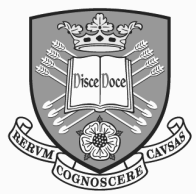
Scalable decision trees and ensembles

Mauricio A. Álvarez

Scalable Machine Learning,
University of Sheffield



The University Of Sheffield.

Decision trees and ensembles in MLAI

- Last semester, as part of the MLAI module, we studied decision trees and ensemble methods.
- There was also a Jupyter Notebook and an Assignment looking at how to use `scikit-learn` to apply these ML methods to actual data.
- In this lecture, we will not explain decision trees and ensemble methods to the same level of detail but will review some relevant aspects necessary to use both models in PySpark.
- The APIs that we use in `spark.ml` are very similar to the ones used in `scikit-learn`, with subtle differences.

Why do we like decision trees?

- easy to interpret.
- handle categorical features.
- extend to the multiclass classification setting.
- do not require feature scaling.
- able to capture non-linearities and feature interactions.

Contents

Nodes in a decision tree

- A decision tree consists of:
  1. a **root node** (or starting node),
  2. **interior nodes**
  3. and **leaf nodes** (or terminating nodes).

- Each of the non-leaf nodes (root and interior) in the tree specifies a test to be carried out on one of the query's descriptive features.

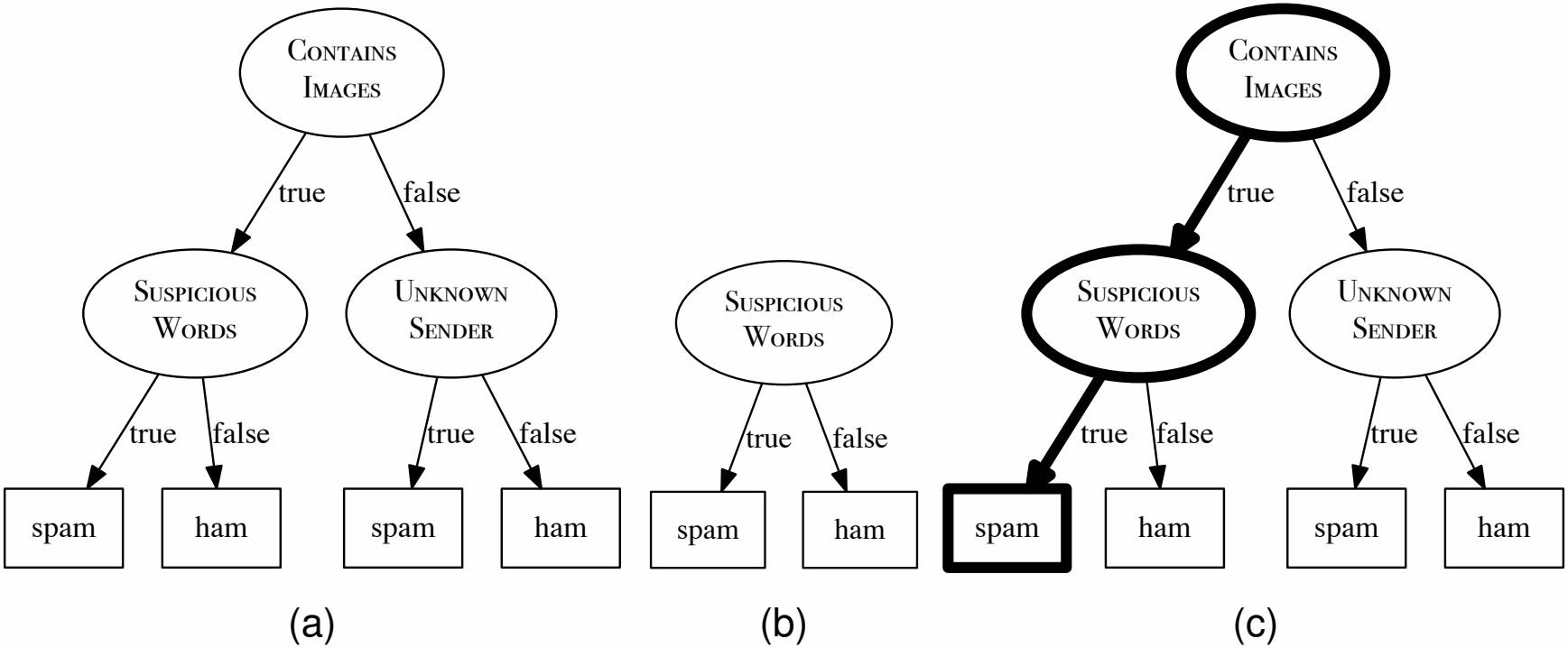
- Each of the leaf nodes specifies a predicted classification or predicted regression value for the query.

Binary classification: spam prediction

An email spam prediction dataset.

ID	Suspicious Words	Unknown Sender	Contains Images	Class
376	true	false	true	spam
489	true	true	false	spam
541	true	true	false	spam
693	false	true	true	ham
782	false	false	false	ham
976	false	false	false	ham

Two decision trees and a query instance

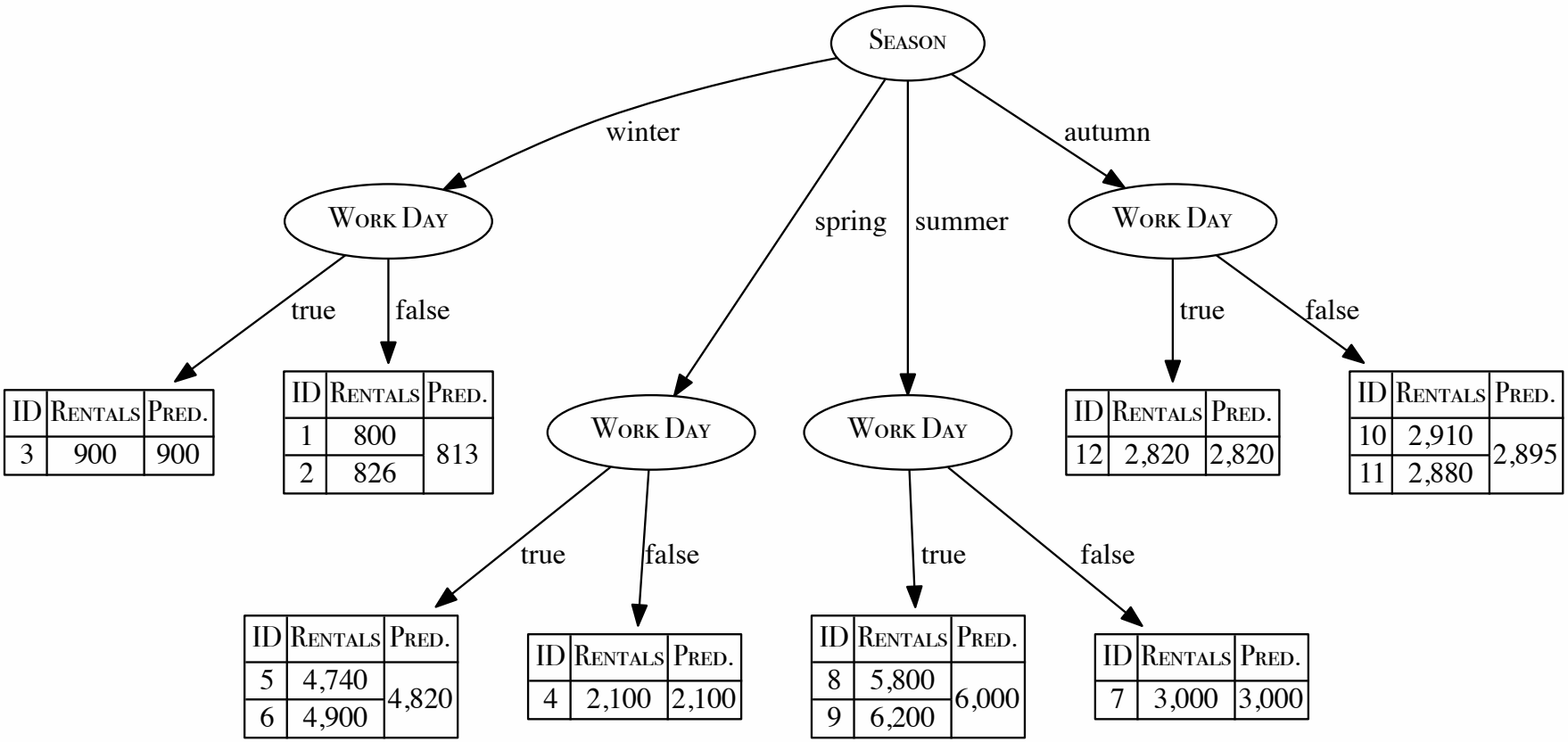


(a) and (b) show two decision trees that are consistent with the instances in the spam dataset. (c) shows the path taken through the tree shown in (a) to make a prediction for the query instance: SUSPICIOUS WORDS = *'true'*, UNKNOWN SENDER = *'true'*, CONTAINS IMAGES = *'true'*.

Regression: bike rentals per day

ID	Season	Work Day	Rentals
1	winter	false	800
2	winter	false	826
3	winter	true	900
4	spring	false	2 100
5	spring	true	4 740
6	spring	true	4 900
7	summer	false	3 000
8	summer	true	5 800
9	summer	true	6 200
10	autumn	false	2 910
11	autumn	false	2 880
12	autumn	true	2 820

Regression tree



ID	RENTALS	PRED.
3	900	900

ID	RENTALS	PRED.
1	800	813
2	826	

ID	RENTALS	PRED.
5	4,740	4,820
6	4,900	

ID	RENTALS	PRED.
4	2,100	2,100

ID	RENTALS	PRED.
8	5,800	6,000
9	6,200	

ID	RENTALS	PRED.
7	3,000	3,000

ID	RENTALS	PRED.
12	2,820	2,820

ID	RENTALS	PRED.
10	2,910	2,895
11	2,880	

The final decision tree induced from the dataset. This tree lists the instances that ended up at each leaf node and the prediction (PRED.) made by each leaf node.

Contents

Contents

Binary decision trees

- Apache Spark only builds binary decision trees: each node can only have two branches.
- Binary partitions at each node are done recursively.
- Each partition is chosen greedily by selecting the *best split* from a set of possible splits.

Best split

- The best split maximizes the *information gain* at a tree node.

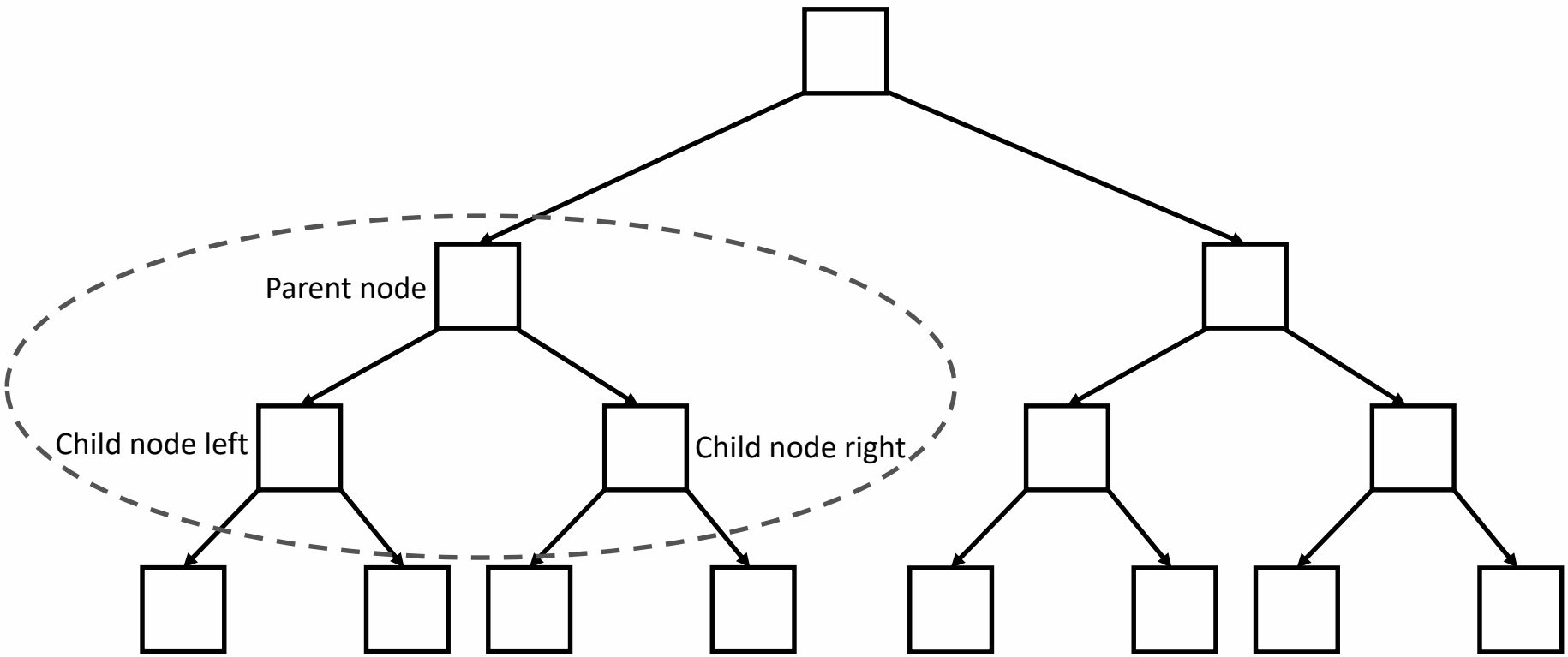
- The split chosen at each tree node is chosen from the set

$$\arg\max_{s} IG(D, s),$$

where $IG(D, s)$ is the information gain when split s is applied to dataset D.

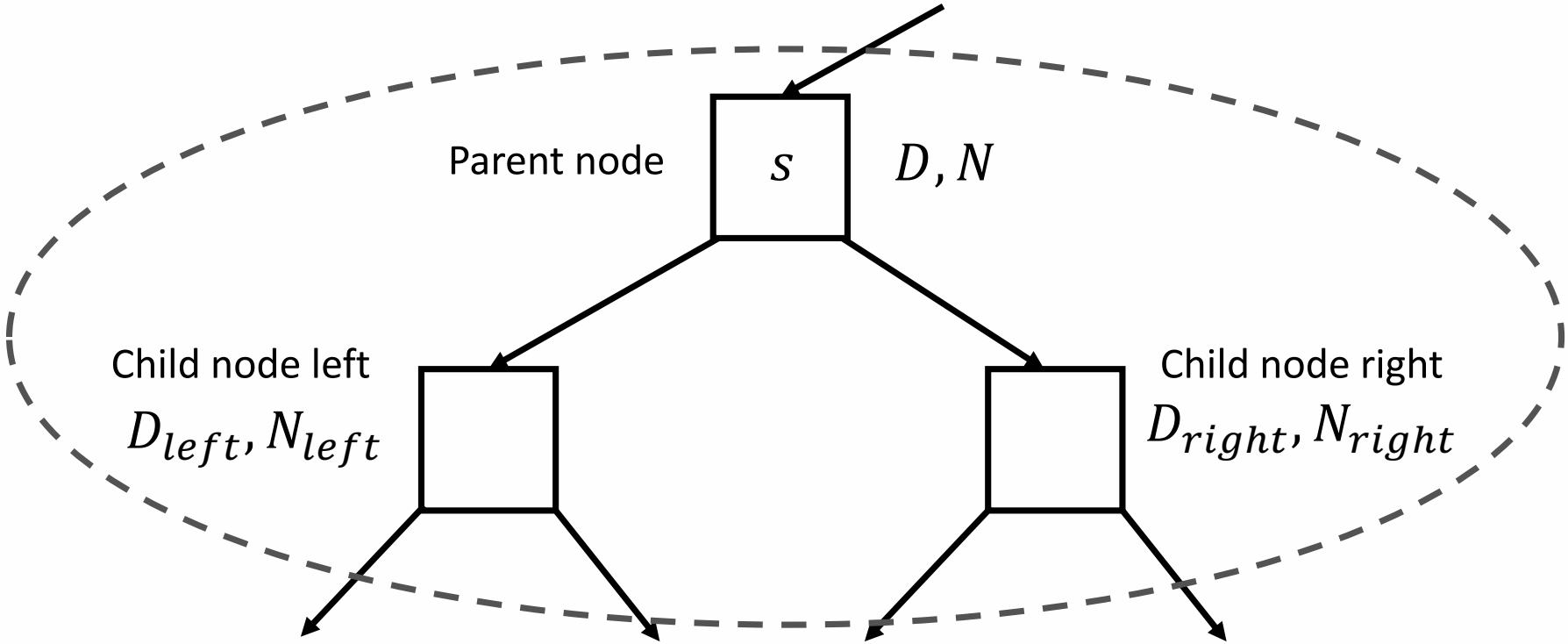
Information gain

The *information gain* is the difference between the parent node impurity and the weighted sum of the two child node impurities.



Information gain

Assuming that a split s partitions the dataset D of size N into two datasets D_{left} and D_{right} of sizes N_{left} and N_{right},



the information gain is computed as

$$IG(D, s) = \text{Impurity}(D) - \left(\frac{N_{\text{left}}}{N} \text{Impurity}(D_{\text{left}}) + \frac{N_{\text{right}}}{N} \text{Impurity}(D_{\text{right}}) \right).$$

Node impurity

- Remember that the node impurity is a measure of how homogeneous are the labels at a node.

- The table below shows the impurity measures currently implemented in Spark

Impurity	Task	Formula	Description
Gini impurity	Classification	$\sum_{i=1}^{C} f_i(1-f_i)$	f_i is the frequency of label i at a node and C is the number of unique labels.
Entropy	Classification	$\sum_{i=1}^{C} -f_i log(f_i)$	f_i is the frequency of label i at a node and C is the number of unique labels.
Variance	Regression	$\frac{1}{N}\sum_{i=1}^{N}(y_i-\mu)^2$	y_i is label for an instance, N is the number of instances and μ is the mean given by $\frac{1}{N}\sum_{i=1}^{N} y_i$.

Frequency refers to probability. Link here

Contents

How are the split candidates chosen?

- We saw before that Spark only implements binary decision trees.

- This means the split is always of the kind:

  if feature $i \leq$ `threshold` **then**
  move to the left branch (or the right branch)
  else
  move to the right branch (or the left branch)
  end if

- So, how is `threshold` computed? It depends on whether the feature is continuous or categorical.

Handling continuous features

- When working with small datasets, we can either:
  - directly use the values of the feature as the thresholds.
  - sort the values of the feature and compute the corresponding thresholds from the sorted values.

- This is what we saw in MLAI.

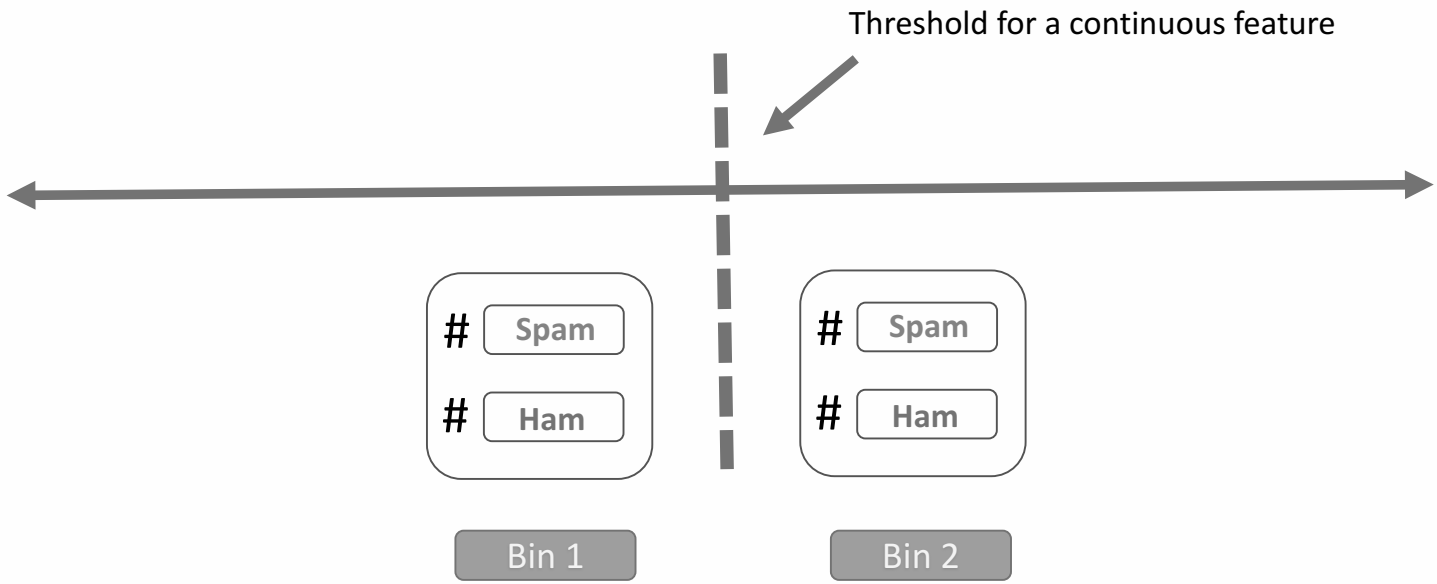
- However, these alternatives are not possible to use for large distributed datasets
  - it is not feasible to use the values of the feature as the thresholds.
  - although sorting is an option, it is expensive.

Handling continuous features

- Spark's implementation computes an approximate set of split candidates by performing a quantile calculation over a sampled fraction of the data.

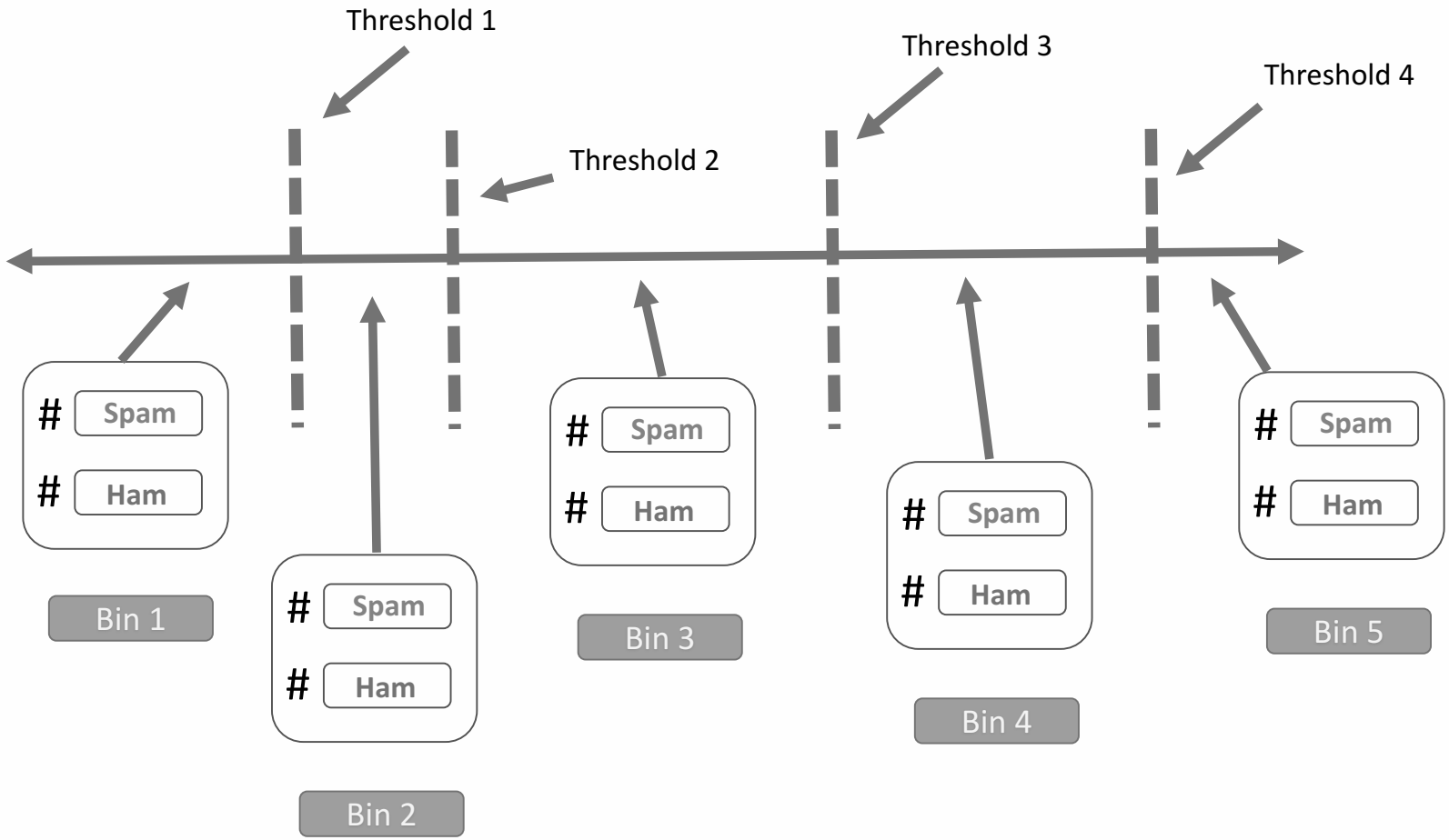
- The ordered splits create "bins" and the maximum number of such bins can be specified using the `maxBins` parameter.

Bins for continuous features



Bins in continuous features

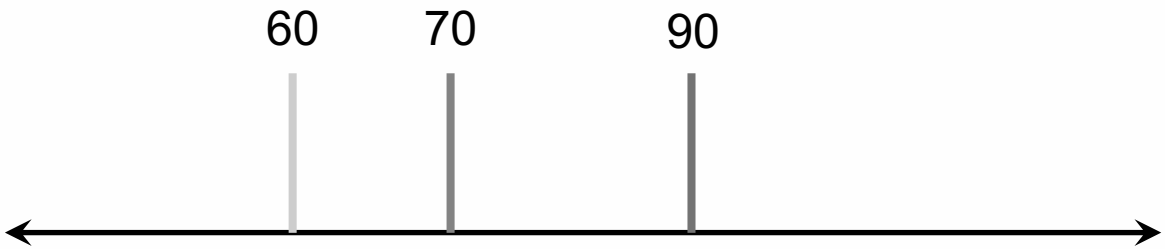
Bins for continuous features



Bins in continuous features

Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham

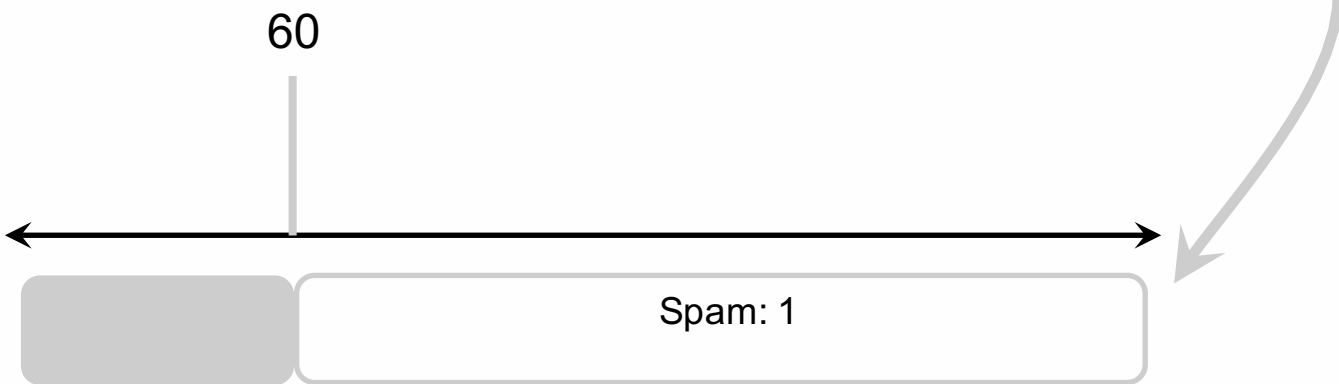


Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham

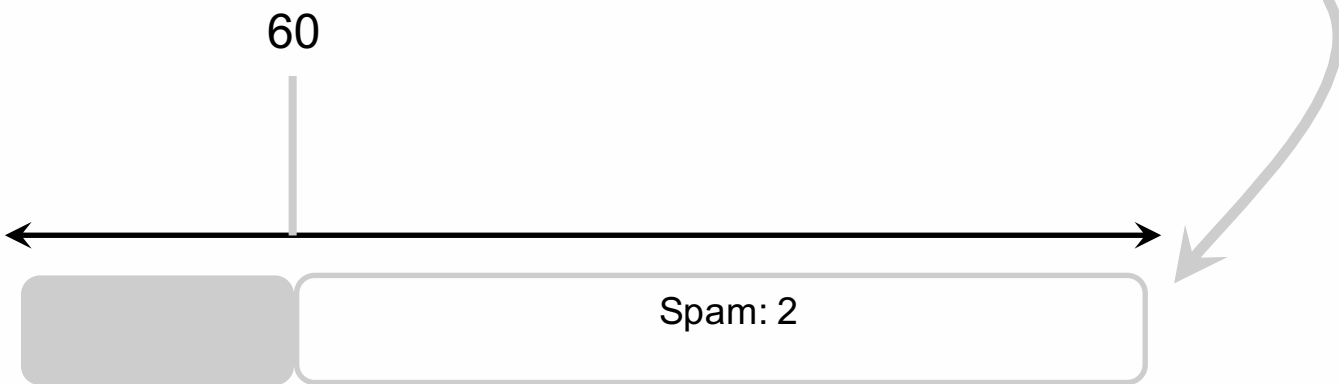
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



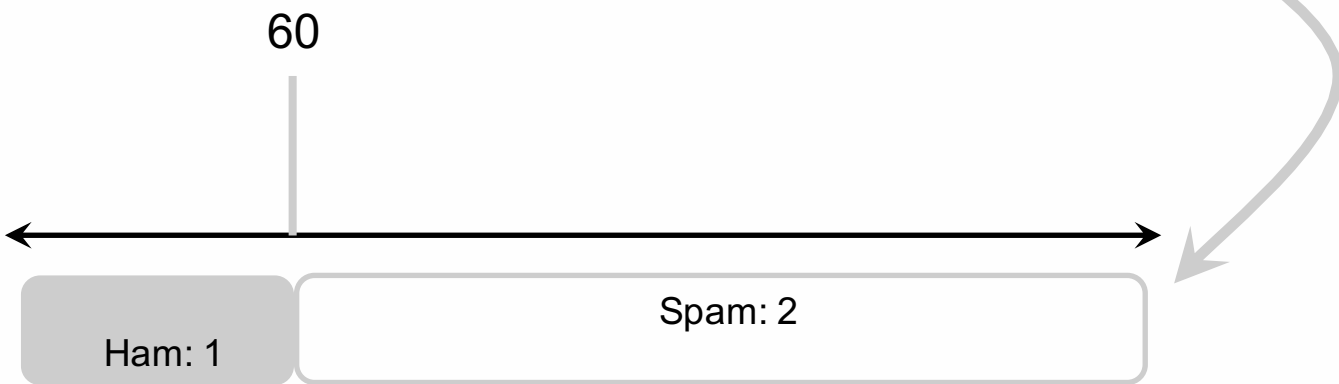
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



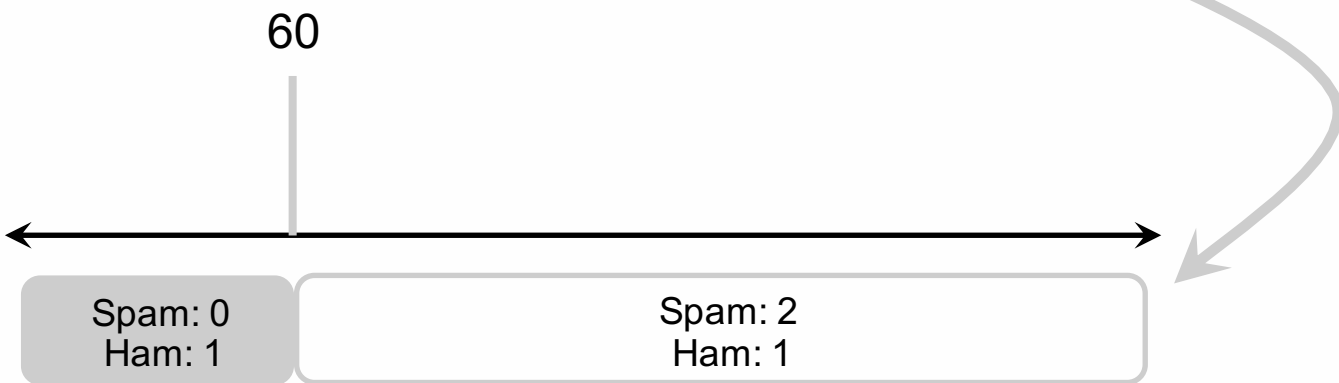
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



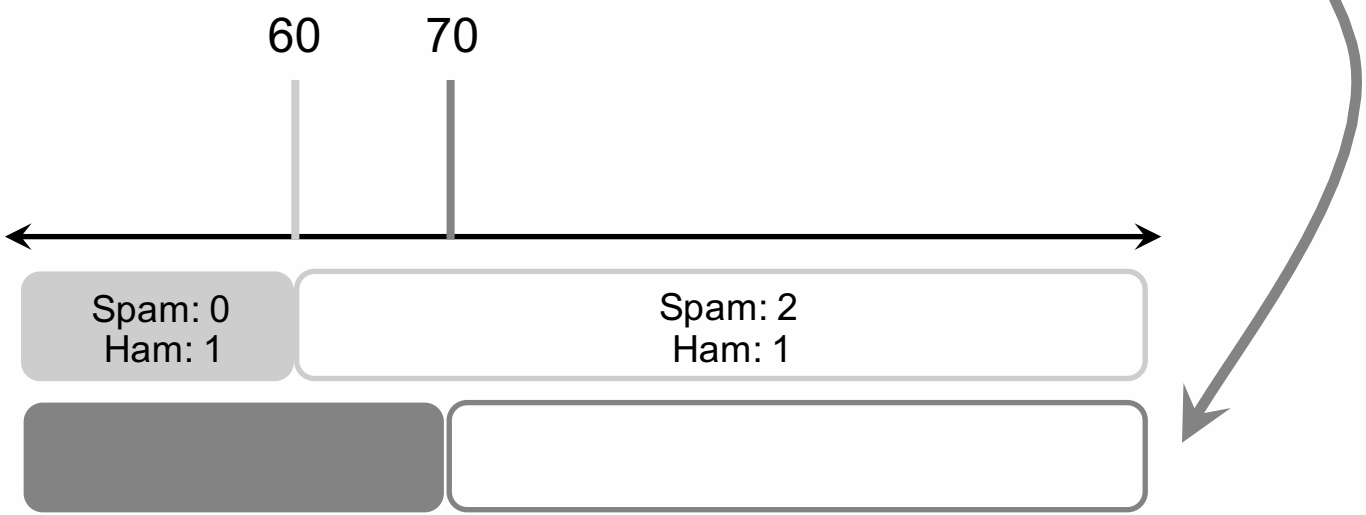
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



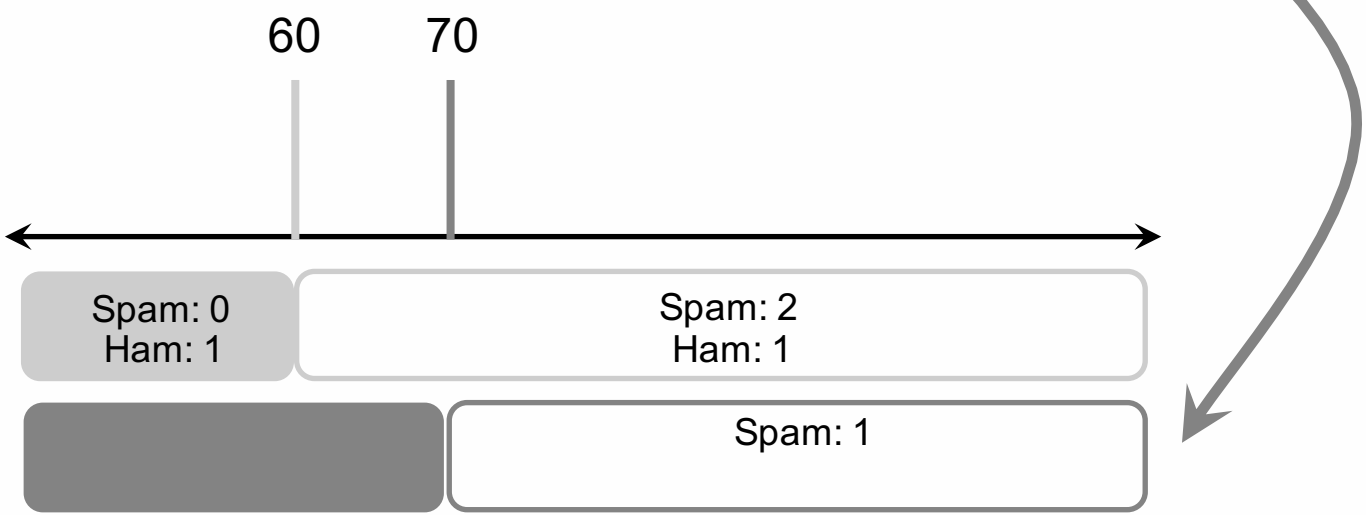
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



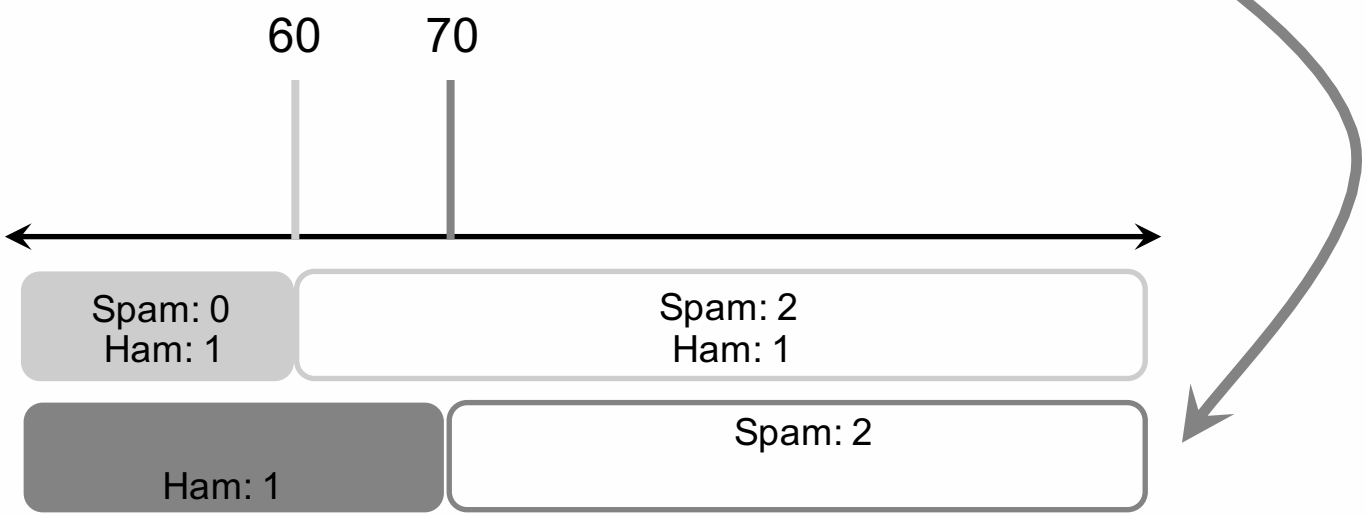
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



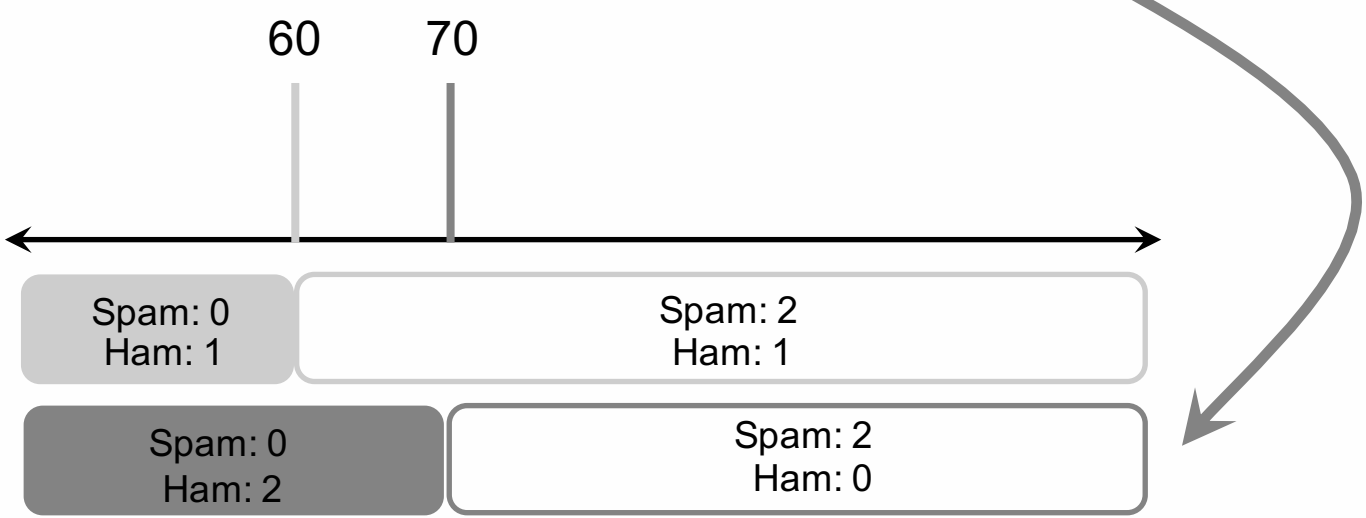
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



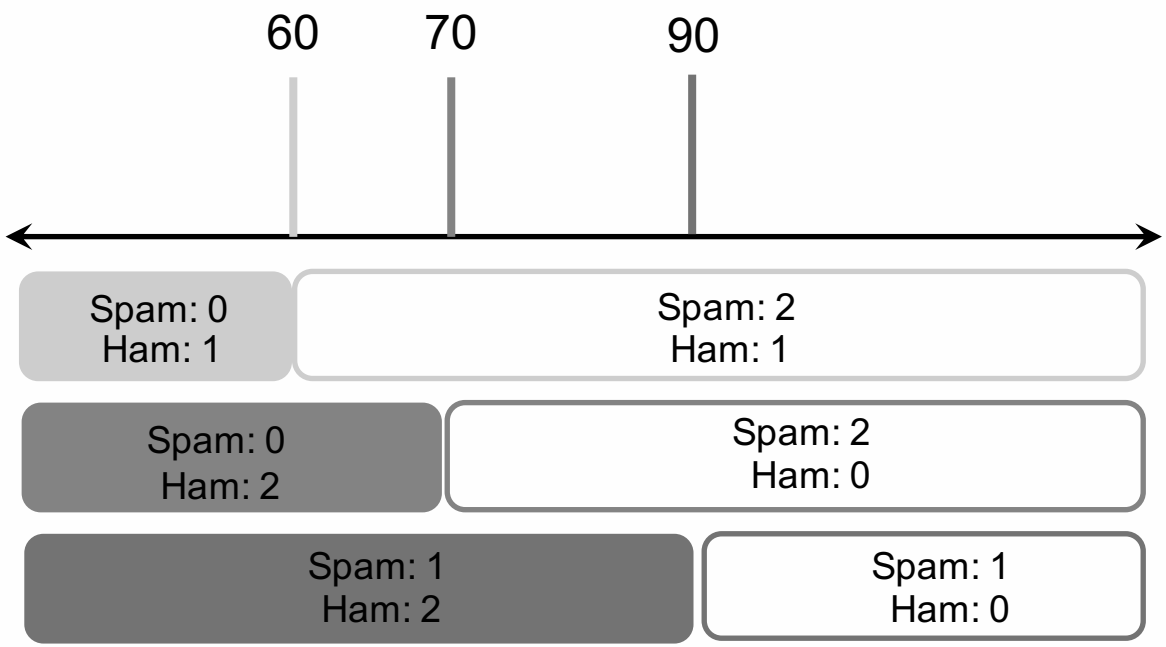
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



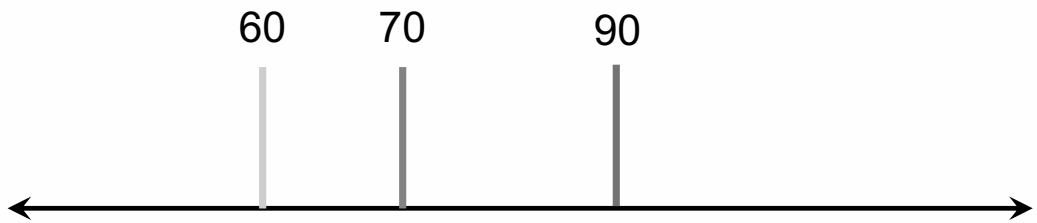
Why bins? Naive approach

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



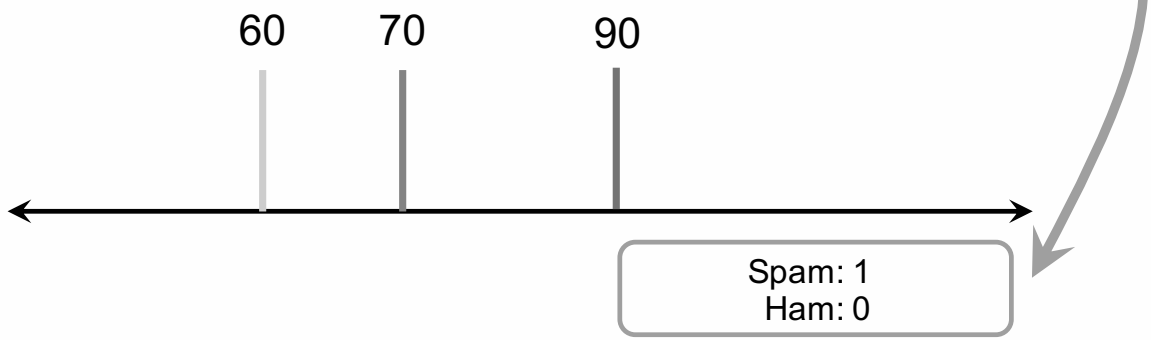
Why bins? Binary search

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



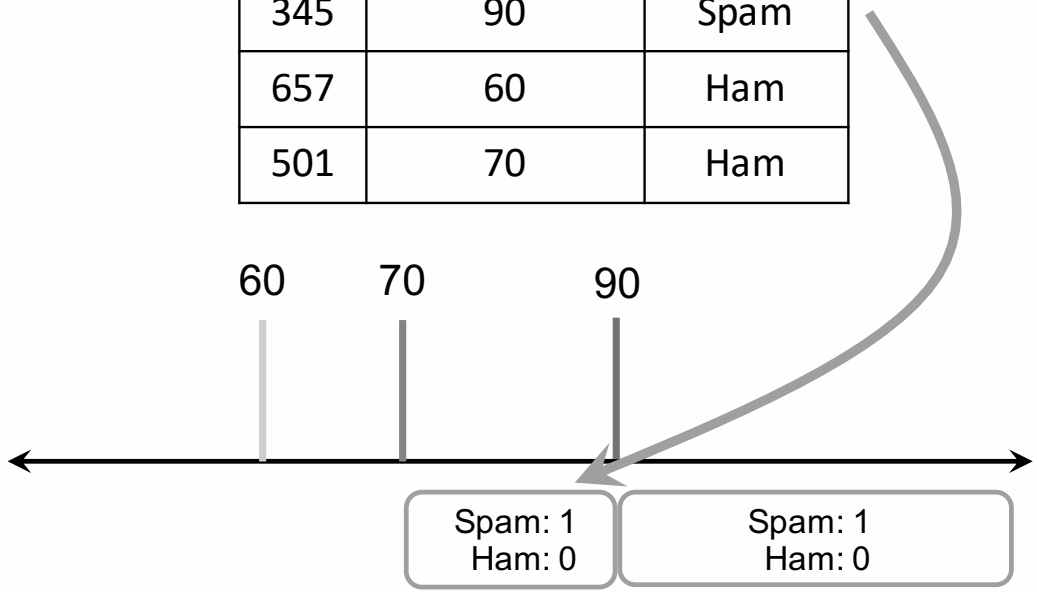
Why bins? Binary search

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



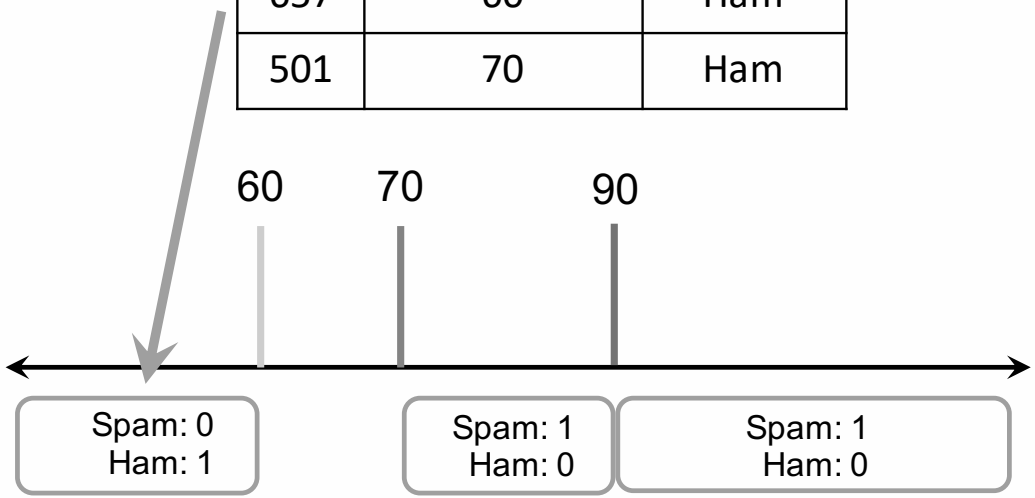
Why bins? Binary search

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



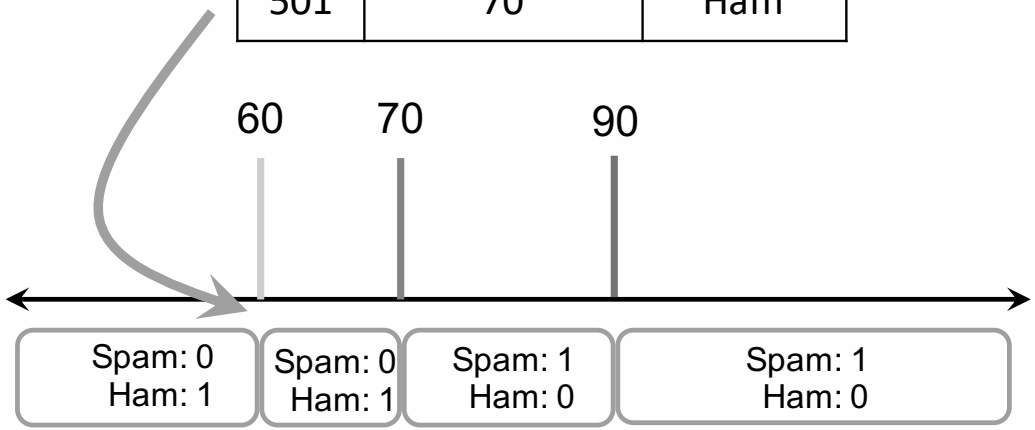
Why bins? Binary search

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



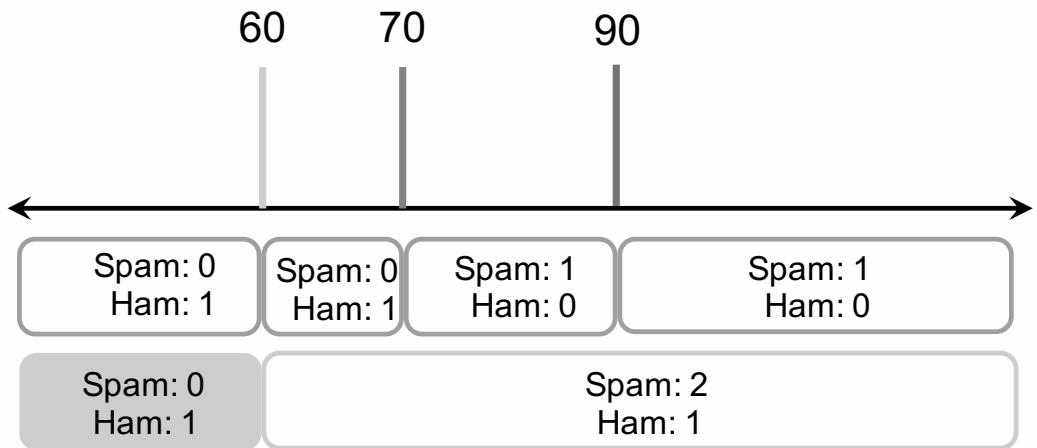
Why bins? Binary search

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



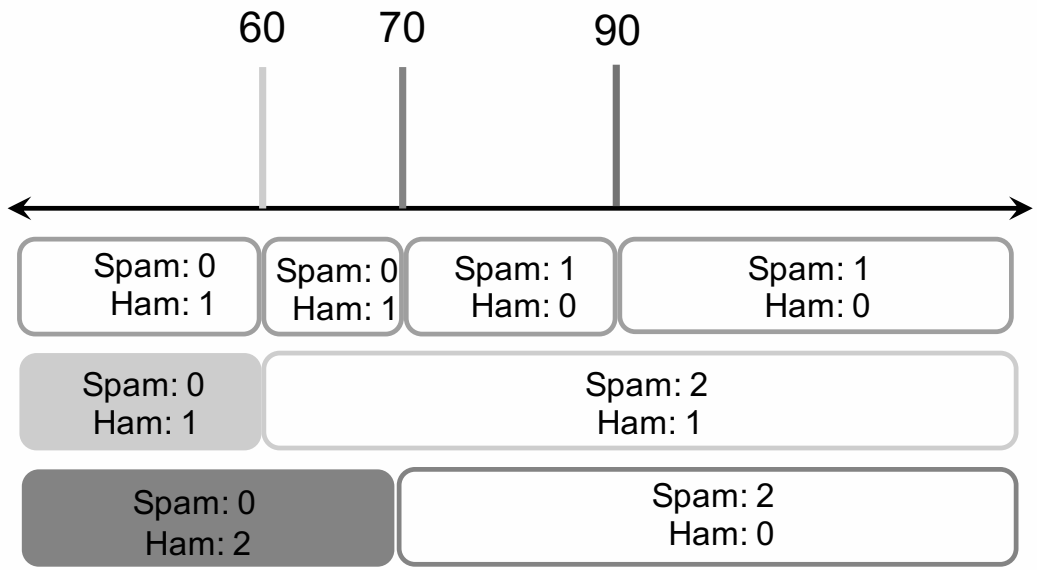
Why bins? Binary search

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



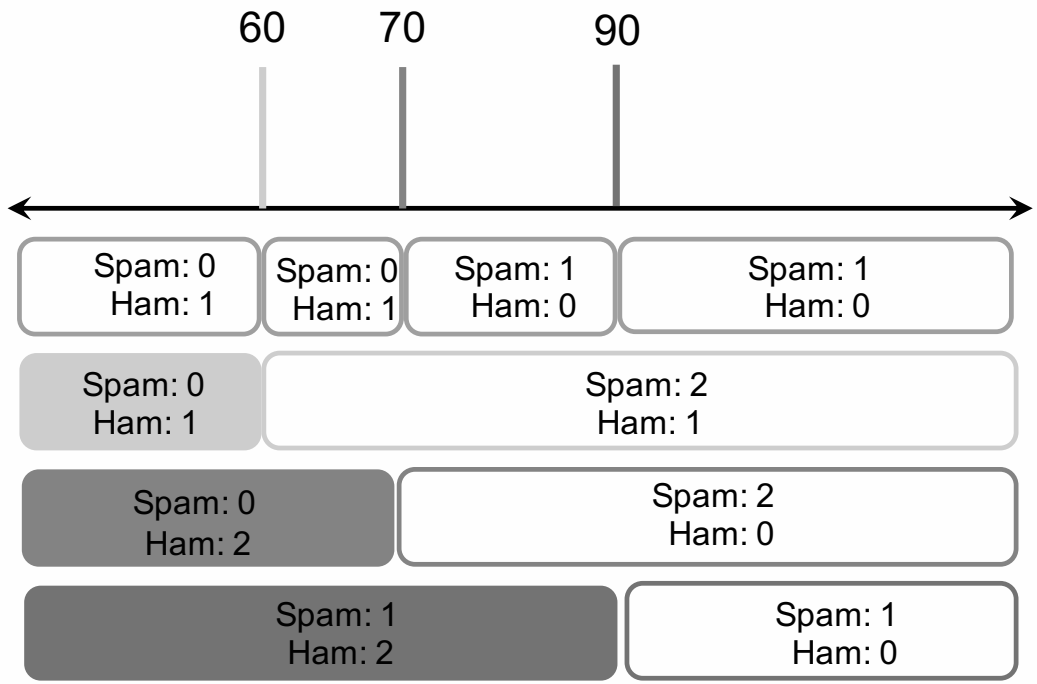
Why bins? Binary search

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



Why bins? Binary search

ID	Frequency of word “free”	Class
908	95	Spam
345	90	Spam
657	60	Ham
501	70	Ham



What is the difference?

- Assume there are m threholds.
- In the naive approach, for each observation, we need to perform m operations.
- When using binning, it is possible to use binary search for each observation to fill out the bins.
- Binary search takes $\log(m)$ operations.

Categorical features

- Say we have a categorical feature that can take M unordered values.
- We need to make a binary partition based on this categorical feature.
- It can be shown there are $2^{M-1} - 1$ possible partitions of the M values into two groups.

Categorical features

- As an example, consider the feature `weather` that takes values `[spring, summer, autumn, winter]`

- We then have $2^{M-1} - 1 = 2^{4-1} - 1 = 7$ possible partitions of two groups

Option	Group 1	Group 2
1	`spring`	`summer, autumn, winter`
2	`summer`	`spring, autumn, winter`
3	`autumn`	`summer, spring, winter`
4	`winter`	`summer, autumn, spring`
5	`spring, summer`	`autumn, winter`
6	`spring, autumn`	`summer, winter`
7	`spring, winter`	`summer, autumn`

- For M large, computation becomes prohibitive.

- There are heuristics to optimally reduce the number of partitions depending on the type of prediction problem.

Categorical features: binary classification

- For binary classification, the number of partitions can be reduced to $M - 1$ by ordering the categorical feature values by the proportion falling in outcome class 1.

- For example, for the feature `weather` say that the proportions for outcome class 1 according to the categorical values are

Categorical value	Proportion
`spring`	0.2
`summer`	0.3
`autumn`	0.05
`winter`	0.45

- Ordering the categorical values, we get `[autumn, spring, summer, winter]` and the $M - 1 = 3$ partitions are

Option	Group 1	Group 2
1	`autumn`	`spring, summer, winter`
2	`autumn, spring`	`summer, winter`
3	`autumn, spring, summer`	`winter`

- Essentially, we are transforming an unordered feature into an ordered feature.

Categorical features: regression

- One can show this gives the optimal split, in terms of cross-entropy or Gini index, among all possible $2^{M-1} - 1$ splits.
- This result also holds for a regression problem when using the square error loss.
- In this case, the categories are ordered by increasing mean of the outcome.
- For more details, see The Elements of Statistical Learning, 2nd Edition, section 9.2.4

Categorical features: multi-class classification

- In Spark, for multiclass classification, all $2^{M-1} - 1$ possible splits are used whenever possible.
- When $2^{M-1} - 1$ is greater than the `maxBins` parameter, Spark uses a (heuristic) method similar to the method used for binary classification and regression.
- The M categorical feature values are ordered by impurity, and the resulting $M - 1$ split candidates are considered.

Contents

APIs for decision trees

- The class for classification is `DecisionTreeClassifier`.
- The class for regression is `DecisionTreeRegressor`.

The `DecisionTreeClassifier` class

class `sklearn.tree.` **`DecisionTreeClassifier`** (*criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, class_weight=None, presort=False*)

[source]

`DecisionTreeClassifier` class in scikit-learn

class `pyspark.ml.classification.`**`DecisionTreeClassifier`**(*featuresCol='features', labelCol='label', predictionCol='prediction', probabilityCol='probability', rawPredictionCol='rawPrediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, impurity='gini', seed=None*)

[source]

`DecisionTreeClassifier` class in pyspark

The `DecisionTreeRegressor` class

class `sklearn.tree.` **`DecisionTreeRegressor`**(**, criterion='mse', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, ccp_alpha=0.0*) [source]

`DecisionTreeRegressor` class in scikit-learn

class `pyspark.ml.regression.` **`DecisionTreeRegressor`**(*featuresCol='features', labelCol='label', predictionCol='prediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, impurity='variance', seed=None, varianceCol=None, weightCol=None, leafCol='', minWeightFractionPerNode=0.0*) [source]

`DecisionTreeRegressor` class in pyspark

Parameters to adjust

- **`maxDepth`**: Maximum depth of a tree.
- **`maxBins`**: Max number of bins for discretizing continuous features. Must be $>=$ 2 and $>=$ number of categories for any categorical feature.
- **`impurity`**: Criterion used for information gain calculation (case-insensitive). Supported options: `entropy` or `gini` for classification and `variance` for regression.
- There are several other parameters some of which will be reviewed in the Lab session.

Contents

PLANET: horizontal partitioning

- PLANET is the standard algorithm to train a decision tree in a distributed dataset or horizontal partitioning.

- Let $\mathbf{X} \in \mathbb{R}^{n \times p}$ be the input matrix

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix},$$

  where $\mathbf{x}_k \in \mathbb{R}^{p \times 1}$, meaning that there are p features.

- Horizontal partitioning refers to the fact that each worker will receive a subset of the n instances or row vectors.

Setup

- The following description is in the paper Yggdrasil: An Optimized System for Training Deep Decision Trees at Scale
- Assume there are B potential thresholds for each of the p features considered.
- Let us define $\mathcal{S}$ as the set of cardinality $p \times B$ that contains all the split candidates.
- Assume there are k workers. Each worker j computes c sufficient statistics over a subset of the original data, $\mathbf{X}_j$.
- What are these sufficient statistics?
  - classification: label counts (e.g. $c = 4$ for binary classification, 2 per branch).
  - regression: count, sum, and sum^2 (so, $c = 6$, 3 per branch).

Setup

- Each worker j then computes the sufficient statistics $g_j(s)$ over $\mathbf{X}_j$ for each $s \in \mathcal{S}$.

- For a parent node i in the tree, the optimal split s^* is found by solving

$$s^* = \arg\max_{s \in \mathcal{S}} f\left(\sum_{j=1}^{k} g_j(s)\right),$$

  where $f : \mathbb{R}^c \to \mathbb{R}$ computes the information gains.

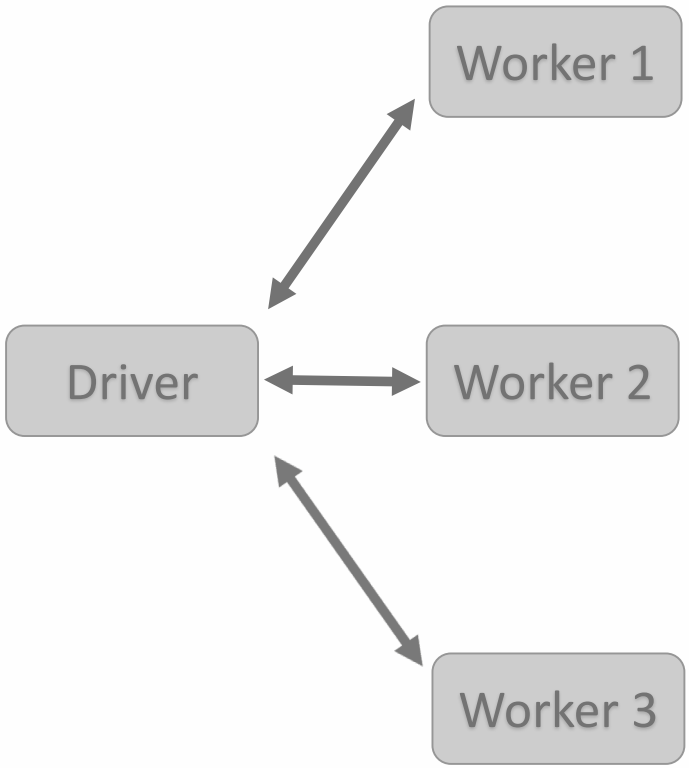
- Notice also that the function $g_j(s)$ computed using the subset of the data $\mathbf{X}_j$ can be computed using a map-reduce approach at the level of the worker.

- The description above leads to a natural distributed algorithm.

Algorithm to compute s^* in a distributed fashion

- We start by assuming the depth of the tree is fixed to D.
- At iteration t,
  - the algorithm computes the optimal splits for **all** the nodes on the level t of the tree.
  - there is a single round trip of communication between the master and the workers.
- Each tree node i is split as follows
  1. The j-th worker locally computes sufficient statistics $g_j(s)$ for all $s \in \mathcal{S}$.
  2. Each worker communicates all statistics $g_j(s)$ to the master (Bp in total).
  3. The master computes the best split s^*.
  4. The master broadcasts s^* to the workers, who update their local states to keep track of which instances are assigned to which child nodes.

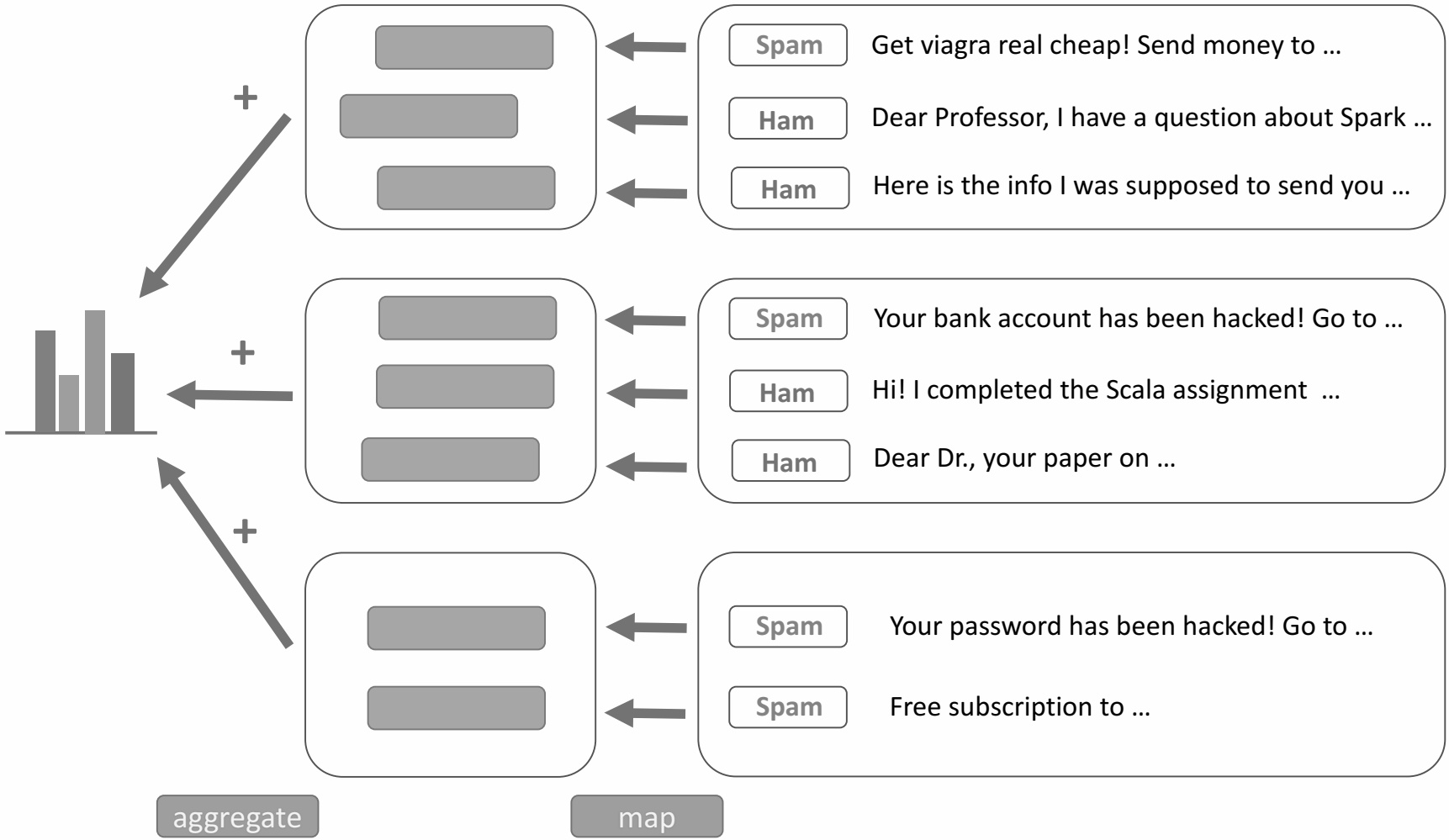
The distributed dataset



Spam	Get viagra real cheap! Send money to …
Ham	Dear Professor, I have a question about Spark …
Ham	Here is the info I was supposed to send you …
Spam	Your bank account has been hacked! Go to …
Ham	Hi! I completed the Scala assignment …
Ham	Dear Dr., your paper on …
Spam	Your password has been hacked! Go to …
Spam	Free subscription to …

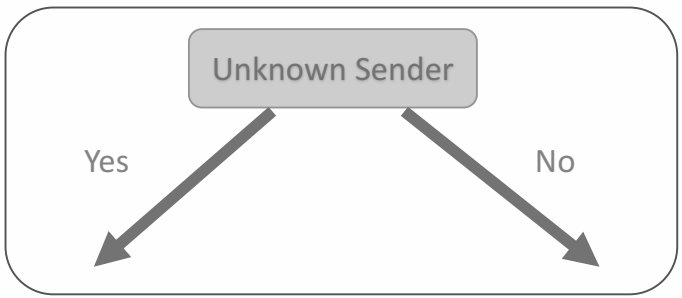
A distributed dataset

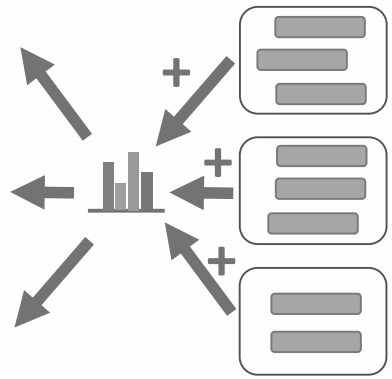
The distributed dataset



A distributed dataset

Building a decision tree

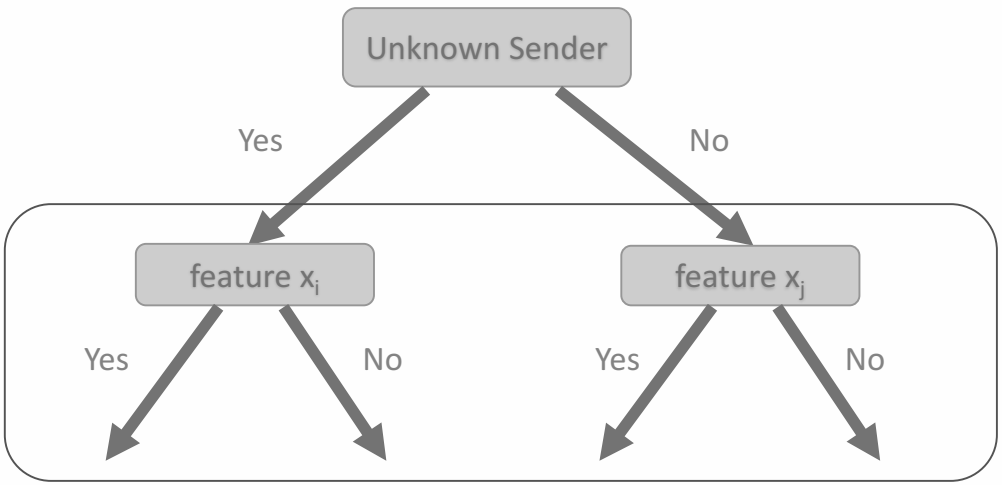


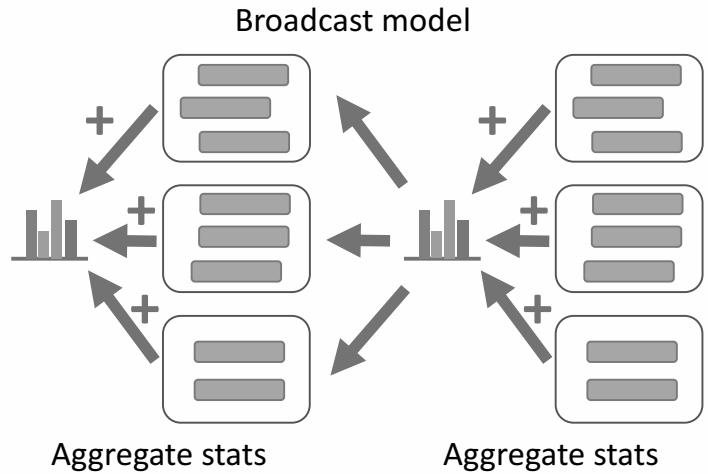


Aggregate stats

Building the tree in Spark

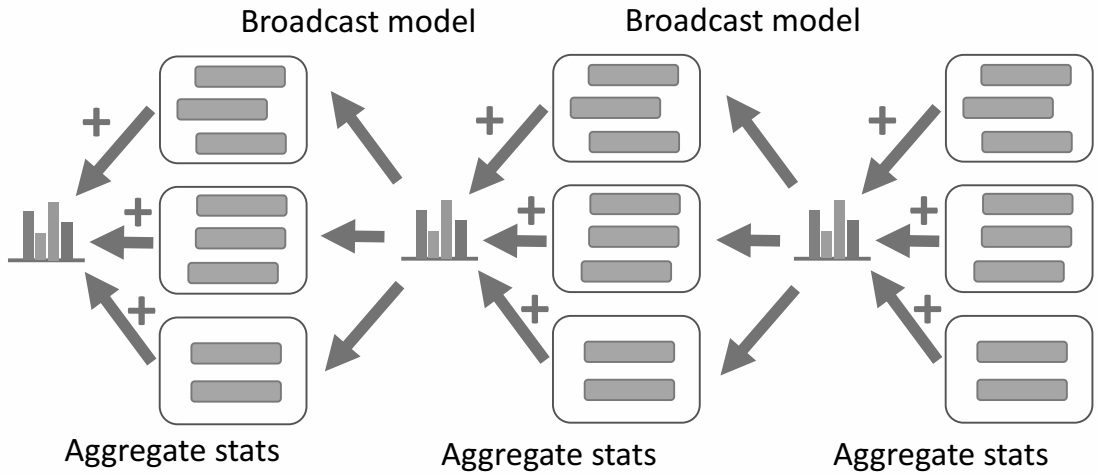
Building a decision tree

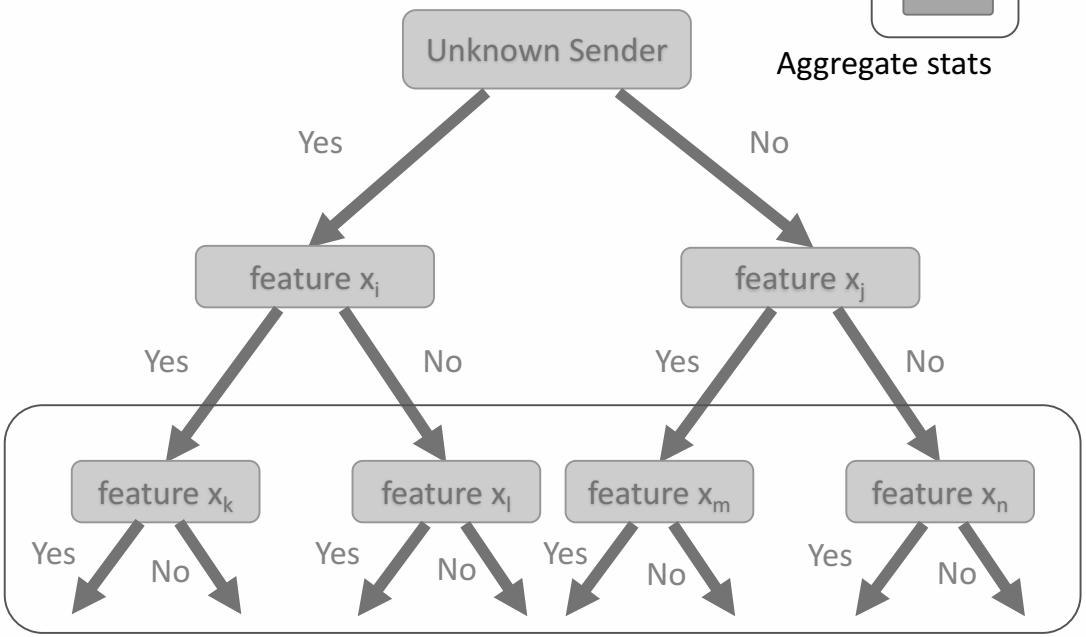




Building the tree in Spark

Building a decision tree





Building the tree in Spark

Computational and communication complexity

- Computation is linear in n, p, and D, so easy to parallelize.
- The issue is the communication overhead.
- For each tree node, step 2 in the algorithm before, requires communicating kBp tuples of size c.
- With 2^D total nodes, the total communication is $2^D kBpc$ floating point values.
- Communication is then exponential in tree depth D and linear in thresholds B.

Contents

Bagging and boosting

- Rather than creating a single model they generate a set of models and then make predictions by aggregating the outputs of these models.
- A prediction model that is composed of a set of models is called a **model ensemble**.
- In order for this approach to work the models that are in the ensemble must be different from each other.
- There are two standard approaches to creating ensembles: bagging and boosting.

Bagging: Definition

- When we use **bagging** (or **bootstrap aggregating**) each model in the ensemble is trained on a random sample of the dataset known as **bootstrap samples**.

- Each random sample is the same size as the dataset and **sampling with replacement** is used.

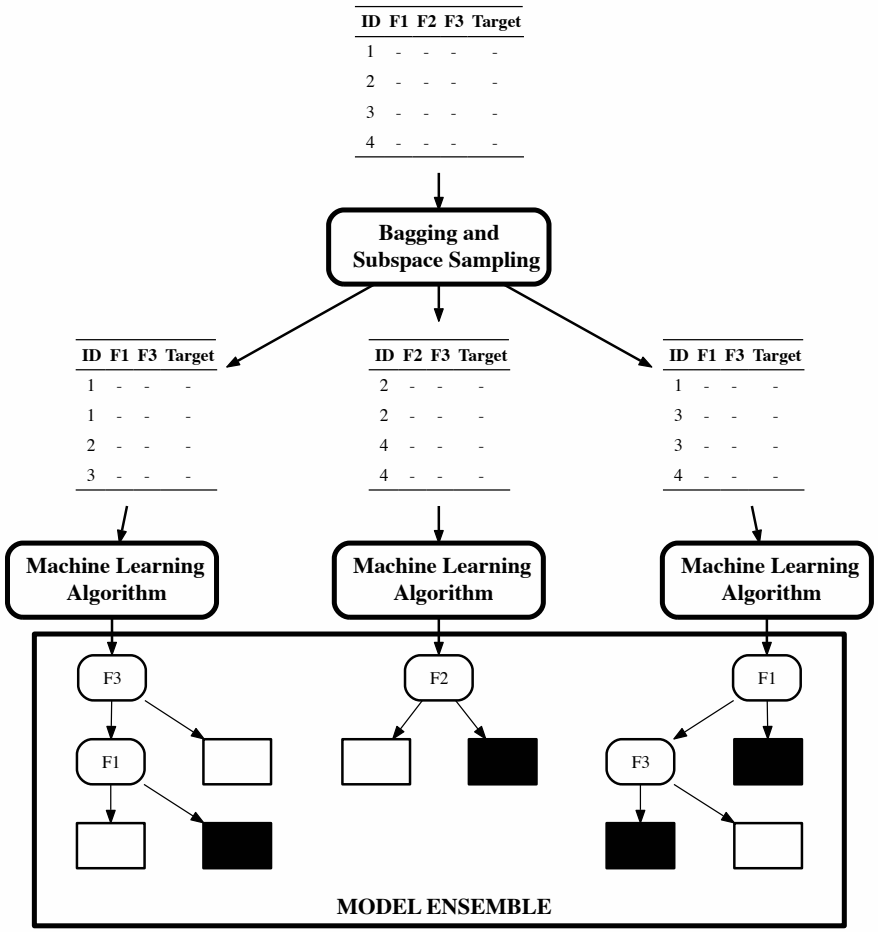
- Consequently, every bootstrap sample will be missing some of the instances from the dataset so each bootstrap sample will be different and this means that models trained on different bootstrap samples will also be different

Bagging: Random forest

- When bagging is used with decision trees each bootstrap sample only uses a randomly selected subset of the descriptive features in the dataset. This is known as **subspace sampling**.

- The combination of bagging, subspace sampling, and decision trees is known as a **random forest** model.

Example of using bagging



The process of creating a model ensemble using bagging and subspace sampling.

Boosting: How does it work?

- Boosting works by iteratively creating models and adding them to the ensemble.
- The iteration stops when a predefined number of models have been added.
- When we use **boosting** each new model added to the ensemble is biased to pay more attention to instances that previous models miss-classified.
- This is done by incrementally adapting the dataset used to train the models. To do this we use a **weighted dataset**

Boosting: Weighted Dataset

- Each instance has an associated weight $\mathbf{w}_i \geq 0$,
- Initially set to $\frac{1}{n}$ where n is the number of instances in the dataset.
- After each model is added to the ensemble it is tested on the training data and the weights of the instances the model gets correct are decreased and the weights of the instances the model gets incorrect are increased.
- These weights are used as a distribution over which the dataset is sampled to created a replicated training set, where the replication of an instance is proportional to its weight.

Algorithm

During each training iteration the algorithm:

1. Induces a model and calculates the total error, ϵ, by summing the weights of the training instances for which the predictions made by the model are incorrect.
2. Increases the weights for the instances misclassified using:

$$\mathbf{w}[i] \leftarrow \mathbf{w}[i] \times \left(\frac{1}{2 \times \epsilon} \right)$$

3. Decreases the weights for the instances correctly classified:

$$\mathbf{w}[i] \leftarrow \mathbf{w}[i] \times \left(\frac{1}{2 \times (1 - \epsilon)} \right)$$

4. Calculate a **confidence factor**, α, for the model such that α increases as ϵ decreases:

$$\alpha = \frac{1}{2} \times log_e \left(\frac{1 - \epsilon}{\epsilon} \right)$$

Predictions

- Once the set of models have been created the ensemble makes predictions using a weighted aggregate of the predictions made by the individual models.

- The weights used in this aggregation are simply the confidence factors associated with each model.

Contents

Bagging and Boosting in PySpark

- Random Forests are implemented in PySpark.
- The boosting model implemented in PySpark is the Gradient-Boosted Trees (GBTs).

Random forests

- `spark.ml` supports random forests for binary and multiclass classification and for regression, using both continuous and categorical features.

- `spark.ml` implements random forests using the existing decision tree implementation.

API Random forests

class `pyspark.ml.classification.`**`RandomForestClassifier`**(*featuresCol='features', labelCol='label', predictionCol='prediction', probabilityCol='probability', rawPredictionCol='rawPrediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, impurity='gini', numTrees=20, featureSubsetStrategy='auto', seed=None, subsamplingRate=1.0, leafCol='', minWeightFractionPerNode=0.0, weightCol=None, bootstrap=True*) [source]

`RandomForestClassifier` class in pyspark

class `pyspark.ml.regression.`**`RandomForestRegressor`**(*featuresCol='features', labelCol='label', predictionCol='prediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, impurity='variance', subsamplingRate=1.0, seed=None, numTrees=20, featureSubsetStrategy='auto', leafCol='', minWeightFractionPerNode=0.0, weightCol=None, bootstrap=True*) [source]

`RandomForestRegressor` class in pyspark

Two important parameters to adjust

- **`numTrees`**: Number of trees in the forest.
- **`maxDepth`**: Maximum depth of each tree in the forest.
- There are several other parameters some of which will be reviewed in the Lab session.

Gradient-boosted trees

- `spark.ml` supports GBTs for binary classification and for regression, using both continuous and categorical features.
- `spark.ml` implements GBTs using the existing decision tree implementation.
- GBTs do not yet support multiclass classification.

API Gradient-boosted trees

class `pyspark.ml.classification.`**`GBTClassifier`**(*featuresCol='features', labelCol='label', predictionCol='prediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, lossType='logistic', maxIter=20, stepSize=0.1, seed=None, subsamplingRate=1.0, impurity='variance', featureSubsetStrategy='all', validationTol=0.01, validationIndicatorCol=None, leafCol='', minWeightFractionPerNode=0.0, weightCol=None*) [source]

`GBTClassifier` class in pyspark

class `pyspark.ml.regression.`**`GBTRegressor`**(*featuresCol='features', labelCol='label', predictionCol='prediction', maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0, maxMemoryInMB=256, cacheNodeIds=False, subsamplingRate=1.0, checkpointInterval=10, lossType='squared', maxIter=20, stepSize=0.1, seed=None, impurity='variance', featureSubsetStrategy='all', validationTol=0.01, validationIndicatorCol=None, leafCol='', minWeightFractionPerNode=0.0, weightCol=None*) [source]

`GBTRegressor` class in pyspark

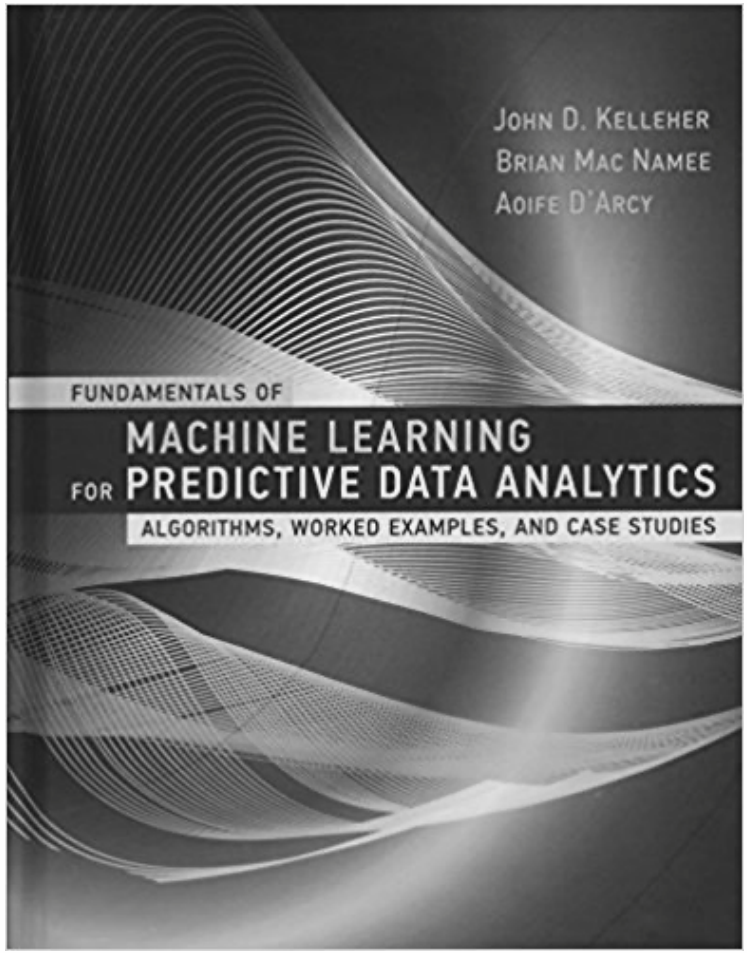
Parameters to adjust

- The parameters to adjust are similar to the ones used for random forests.
- They will be reviewed in the Lab session.

Contents

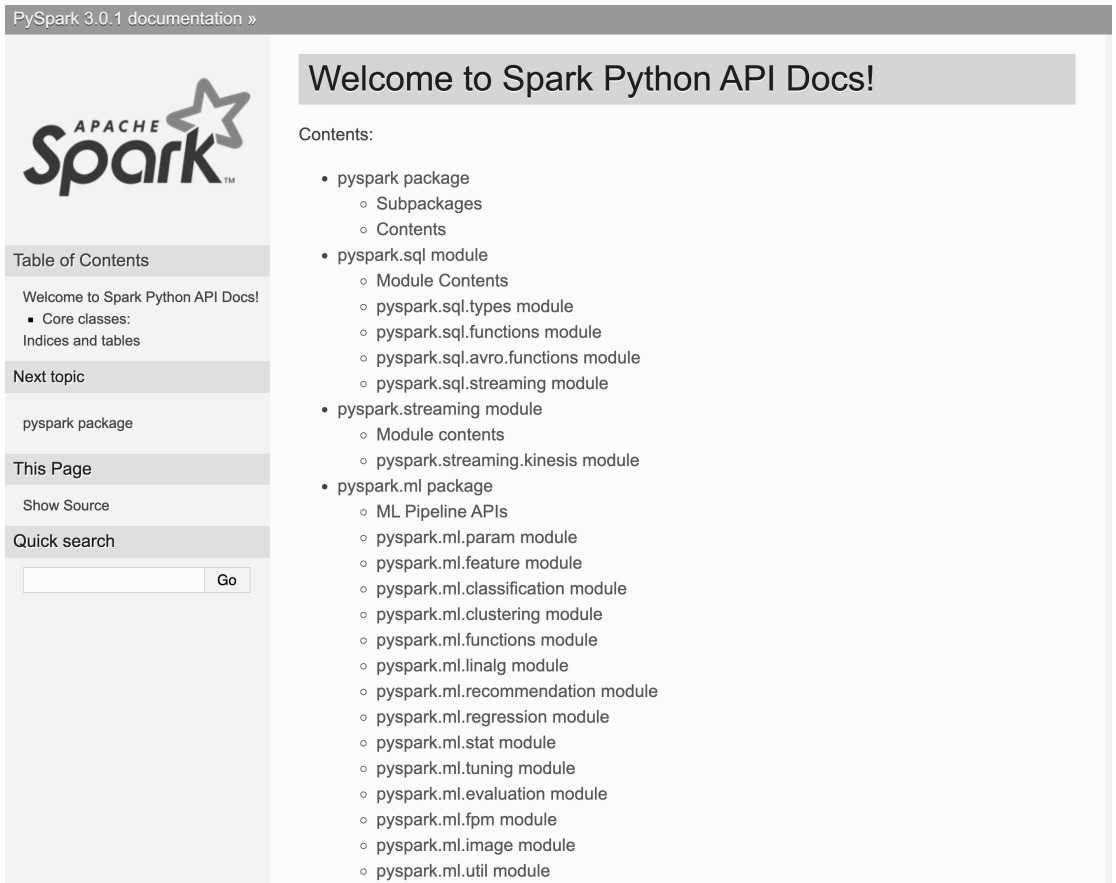
References used in this lecture

Book: *Fundamentals of Machine Learning for Predictive Analytics* by Kelleher, Mac Namee and D'Arcy, 2015.



References used in this lecture

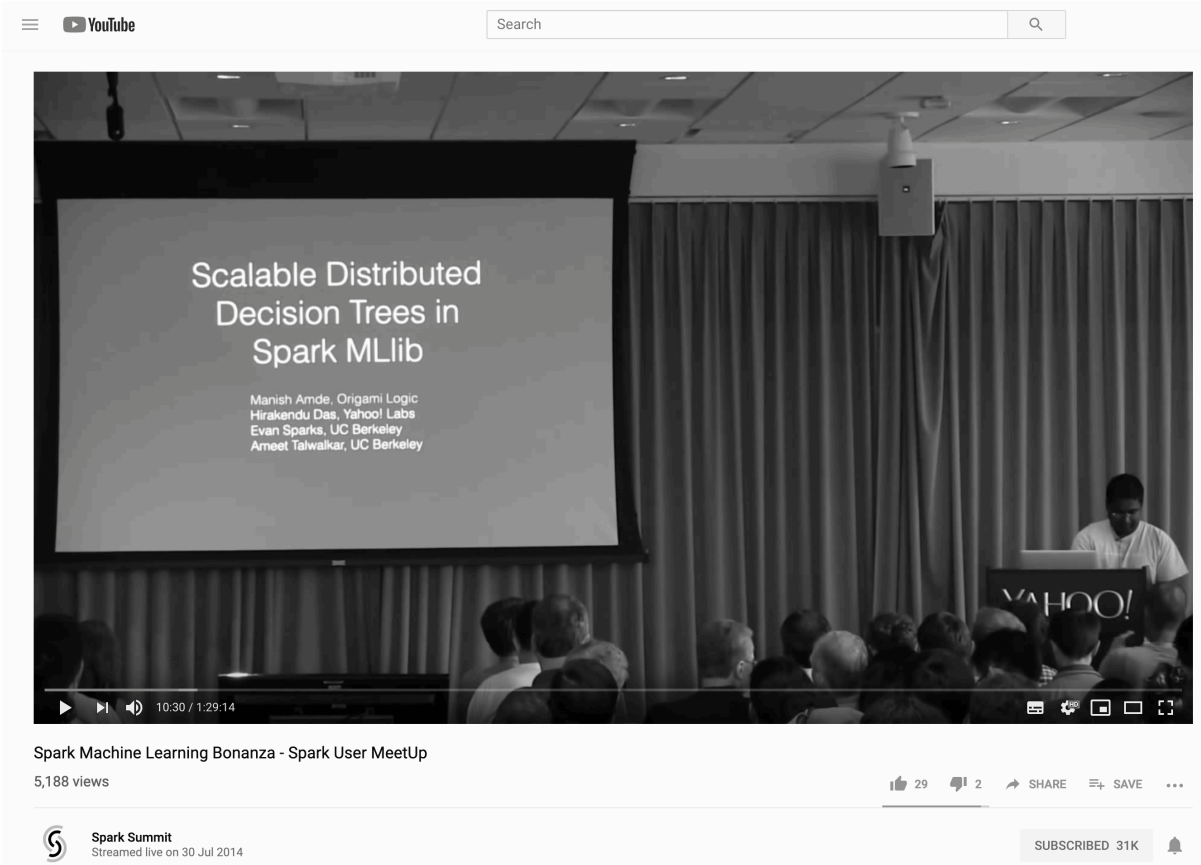
Website: *Spark Python API Documentation*.



`https://spark.apache.org/docs/3.0.1/api/python/index`

References used in this lecture

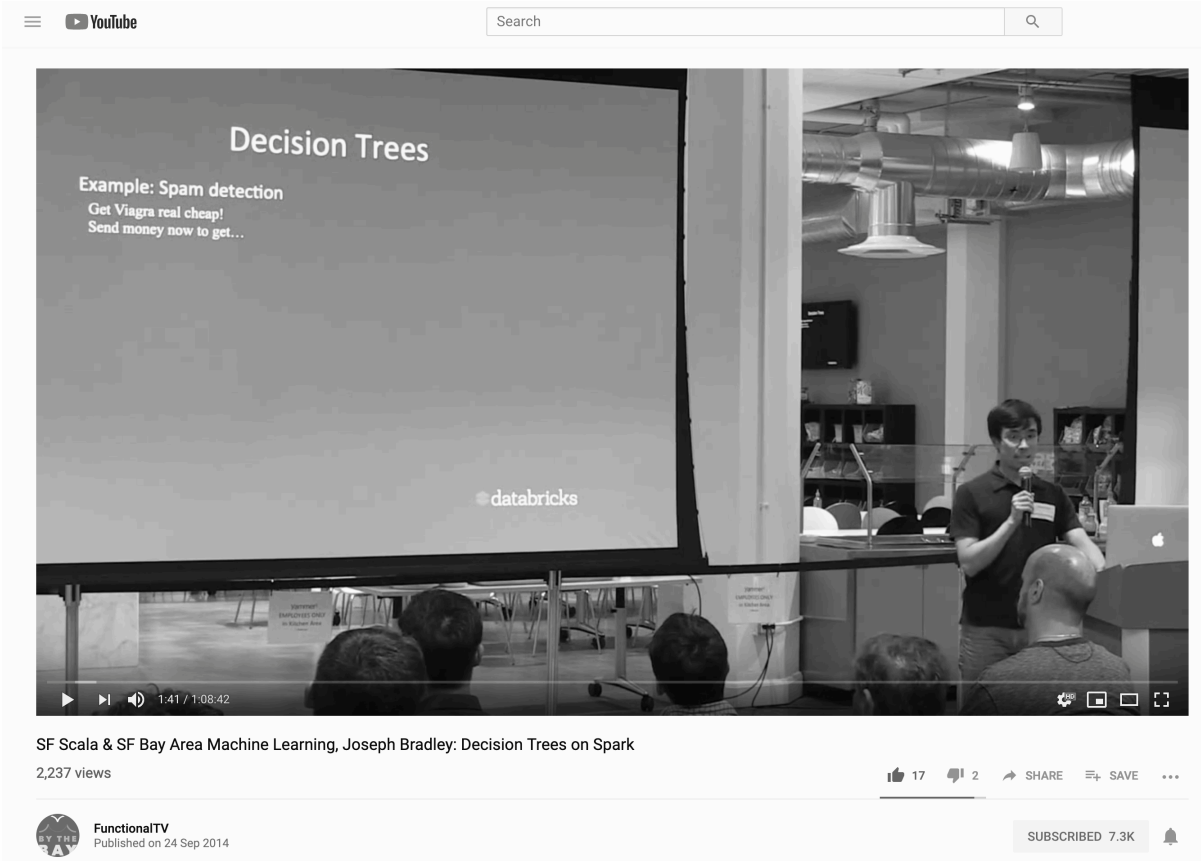
Youtube video: *Scalable Decision Trees in Spark MLlib* by Manish Amde (contributor to the MLlib implementation of Decision Trees in Spark).



`https://www.youtube.com/watch?v=N453EV5gHRA&t=10m30s`

References used in this lecture

Youtube video: *Decision Trees on Spark* by Joseph Bradley (from databricks).



`https://www.youtube.com/watch?v=3WS9OK3EXVA`

References used in this lecture

Paper: *PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce* by B. Panda et al. (from Google).

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 reside in memory on a single machine. While more scalable implementations of tree learning have been proposed, they typically require specialized parallel computing architectures. In contrast, the majority of Google's computing infrastructure is based on commodity hardware.

In this paper, we describe PLANET: a scalable distributed framework for learning tree models over large datasets. PLANET defines tree learning as a series of distributed computations, and implements each one using the *MapReduce* model of distributed computation. We show how this framework supports scalable construction of classification and regression trees, as well as ensembles of such models. We discuss the benefits and challenges of using a MapReduce compute cluster for tree learning, and demonstrate the scalability of this approach by applying it to a real world learning task from the domain of computational advertising.

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 frameworks on their compute clusters for data analysis and other data management tasks. MapReduce has become in some sense an industry standard. For example, there are open source implementations such as Hadoop that can be run either in-house or on cloud computing services such as Amazon EC2.[1] Startups like Cloudera[2] offer software and services to simplify Hadoop deployment, and companies including Google, IBM and Yahoo! have granted several universities access to Hadoop clusters to further cluster computing research.[3]

Despite the growing popularity of MapReduce [12], its application to certain standard data mining and machine learning tasks remains poorly understood. In this paper we focus on one such task: tree learning. We believe that a tree learner capable of exploiting a MapReduce cluster can effectively address many scalability issues that arise in building tree models on massive datasets. Our choice of focusing