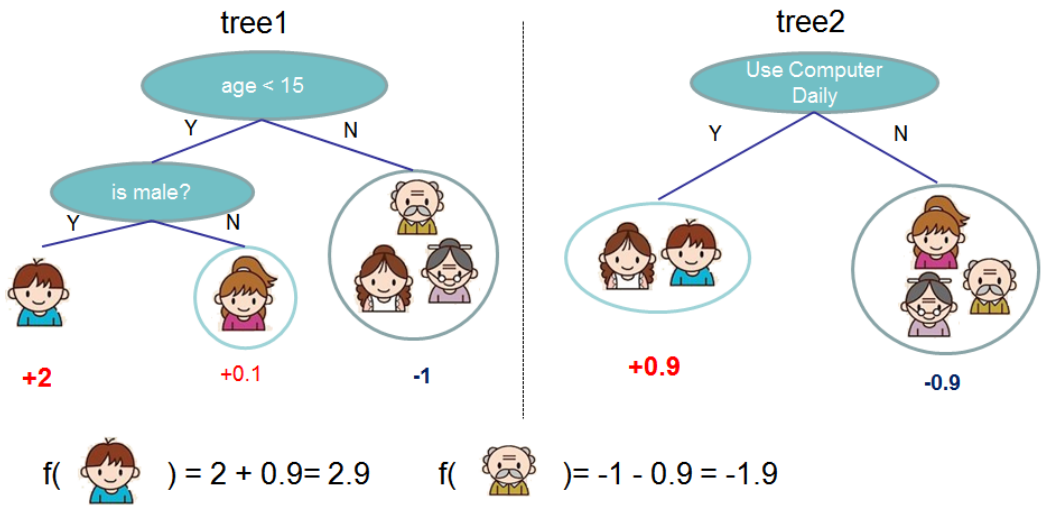
- Thus, in order for α_i to be positive, we only need $\varepsilon_i \leq 1 - \frac{1}{K}$

Zhu, Zou, Rosset, Hastie: *Multi-class AdaBoost*, Statistics and Its Interface, 2009.

XGBoost

Task

- Training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$

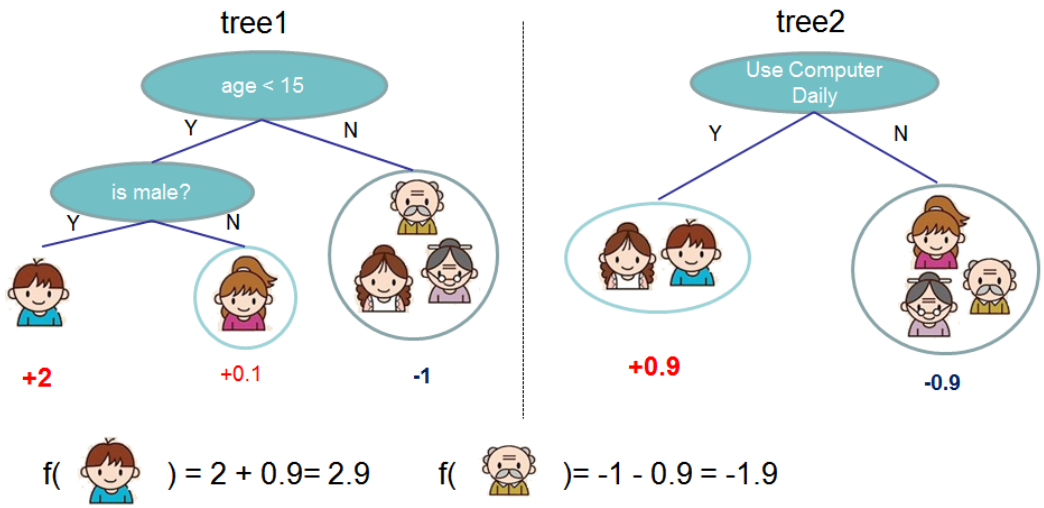


https://xgboost.readthedocs.io

XGBoost

Task

- Training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$
- The final model $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ is based on K trees $f_k \in \mathcal{F}$, where $\mathcal{F}$ is the space of decision trees.



https://xgboost.readthedocs.io

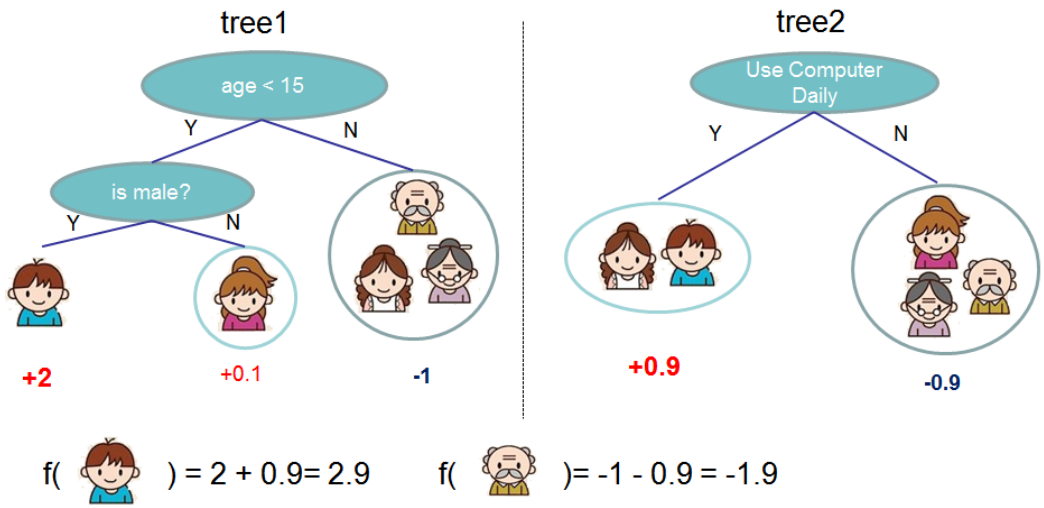
XGBoost

Task

- Training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$
- The final model $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ is based on K trees $f_k \in \mathcal{F}$, where $\mathcal{F}$ is the space of decision trees.
- Learning goal: Find K such trees that minimize the objective

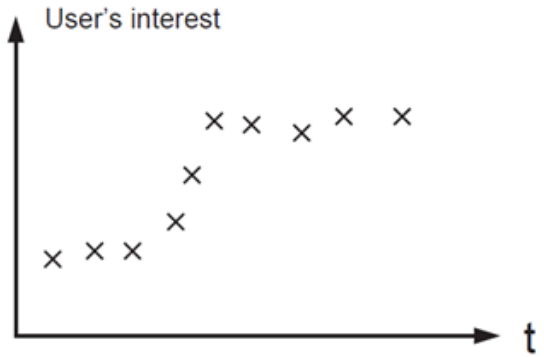
$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

where $\mathcal{L}$ is a loss function and $\Omega(f_k)$ a measure for the complexity of a tree.

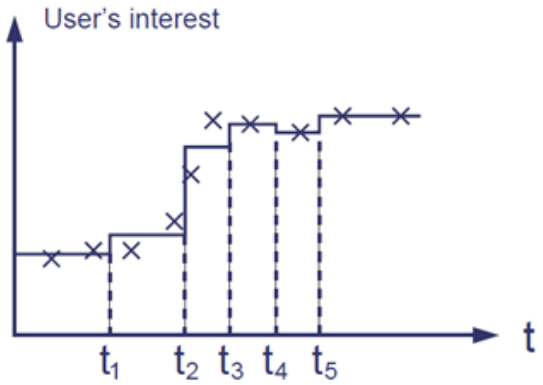


https://xgboost.readthedocs.io

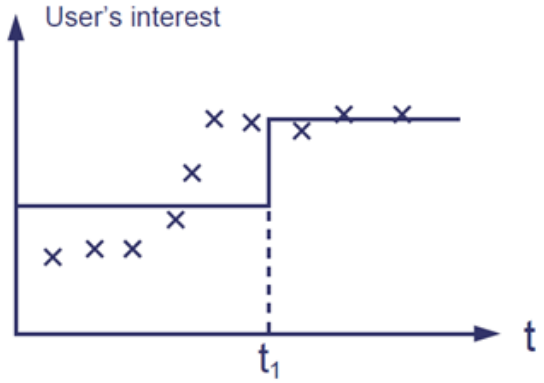
Trade-Off: Small Loss vs. Complexity



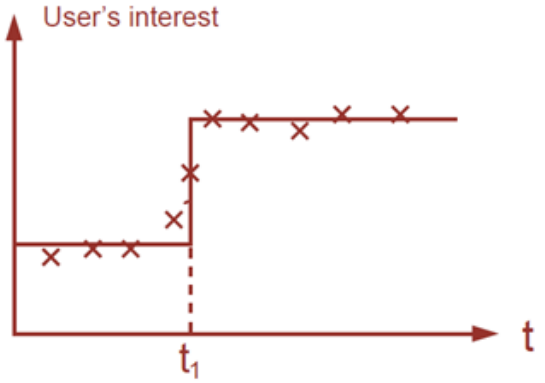
Observed user's interest on topic k against time t



☒ Too many splits, $\Omega(f)$ is high



☒ Wrong split point, $L(f)$ is high



☑ Good balance of $\Omega(f)$ and $L(f)$

https://xgboost.readthedocs.io

XGBoost: Goal & Training

Objective

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_k)$ specifying the complexity of the tree f_k.

Comment: It is, in general, difficult to find a "global" optimal solution.

XGBoost: Goal & Training

Objective

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_k)$ specifying the complexity of the tree f_k.

Comment: It is, in general, difficult to find a "global" optimal solution.

Additive Training (Boosting): Start with constant predictions and add a new function (tree) in each iteration:

1. $\hat{y}_i^{(0)} = 0$

XGBoost: Goal & Training

Objective

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_k)$ specifying the complexity of the tree f_k.

Comment: It is, in general, difficult to find a "global" optimal solution.

Additive Training (Boosting): Start with constant predictions and add a new function (tree) in each iteration:

1. $\hat{y}_i^{(0)} = 0$
2. $\hat{y}_i^{(1)} = f_1(\mathbf{x}_i) = \hat{y}_i^{(0)} + f_1(\mathbf{x}_i)$

XGBoost: Goal & Training

Objective

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_k)$ specifying the complexity of the tree f_k.

Comment: It is, in general, difficult to find a "global" optimal solution.

Additive Training (Boosting): Start with constant predictions and add a new function (tree) in each iteration:

1. $\hat{y}_i^{(0)} = 0$
2. $\hat{y}_i^{(1)} = f_1(\mathbf{x}_i) = \hat{y}_i^{(0)} + f_1(\mathbf{x}_i)$
3. $\hat{y}_i^{(2)} = f_1(\mathbf{x}_i) + f_2(\mathbf{x}_i) = \hat{y}_i^{(1)} + f_2(\mathbf{x}_i)$

XGBoost: Goal & Training

Objective

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_k)$ specifying the complexity of the tree f_k.

Comment: It is, in general, difficult to find a "global" optimal solution.

Additive Training (Boosting): Start with constant predictions and add a new function (tree) in each iteration:

1. $\hat{y}_i^{(0)} = 0$
2. $\hat{y}_i^{(1)} = f_1(\mathbf{x}_i) = \hat{y}_i^{(0)} + f_1(\mathbf{x}_i)$
3. $\hat{y}_i^{(2)} = f_1(\mathbf{x}_i) + f_2(\mathbf{x}_i) = \hat{y}_i^{(1)} + f_2(\mathbf{x}_i)$
4. ...

XGBoost: Goal & Training

Objective

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_k)$ specifying the complexity of the tree f_k.

Comment: It is, in general, difficult to find a "global" optimal solution.

Additive Training (Boosting): Start with constant predictions and add a new function (tree) in each iteration:

1. $\hat{y}_i^{(0)} = 0$
2. $\hat{y}_i^{(1)} = f_1(\mathbf{x}_i) = \hat{y}_i^{(0)} + f_1(\mathbf{x}_i)$
3. $\hat{y}_i^{(2)} = f_1(\mathbf{x}_i) + f_2(\mathbf{x}_i) = \hat{y}_i^{(1)} + f_2(\mathbf{x}_i)$
4. $\ldots$
5. $\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)$

Simplifying the Objective

- At each step t, we have $\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)$
  - Keep functions $\hat{y}_i^{(t-1)}$ that were added in the previous rounds ...
  - Find new function $f_t(\mathbf{x}_i)$ that minimizes the objective!

Simplifying the Objective

- At each step t, we have $\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)$
  - Keep functions $\hat{y}_i^{(t-1)}$ that were added in the previous rounds ...
  - Find new function $f_t(\mathbf{x}_i)$ that minimizes the objective!
- We want to minimize

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{k=1}^{t} \Omega(f_k) \\ &= \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)\right) + \Omega(f_t) + \text{constant} \\ &= \dots \end{aligned}$$

Simplifying the Objective

- At each step t, we have $\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)$
  - Keep functions $\hat{y}_i^{(t-1)}$ that were added in the previous rounds ...
  - Find new function $f_t(\mathbf{x}_i)$ that minimizes the objective!
- We want to minimize

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{k=1}^{t} \Omega(f_k) \\ &= \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)\right) + \Omega(f_t) + \text{constant} \\ &= \dots \end{aligned}$$

(to be continued L12/L13)

XGBoost: In a Nutshell

Objective and Parameters

Find an ensemble $\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$ of K trees $f_k \in \mathcal{F}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

with $\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$.

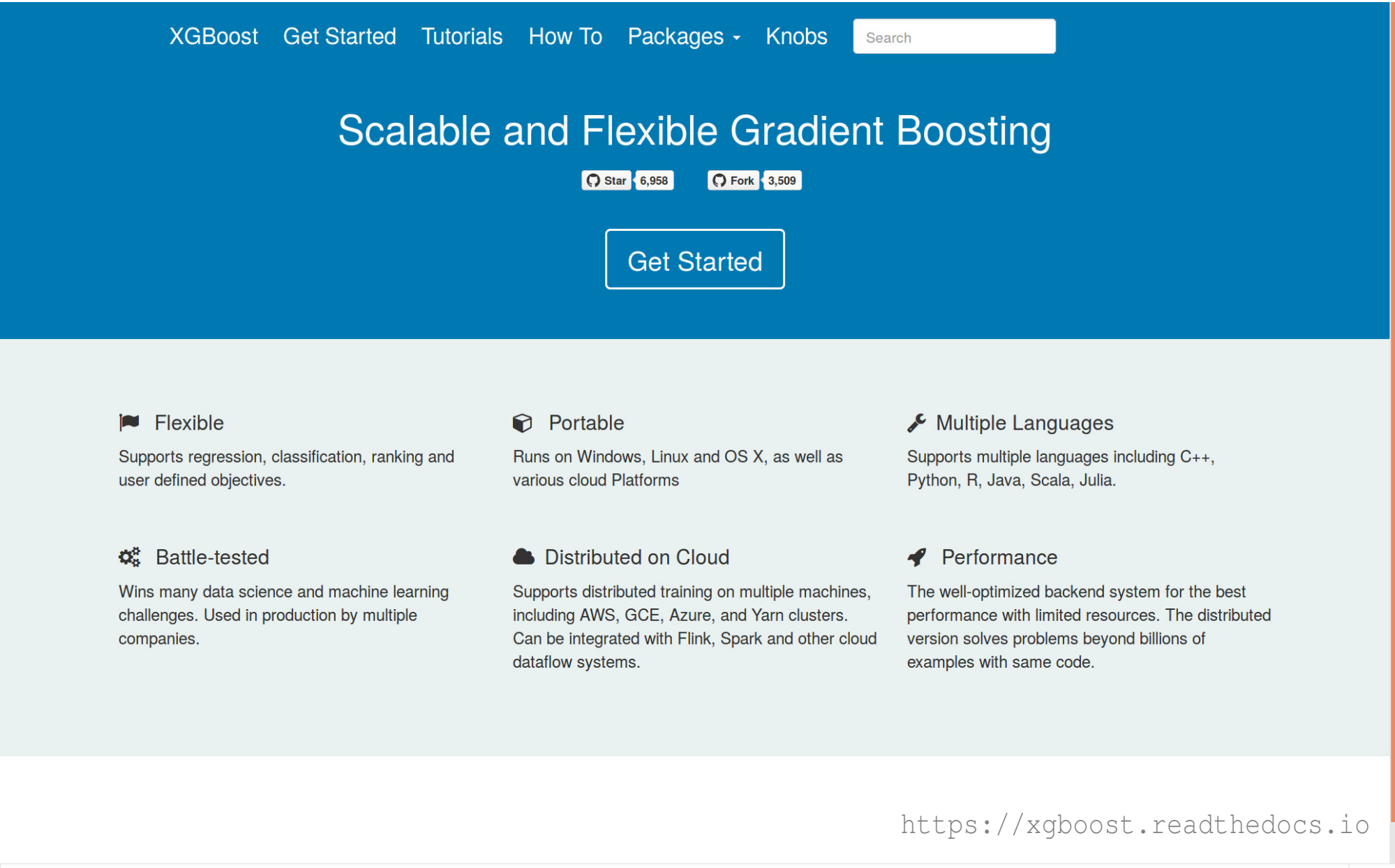
- K specifies the number of trees to be fitted (e.g., 100).
- One can restrict the trees to have a maximum depths (e.g., 5).
- One usually specifies a learning rate η (default $\eta = 0.3$):

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta \cdot f_t(\mathbf{x}_i)$$

- One can adapt γ and λ (see above)
- One can consider column-subsampling (proportion of features to be checked per tree (e.g., 0.3).
- ...

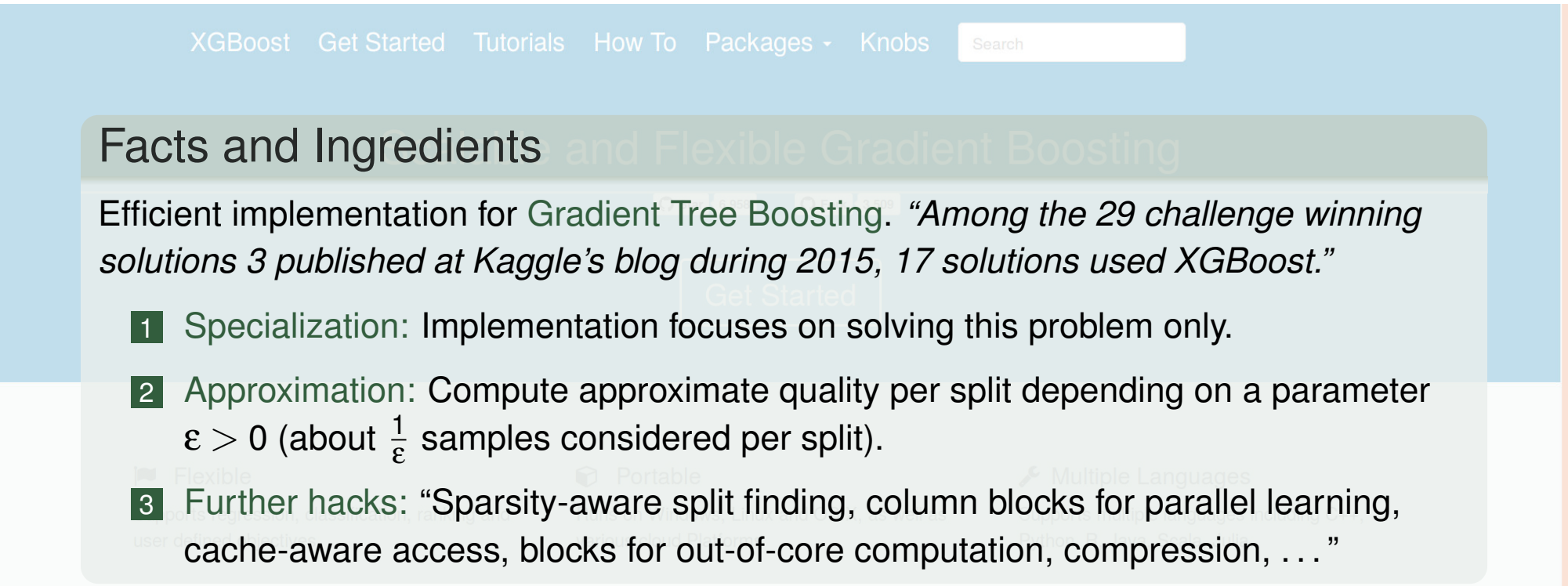
https://xgboost.readthedocs.io/en/latest/python/python_api.html

XGBoost: Implementation



https://xgboost.readthedocs.io

XGBoost: Implementation



Facts and Ingredients

Efficient implementation for Gradient Tree Boosting. *"Among the 29 challenge winning solutions 3 published at Kaggle's blog during 2015, 17 solutions used XGBoost."*

1. Specialization: Implementation focuses on solving this problem only.
2. Approximation: Compute approximate quality per split depending on a parameter $\varepsilon > 0$ (about $\frac{1}{\varepsilon}$ samples considered per split).
3. Further hacks: "Sparsity-aware split finding, column blocks for parallel learning, cache-aware access, blocks for out-of-core computation, compression, ..."

Tianqi Chen and Carlos Guestrin. *XGBoost: A Scalable Tree Boosting System*, KDD, 2016.

Demo Time!

jupyter XGBoost Boston Last Checkpoint: a few seconds ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

This exercise is about applying XGBoost to a well-known dataset, the Boston Housing Dataset (https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html). In the first step, you will load and quickly analyze the dataset. In the second step, you will training and apply a XGBoost regression model.

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt

import numpy as np
np.random.seed(0)
```

Let's load the dataset. Luckily, the Scikit-Learn package provides a very simple way to load the Boston dataset ...

In [2]:
```
from sklearn.datasets import load_boston

boston = load_boston()

print("Number of instances: {}".format(boston.data.shape[0]))
print("Number of features: {}".format(boston.data.shape[1]))

# YOUR TASK: Print details related to the dataset
# print(boston.DESCR)
```

```
Number of instances: 506
Number of features: 13
```

Let's quickly analyze the main characteristics of the data at hand. The Pandas package (https://pandas.pydata.org/) provides some nice out-of-the-box methods to quickly analyze data and to do some preprocessing (if needed).

In [3]:
```
import pandas as pd

df = pd.DataFrame(boston.data)
df.columns = boston.feature_names
df['PRICE (REGRESSION TARGET)'] = boston.target
df.head()
```

Out[3]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE (REGRESSION TARGET)
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4

Outline

Summary & Outlook

Today

1. Large-Scale Random Forests
2. AdaBoost & XGBoost I

Outlook

- Thursday (Lecture): Neural Networks (CI)
- Tuesday (Lecture): Neural Networks (CI)