

[544] Spark MLlib

Tyler Caraza-Harter

Learning Objectives

- perform common machine learning tasks (train/test split, preprocessing, pipelining, training, prediction, and evaluation) using Spark MLlib
- describe how decision trees make predictions
- describe how the PLANET algorithm (using by Spark decision trees) efficiently trains on large, distributed data

Outline

ML Overview

Training/Predicting APIs

Demos

Decision Trees

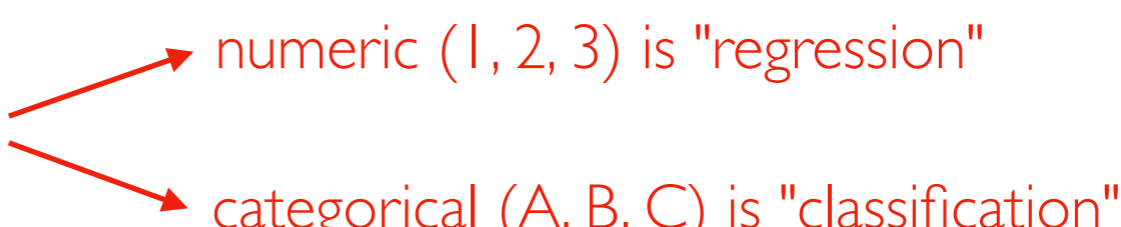
Machine Learning, Major Ideas

Classic Categories of Machine Learning:

- **Reinforcement learning:** agent makes series of actions to maximize reward
- **Unsupervised learning:** looking for general patterns
- **Supervised learning:** train models to predict outputs given inputs

Models are functions that return predictions:

```
def my_model(some_info):  
    ...  
    return some_prediction
```



numeric (1, 2, 3) is "regression"
categorical (A, B, C) is "classification"

Example:

```
def weather_forecast(temp_today, temp_yesterday):  
    ...  
    return temp_tomorrow
```

Machine Learning, Major Ideas

Classic Categories of Machine Learning:

- **Reinforcement learning:** agent makes series of actions to maximize reward
- **Unsupervised learning:** looking for general patterns
- **Supervised learning:** train models to predict outputs given inputs

Models are functions that return predictions:

```
def my_model(some_info) :  
    ...  
    return some_prediction
```

computation usually involves some calculations (multiply, add) with various numbers (parameters). Training is finding parameters that result in good predictions for known training data

Example:

```
def weather_forecast(temp_today, temp_yesterday) :  
    ...  
    return temp_tomorrow
```

Learning from Data

	x1	x2	y
0	2	8	5
1	9	2	6
2	4	1	0
3	7	9	7
4	2	2	3
5	3	4	3
6	3	5	9
7	7	1	4
8	6	6	3
9	4	3	?
10	1	2	?
11	2	9	?

- feature columns: x1 and x2
- label column: y

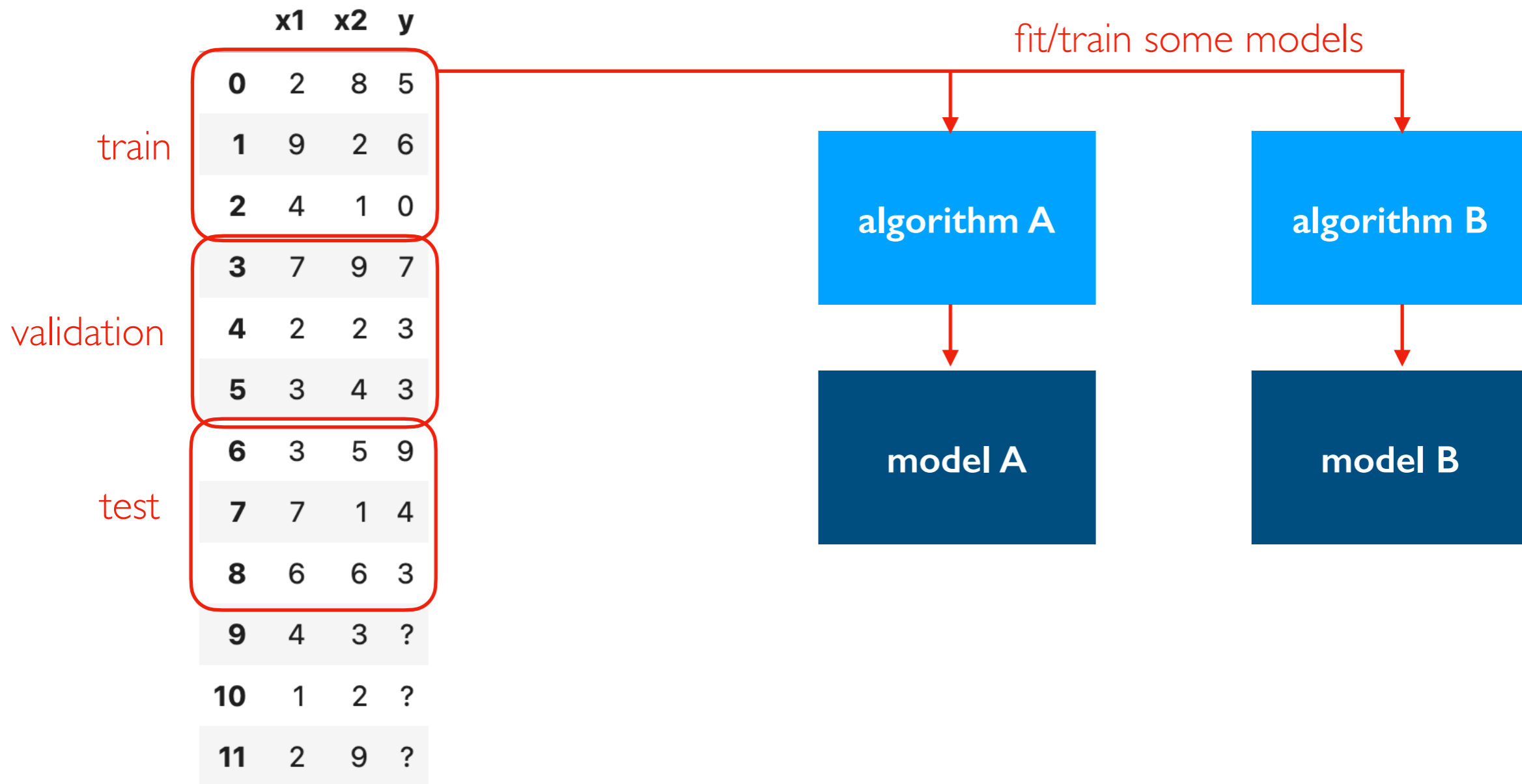
how can the cases where we DO know y help us predict the cases where we do not?

Learning from Data

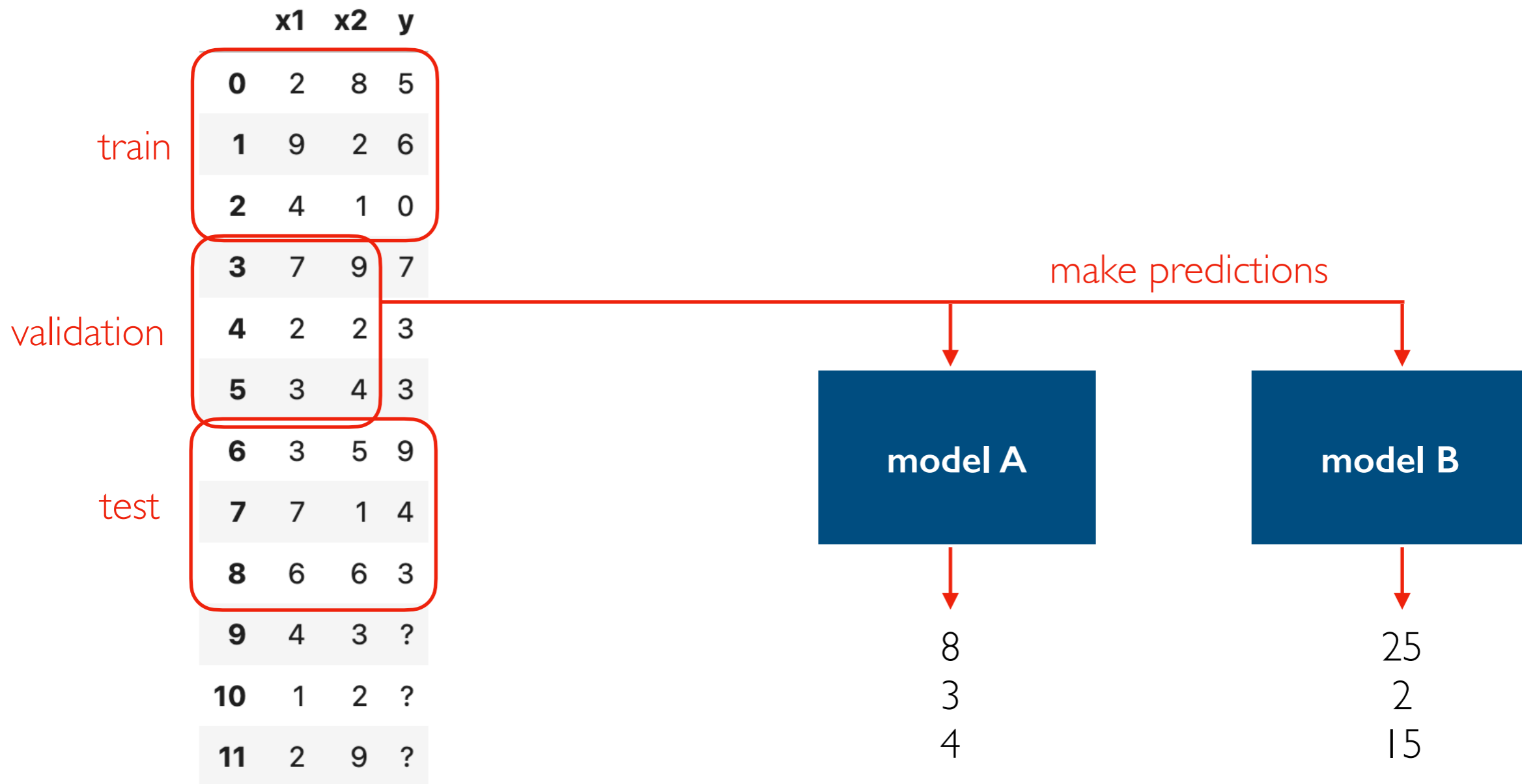
	x1	x2	y	
train	0	2	8	5
	1	9	2	6
	2	4	1	0
validation	3	7	9	7
	4	2	2	3
	5	3	4	3
test	6	3	5	9
	7	7	1	4
	8	6	6	3
	9	4	3	?
	10	1	2	?
	11	2	9	?

random split

Learning from Data



Learning from Data



Learning from Data

	x1	x2	y	
train	0	2	8	5
	1	9	2	6
	2	4	1	0
validation	3	7	9	7
	4	2	2	3
	5	3	4	3
test	6	3	5	9
	7	7	1	4
	8	6	6	3
	9	4	3	?
	10	1	2	?
	11	2	9	?

which model predicts better?

winner!

model A

8
3
4

model B

25
2
15

Learning from Data

	x1	x2	y	
train	0	2	8	5
	1	9	2	6
	2	4	1	0
validation	3	7	9	7
	4	2	2	3
	5	3	4	3
test	6	3	5	9
	7	7	1	4
	8	6	6	3
	9	4	3	?
	10	1	2	?
	11	2	9	?

why might the winning model do worse on the test data than the validation data?

winner!

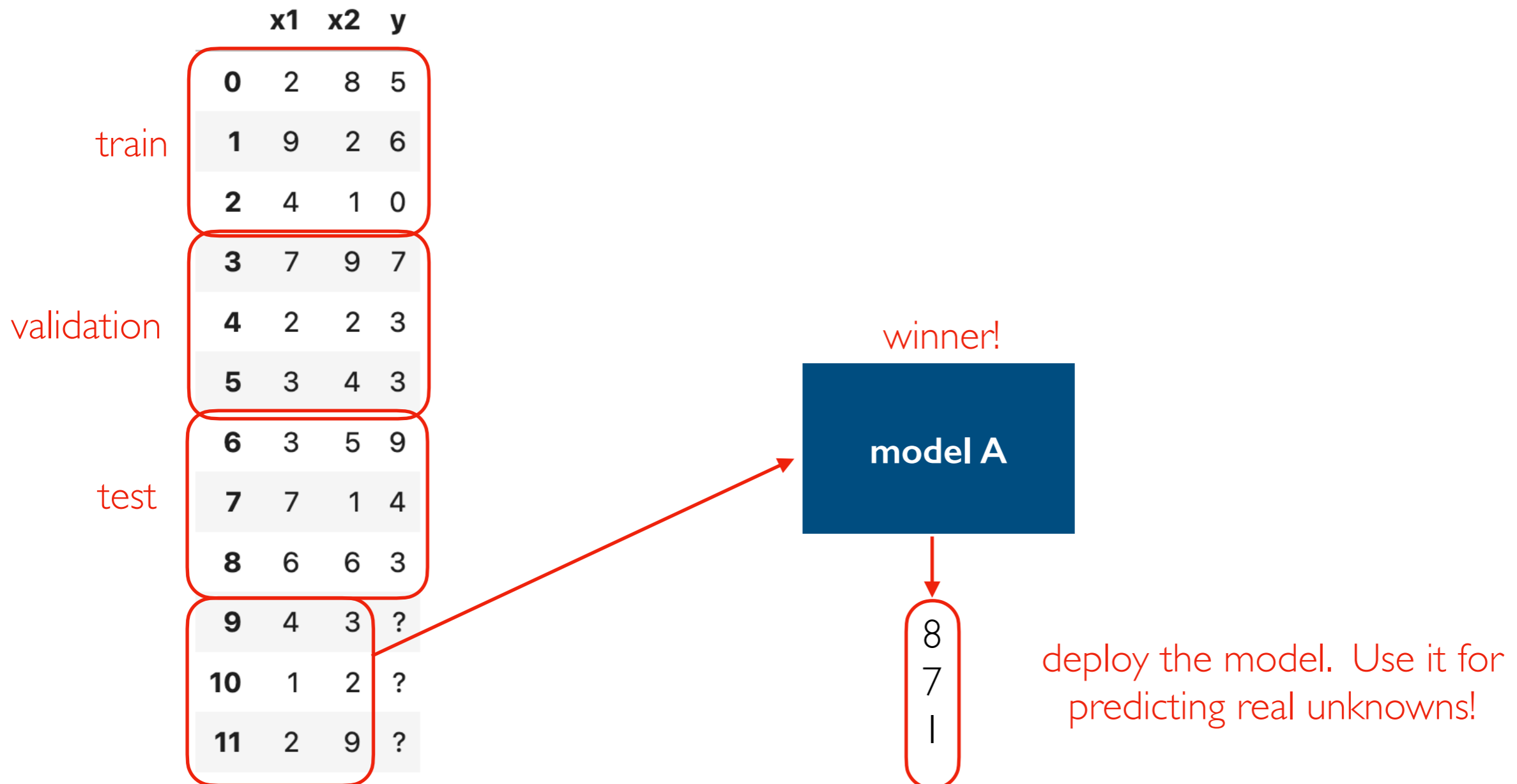


10
3
3

how good does the chosen model do on the test data?

models that do good on train data but bad on validation/test data have "overfitted"

Learning from Data



Outline

ML Overview

Training/Predicting APIs

- sklearn
- PyTorch
- Spark MLlib

Demos

Decision Trees

Training

scikit-learn

```
model = ????  
model.fit(X, y)  
# model parameters can relate X to y
```

pytorch

```
model = ????  
# TODO: optimizer, loss function  
# training loop  
for epoch in range(????):  
    for X, y in ????:  
        ...  
# model parameters can relate X to y
```

- models are **mutable**
- fitting sets/improves parameters

Spark MLlib

```
unfit_model = ????  
fit_model = unfit_model.fit(df)  
# fit_model params can relate x to y
```

- models are **immutable**
- fitting returns new model object

Predicting

scikit-learn

```
y = model.predict(X)
```

pytorch

```
y = model(X)
```

Spark MLlib

```
df2 = fit_model.transform(df)
```

Data

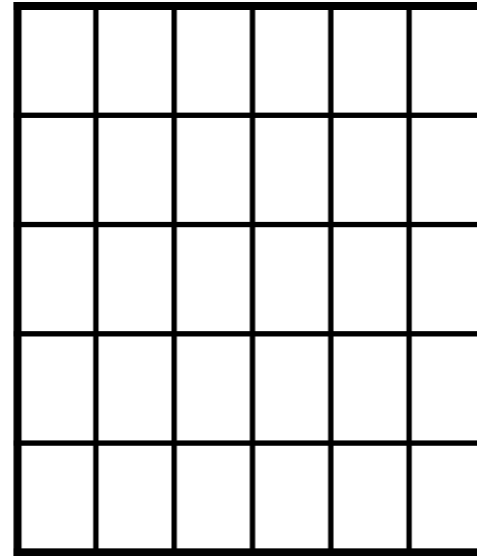
scikit-learn

```
y = model.predict(X)
```

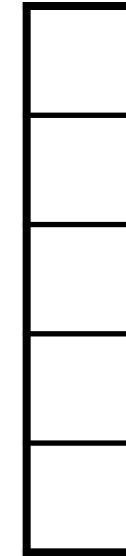
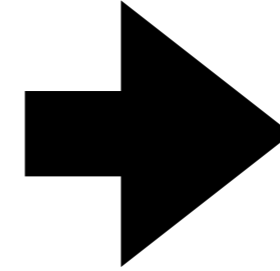
pytorch

```
y = model(X)
```

X (features)



y (label)

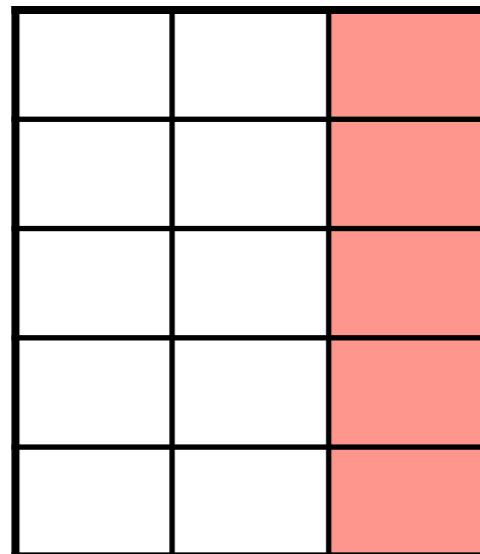


Spark MLlib

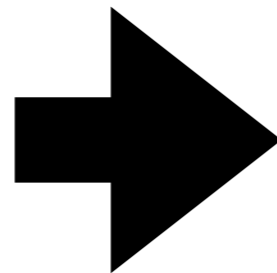
```
df2 = fit_model.transform(df)
```

X (features)

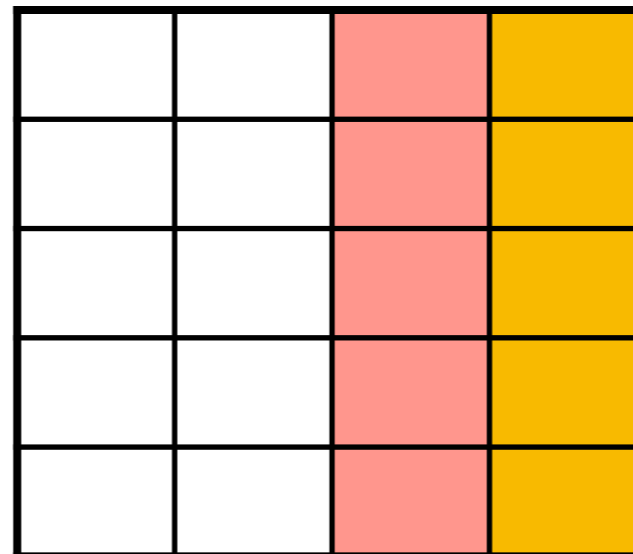
df



df2

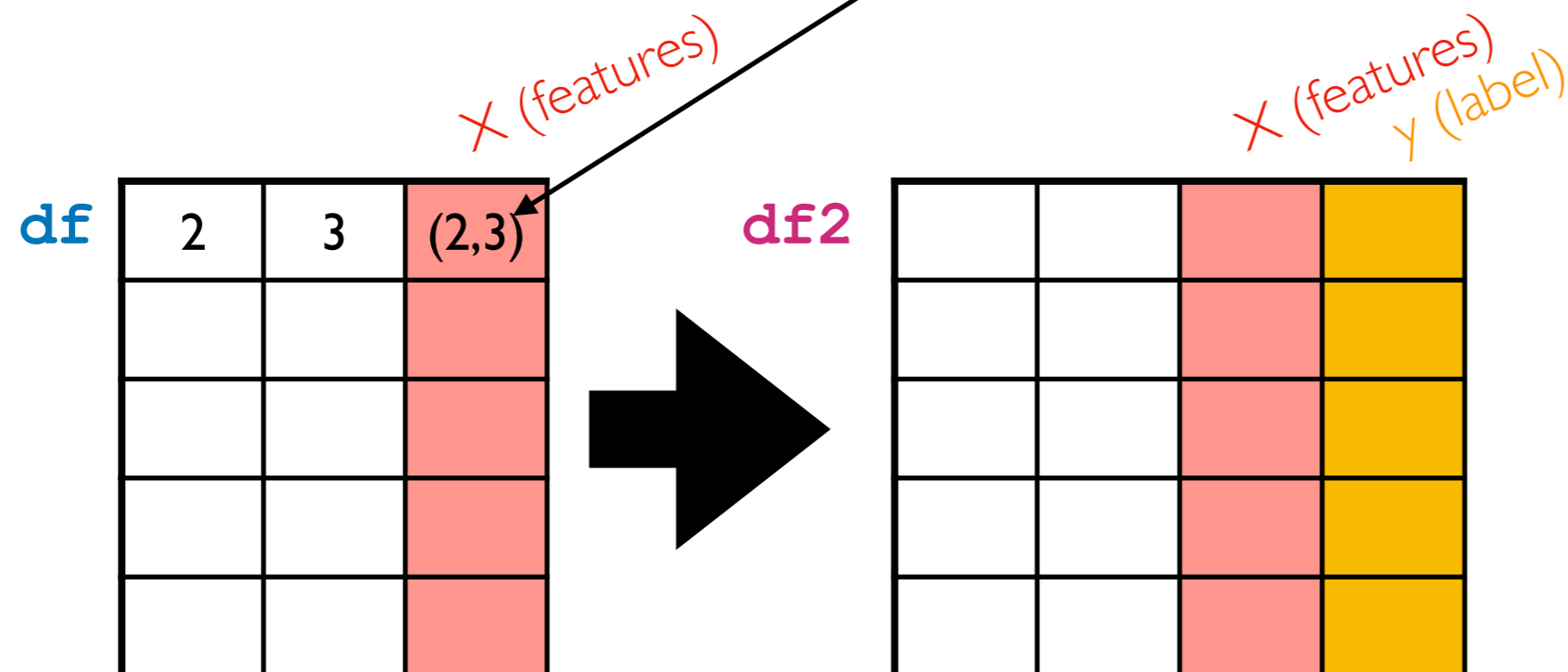


X (features)
y (label)



Features Column

- we only get one features column
- it usually contains vectors
- those vectors typically contain values from other columns
- example:
(2, 3)



Terminology

Spark and scikit-learn use many of the same terms, with very different meaning.

Transformer (scikit-learn)

- object has `.transform` method
- takes a DataFrame, returns a different DataFrame
- used as **preprocessing step** for a model

Transformer (Spark)

- object has `.transform` method
- takes a DataFrame, returns original **with 1 or more additional columns**
- **a fitted model is a transformer** that adds a prediction column

Estimator (scikit-learn)

- object has `.fit` and `.predict` methods
- `.fit` **modifies** the object
- makes predictions after learning params

Estimator (Spark)

- object has `.fit` method that **returns new object**
- an **unfitted model** is an estimator; calling `.fit` returns a **fitted model** (a transformer)

Pipeline

Both scikit-learn and Spark: a pipeline is a series of stages (transformers/estimators). fit/transform/etc. are called as appropriate on each stage.

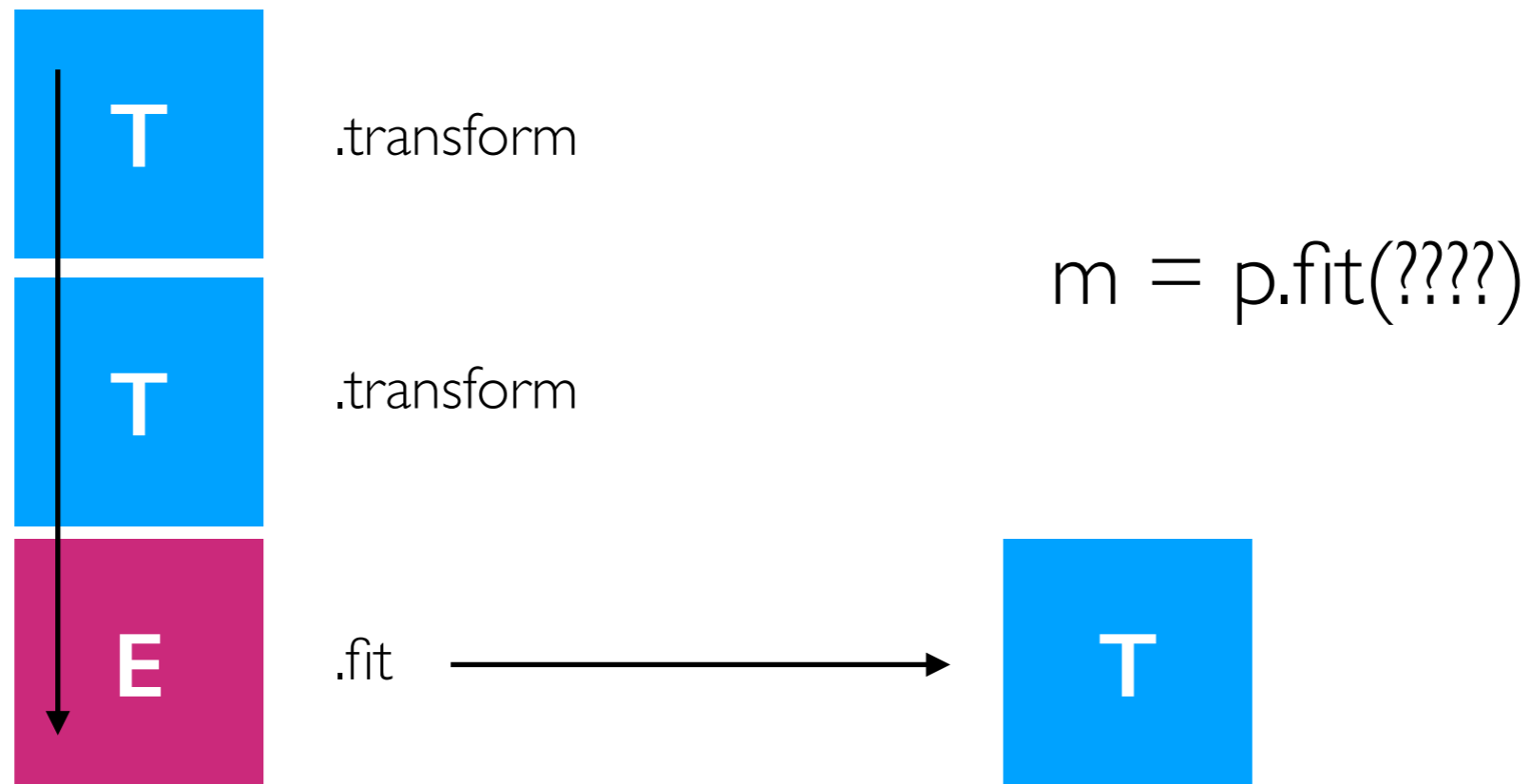
Pipeline (p)



Pipeline (Spark Example)

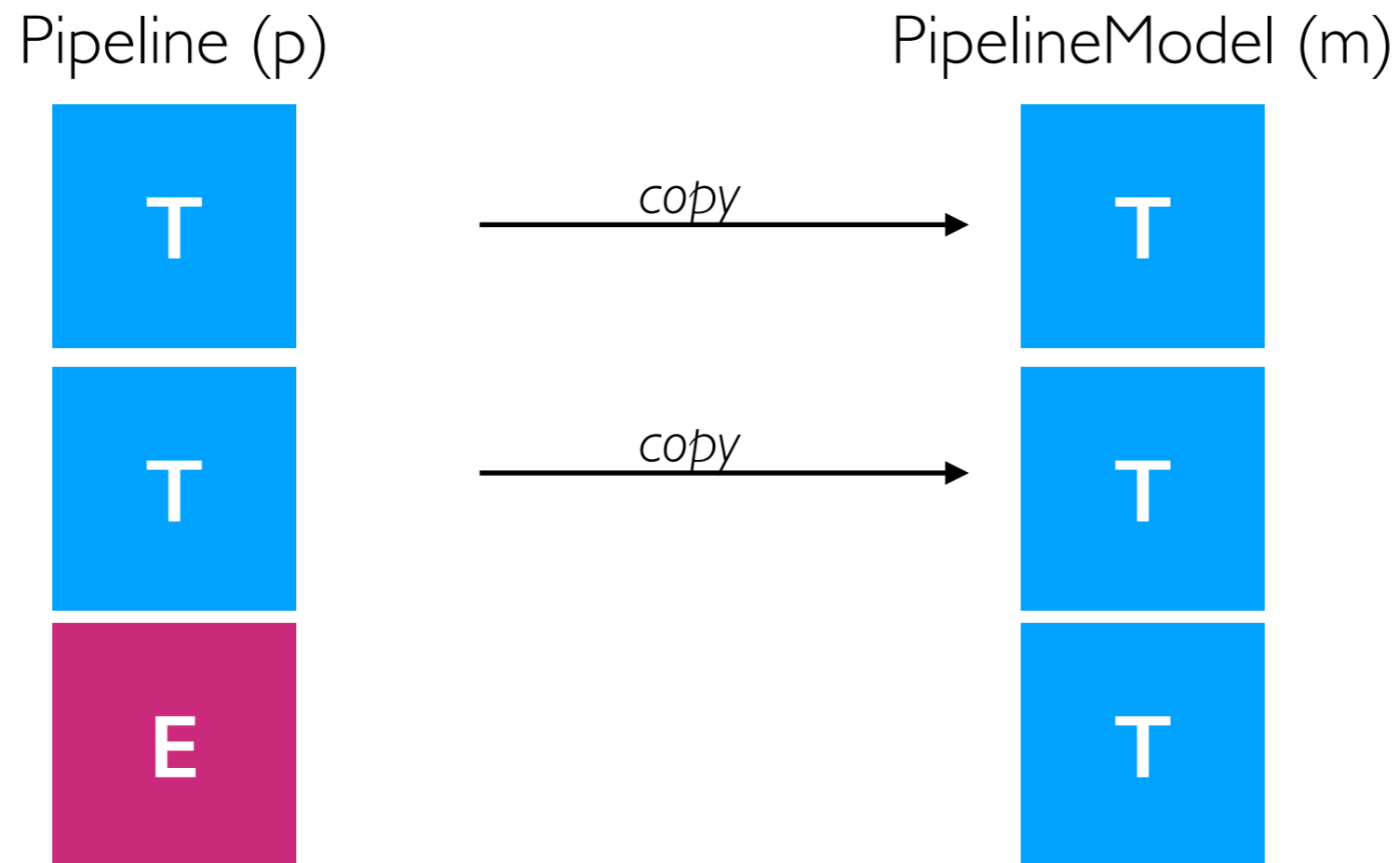
Both scikit-learn and Spark: a pipeline is a series of stages (transformers/estimators). `fit`/`transform`/etc. are called as appropriate on each stage.

Pipeline (p)



Pipeline (Spark Example)

Both scikit-learn and Spark: a pipeline is a series of stages (transformers/estimators). fit/transform/etc. are called as appropriate on each stage.



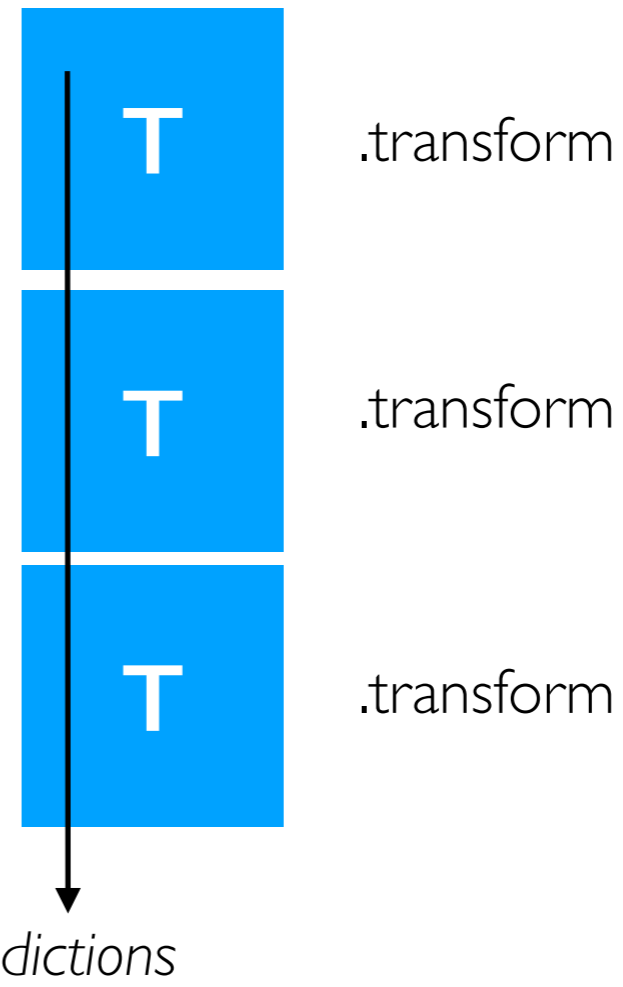
Pipeline (Spark Example)

Both scikit-learn and Spark: a pipeline is a series of stages (transformers/estimators). fit/transform/etc. are called as appropriate on each stage.

Pipeline (p)



PipelineModel (m)



`m.transform(????)`

TopHat

Outline

ML Overview

Training/Predicting APIs

Demos

Decision Trees

Spark mllib packages

- `pyspark.mllib` -- based on RDDs
- `pyspark.ml` -- based on DataFrames

Distributed ML Outline

ML Overview

Training/Predicting APIs

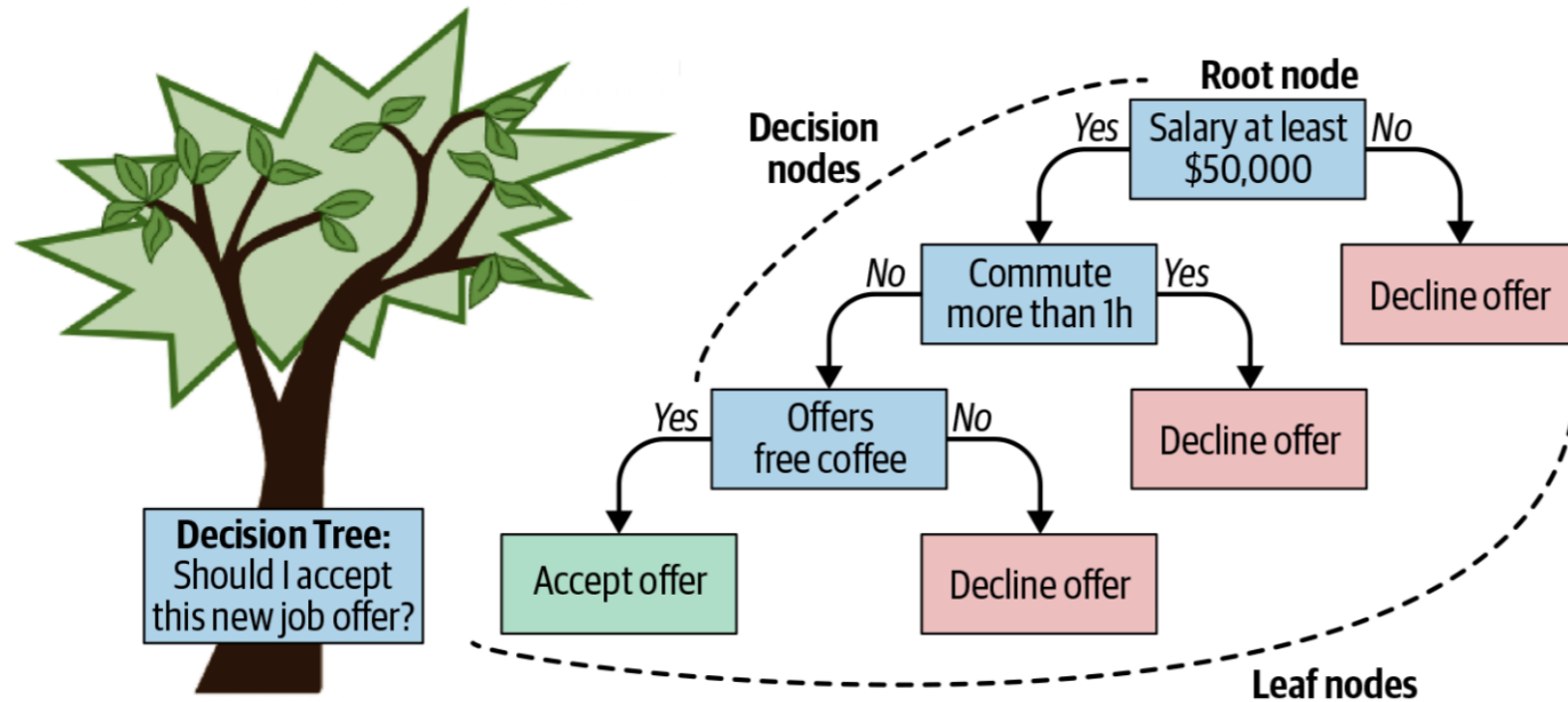
Demos

Decision Trees:

- Background
- Training in memory
- PLANET algorithm

Decision Trees

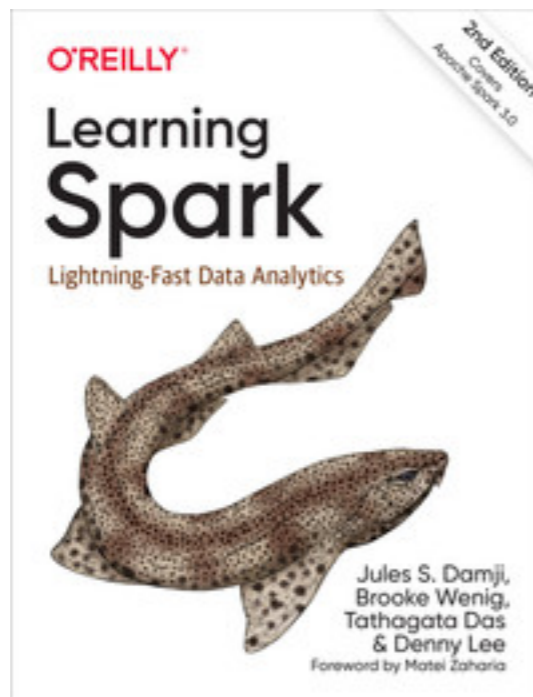
problem: if the tree is large,
many subtrees might be similar



decision trees are like nested
if/else statements

features and labels can be
numeric or categorical

Figure 10-9. Decision tree example



```
def predict(row):
    if row.salary < 50K:
        return False
    else:
        if row.commute > 1h:
            return False
        else:
            if row.coffee == "free":
                return True
            else:
                return False
```

Ensemble Methods

Ensemble: many simple models vote. Many simple decision trees (each trained on subset of rows/columns) together are often better than one big tree. Examples:

- random forest
- gradient-boosted trees

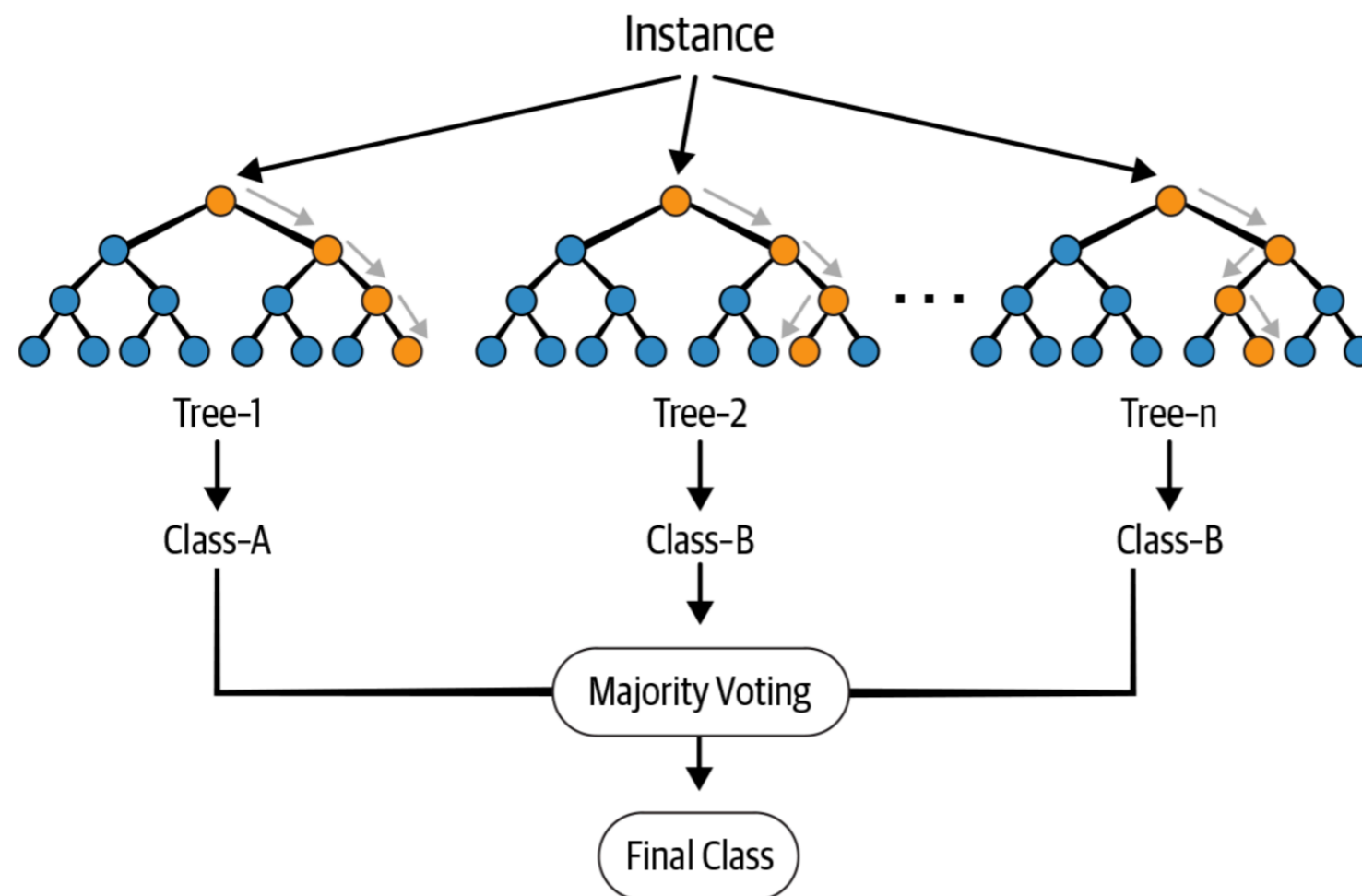
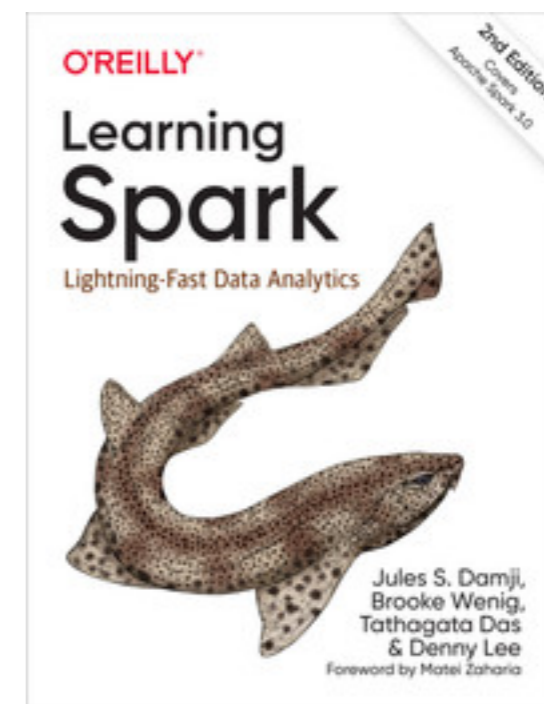


Figure 10-12. Random forest predictions



A Spark cluster can train many trees in a random forest simultaneously!

Tree methods vs. Deep Learning

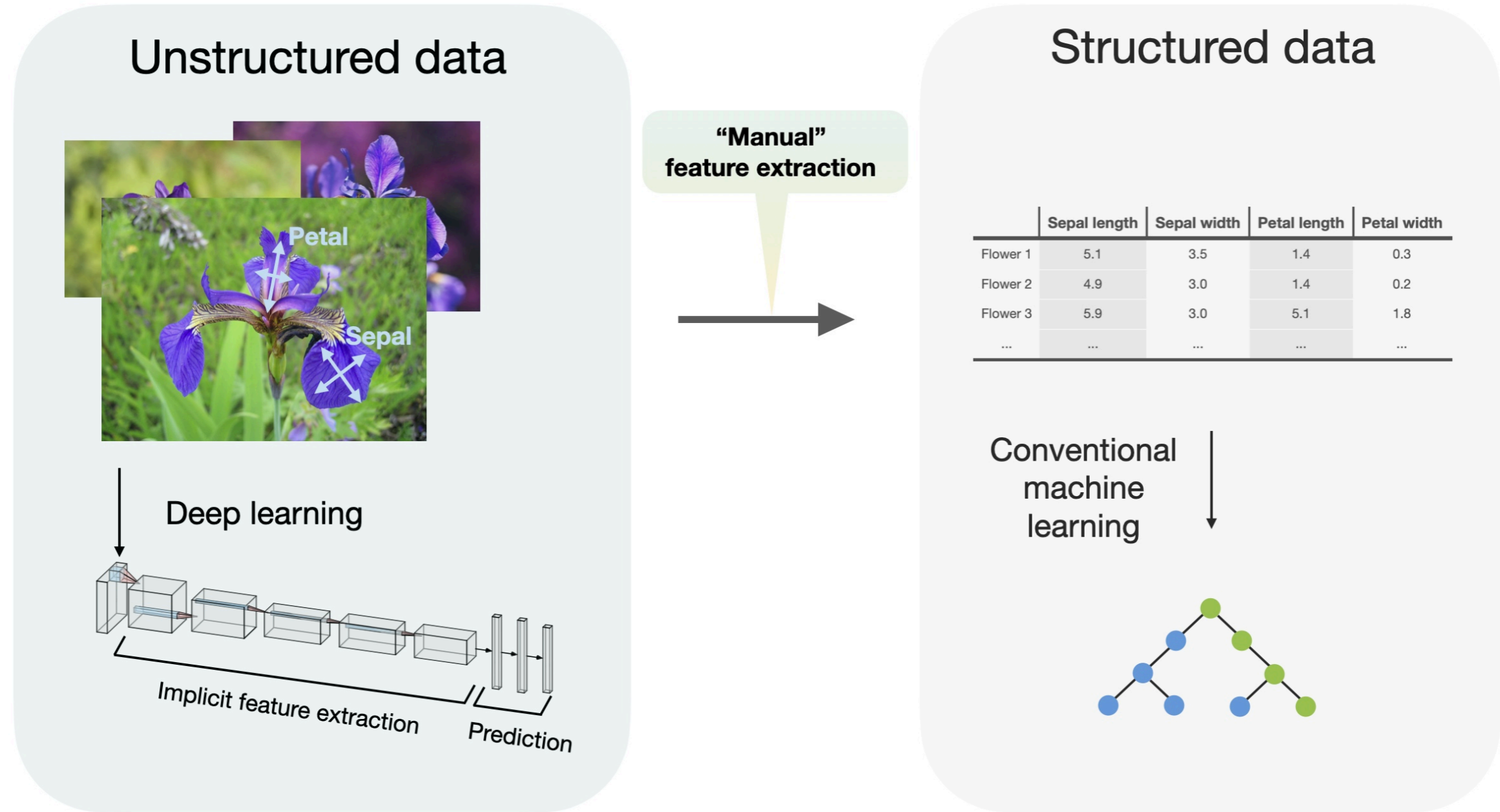


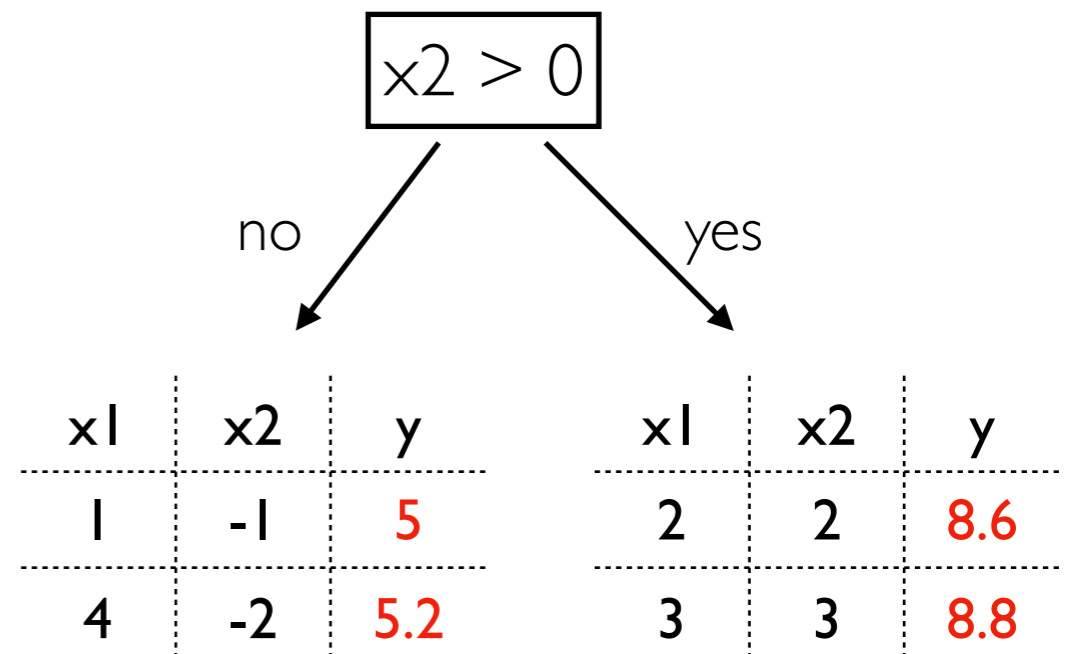
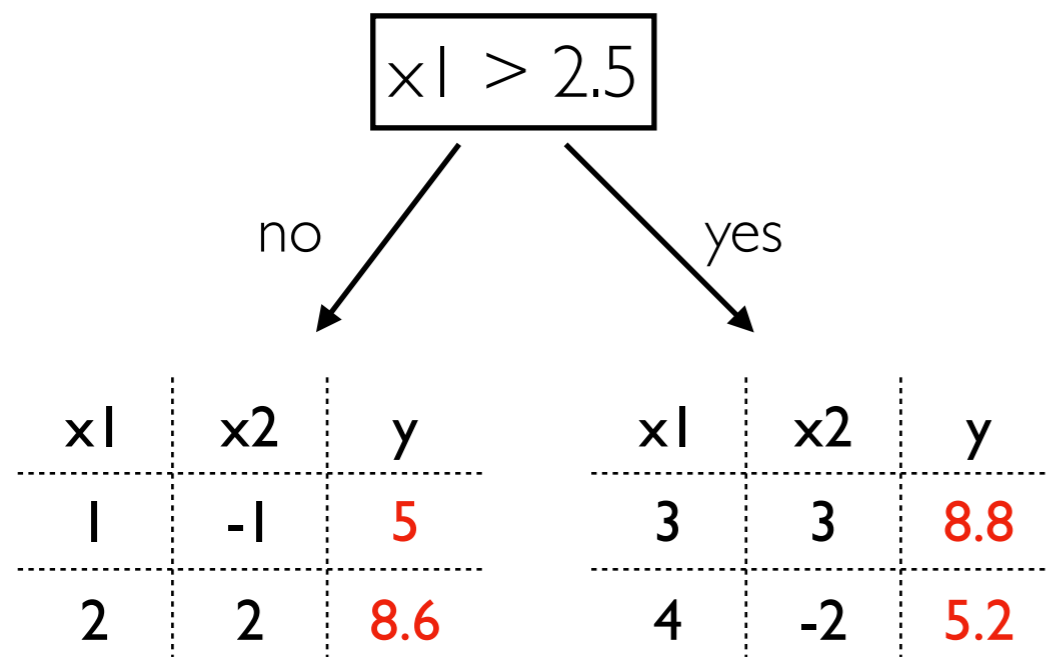
Image from Blog Post: A Short Chronology Of Deep Learning For Tabular Data, by Sebastian Raschka
<https://sebastianraschka.com/blog/2022/deep-learning-for-tabular-data.html>

*Tree-based methods are **still relevant** in the age of deep learning because there are many important tabular datasets.*

Is a Tree Good?

data:

x1	x2	y
1	-1	5
2	2	8.6
3	3	8.8
4	-2	5.2



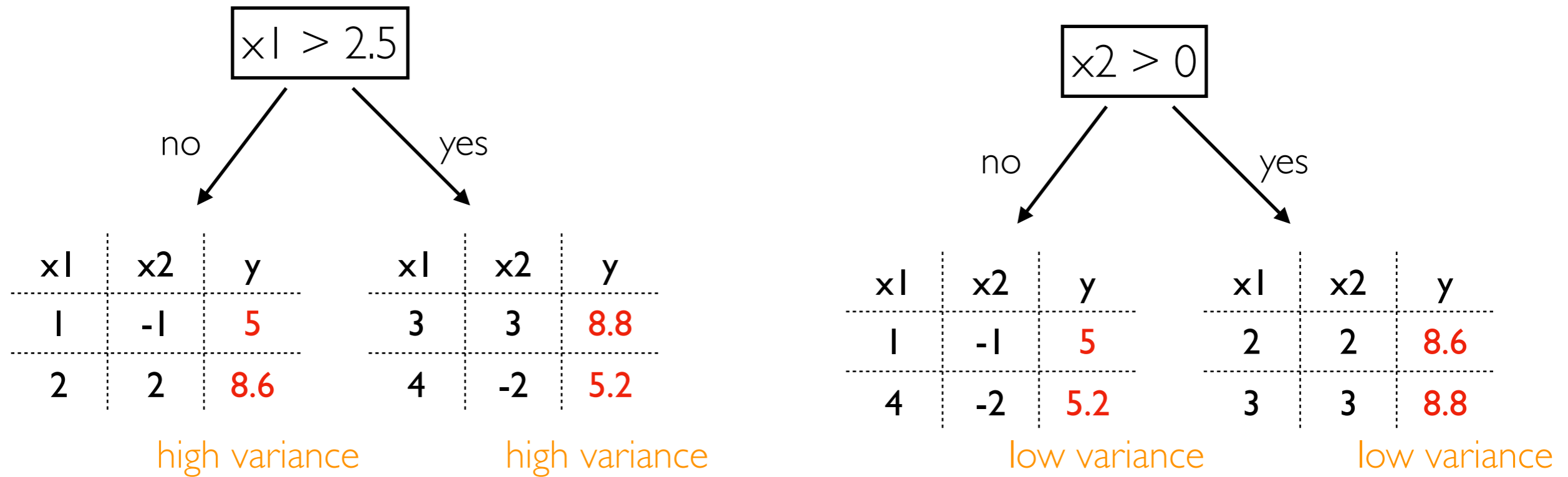
which tree asks better questions about x values if we want to predict y?

Impurity

data:

x1	x2	y
1	-1	5
2	2	8.6
3	3	8.8
4	-2	5.2

better tree



"impurity" measures (like variance) measure how non-uniform label (y) values are in leaves

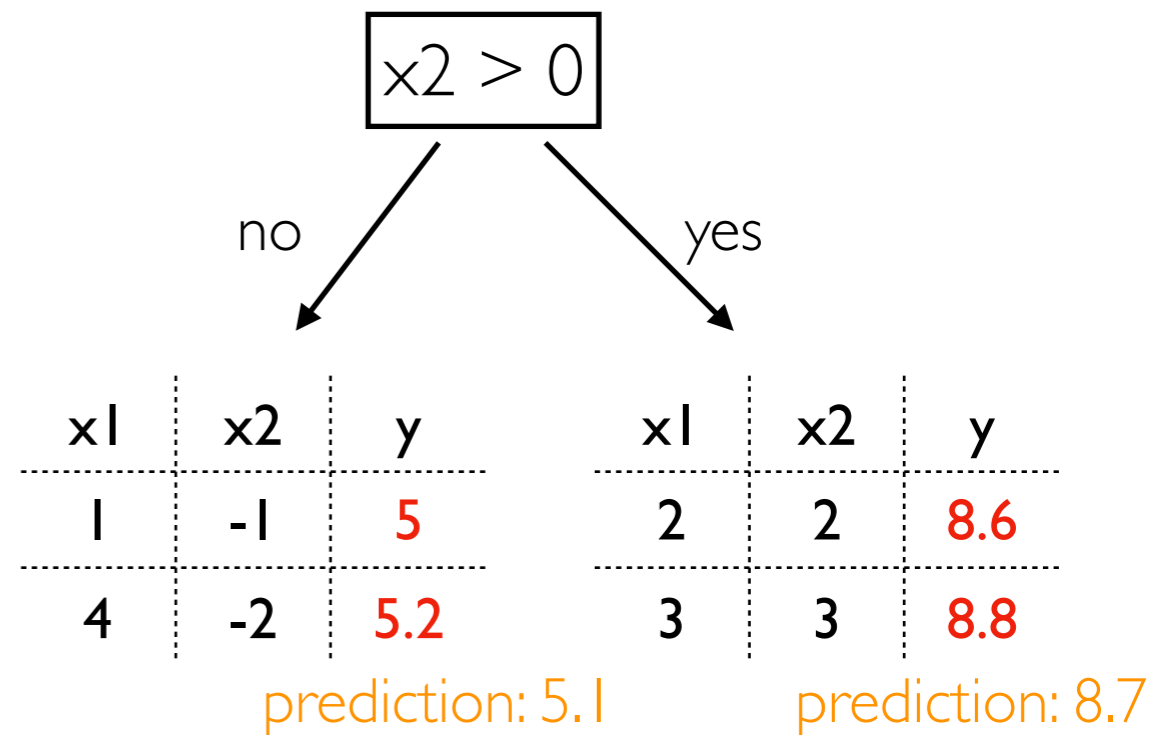
Predictions

if a new data point lands in a leaf, assume it is similar to other rows in that leaf...

data:

x1	x2	y
1	-1	5
2	2	8.6
3	3	8.8
4	-2	5.2
3	50	????

better tree



Distributed ML Outline

ML Overview

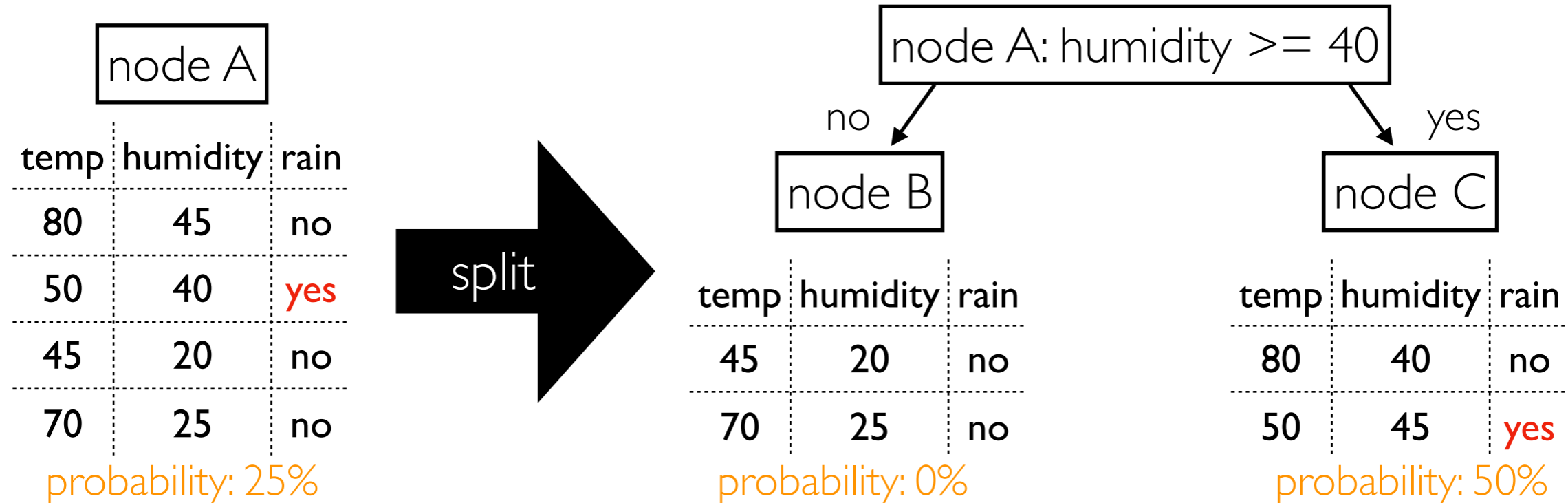
Training/Predicting APIs

Demos

Decision Trees:

- Background
- Training in memory
- PLANET algorithm

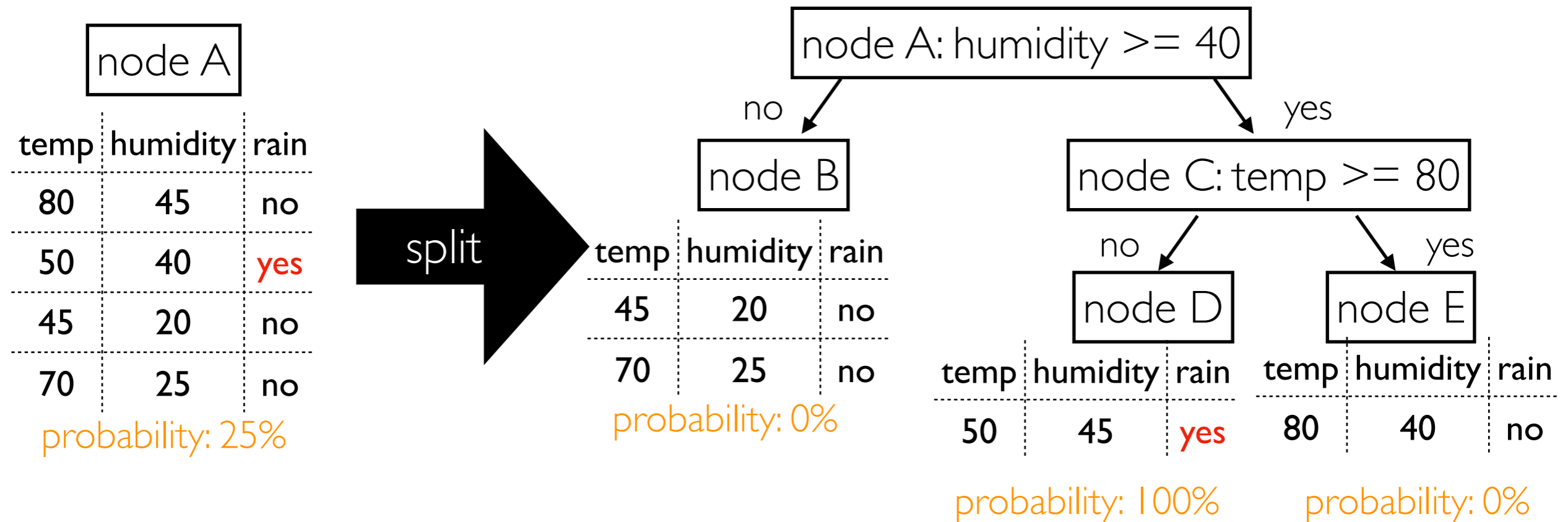
Splitting Nodes



Algorithm

- start with one node with all data
- find split point in some column to create two children
- identify another node, recursively split
- eventually stop

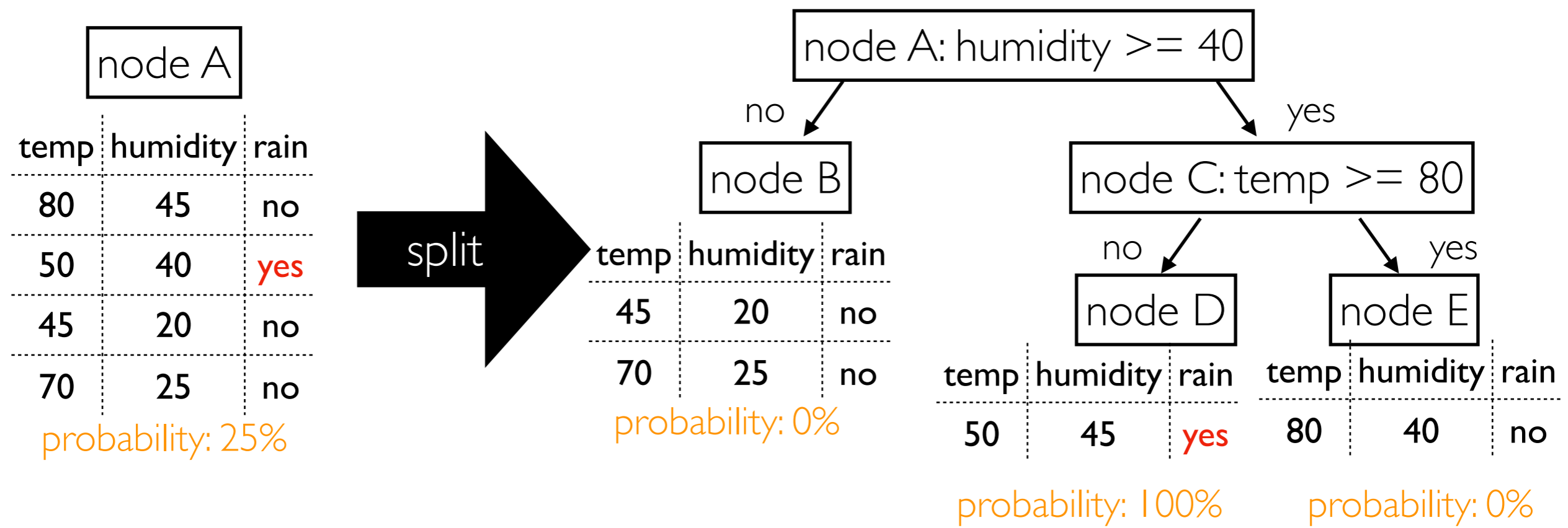
Splitting Nodes



Algorithm

- start with one node with all data
- find split point in some column to create two children
- identify another node, recursively split
- eventually stop

When to Stop Splitting?



Some Approaches

- set maximum tree height
- set minimum number of rows in node required for split
- prune tree later to get rid of unhelpful/excessive splitting

Overfitting

"simple" tree

node A: humidity \geq 40

no

node B

yes

node C

temp	humidity	rain
45	20	no
70	25	no

probability: 0%

temp	humidity	rain
80	40	no
50	45	yes

probability: 50%

temp=65, humidity=42

"complex" tree

node A: humidity \geq 40

no

node B

yes

node C: temp \geq 80

no

node D

yes

node E

temp	humidity	rain
45	20	no
70	25	no

probability: 0%

temp	humidity	rain
50	45	yes

probability: 100%

temp	humidity	rain
80	40	no

probability: 0%

50% chance of rain

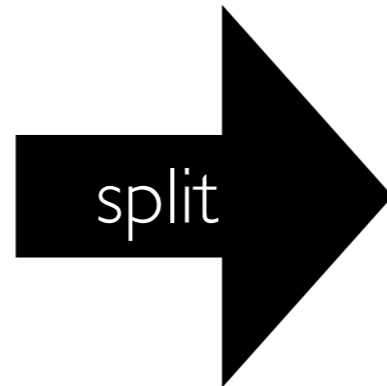
100% chance of rain

which tree will make better predictions?

Choosing Splits

node A		
temp	humidity	rain
80	45	no
50	40	yes
45	20	no
70	25	no

probability: 25%



Which node to split?

- 2 feature columns
- 3 ways to divide 4 rows into big/small
- $2 * 3 = 6$ choices
- try all, choose one that reduces impurity the most!
- how to do so efficiently?

Choosing Splits

node A

temp	humidity	rain
80	45	no
50	40	yes
45	20	no
70	25	no

sort by each column

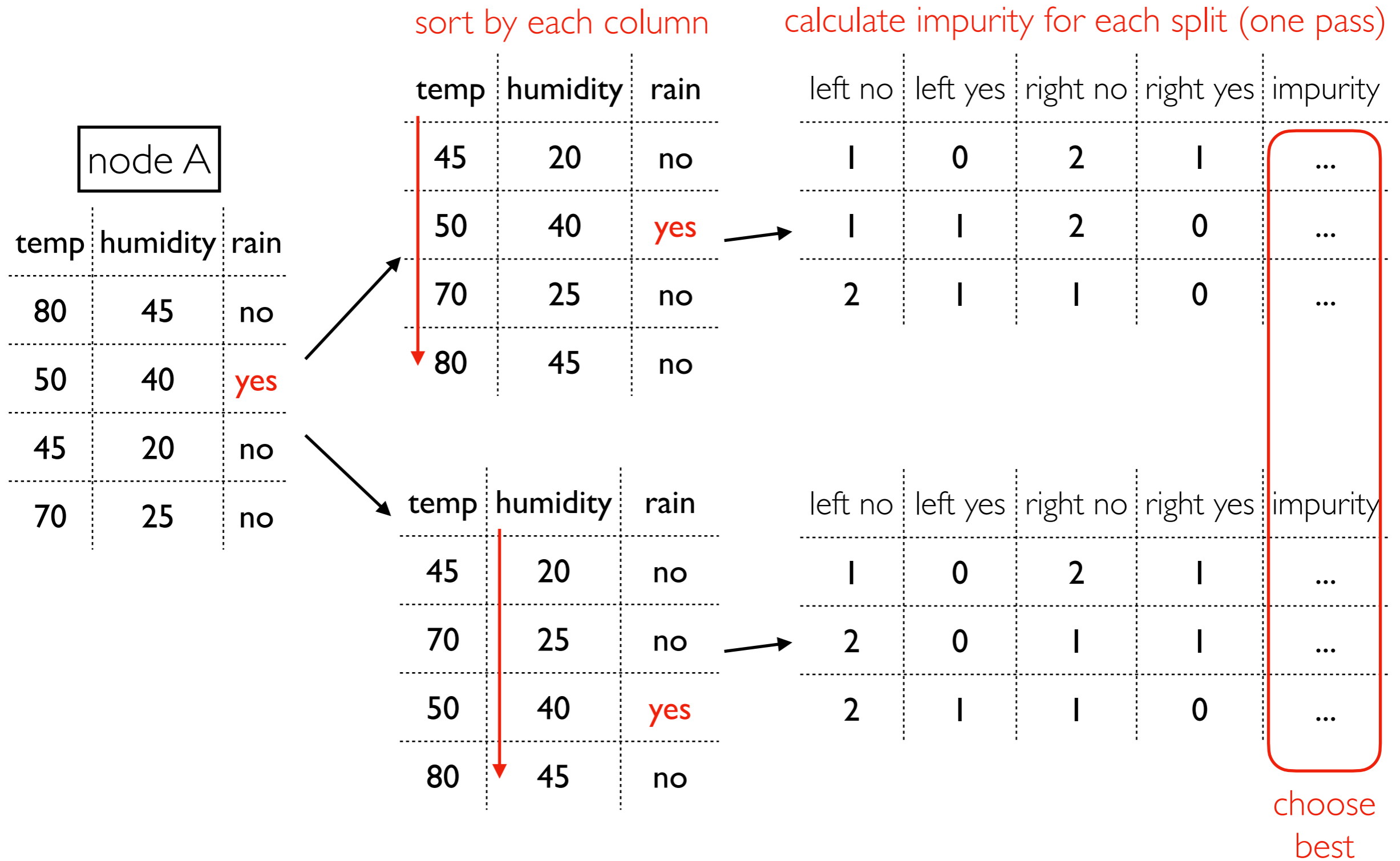
temp	humidity	rain
45	20	no
50	40	yes
70	25	no
80	45	no

calculate impurity for each split (one pass)

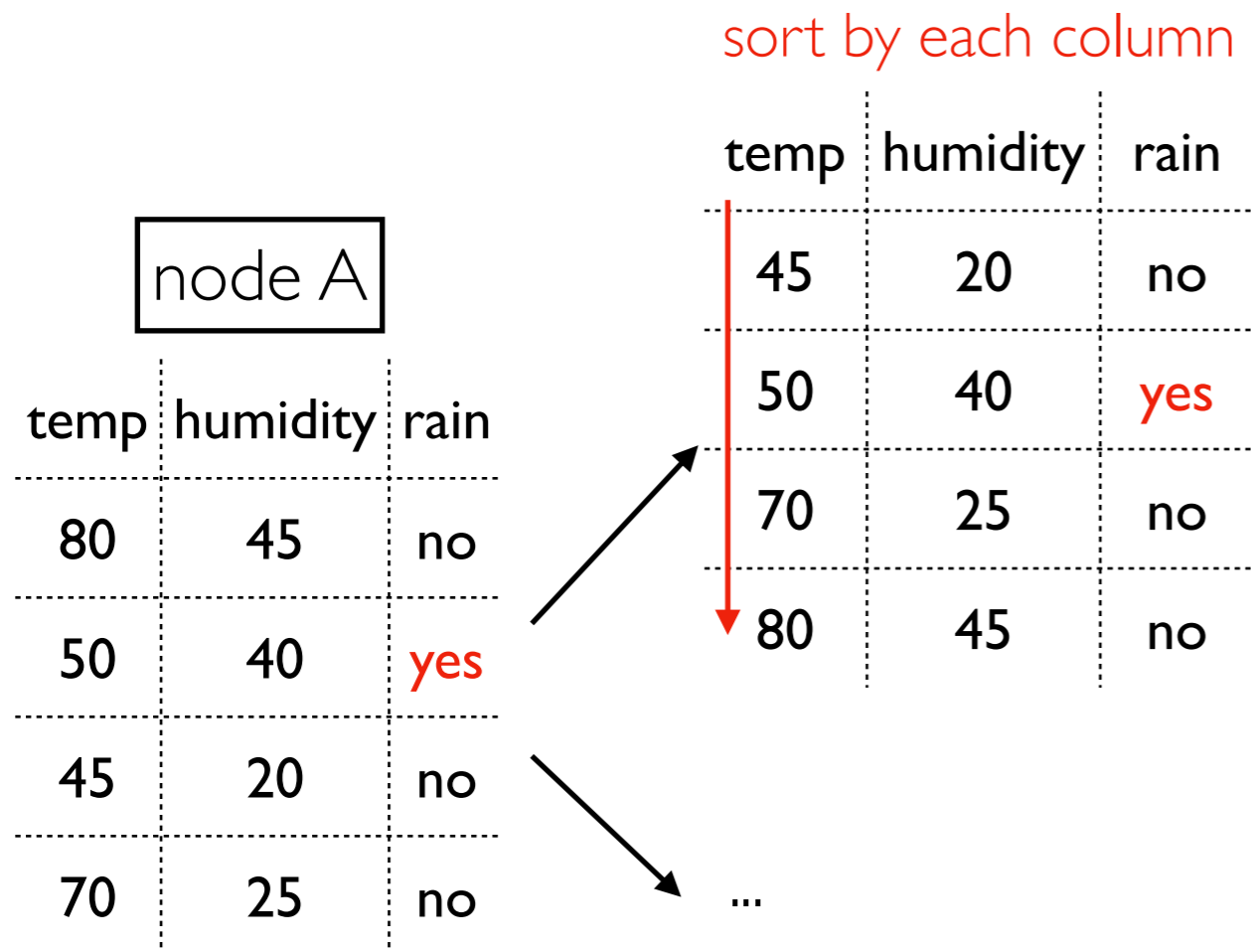
left no	left yes	right no	right yes	impurity
1	0	2	1	...
1	1	2	0	...
2	1	1	0	...

Observation: we can incrementally compute impurity for each split point by looking at just one more row of data. Don't need to loop over all rows for every possible split point.

Choosing Splits



Challenge: Big Data



What if rows for a node are **too big to fit in RAM** on one worker?

- partitioned across many Spark workers
- maybe fits in cumulative RAM of many workers (or maybe not)
- each sort would be expensive (network shuffle/exchange)
- if looping over every possible split point, we'll be computing on one worker at any given time (the one that has data around the split point). Not parallel!

Distributed ML Outline

ML Overview

Training/Predicting APIs

Demos

Decision Trees:

- Background
- Training in memory
- PLANET algorithm

PLANET Algorithm

PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce

Biswanath Panda, Joshua S. Herbach, Sugato Basu, Roberto J. Bayardo
Google, Inc.

[bpanda, jsherbach, sugato]@google.com, bayardo@alum.mit.edu

ABSTRACT

Classification and regression tree learning on massive datasets is a common data mining task at Google, yet many state of the art tree learning algorithms require training data to

plexities such as data partitioning, scheduling tasks across many machines, handling machine failures, and performing inter-machine communication. These properties have motivated many technology companies to run MapReduce

<https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36296.pdf>

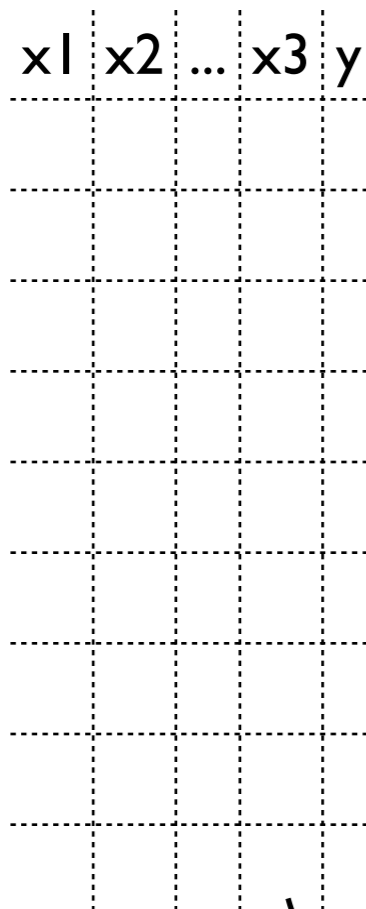
PLANET: Parallel Learner for Assembling Numerous Ensemble Trees

- originally implemented as MapReduce jobs
- Spark **DecisionTreeRegressor** and **DecisionTreeClassifier** use it too

Hybrid Approach

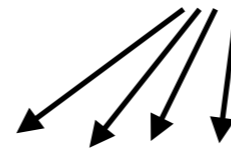
- in-memory splitting for nodes with few enough rows to fit in worker memory
- simplified (fewer split points) and distributed approach for nodes with lots of data

Step 1: Compute Equi-Depth Histograms

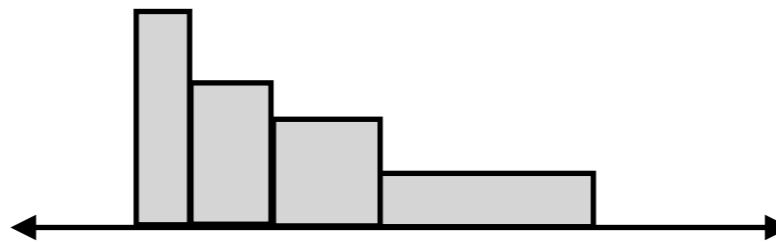


Warning: number of bins must be greater than biggest number of unique values in any categorical column

```
dt = DecisionTreeClassifier(labelCol="y")  
dt.setMaxBins(4)
```

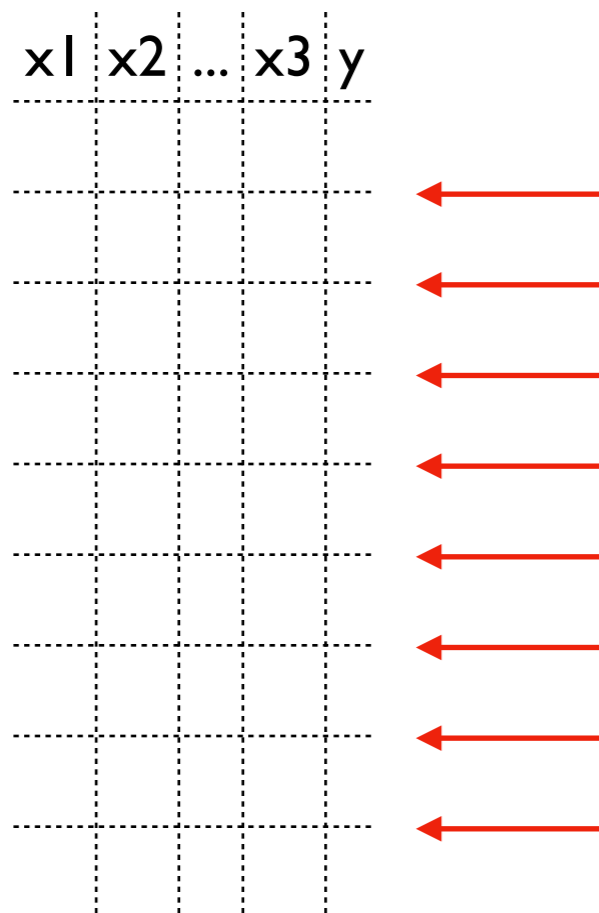


equi-depth histogram for x3 feature
(each bucket has approx same number of samples)



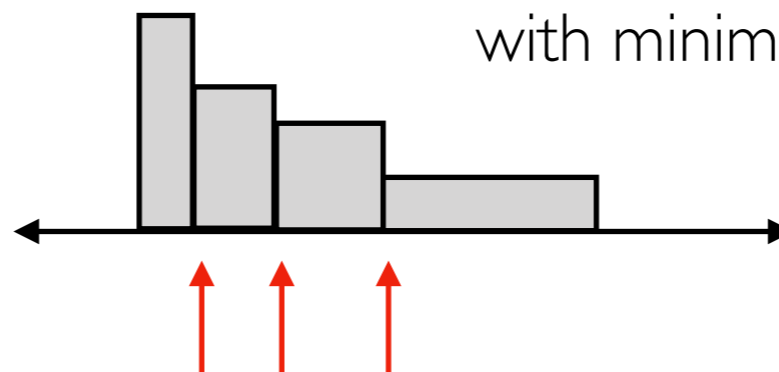
histograms for every feature...

Split Points: In-Mem vs. Distributed



split points for one-node,
in-memory algorithm

- splits based on initial data
- every task has split info for every column
- small number of pre-defined split points make it easy for executors to collaborate with minimal coordination

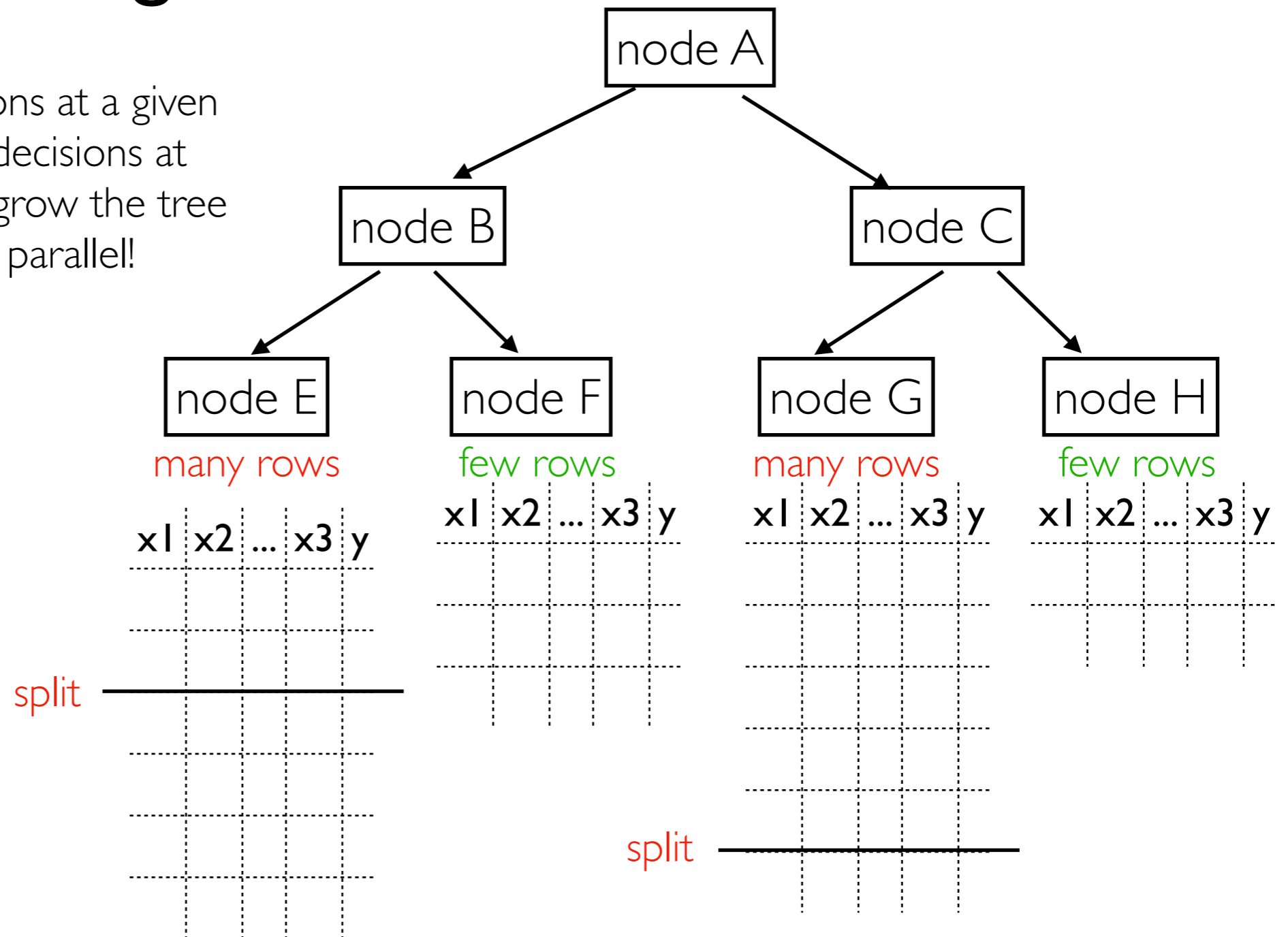


split points for **distributed** algorithm

Parallel Splitting

Observation: split decisions at a given node are unrelated to decisions at other nodes. It's easy to grow the tree at different places in parallel!

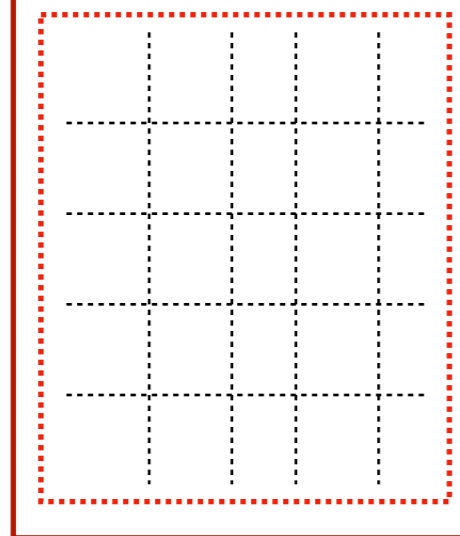
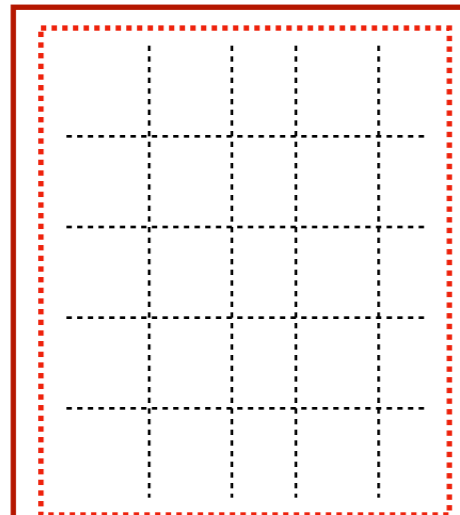
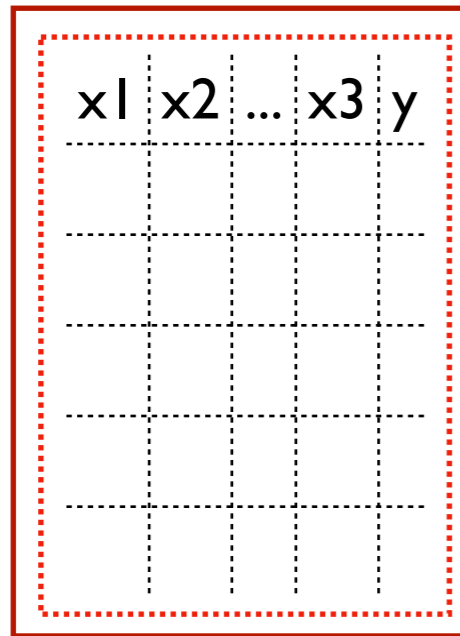
Decision Tree



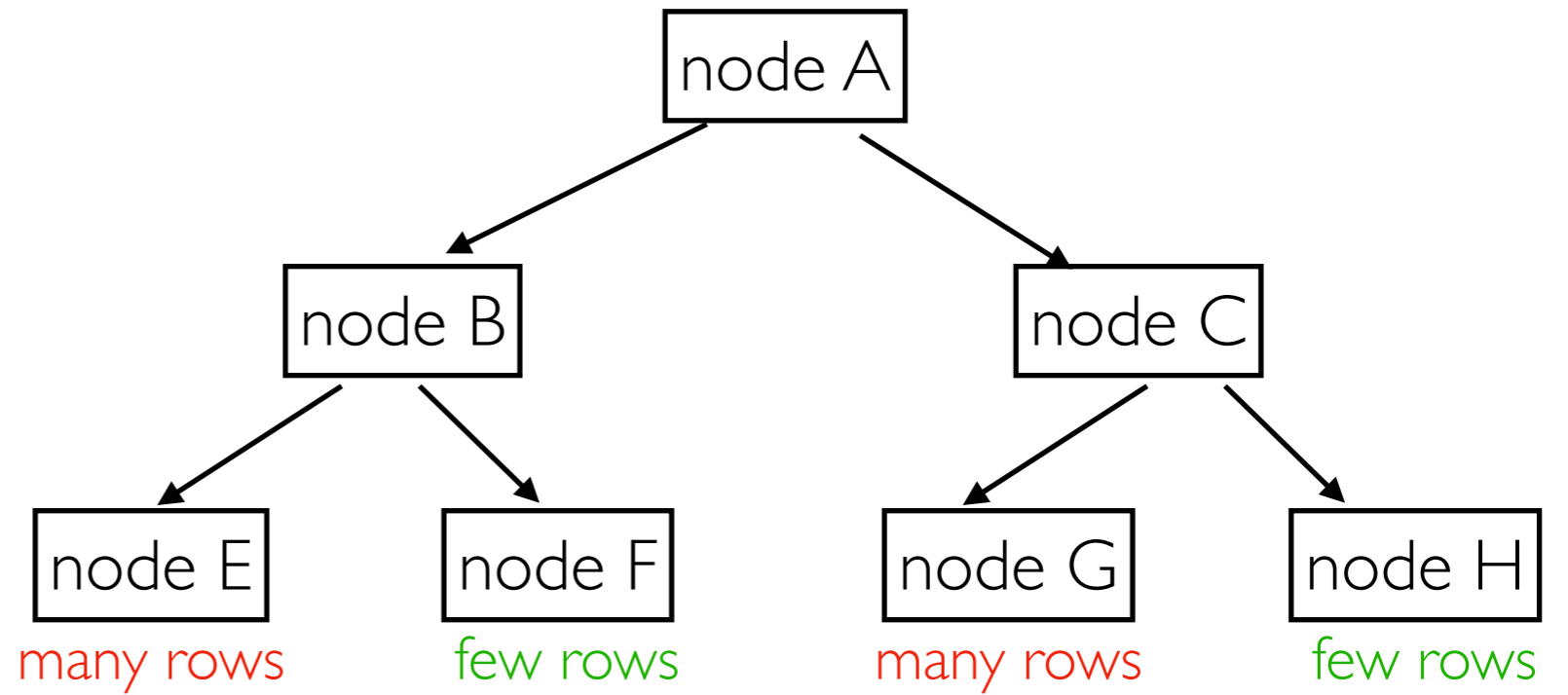
Clarification: nodes in the tree data structure DO NOT correspond to nodes in the Spark cluster.



Logical View of Rows
(rows in a DT node are NOT in the same place physically)

Physical Layout



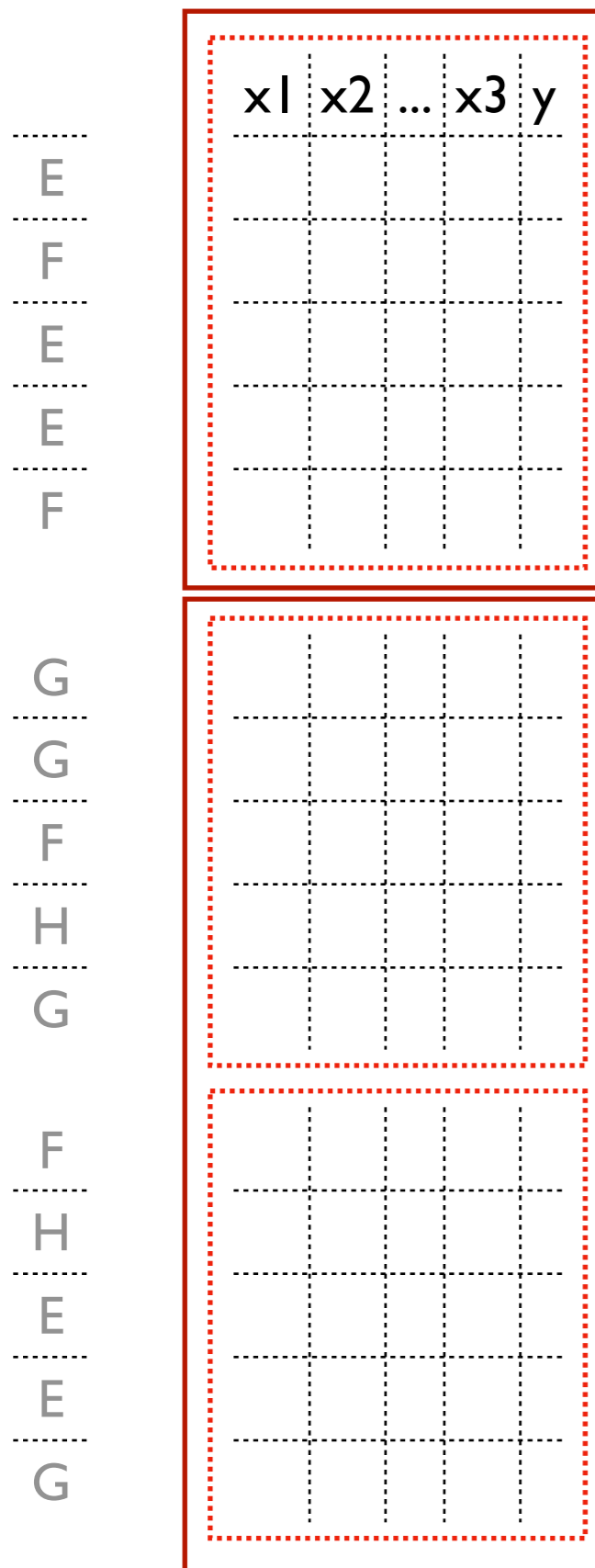
Decision Tree



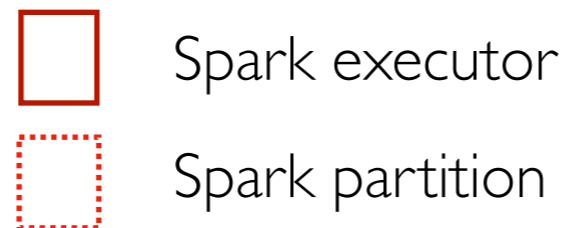
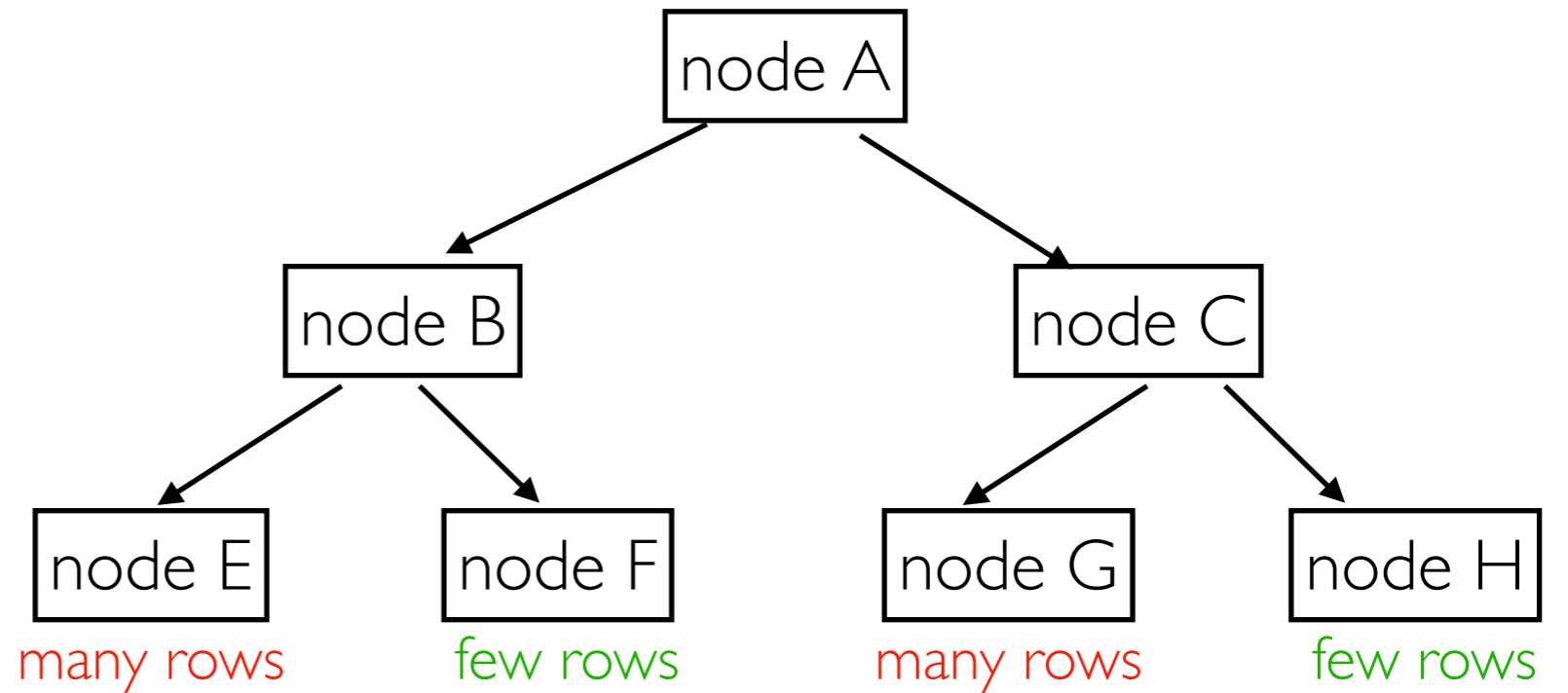
-  Spark executor
-  Spark partition

- all rows are in one big Spark DataFrame
- no particular order for rows

Physical Layout

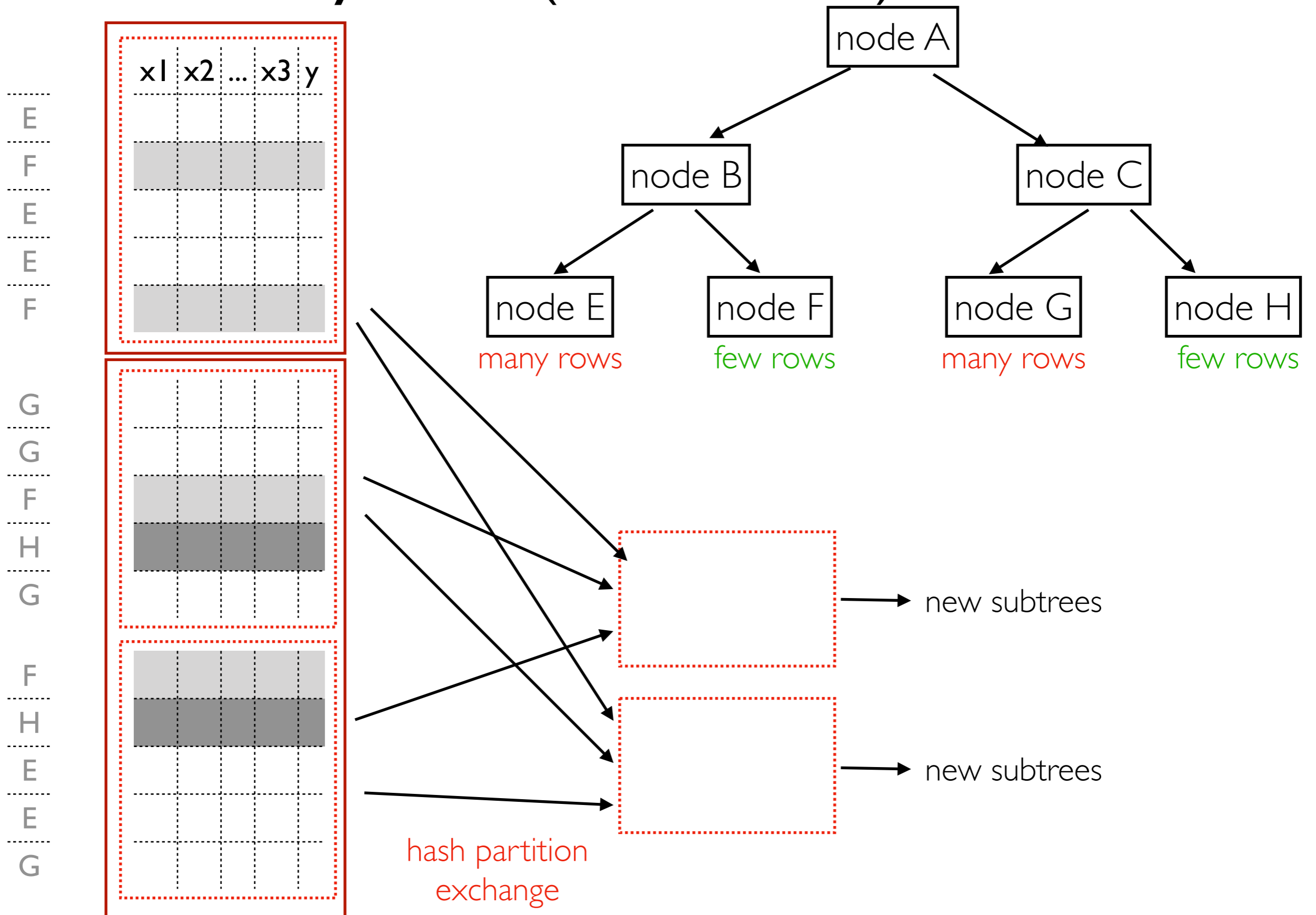


Decision Tree

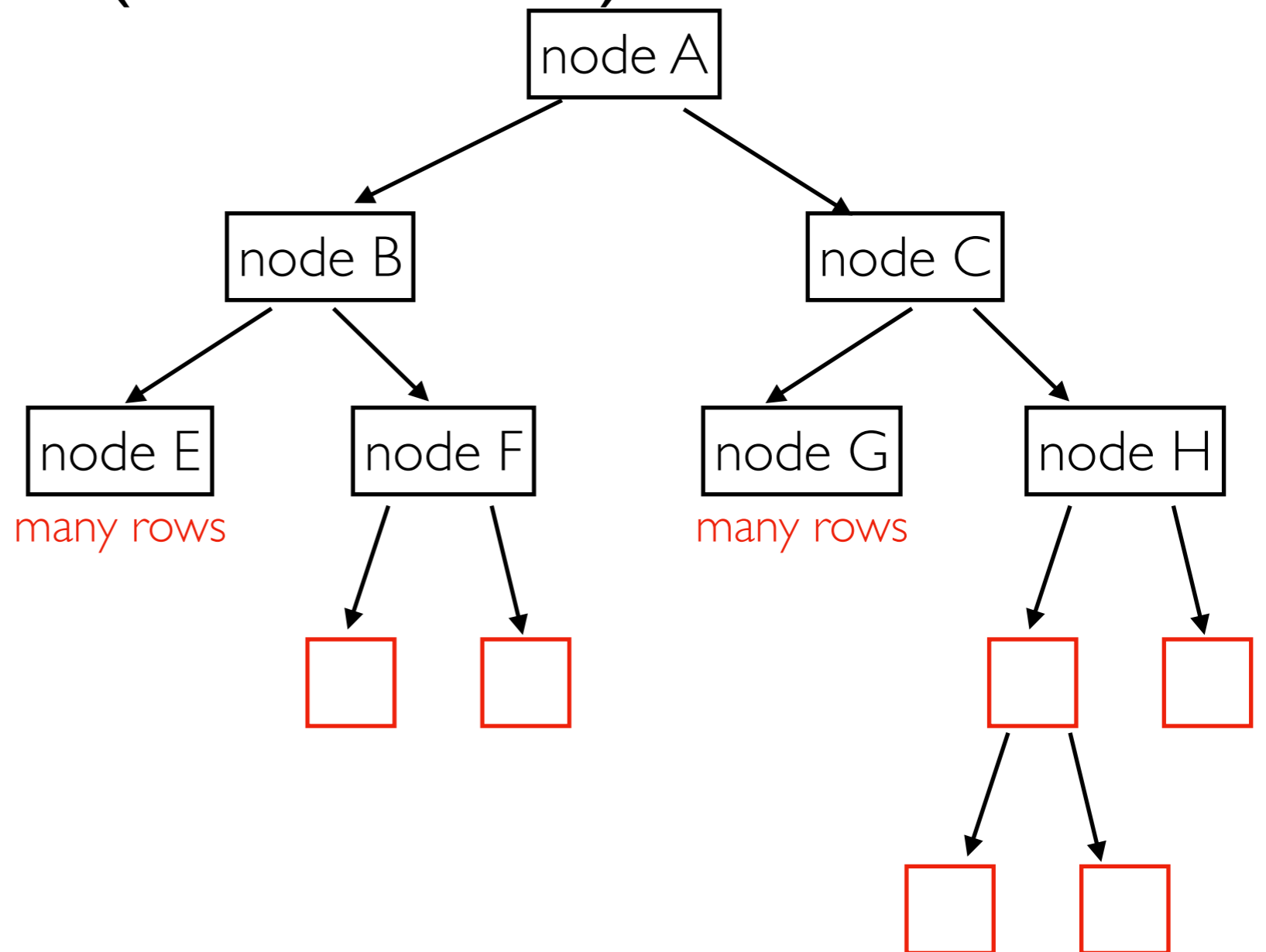
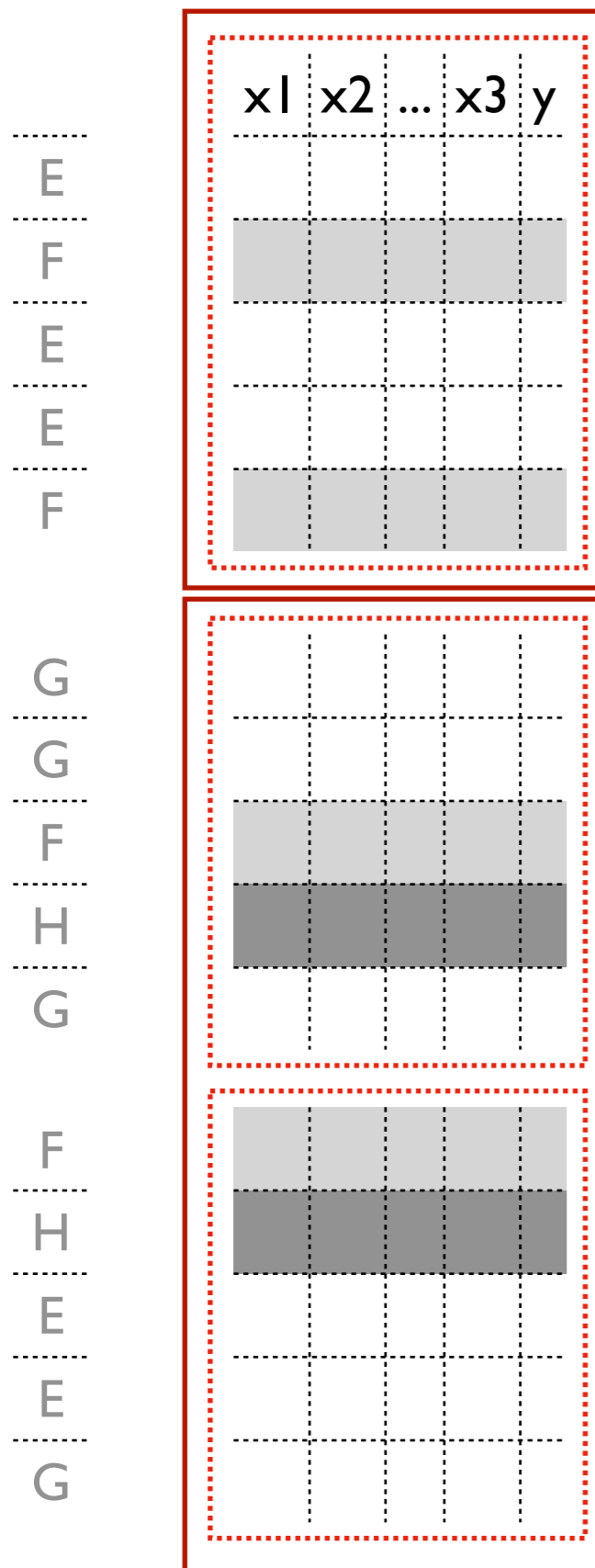


- all rows are in one big Spark DataFrame
- no particular order for rows
- given current tree and $x_1 \dots x_N$ values, we can infer what leaf node in the tree owns each row

In Memory Build (small nodes)

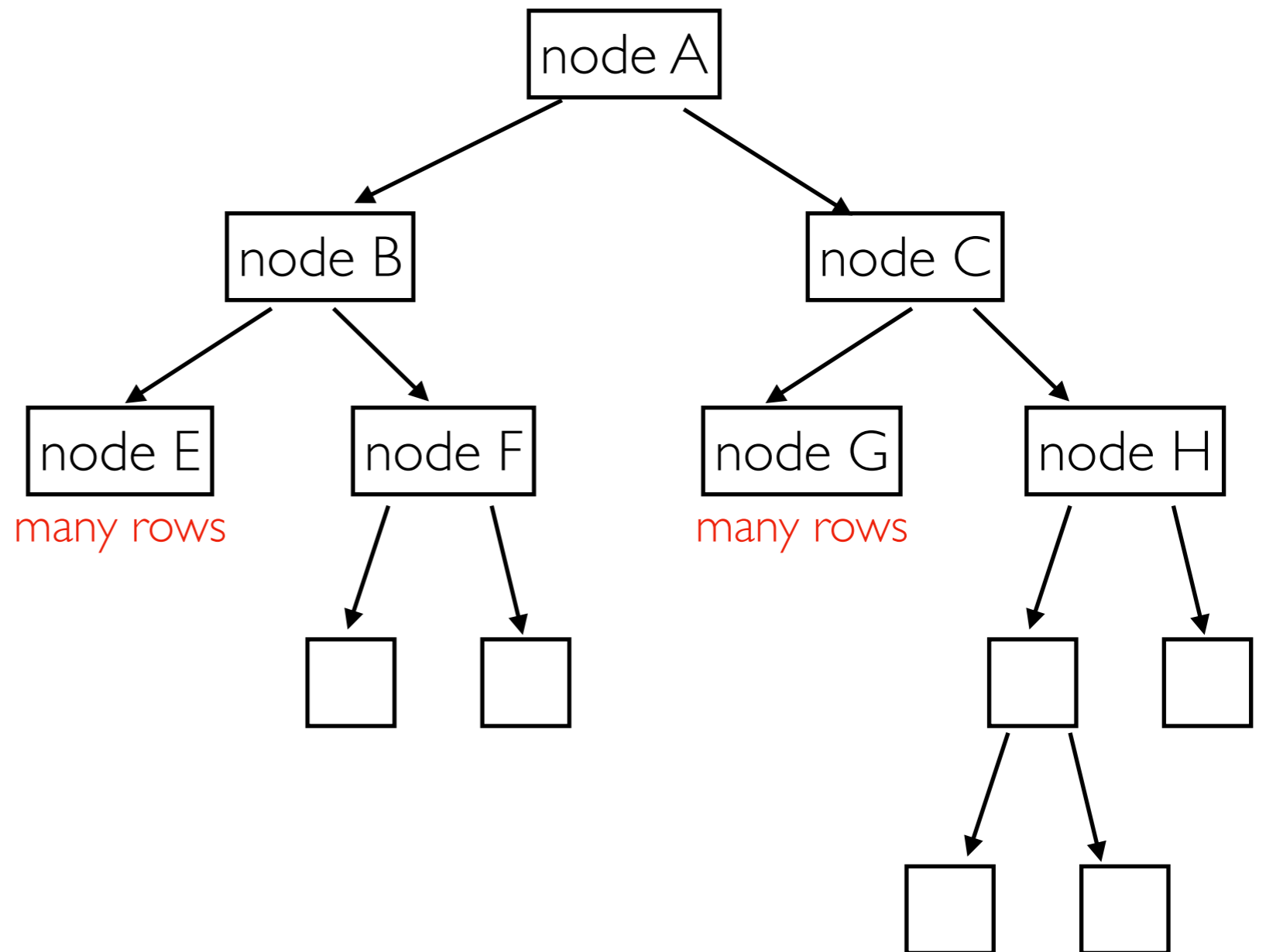
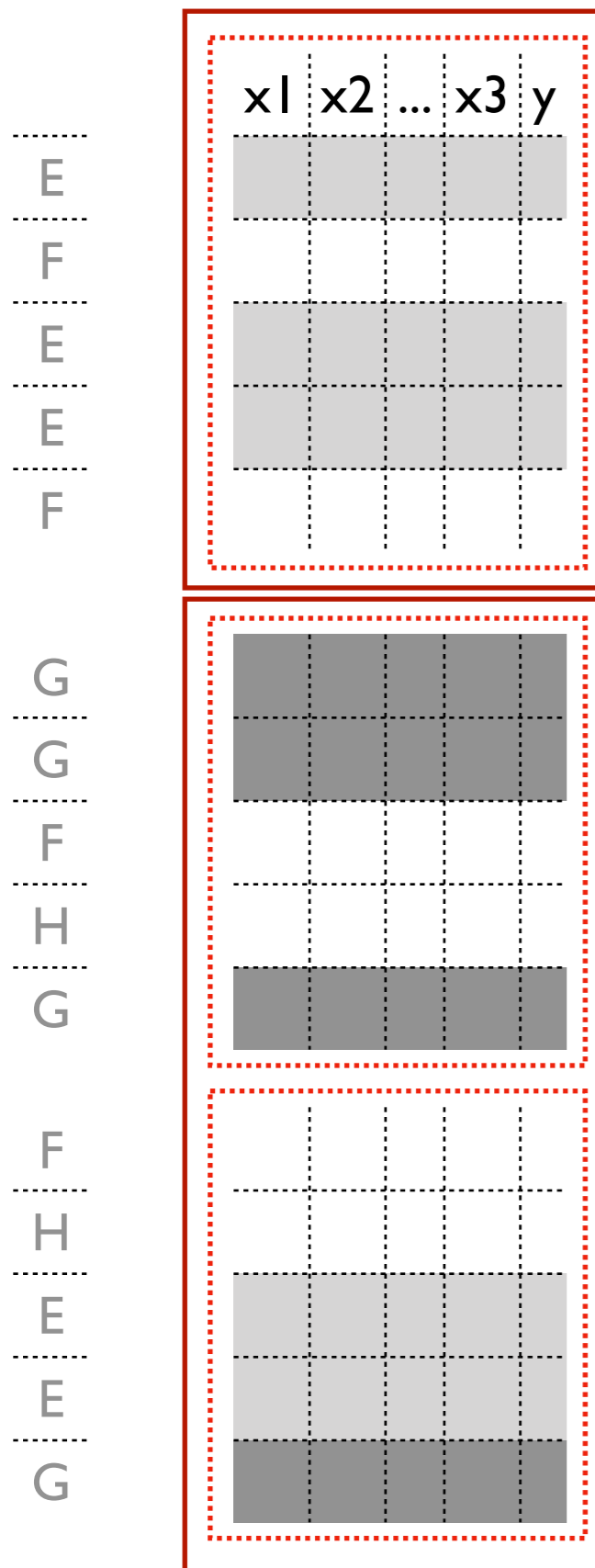


In Memory Build (small nodes)



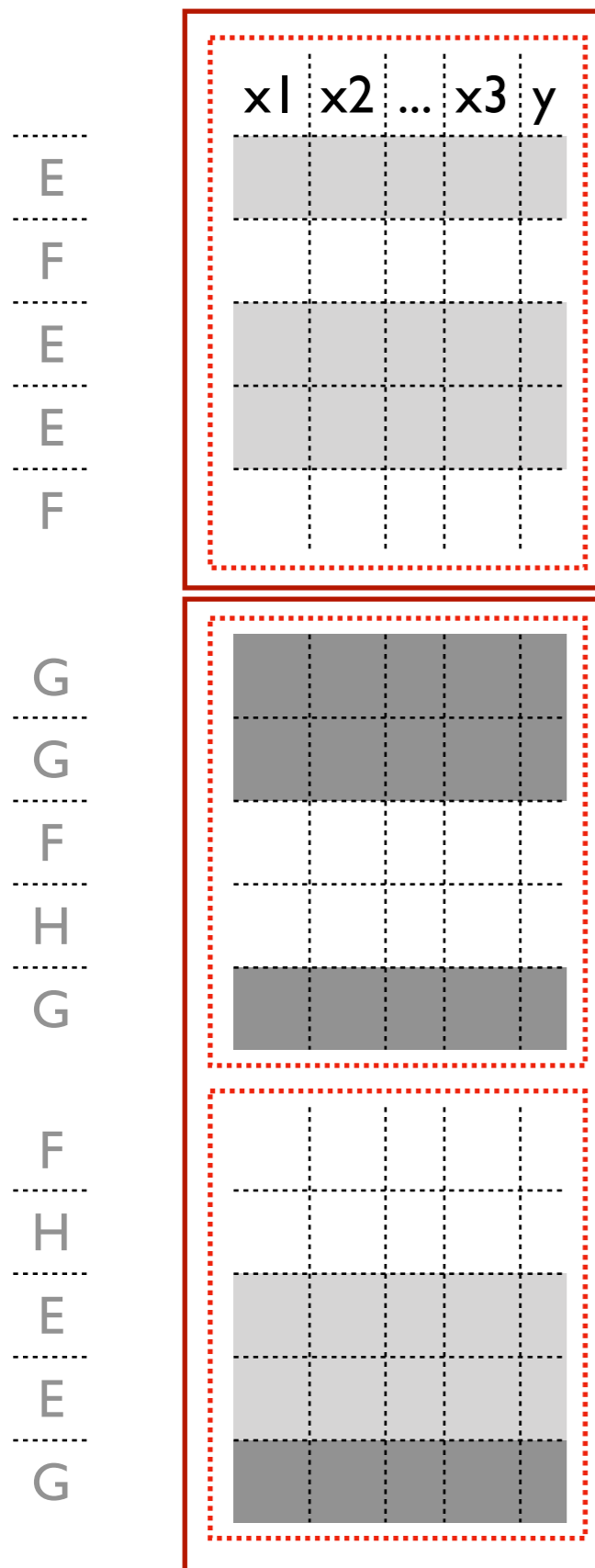
once in memory, splits keep happening recursively, so these nodes are done.

Big Nodes



- don't move row data between machines!
- just output stats per partition for every split/feature option

Big Nodes



node E (partial)

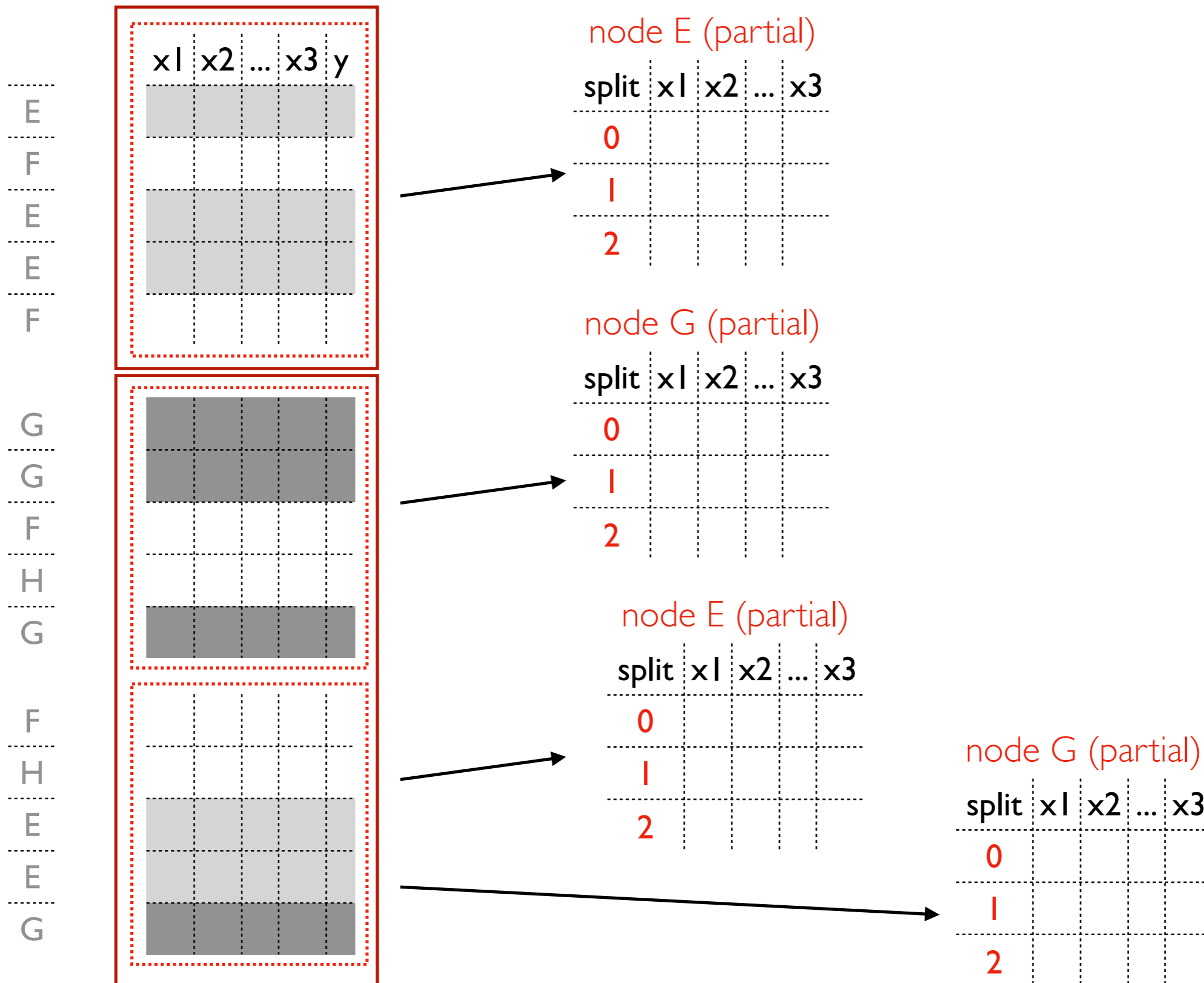
split	x1	x2	...	x3
0				
1				
2				

stats per feature/split combo

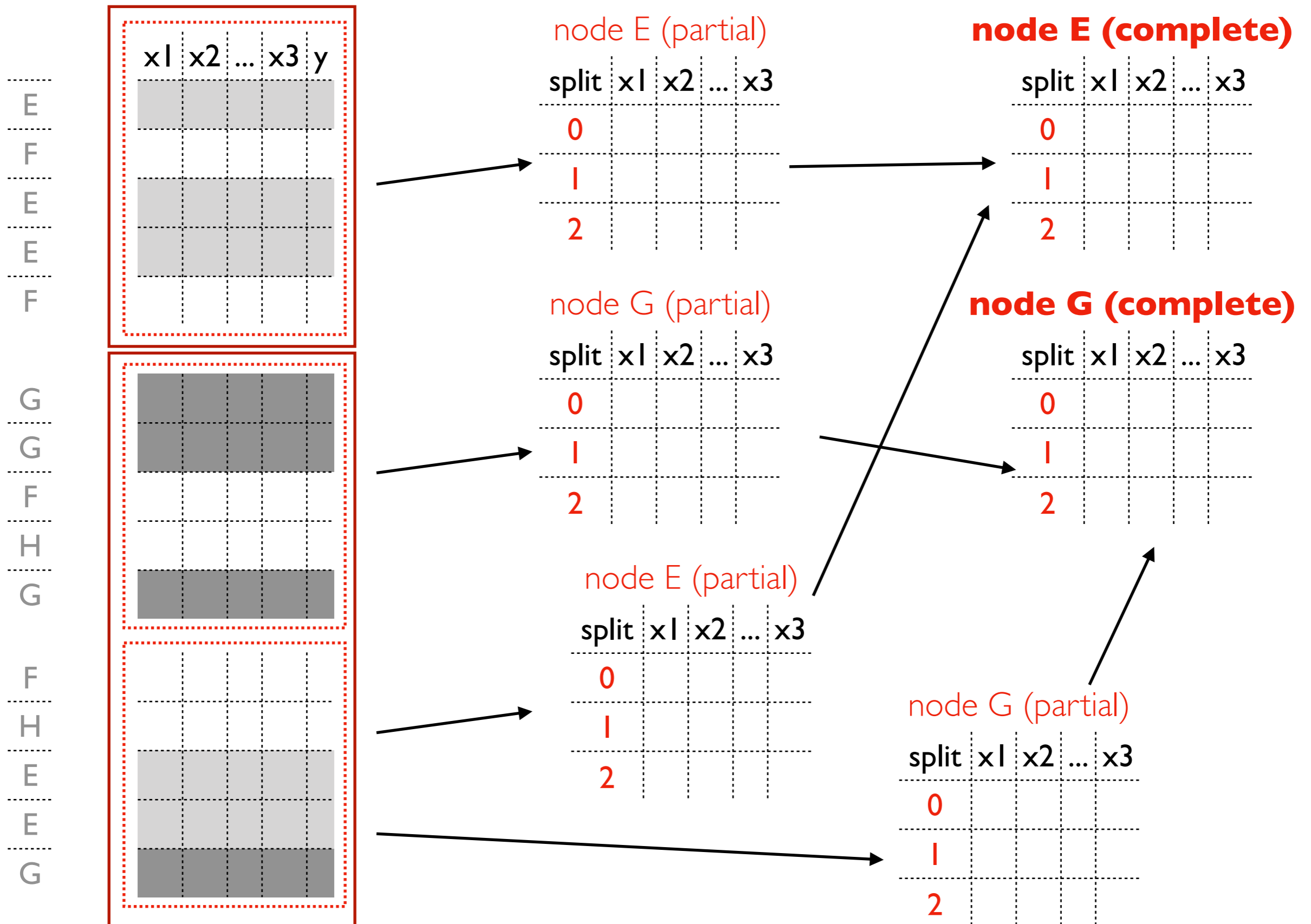
- left no: *number*
- left yes: *number*
- right no: *number*
- right yes: *number*

```
dt = DecisionTreeClassifier(labelCol="y")
dt.setMaxBins(4)
```

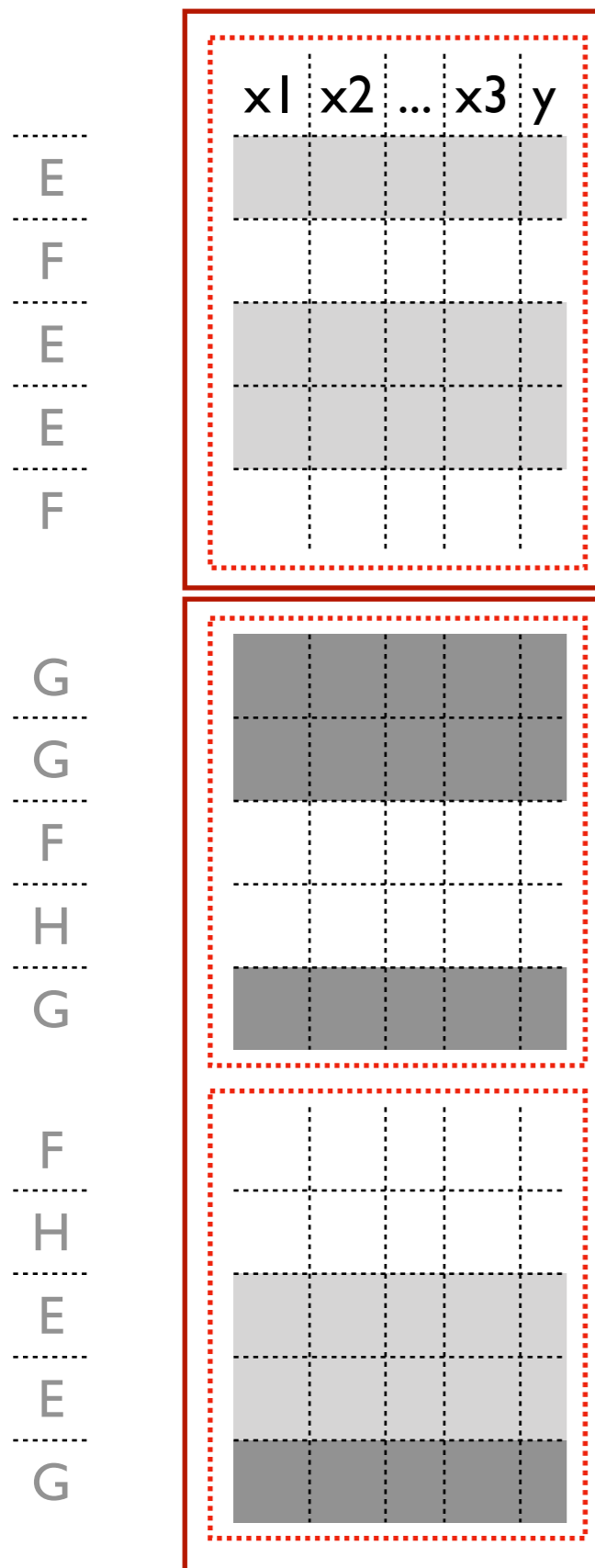
Big Nodes



Big Nodes



Big Nodes



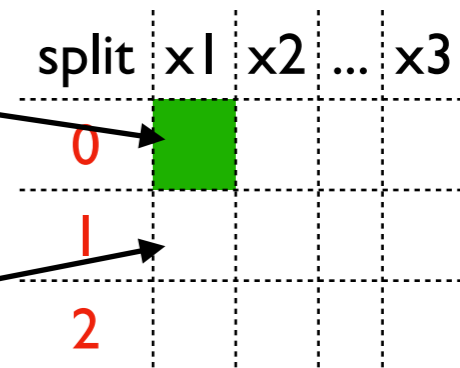
best:

left Y/N = 10/20
right Y/N = 40/10

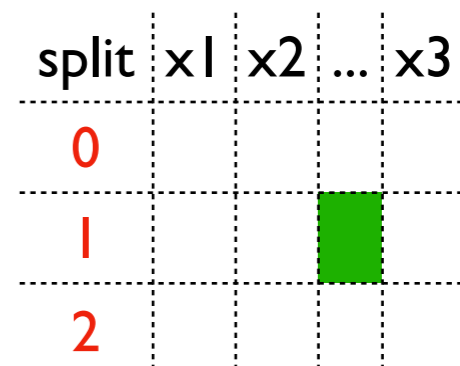
not as good:

left Y/N = 25/20
right Y/N = 25/10

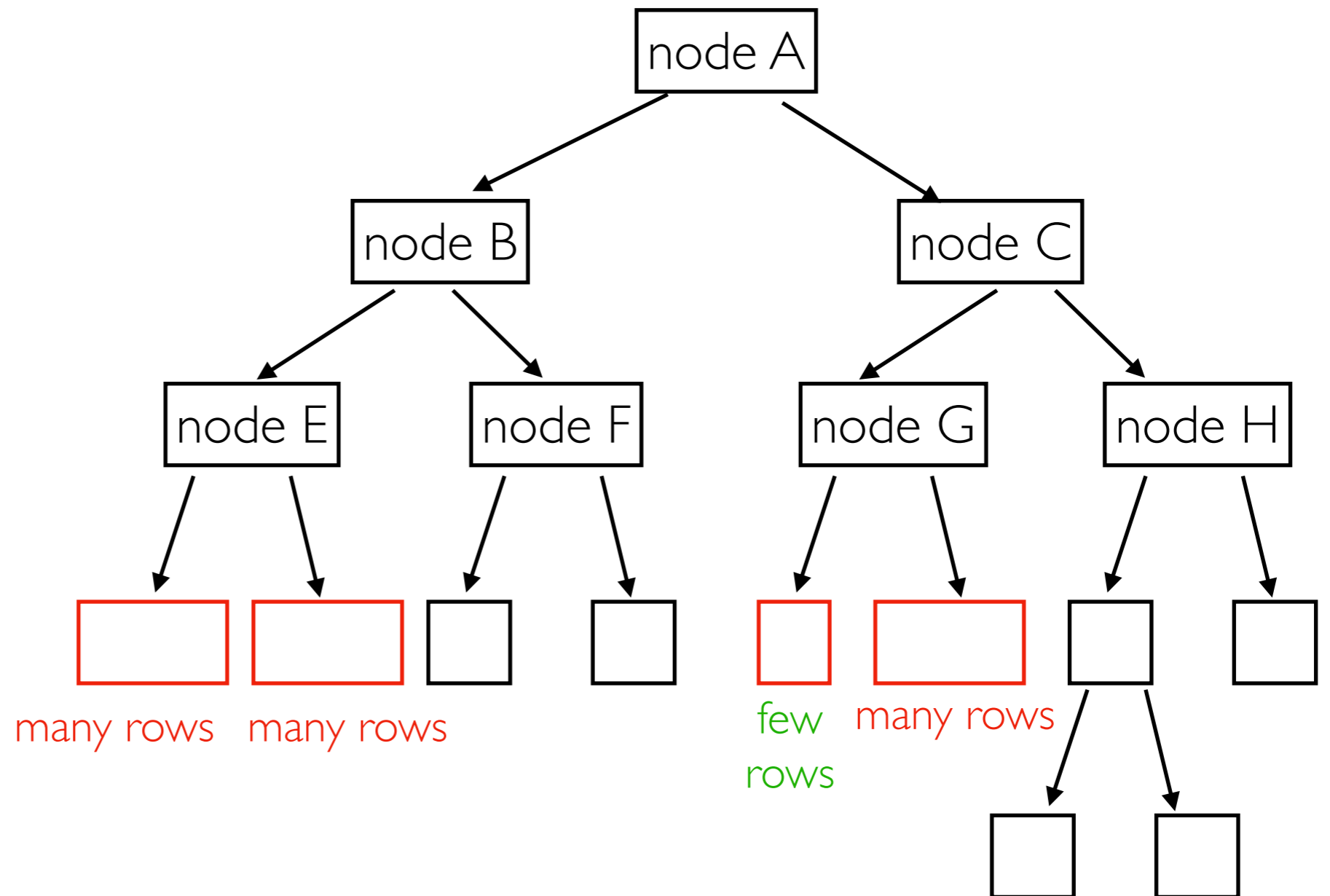
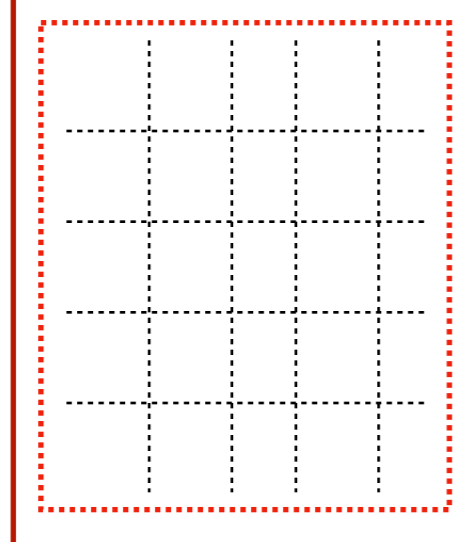
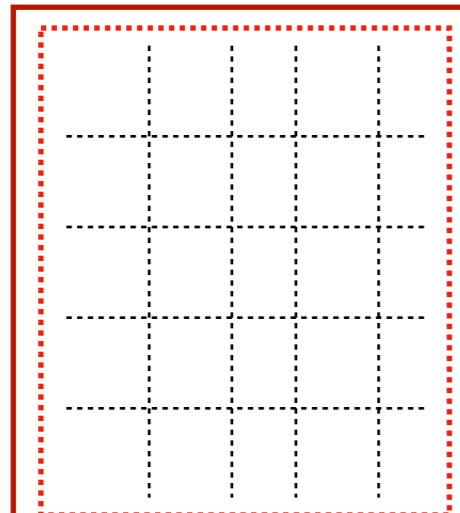
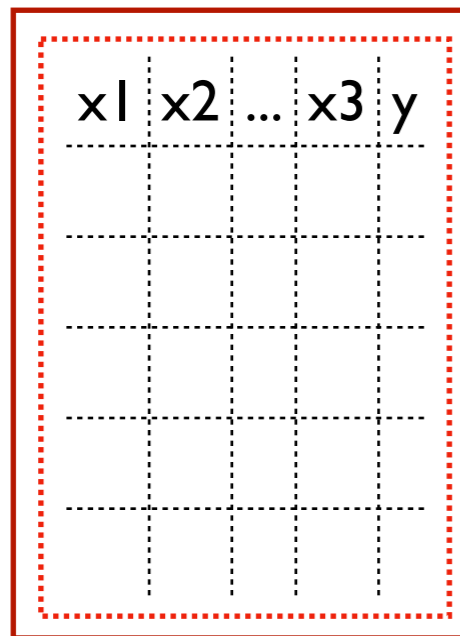
node E (complete)



node G (complete)



- each stats **table** corresponds to a DT **node** we can split (we will choose best split for each node)
- each **column** represents a **feature** we could split on
- each **row** represents a **threshold** we could use for that split



- we split E and G, creating 4 new nodes
- we DID NOT shuffle rows of data
- we DID shuffle statistics about split choices
- recursively keep splitting (either distributed or in-memory, depending on remaining size)

TopHat