Machine Learning & Spark

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

Building the perfect waffle (an analogy)

Archetype Waffle

Find waffle recipe. Give explicit instructions:

- 125 g flour
- 1 t baking powder
- 1 egg
- 225 ml milk
- 1 T melted butter

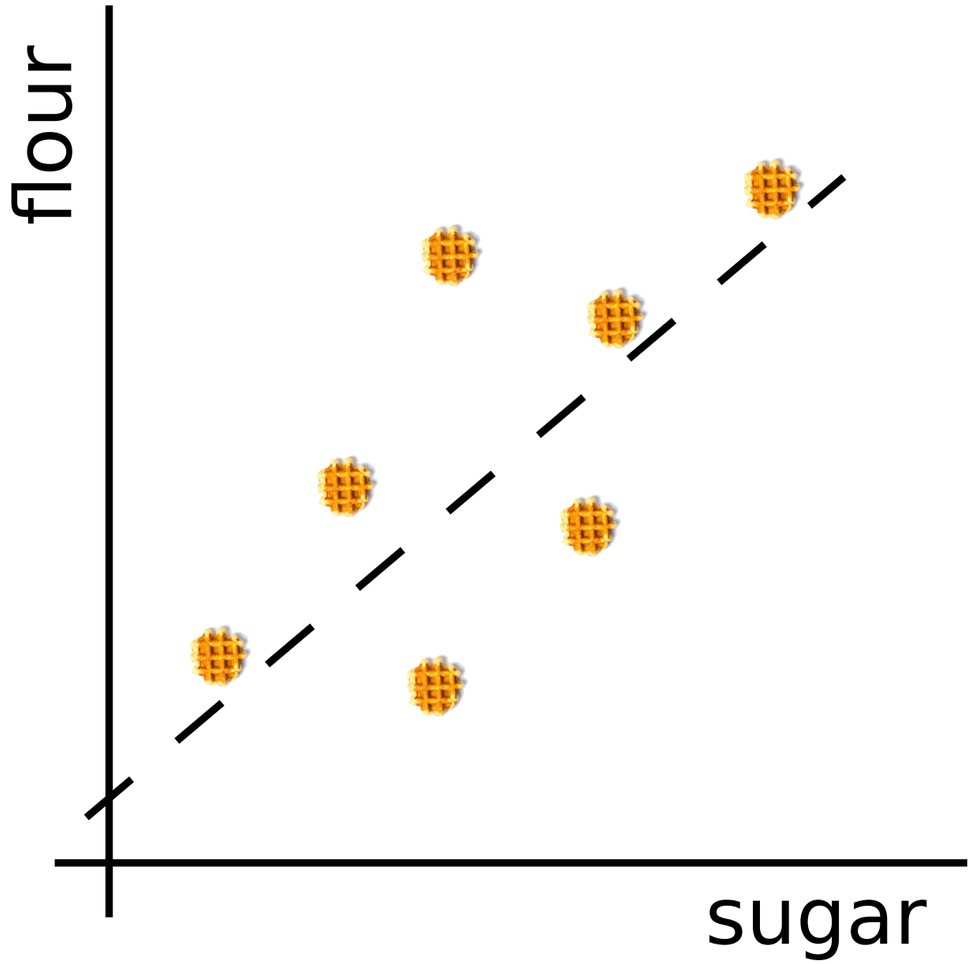
Find many waffle recipes.

Learn the perfect recipe:

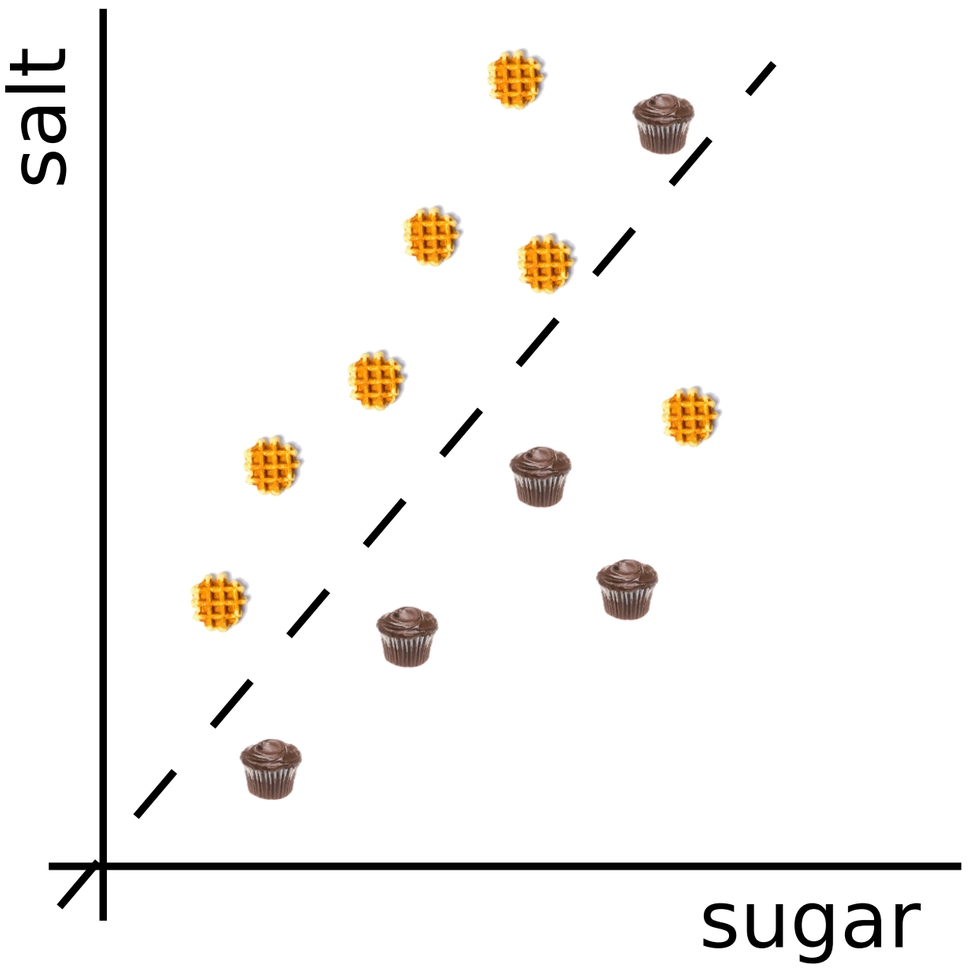
1. Look at lots of recipes.
2. What ingredients?
3. What proportions?

Computer generates its own instructions.

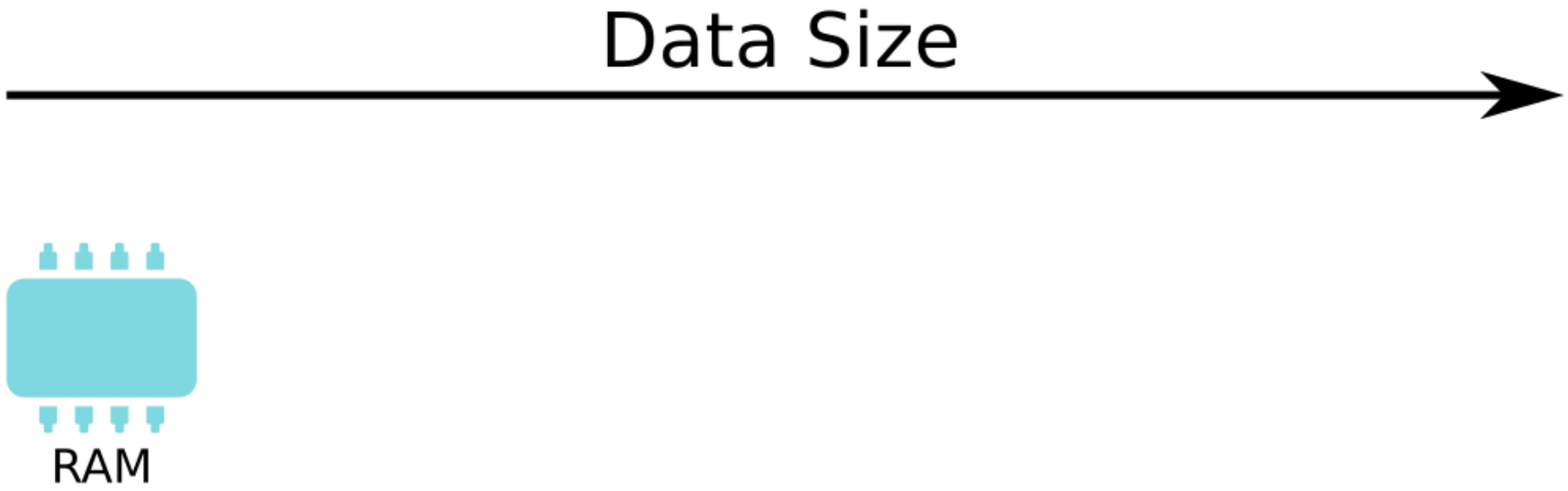
Regression



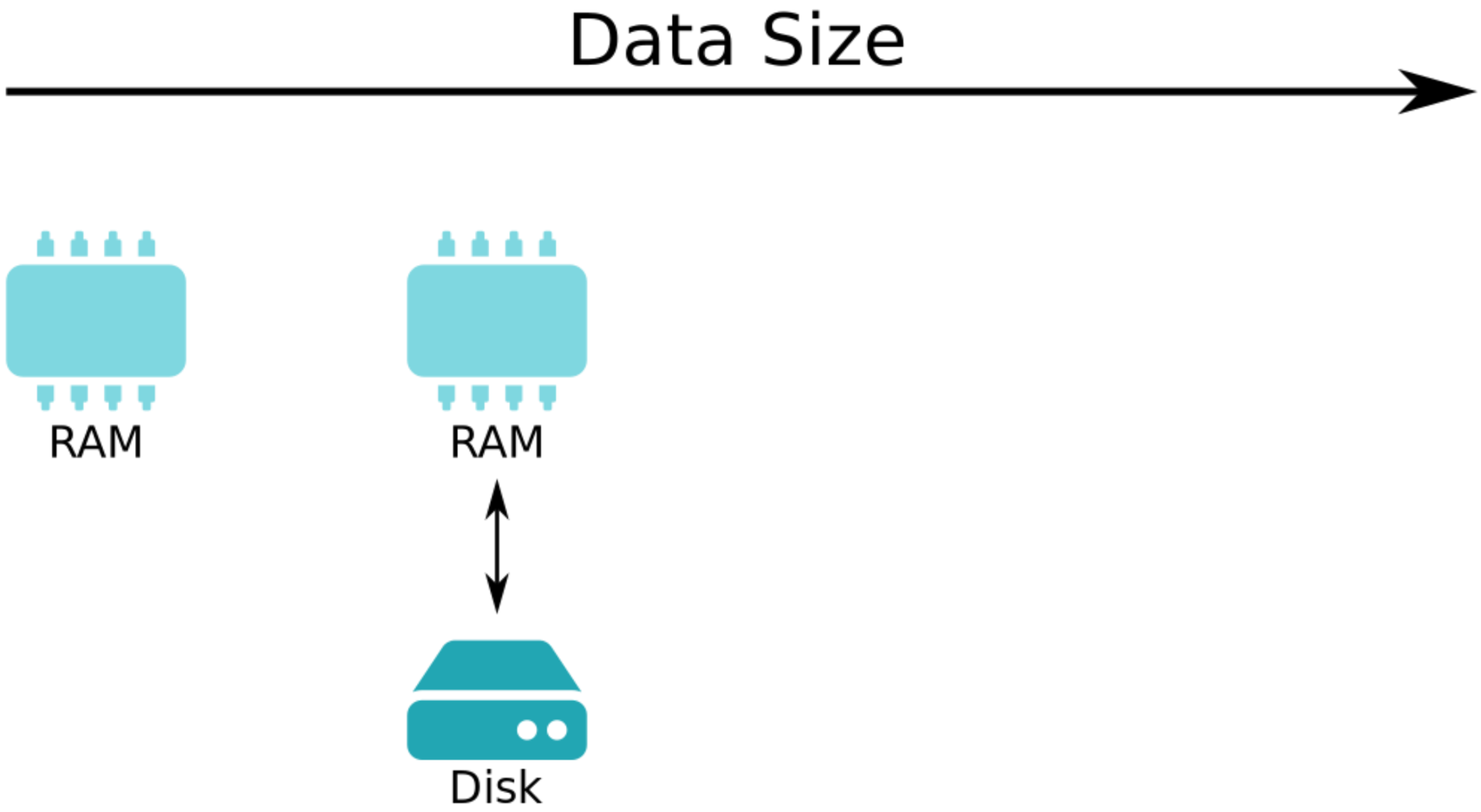
Classification



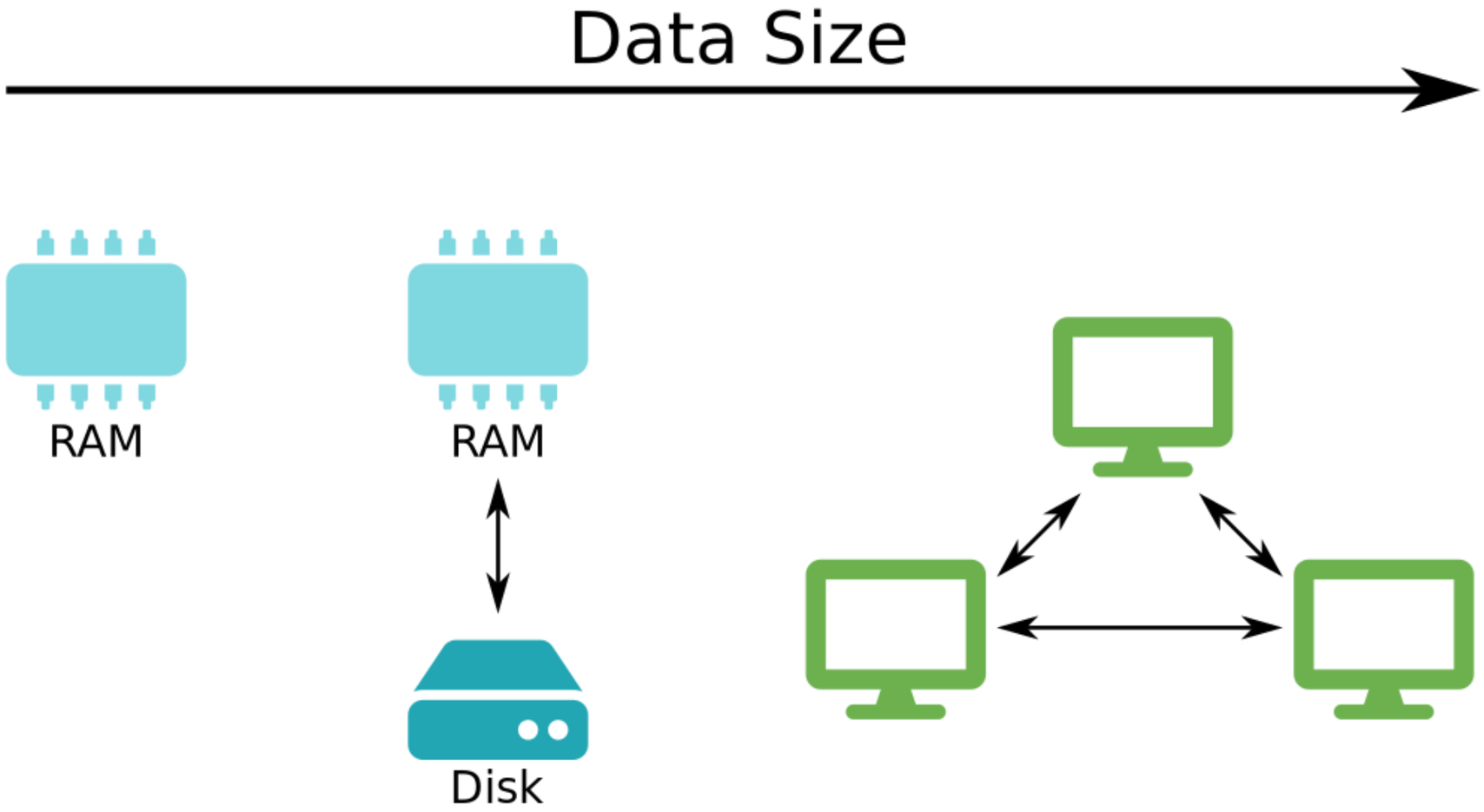
Data in RAM



Data exceeds RAM

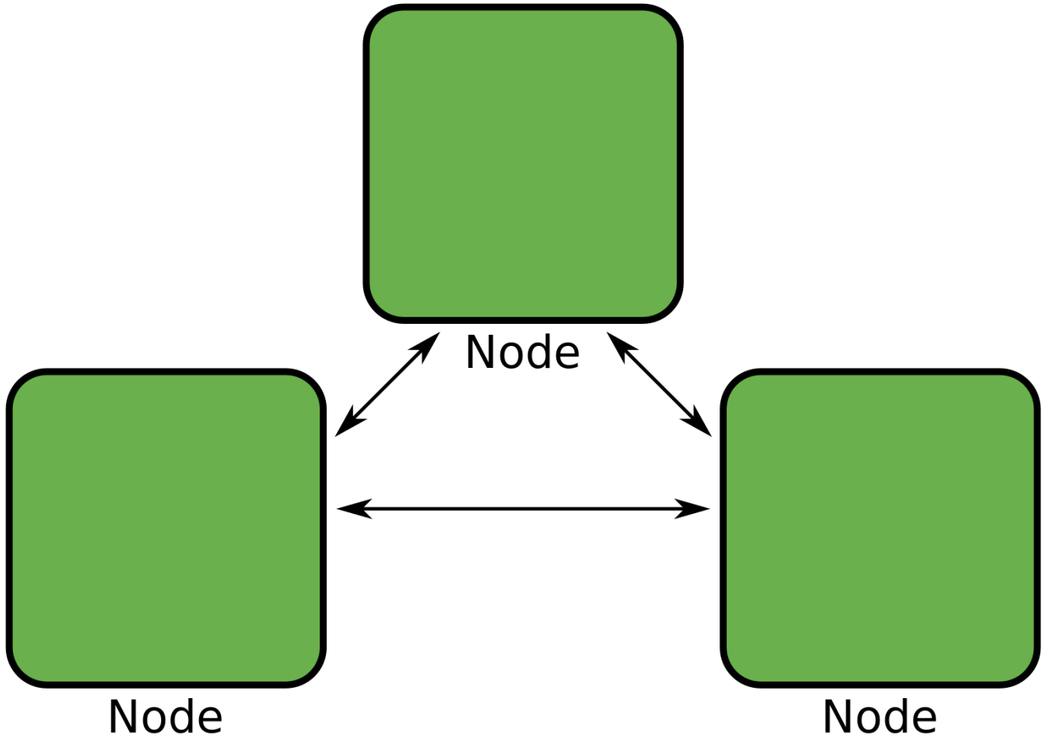


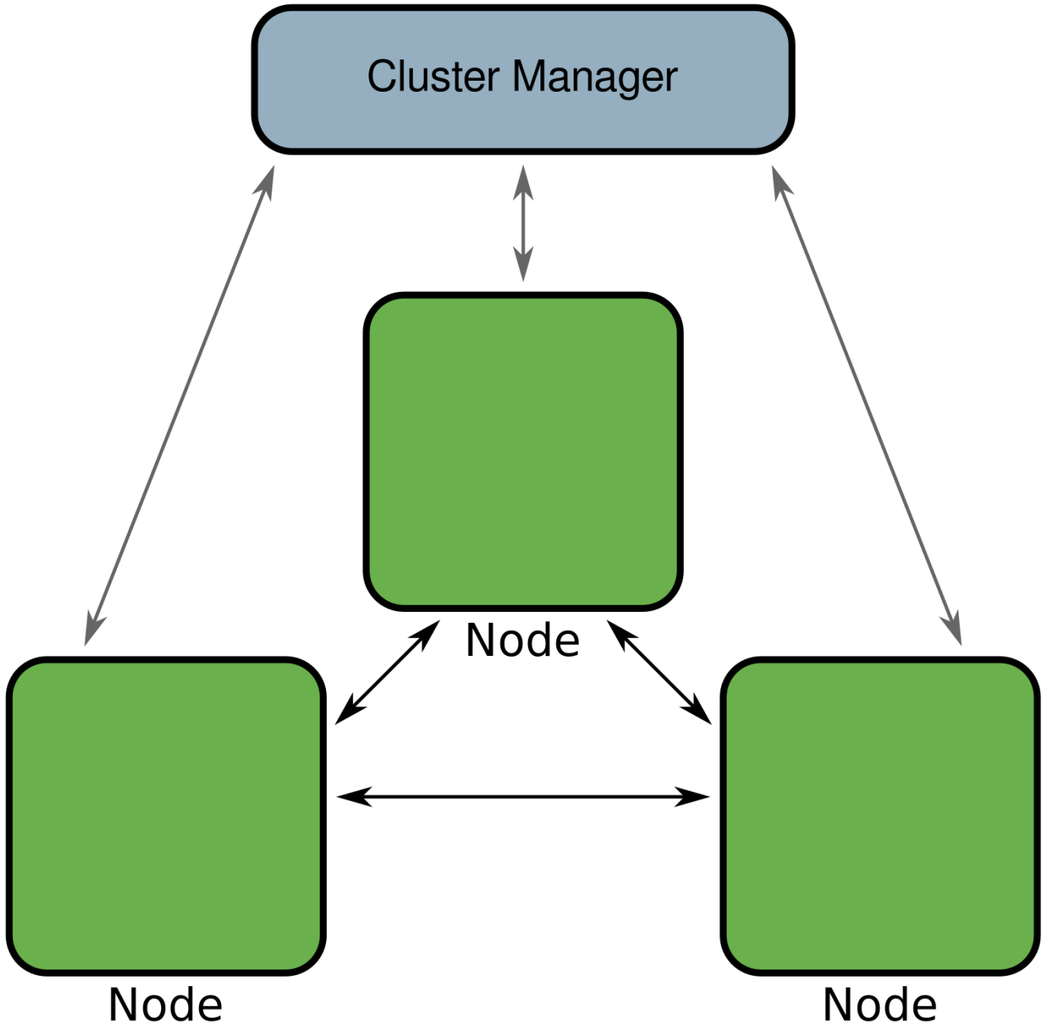
Data distributed across a cluster

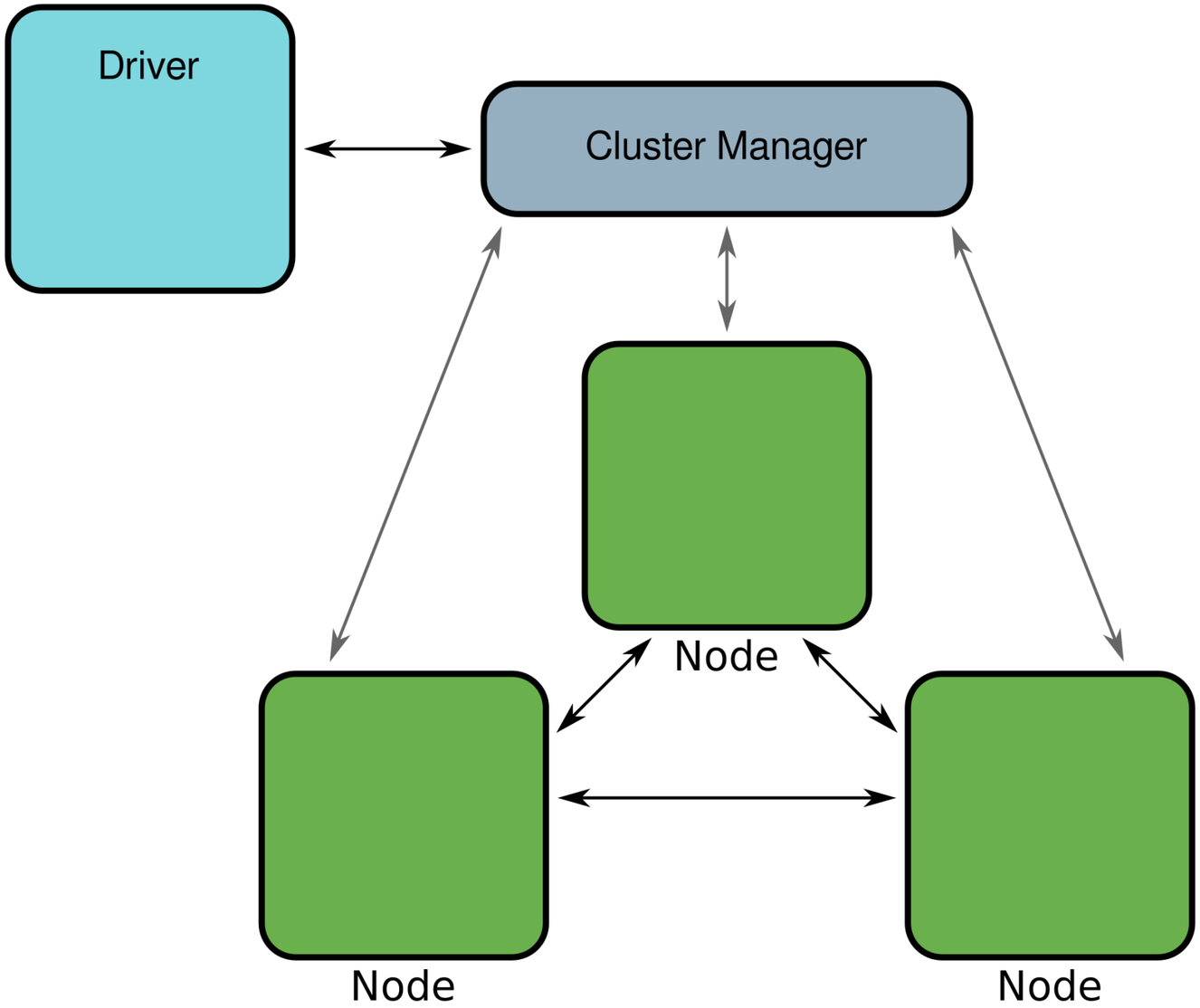


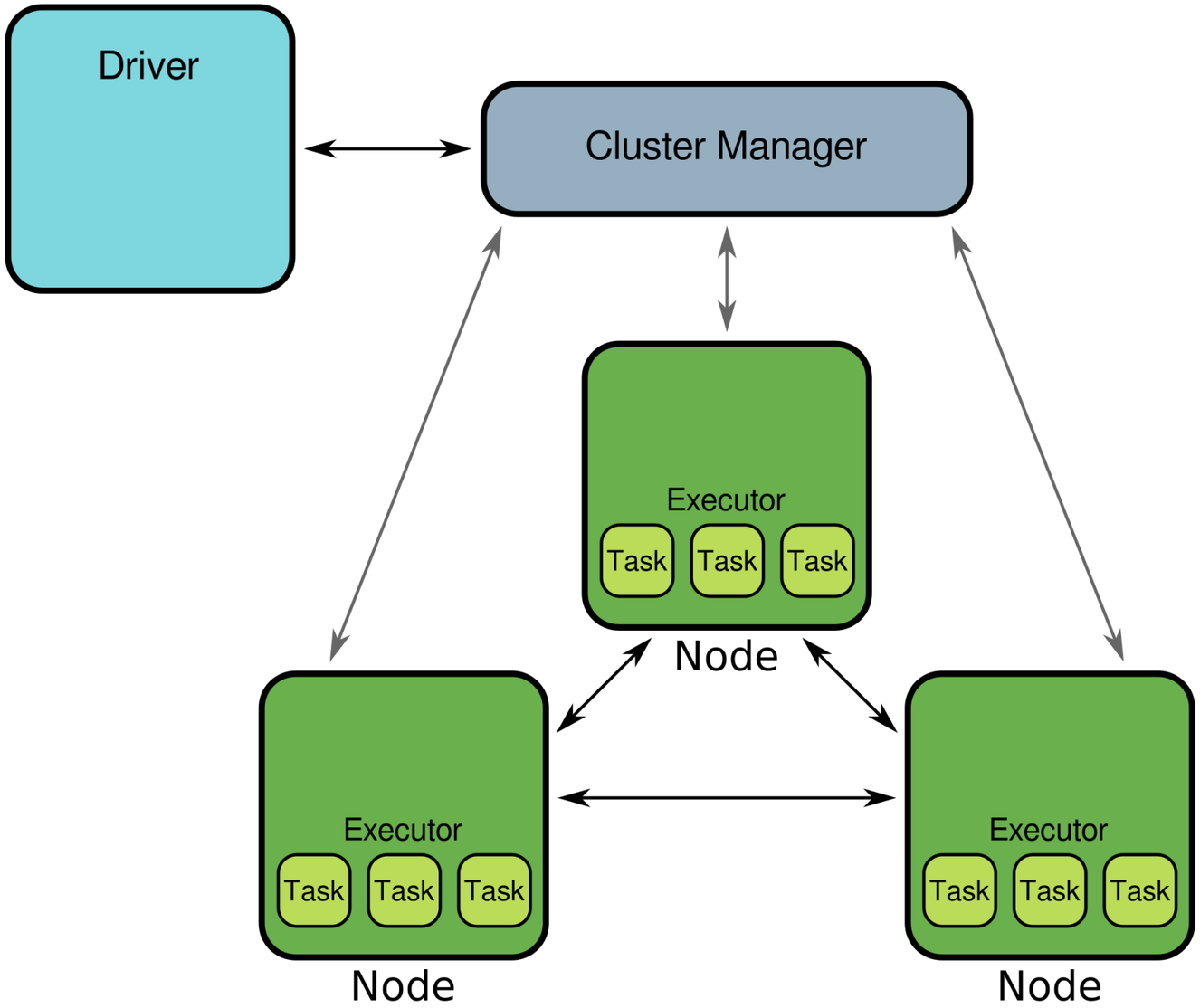
What is Spark?

- Compute across a distributed **cluster**.
- Data processed in memory.
- Well documented high-level **API**.


Node
Node
Node


Cluster Manager
Node
Node
Node


Driver
Cluster Manager
Node
Node
Node


Driver
Cluster Manager
Executor
Task
Task
Task
Node
Executor
Task
Task
Task
Node
Executor
Task
Task
Task
Node

Onward!

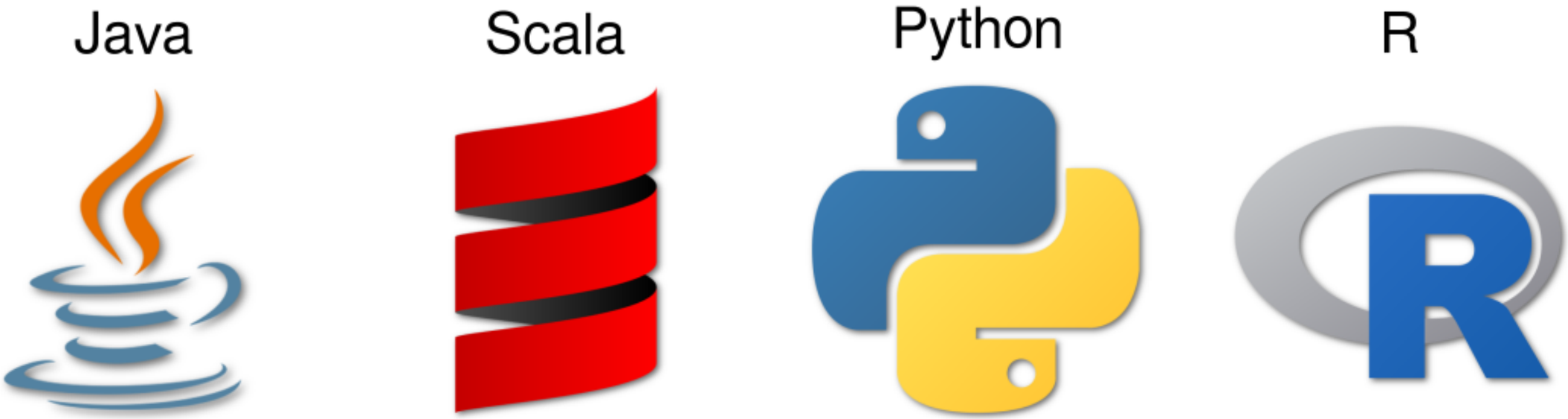
MACHINE LEARNING WITH PYSPARK

Connecting to Spark

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

Interacting with Spark



Languages for interacting with Spark.

- Java — low-level, compiled
- Scala, Python and R — high-level with interactive REPL

Importing pyspark

From Python import the `pyspark` module.

```
import pyspark
```

Check version.

```
pyspark.__version__
```

```
'2.4.1'
```

Sub-modules

In addition to `pyspark` there are

- Structured Data — **`pyspark.sql`**
- Streaming Data — **`pyspark.streaming`**
- Machine Learning — **`pyspark.mllib`** (deprecated) and **`pyspark.ml`**

Spark URL

Remote Cluster using Spark URL — `spark://<IP address | DNS name>:<port>`

Example:

- `spark://13.59.151.161:7077`

Local Cluster

Examples:

- `local` — only 1 core;
- `local[4]` — 4 cores; or
- `local[*]` — all available cores.

Creating a SparkSession

```
from pyspark.sql import SparkSession
```

Create a local cluster using a `SparkSession` builder.

```
spark = SparkSession.builder \
                    .master('local[*]') \
                    .appName('first_spark_application') \
                    .getOrCreate()
```

Interact with Spark...

```
# Close connection to Spark
>>> spark.stop()
```

Let's connect to Spark!

MACHINE LEARNING WITH PYSPARK

Loading Data

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

DataFrames: A refresher

`DataFrame` for tabular data.

123	abc	123	abc
123	abc	123	abc
123	abc	123	abc
123	abc	123	abc
123	abc	123	abc

Selected methods:

- `count()`
- `show()`
- `printSchema()`

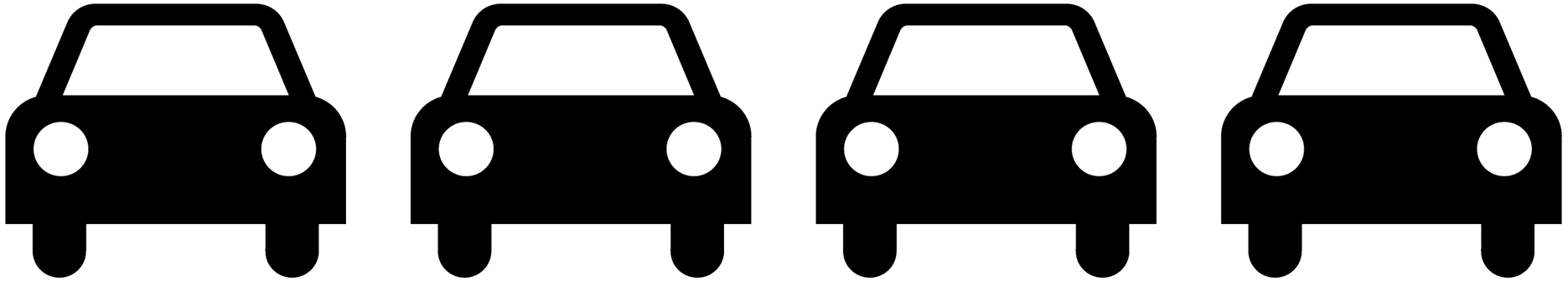
Selected attributes:

- `dtypes`

CSV data for cars

The first few lines from the 'cars.csv' file.

```
mfr,mod,org,type,cyl,size,weight,len,rpm,cons
Mazda,RX-7,non-USA,Sporty,NA,1.3,2895,169,6500,9.41
Nissan,Maxima,non-USA,Midsize,6,3,3200,188,5200,9.05
Chevrolet,Cavalier,USA,Compact,4,2.2,2490,182,5200,6.53
Subaru,Legacy,non-USA,Compact,4,2.2,3085,179,5600,7.84
Ford,Escort,USA,Small,4,1.8,2530,171,6500,7.84
```



Reading data from CSV

The `.csv()` method reads a CSV file and returns a `DataFrame` .

```
cars = spark.read.csv('cars.csv', header=True)
```

Optional arguments:

- `header` — is first row a header? (default: `False`)
- `sep` — field separator (default: a comma `','`)
- `schema` — explicit column data types
- `inferSchema` — deduce column data types from data?
- `nullValue` — placeholder for missing data

Peek at the data

The first five records from the `DataFrame` .

```
cars.show(5)
```

```
+---------+--------+-------+-------+---+----+------+---+----+----+
|      mfr|     mod|    org|   type|cyl|size|weight|len| rpm|cons|
+---------+--------+-------+-------+---+----+------+---+----+----+
|    Mazda|    RX-7|non-USA| Sporty| NA| 1.3|  2895|169|6500|9.41|
|   Nissan|  Maxima|non-USA|Midsize|  6|   3|  3200|188|5200|9.05|
|Chevrolet|Cavalier|    USA|Compact|  4| 2.2|  2490|182|5200|6.53|
|   Subaru|  Legacy|non-USA|Compact|  4| 2.2|  3085|179|5600|7.84|
|     Ford|  Escort|    USA|  Small|  4| 1.8|  2530|171|6500|7.84|
+---------+--------+-------+-------+---+----+------+---+----+----+
```

Check column types

```
cars.printSchema()
```

```
root
|-- mfr: string (nullable = true)
|-- mod: string (nullable = true)
|-- org: string (nullable = true)
|-- type: string (nullable = true)
|-- cyl: string (nullable = true)
|-- size: string (nullable = true)
|-- weight: string (nullable = true)
|-- len: string (nullable = true)
|-- rpm: string (nullable = true)
|-- cons: string (nullable = true)
```

Inferring column types from data

```
cars = spark.read.csv("cars.csv", header=True, inferSchema=True)
cars.dtypes
```

```
 [('mfr', 'string'),
 ('mod', 'string'),
 ('org', 'string'),
 ('type', 'string'),
 ('cyl', 'string'),
 ('size', 'double'),
 ('weight', 'int'),
 ('len', 'int'),
 ('rpm', 'int'),
 ('cons', 'double')]
```

Dealing with missing data

Handle missing data using the `nullValue` argument.

```
cars = spark.read.csv("cars.csv", header=True, inferSchema=True, nullValue='NA')
```

The `nullValue` argument is case sensitive.

Specify column types

```
schema = StructType([
    StructField("maker", StringType()),
    StructField("model", StringType()),
    StructField("origin", StringType()),
    StructField("type", StringType()),
    StructField("cyl", IntegerType()),
    StructField("size", DoubleType()),
    StructField("weight", IntegerType()),
    StructField("length", DoubleType()),
    StructField("rpm", IntegerType()),
    StructField("consumption", DoubleType())
])
cars = spark.read.csv("cars.csv", header=True, schema=schema, nullValue='NA')
```

Final cars data

```
+----------+-------------+-------+-------+----+----+------+------+----+-----------+
|maker     |model        |origin |type   |cyl |size|weight|length|rpm |consumption|
+----------+-------------+-------+-------+----+----+------+------+----+-----------+
|Mazda     |RX-7         |non-USA|Sporty |null|1.3 |2895  |169.0 |6500|9.41       |
|Nissan    |Maxima       |non-USA|Midsize|6   |3.0 |3200  |188.0 |5200|9.05       |
|Chevrolet |Cavalier     |USA    |Compact|4   |2.2 |2490  |182.0 |5200|6.53       |
|Subaru    |Legacy       |non-USA|Compact|4   |2.2 |3085  |179.0 |5600|7.84       |
|Ford      |Escort       |USA    |Small  |4   |1.8 |2530  |171.0 |6500|7.84       |
|Mercury   |Capri        |USA    |Sporty |4   |1.6 |2450  |166.0 |5750|9.05       |
|Oldsmobile|Cutlass Ciera|USA    |Midsize|4   |2.2 |2890  |190.0 |5200|7.59       |
|Saab      |900          |non-USA|Compact|4   |2.1 |2775  |184.0 |6000|9.05       |
|Dodge     |Caravan      |USA    |Van    |6   |3.0 |3705  |175.0 |5000|11.2       |
+----------+-------------+-------+-------+----+----+------+------+----+-----------+
```

Let's load some data!

MACHINE LEARNING WITH PYSPARK

Data Preparation

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

Do you need all of those columns?

```
+-----+-------+-------+------+----+----+------+------+----+-----------+
|maker|  model| origin|  type| cyl|size|weight|length| rpm|consumption|
+-----+-------+-------+------+----+----+------+------+----+-----------+
|Mazda|   RX-7|non-USA|Sporty|null| 1.3|  2895| 169.0|6500|       9.41|
|  Geo|  Metro|non-USA| Small|   3| 1.0|  1695| 151.0|5700|        4.7|
| Ford|Festiva|    USA| Small|   4| 1.3|  1845| 141.0|5000|       7.13|
+-----+-------+-------+------+----+----+------+------+----+-----------+
```

Remove the `maker` and `model` fields.

Dropping columns

```
# Either drop the columns you don't want...
cars = cars.drop('maker', 'model')

# ... or select the columns you want to retain.
cars = cars.select('origin', 'type', 'cyl', 'size', 'weight', 'length', 'rpm', 'consumption')
```

```
+-------+------+----+----+------+------+----+-----------+
| origin|  type| cyl|size|weight|length| rpm|consumption|
+-------+------+----+----+------+------+----+-----------+
|non-USA|Sporty|null| 1.3|  2895| 169.0|6500|       9.41|
|non-USA| Small|   3| 1.0|  1695| 151.0|5700|        4.7|
|    USA| Small|   4| 1.3|  1845| 141.0|5000|       7.13|
+-------+------+----+----+------+------+----+-----------+
```

Filtering out missing data

```
# How many missing values?
cars.filter('cyl IS NULL').count()
```

```
1
```

Drop records with missing values in the `cylinders` column.

```
cars = cars.filter('cyl IS NOT NULL')
```

Drop records with missing values in *any* column.

```
cars = cars.dropna()
```

Mutating columns

```
from pyspark.sql.functions import round

# Create a new 'mass' column
cars = cars.withColumn('mass', round(cars.weight / 2.205, 0))

# Convert length to metres
cars = cars.withColumn('length', round(cars.length * 0.0254, 3))
```

```
+-------+-----+---+----+------+------+----+-----------+-----+
| origin| type|cyl|size|weight|length| rpm|consumption| mass|
+-------+-----+---+----+------+------+----+-----------+-----+
|non-USA|Small|  3| 1.0|  1695| 3.835|5700|        4.7|769.0|
|    USA|Small|  4| 1.3|  1845| 3.581|5000|       7.13|837.0|
|non-USA|Small|  3| 1.3|  1965| 4.089|6000|       5.47|891.0|
+-------+-----+---+----+------+------+----+-----------+-----+
```

Indexing categorical data

```
from pyspark.ml.feature import StringIndexer

indexer = StringIndexer(inputCol='type',
                        outputCol='type_idx')


# Assign index values to strings
indexer = indexer.fit(cars)

# Create column with index values
cars = indexer.transform(cars)
```

```
+-------+--------+
|   type|type_idx|
+-------+--------+
|Midsize|     0.0| <- most frequent value
|  Small|     1.0|
|Compact|     2.0|
| Sporty|     3.0|
|  Large|     4.0|
|    Van|     5.0| <- least frequent value
+-------+--------+
```

Use `stringOrderType` to change order.

Indexing country of origin

```
# Index country of origin:
#
# USA     -> 0
# non-USA -> 1
#
cars = StringIndexer(
  inputCol="origin",
  outputCol="label"
).fit(cars).transform(cars)
```

```
+-------+-----+
| origin|label|
+-------+-----+
|    USA|  0.0|
|non-USA|  1.0|
+-------+-----+
```

Assembling columns

Use a vector assembler to transform the data.

```
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(inputCols=['cyl', 'size'], outputCol='features')
assembler.transform(cars)
```

```
+---+----+---------+
|cyl|size| features|
+---+----+---------+
|  3| 1.0|[3.0,1.0]|
|  4| 1.3|[4.0,1.3]|
|  3| 1.3|[3.0,1.3]|
+---+----+---------+
```

Let's practice!

MACHINE LEARNING WITH PYSPARK

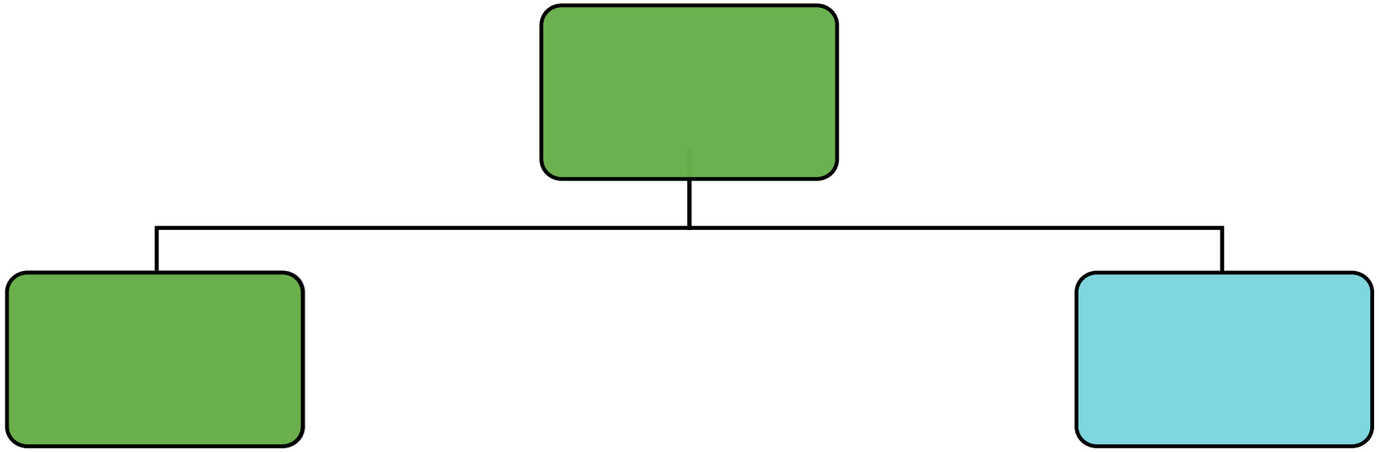
Decision Tree

MACHINE LEARNING WITH PYSPARK

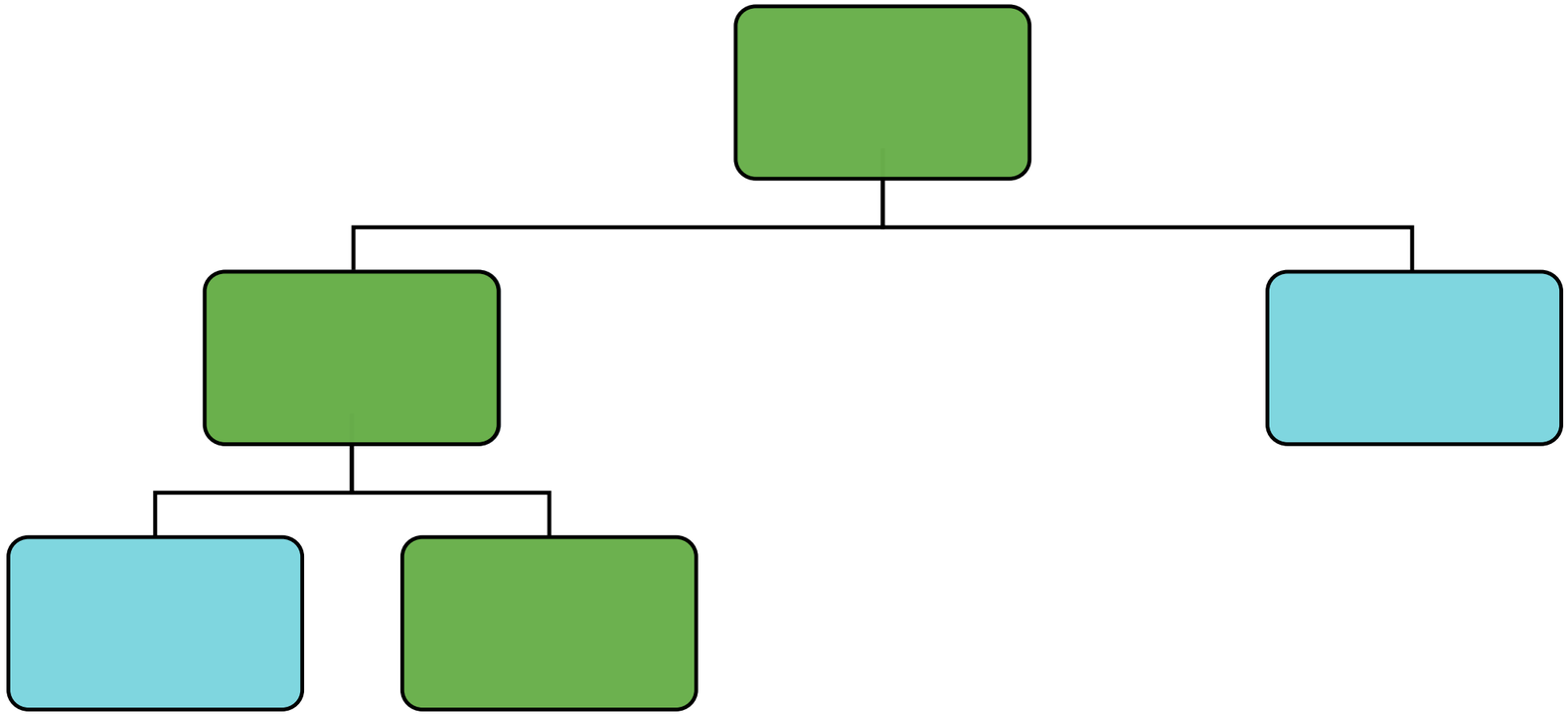
Andrew Collier
Data Scientist, Exegetic Analytics

Anatomy of a Decision Tree: Root node

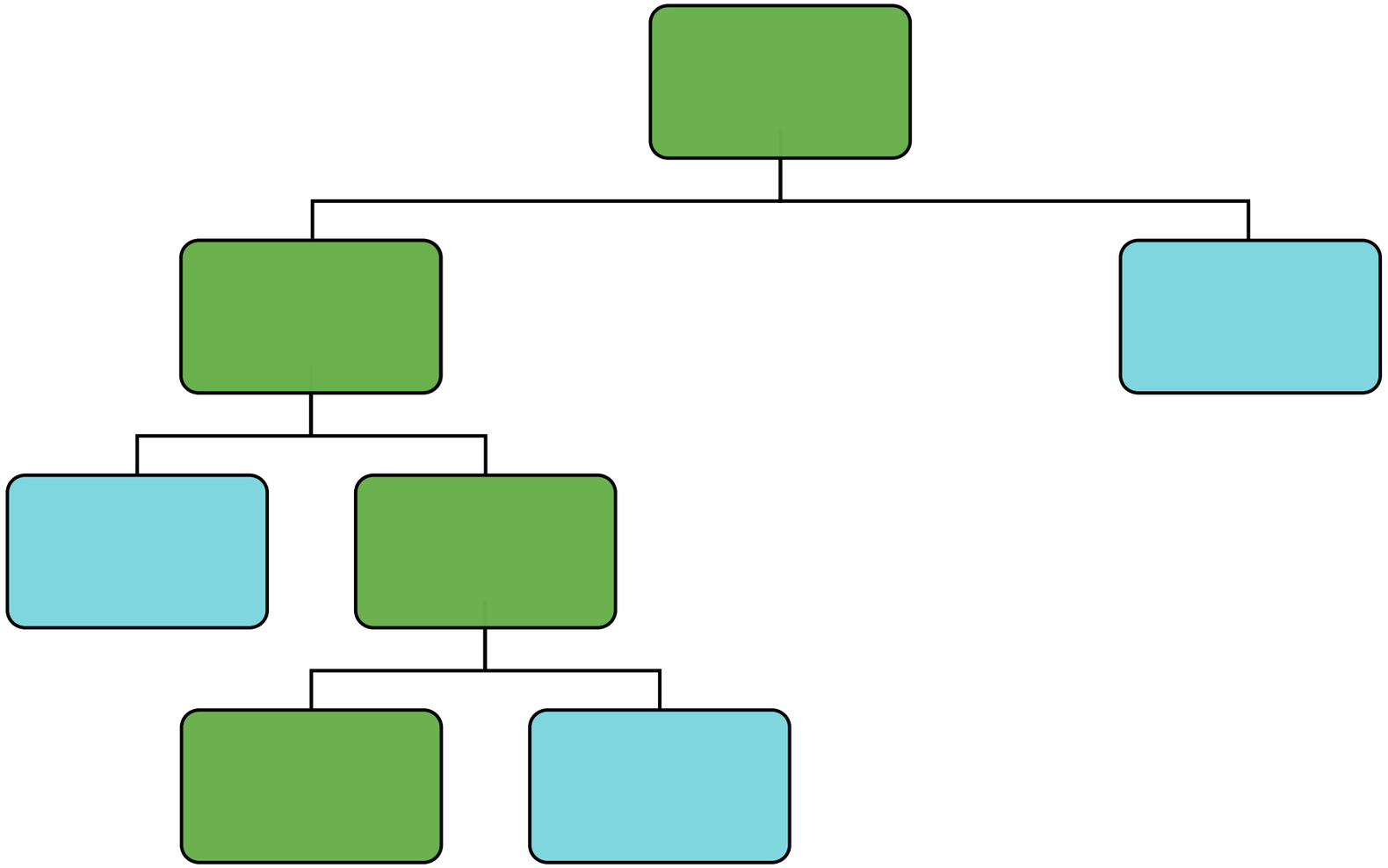
Anatomy of a Decision Tree: First split



Anatomy of a Decision Tree: Second split



Anatomy of a Decision Tree: Third split



Classifying cars

Classify cars according to country of manufacture.

```
+---+----+------+------+----+----------+------------------------------------+-----+
|cyl|size|mass  |length|rpm |consumption|features                            |label|
+---+----+------+------+----+----------+------------------------------------+-----+
|6  |3.0 |1451.0|4.775 |5200|9.05      |[6.0,3.0,1451.0,4.775,5200.0,9.05]|1.0  |
|4  |2.2 |1129.0|4.623 |5200|6.53      |[4.0,2.2,1129.0,4.623,5200.0,6.53]|0.0  |
|4  |2.2 |1399.0|4.547 |5600|7.84      |[4.0,2.2,1399.0,4.547,5600.0,7.84]|1.0  |
|4  |1.8 |1147.0|4.343 |6500|7.84      |[4.0,1.8,1147.0,4.343,6500.0,7.84]|0.0  |
|4  |1.6 |1111.0|4.216 |5750|9.05      |[4.0,1.6,1111.0,4.216,5750.0,9.05]|0.0  |
+---+----+------+------+----+----------+------------------------------------+-----+

label = 0 -> manufactured in the USA
      = 1 -> manufactured elsewhere
```

Split train/test

Split data into training and testing sets.

```
# Specify a seed for reproducibility
cars_train, cars_test = cars.randomSplit([0.8, 0.2], seed=23)
```

Two DataFrames: `cars_train` and `cars_test` .

```
[cars_train.count(), cars_test.count()]
```

```
[79, 13]
```

Build a Decision Tree model

```
from pyspark.ml.classification import DecisionTreeClassifier
```

Create a Decision Tree classifier.

```
tree = DecisionTreeClassifier()
```

Learn from the training data.

```
tree_model = tree.fit(cars_train)
```

Evaluating

Make predictions on the testing data and compare to known values.

```
prediction = tree_model.transform(cars_test)
```

```
+-----+----------+----------------------------------------+
|label|prediction|probability                             |
+-----+----------+----------------------------------------+
|1.0  |0.0       |[0.9615384615384616,0.0384615384615385]|
|1.0  |1.0       |[0.2222222222222222,0.7777777777777778]|
|1.0  |1.0       |[0.2222222222222222,0.7777777777777778]|
|0.0  |0.0       |[0.9615384615384616,0.0384615384615385]|
|1.0  |1.0       |[0.2222222222222222,0.7777777777777778]|
+-----+----------+----------------------------------------+
```

Confusion matrix

A confusion matrix is a table which describes performance of a model on testing data.

```
prediction.groupBy("label", "prediction").count().show()
```

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|  1.0|       1.0|    8| <- True positive  (TP)
|  0.0|       1.0|    2| <- False positive (FP)
|  1.0|       0.0|    3| <- False negative (FN)
|  0.0|       0.0|    6| <- True negative  (TN)
+-----+----------+-----+
```

Accuracy = (TN + TP) / (TN + TP + FN + FP) — proportion of correct predictions.

Let's build Decision Trees!

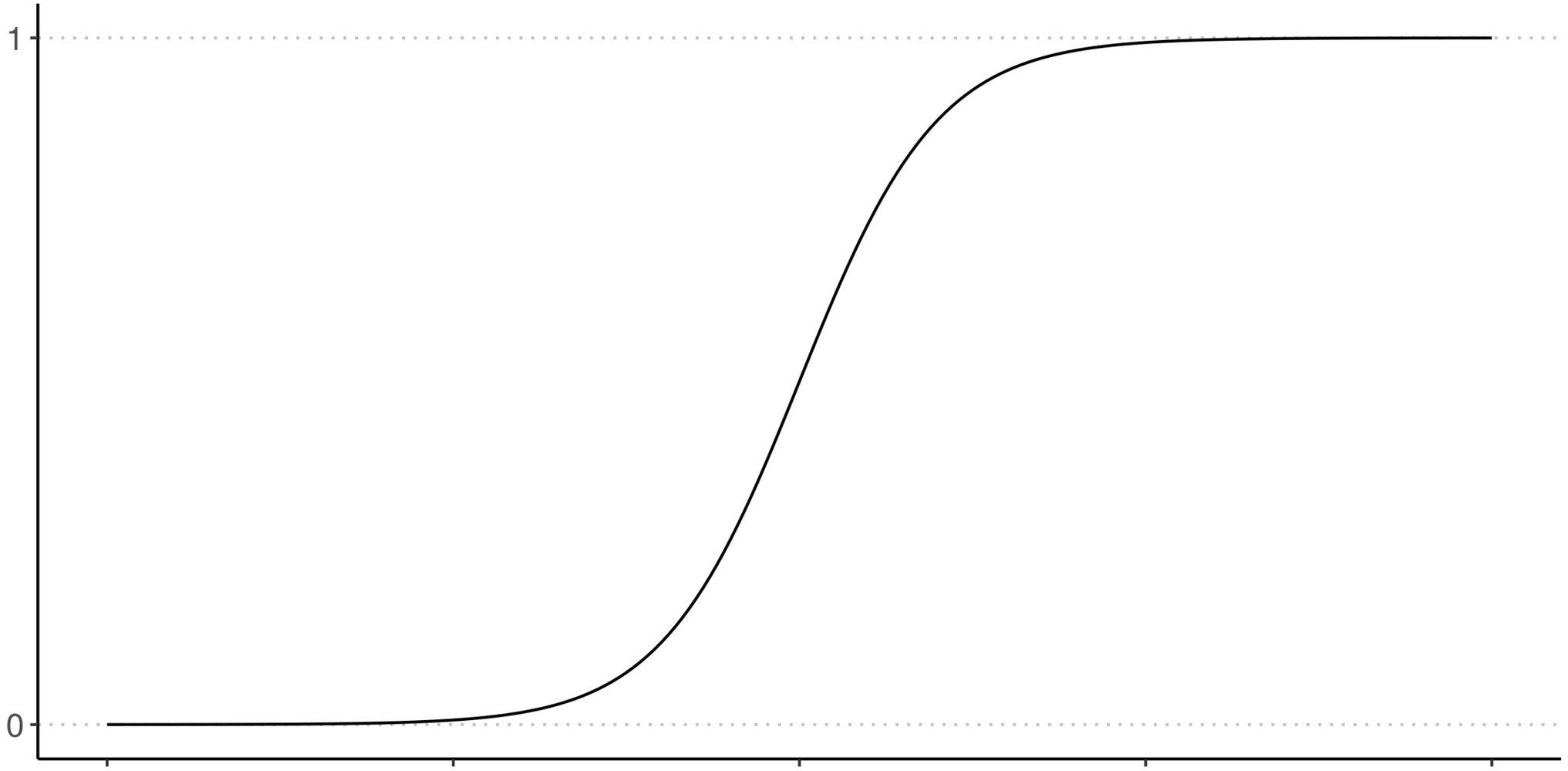
MACHINE LEARNING WITH PYSPARK

Logistic Regression

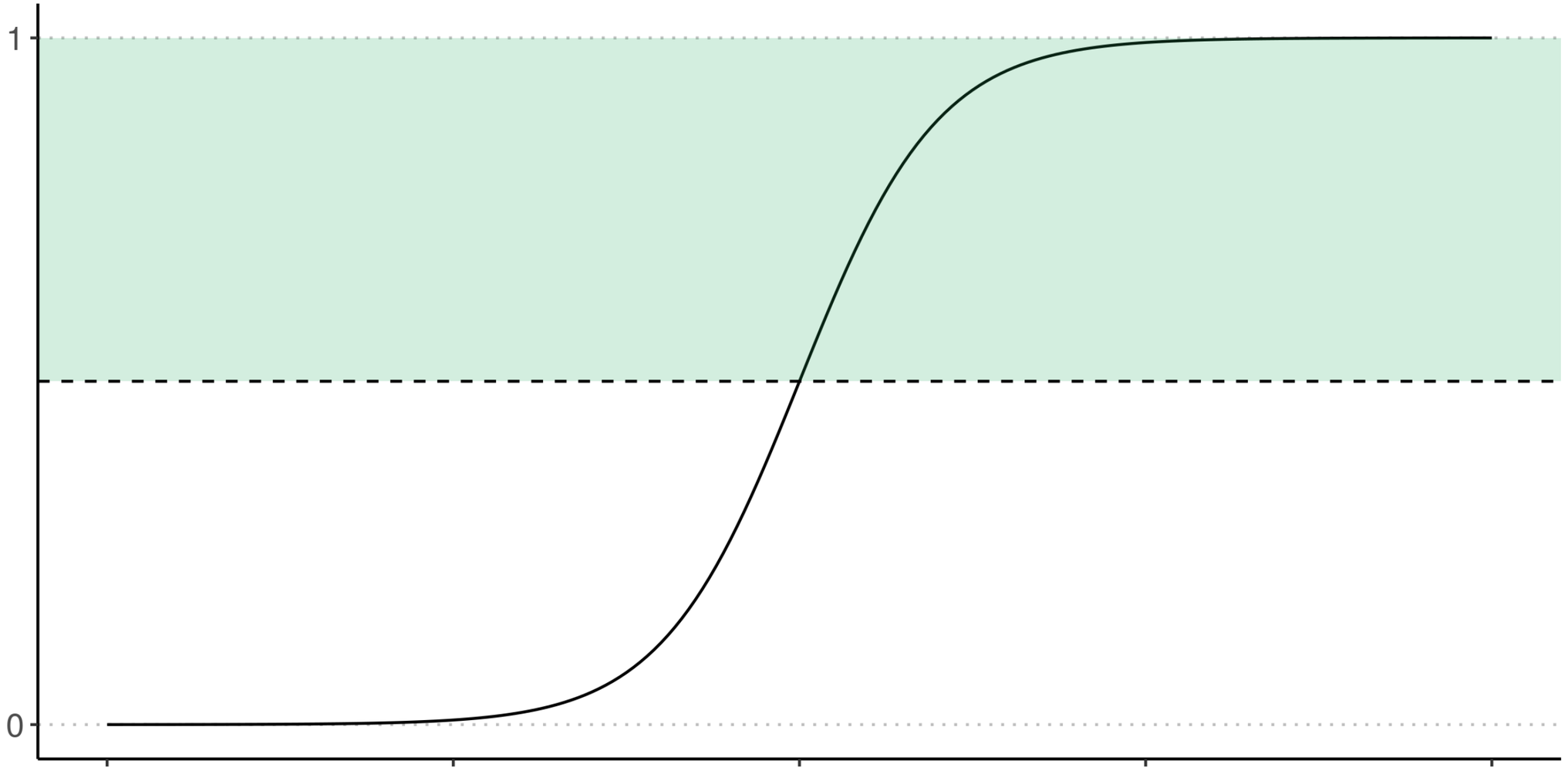
MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

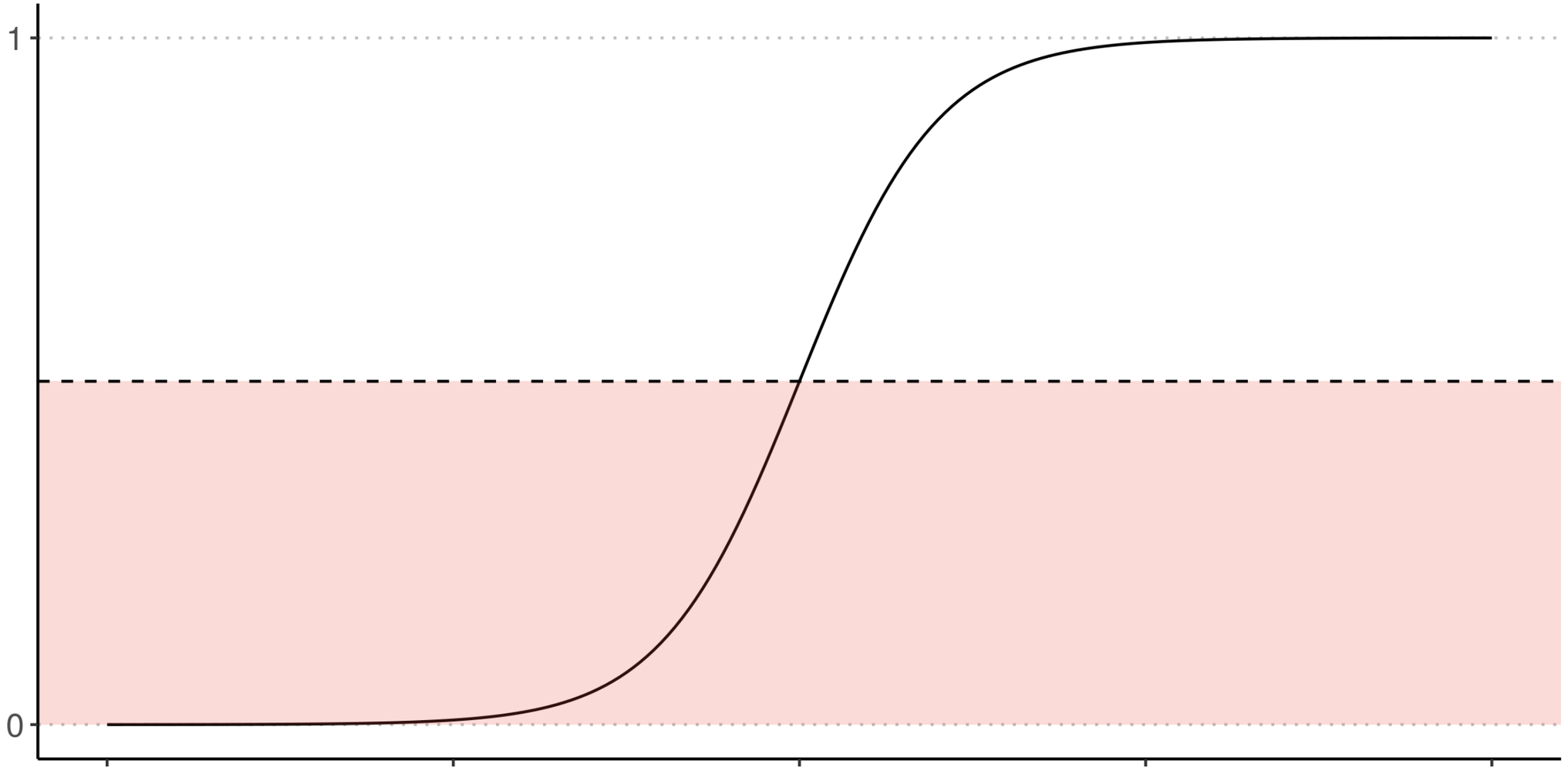
Logistic Curve



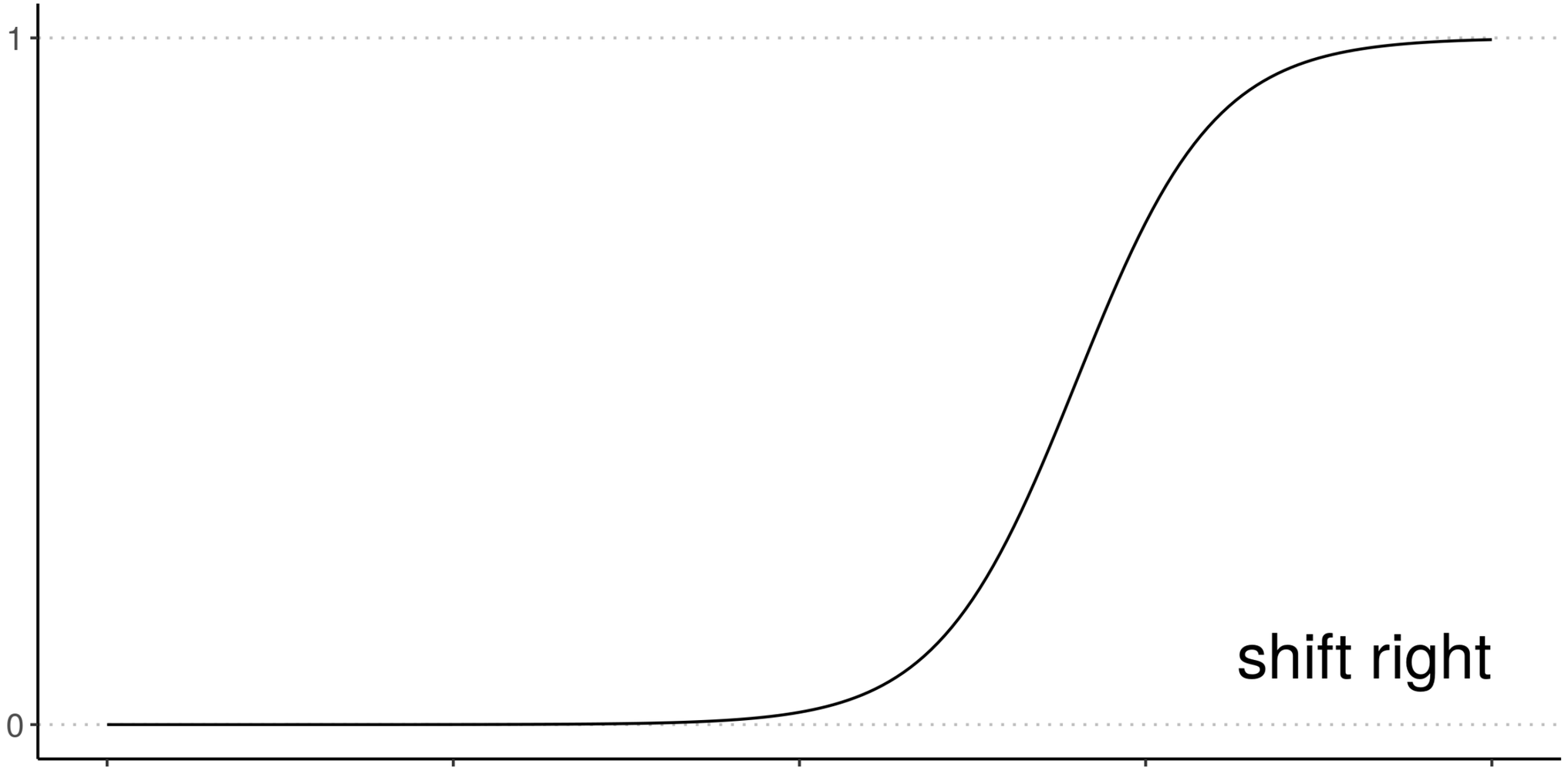
Logistic Curve



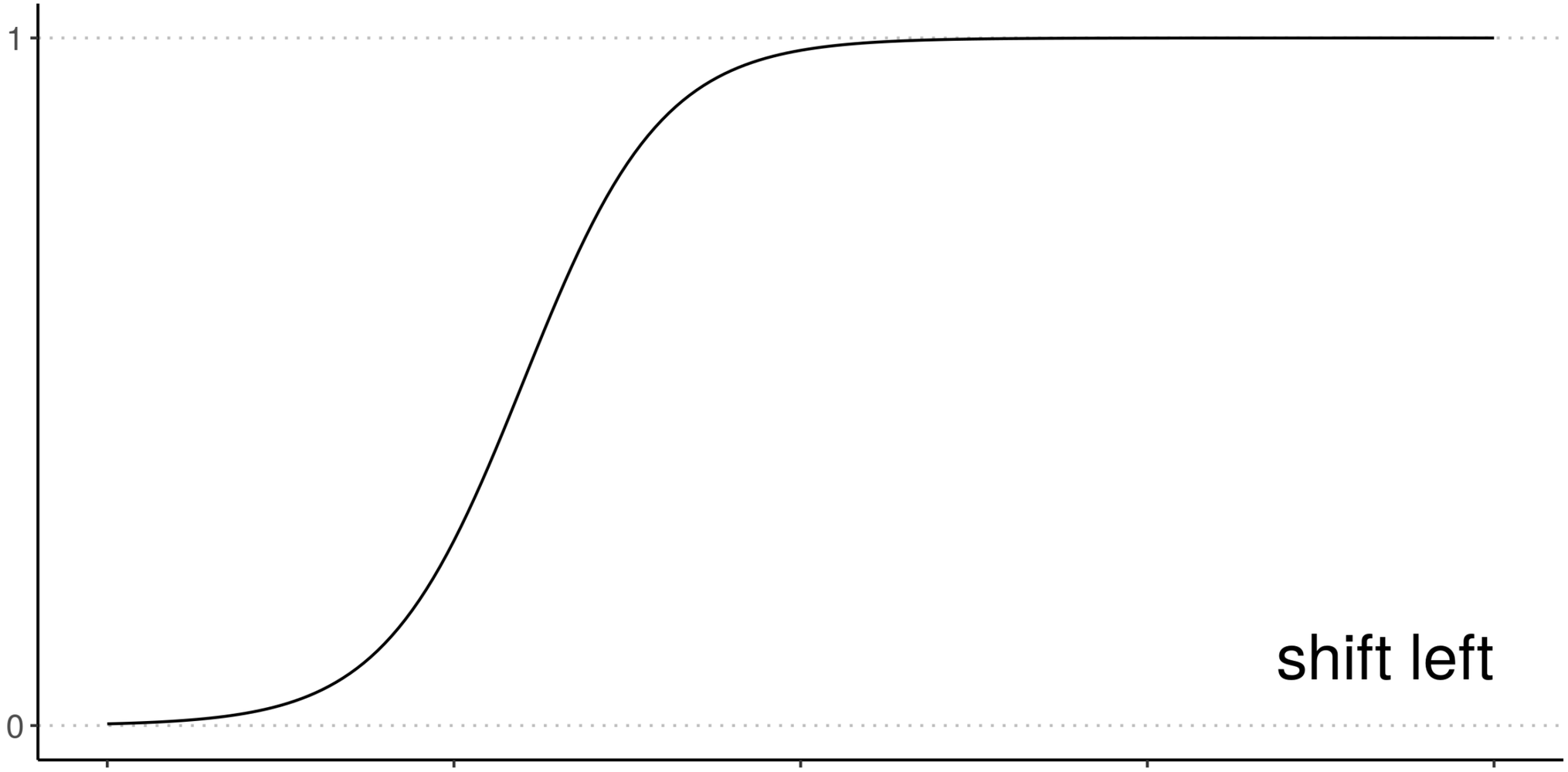
Logistic Curve



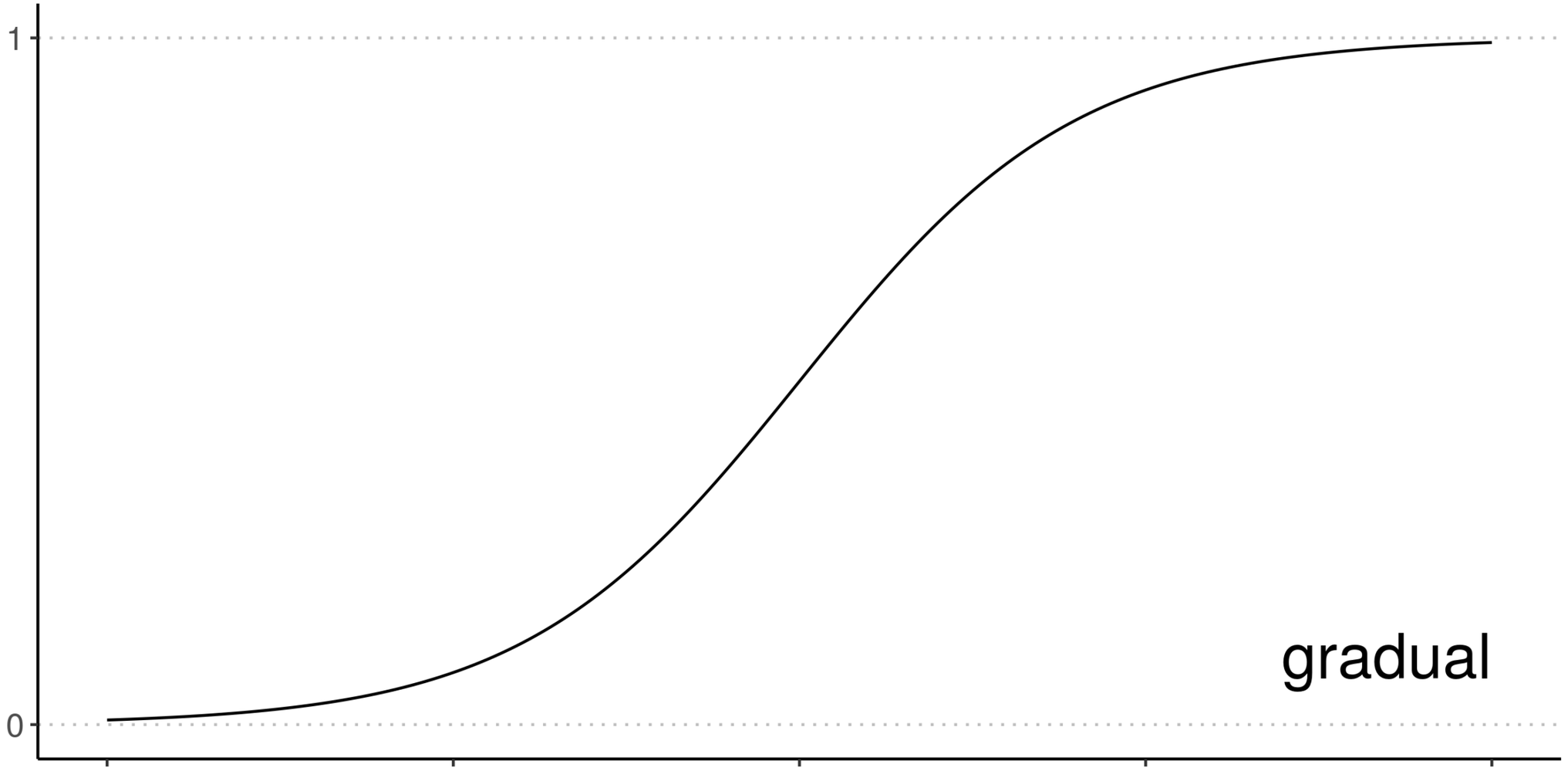
Logistic Curve



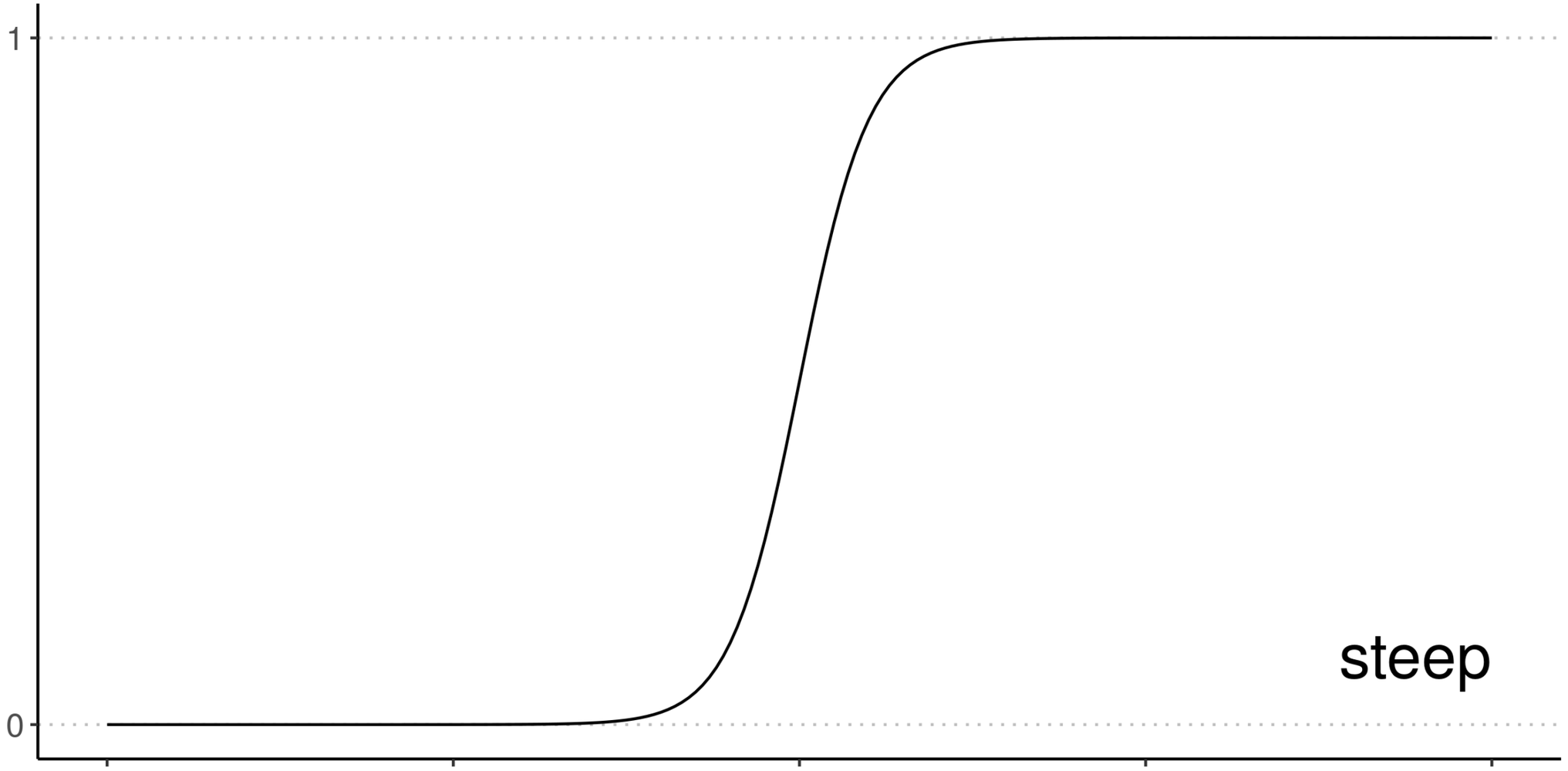
Logistic Curve



Logistic Curve



Logistic Curve



Cars revisited

Prepare for modeling:

- assemble the predictors into a single column (called `features`) and
- split data into training and testing sets.

```
+---+----+------+------+----+-----------+--------------------------------+-----+
|cyl|size|mass  |length|rpm |consumption|features                        |label|
+---+----+------+------+----+-----------+--------------------------------+-----+
|6  |3.0 |1451.0|4.775 |5200|9.05       |[6.0,3.0,1451.0,4.775,5200.0,9.05]|1.0  |
|4  |2.2 |1129.0|4.623 |5200|6.53       |[4.0,2.2,1129.0,4.623,5200.0,6.53]|0.0  |
|4  |2.2 |1399.0|4.547 |5600|7.84       |[4.0,2.2,1399.0,4.547,5600.0,7.84]|1.0  |
|4  |1.8 |1147.0|4.343 |6500|7.84       |[4.0,1.8,1147.0,4.343,6500.0,7.84]|0.0  |
|4  |1.6 |1111.0|4.216 |5750|9.05       |[4.0,1.6,1111.0,4.216,5750.0,9.05]|0.0  |
+---+----+------+------+----+-----------+--------------------------------+-----+
```

Build a Logistic Regression model

```
from pyspark.ml.classification import LogisticRegression
```

Create a Logistic Regression classifier.

```
logistic = LogisticRegression()
```

Learn from the training data.

```
logistic = logistic.fit(cars_train)
```

Predictions

```
prediction = logistic.transform(cars_test)
```

```
+-----+----------+----------------------------------------+
|label|prediction|probability                             |
+-----+----------+----------------------------------------+
|0.0  |0.0       |[0.8683802216422138,0.1316197783577862]|
|0.0  |1.0       |[0.1343792056399585,0.8656207943600416]|
|0.0  |0.0       |[0.9773546766387631,0.0226453233612368]|
|1.0  |1.0       |[0.0170508265586195,0.9829491734413806]|
|1.0  |0.0       |[0.6122241729292978,0.3877758270707023]|
+-----+----------+----------------------------------------+
```

Precision and recall

How well does model work on testing data?

Consult the confusion matrix.

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|  1.0|       1.0|    8| - TP (true positive)
|  0.0|       1.0|    4| - FP (false positive)
|  1.0|       0.0|    2| - FN (true negative)
|  0.0|       0.0|   10| - TN (false negative)
+-----+----------+-----+
```

```
# Precision (positive)
TP / (TP + FP)
```

```
0.6666666666666666
```

```
# Recall (positive)
TP / (TP + FN)
```

```
0.8
```

Weighted metrics

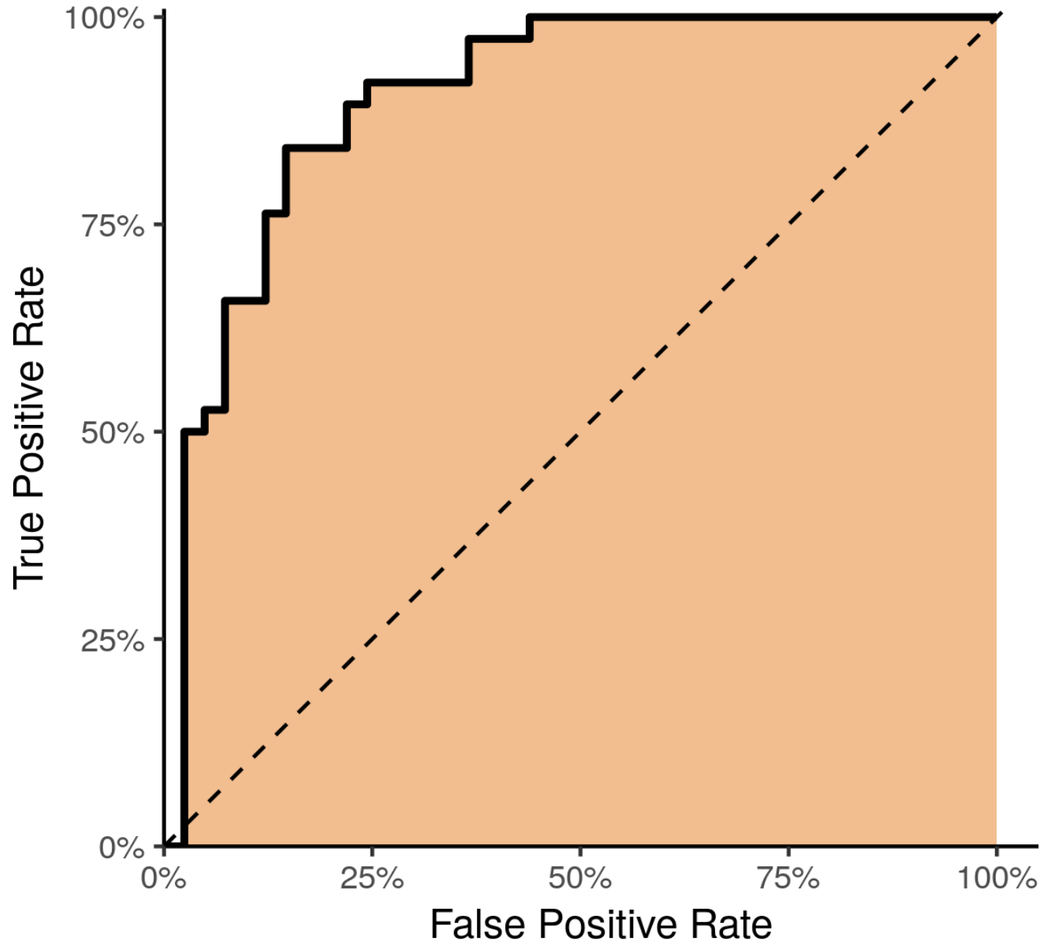
```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator()
evaluator.evaluate(prediction, {evaluator.metricName: 'weightedPrecision'})
```

```
0.7638888888888888
```

Other metrics:

- `weightedRecall`
- `accuracy`
- `f1`

ROC and AUC



ROC = "Receiver Operating Characteristic"

- TP versus FP
- threshold = 0 (top right)
- threshold = 1 (bottom left)

AUC = "Area under the curve"

- ideally AUC = 1

Let's do Logistic Regression!

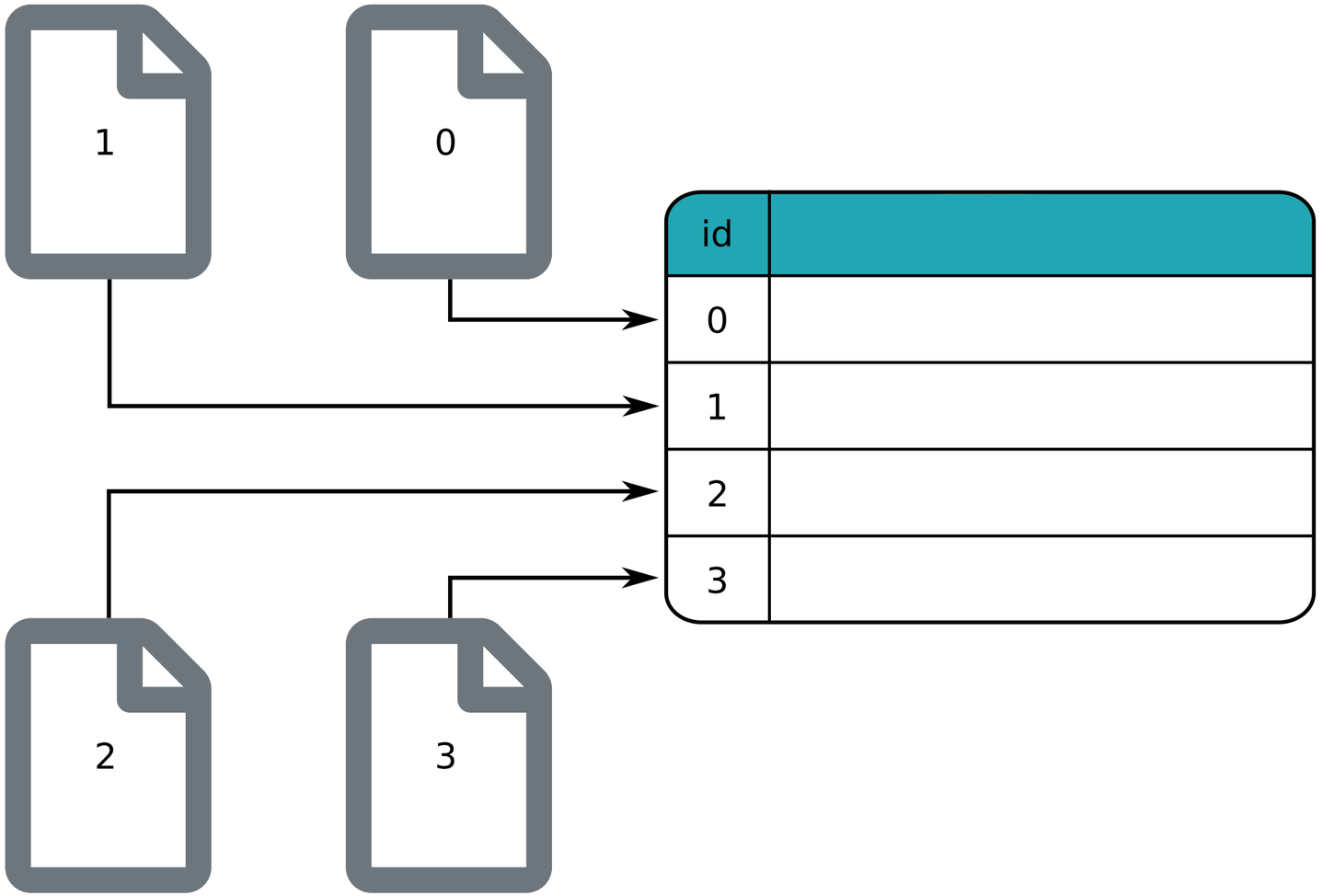
MACHINE LEARNING WITH PYSPARK

Turning Text into Tables

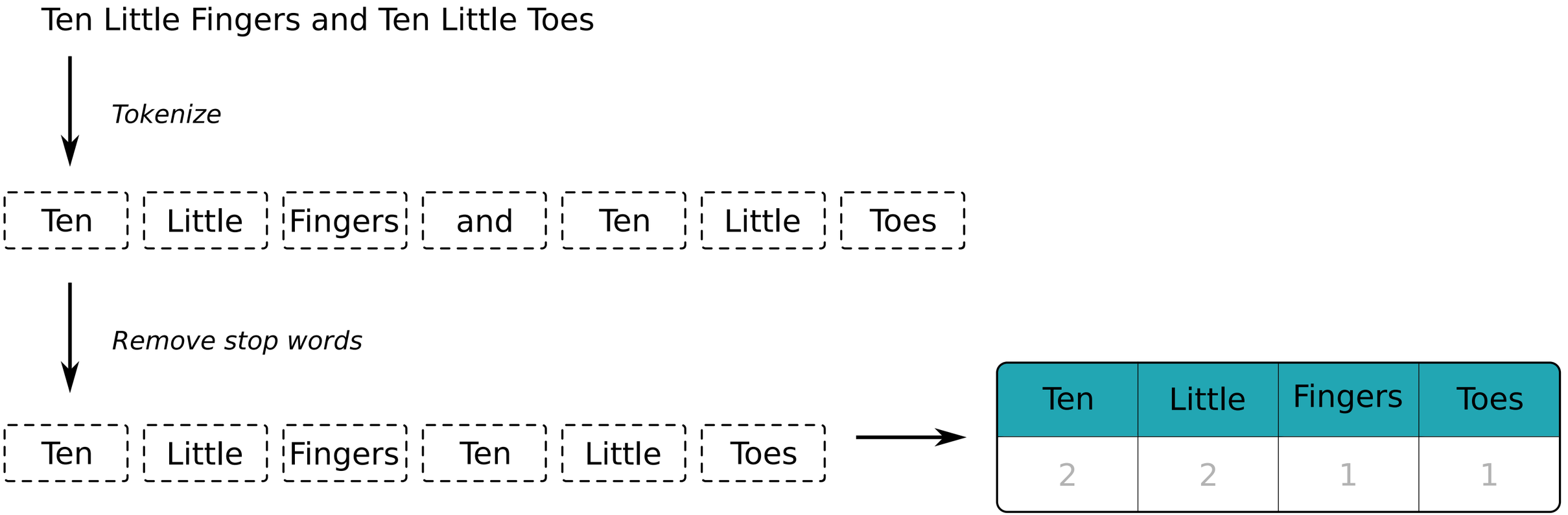
MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

One record per document



One document, many columns



A selection of children's books

```
books.show(truncate=False)
```

```
+---+-------------------------------------+
|id |text                                 |
+---+-------------------------------------+
|0  |Forever, or a Long, Long Time        | ---> 'Long' is only present in this title
|1  |Winnie-the-Pooh                      |
|2  |Ten Little Fingers and Ten Little Toes|
|3  |Five Get into Trouble                | -+-> 'Five' is present in all of these titles
|4  |Five Have a Wonderful Time           |  |
|5  |Five Get into a Fix                  |  |
|6  |Five Have Plenty of Fun              | -+
+---+-------------------------------------+
```

Removing punctuation

```
from pyspark.sql.functions import regexp_replace

# Regular expression (REGEX) to match commas and hyphens
REGEX = '[,\\-]'

books = books.withColumn('text', regexp_replace(books.text, REGEX, ' '))
```

```
Before                                  ->   After
+---+-----------------------------+         +---+-----------------------------+
|id |text                         |         |id |text                         |
+---+-----------------------------+         +---+-----------------------------+
|0  |Forever, or a Long, Long Time|         |0  |Forever  or a Long  Long Time|
|1  |Winnie-the-Pooh              |         |1  |Winnie the Pooh              |
+---+-----------------------------+         +---+-----------------------------+
```

Text to tokens

```
from pyspark.ml.feature import Tokenizer

books = Tokenizer(inputCol="text", outputCol="tokens").transform(books)
```

```
+----------------------------------------+-----------------------------------------------+
|text                                    |tokens                                         |
+----------------------------------------+-----------------------------------------------+
|Forever or a Long Long Time             |[forever, or, a, long, long, time]             |
|Winnie the Pooh                         |[winnie, the, pooh]                            |
|Ten Little Fingers and Ten Little Toes|[ten, little, fingers, and, ten, little, toes]|
|Five Get into Trouble                   |[five, get, into, trouble]                     |
|Five Have a Wonderful Time              |[five, have, a, wonderful, time]               |
+----------------------------------------+-----------------------------------------------+
```

What are stop words?

```
from pyspark.ml.feature import StopWordsRemover

stopwords = StopWordsRemover()

# Take a look at the list of stop words
stopwords.getStopWords()
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours',
'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers', 'herself',
'it', 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which',
'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be',
'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', ...]
```

Removing stop words

```
# Specify the input and output column names
stopwords = stopwords.setInputCol('tokens').setOutputCol('words')

books = stopwords.transform(books)
```

```
+----------------------------------------+----------------------------------------+
|tokens                                  |words                                   |
+----------------------------------------+----------------------------------------+
|[forever, or, a, long, long, time]      |[forever, long, long, time]             |
|[winnie, the, pooh]                     |[winnie, pooh]                          |
|[ten, little, fingers, and, ten, little, toes]|[ten, little, fingers, ten, little, toes]|
|[five, get, into, trouble]              |[five, get, trouble]                    |
|[five, have, a, wonderful, time]        |[five, wonderful, time]                 |
+----------------------------------------+----------------------------------------+
```

Feature hashing

```
from pyspark.ml.feature import HashingTF

hasher = HashingTF(inputCol="words", outputCol="hash", numFeatures=32)
books = hasher.transform(books)
```

```
+---+---------------------------------------+---------------------------------+
|id |words                                  |hash                             |
+---+---------------------------------------+---------------------------------+
|0  |[forever, long, long, time]            |(32,[8,13,14],[2.0,1.0,1.0])     |
|1  |[winnie, pooh]                         |(32,[1,31],[1.0,1.0])            |
|2  |[ten, little, fingers, ten, little, toes]|(32,[1,15,25,30],[2.0,2.0,1.0,1.0])|
|3  |[five, get, trouble]                   |(32,[6,7,23],[1.0,1.0,1.0])      |
|4  |[five, wonderful, time]                |(32,[6,13,25],[1.0,1.0,1.0])     |
+---+---------------------------------------+---------------------------------+
```

Dealing with common words

```
from pyspark.ml.feature import IDF

books = IDF(inputCol="hash", outputCol="features").fit(books).transform(books)
```

```
+----------------------------------------+----------------------------------------+
|words                                   |features                                |
+----------------------------------------+----------------------------------------+
|[forever, long, long, time]             |(32,[8,13,14],[2.598,1.299,1.704])      |
|[winnie, pooh]                          |(32,[1,31],[1.299,1.704])               |
|[ten, little, fingers, ten, little, toes]|(32,[1,15,25,30],[2.598,3.409,1.011,1.704])|
|[five, get, trouble]                    |(32,[6,7,23],[0.788,1.704,1.299])       |
|[five, wonderful, time]                 |(32,[6,13,25],[0.788,1.299,1.011])      |
+----------------------------------------+----------------------------------------+
```

Text ready for Machine Learning!

MACHINE LEARNING WITH PYSPARK

One-Hot Encoding

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

The problem with indexed values

```
# Counts for 'type' category

+-------+-----+
|   type|count|
+-------+-----+
|Midsize|   22|
|  Small|   21|
|Compact|   16|
| Sporty|   14|
|  Large|   11|
|    Van|    9|
+-------+-----+
```

```
# Numerical indices for 'type' category

+-------+--------+
|   type|type_idx|
+-------+--------+
|Midsize|     0.0|
|  Small|     1.0|
|Compact|     2.0|
| Sporty|     3.0|
|  Large|     4.0|
|    Van|     5.0|
+-------+--------+
```

Dummy variables

```
+-------+      +-------+-------+-------+-------+-------+-------+
|   type|      |Midsize|  Small|Compact| Sporty|  Large|    Van|
+-------+      +-------+-------+-------+-------+-------+-------+
|Midsize|      |   X   |       |       |       |       |       |
|  Small|      |       |   X   |       |       |       |       |
|Compact| ===> |       |       |   X   |       |       |       |
| Sporty|      |       |       |       |   X   |       |       |
|  Large|      |       |       |       |       |   X   |       |
|    Van|      |       |       |       |       |       |   X   |
+-------+      +-------+-------+-------+-------+-------+-------+
```

Each categorical level becomes a column.

Dummy variables: binary encoding

```
+-------+      +-------+-------+-------+-------+-------+-------+
|   type|      |Midsize|  Small|Compact| Sporty|  Large|    Van|
+-------+      +-------+-------+-------+-------+-------+-------+
|Midsize|      |   1   |   0   |   0   |   0   |   0   |   0   |
|  Small|      |   0   |   1   |   0   |   0   |   0   |   0   |
|Compact| ===> |   0   |   0   |   1   |   0   |   0   |   0   |
| Sporty|      |   0   |   0   |   0   |   1   |   0   |   0   |
|  Large|      |   0   |   0   |   0   |   0   |   1   |   0   |
|    Van|      |   0   |   0   |   0   |   0   |   0   |   1   |
+-------+      +-------+-------+-------+-------+-------+-------+
```

Binary values indicate the presence (`1`) or absence (`0`) of the corresponding level.

Dummy variables: sparse representation

```
+-------+      +-------+-------+-------+-------+-------+-------+      +------+-----+
|   type|      |Midsize|  Small|Compact| Sporty|  Large|    Van|      |Column|Value|
+-------+      +-------+-------+-------+-------+-------+-------+      +------+-----+
|Midsize|      |   1   |   0   |   0   |   0   |   0   |   0   |      |     0|    1|
|  Small|      |   0   |   1   |   0   |   0   |   0   |   0   |      |     1|    1|
|Compact| ===> |   0   |   0   |   1   |   0   |   0   |   0   | ===> |     2|    1|
| Sporty|      |   0   |   0   |   0   |   1   |   0   |   0   |      |     3|    1|
|  Large|      |   0   |   0   |   0   |   0   |   1   |   0   |      |     4|    1|
|    Van|      |   0   |   0   |   0   |   0   |   0   |   1   |      |     5|    1|
+-------+      +-------+-------+-------+-------+-------+-------+      +------+-----+
```

Sparse representation: store column index and value.

Dummy variables: redundant column

```
+-------+      +-------+-------+-------+-------+-------+      +------+-----+
|   type|      |Midsize|  Small|Compact| Sporty|  Large|      |Column|Value|
+-------+      +-------+-------+-------+-------+-------+      +------+-----+
|Midsize|      |   1   |   0   |   0   |   0   |   0   |      |     0|    1|
|  Small|      |   0   |   1   |   0   |   0   |   0   |      |     1|    1|
|Compact| ===> |   0   |   0   |   1   |   0   |   0   | ===> |     2|    1|
| Sporty|      |   0   |   0   |   0   |   1   |   0   |      |     3|    1|
|  Large|      |   0   |   0   |   0   |   0   |   1   |      |     4|    1|
|    Van|      |   0   |   0   |   0   |   0   |   0   |      |      |     |
+-------+      +-------+-------+-------+-------+-------+      +------+-----+
```

Levels are mutually exclusive, so drop one.

One-hot encoding

```
from pyspark.ml.feature import OneHotEncoderEstimator

onehot = OneHotEncoderEstimator(inputCols=['type_idx'], outputCols=['type_dummy'])
```

Fit the encoder to the data.

```
onehot = onehot.fit(cars)
```

```
# How many category levels?
onehot.categorySizes
```

```
[6]
```

One-hot encoding

```
cars = onehot.transform(cars)
cars.select('type', 'type_idx', 'type_dummy').distinct().sort('type_idx').show()
```

```
+-------+--------+-------------+
|   type|type_idx|   type_dummy|
+-------+--------+-------------+
|Midsize|     0.0|(5,[0],[1.0])|
|  Small|     1.0|(5,[1],[1.0])|
|Compact|     2.0|(5,[2],[1.0])|
| Sporty|     3.0|(5,[3],[1.0])|
|  Large|     4.0|(5,[4],[1.0])|
|    Van|     5.0|    (5,[],[])|
+-------+--------+-------------+
```

Dense versus sparse

```
from pyspark.mllib.linalg import DenseVector, SparseVector
```

Store this vector: [1, 0, 0, 0, 0, 7, 0, 0].

```
DenseVector([1, 0, 0, 0, 0, 7, 0, 0])
```

```
DenseVector([1.0, 0.0, 0.0, 0.0, 0.0, 7.0, 0.0, 0.0])
```

```
SparseVector(8, [0, 5], [1, 7])
```

```
SparseVector(8, {0: 1.0, 5: 7.0})
```

One-Hot Encode categoricals

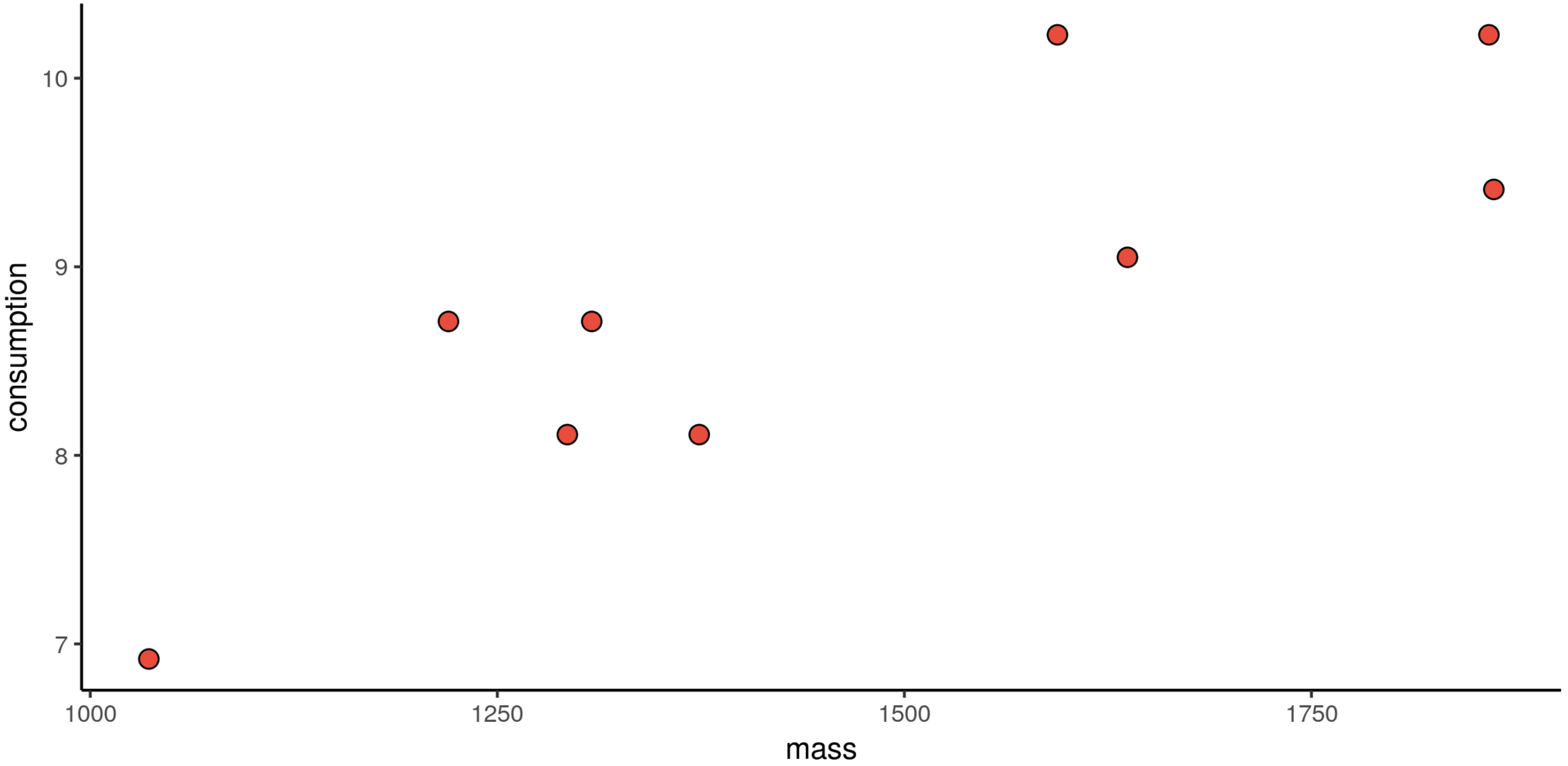
MACHINE LEARNING WITH PYSPARK

Regression

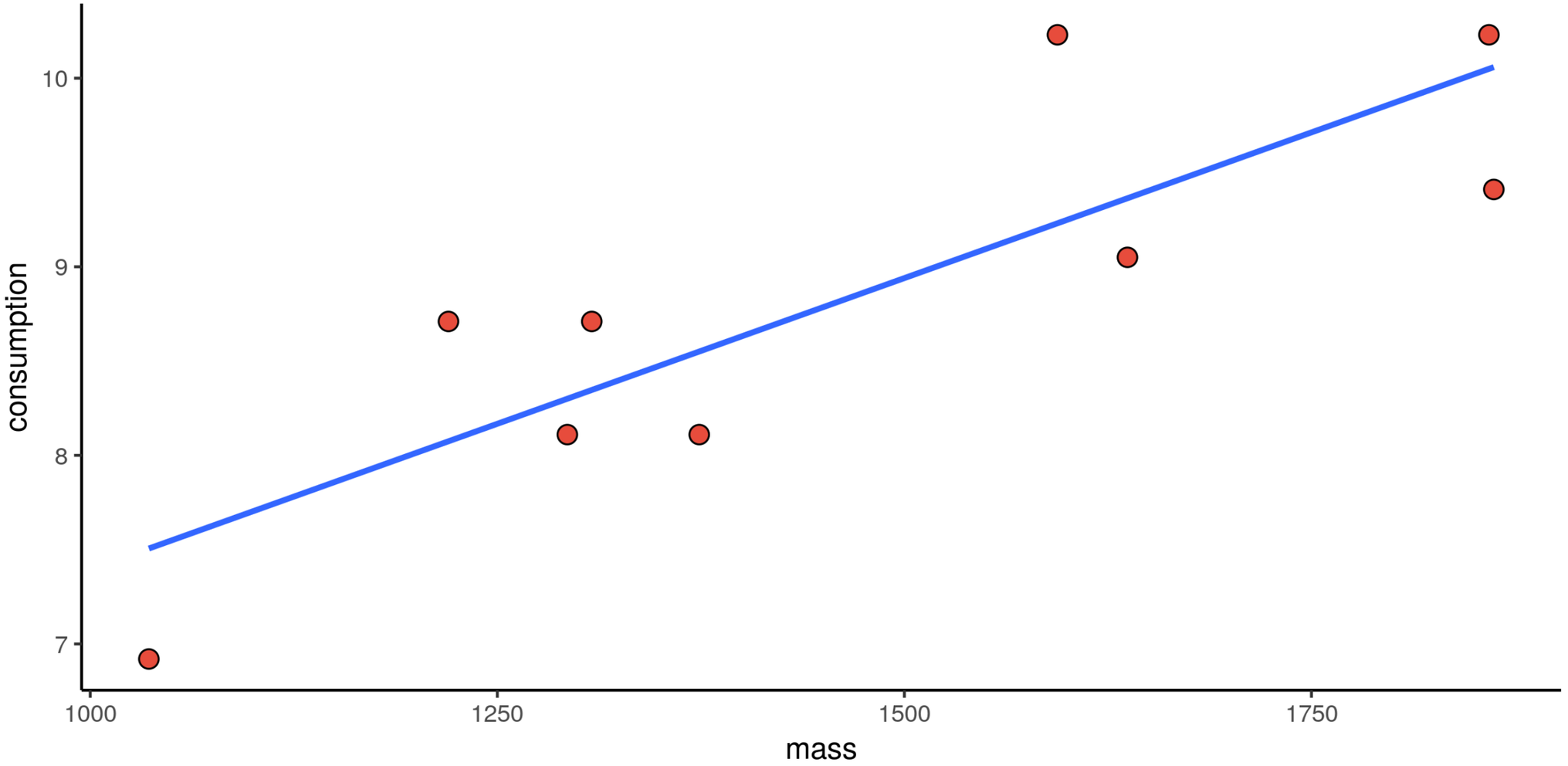
MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

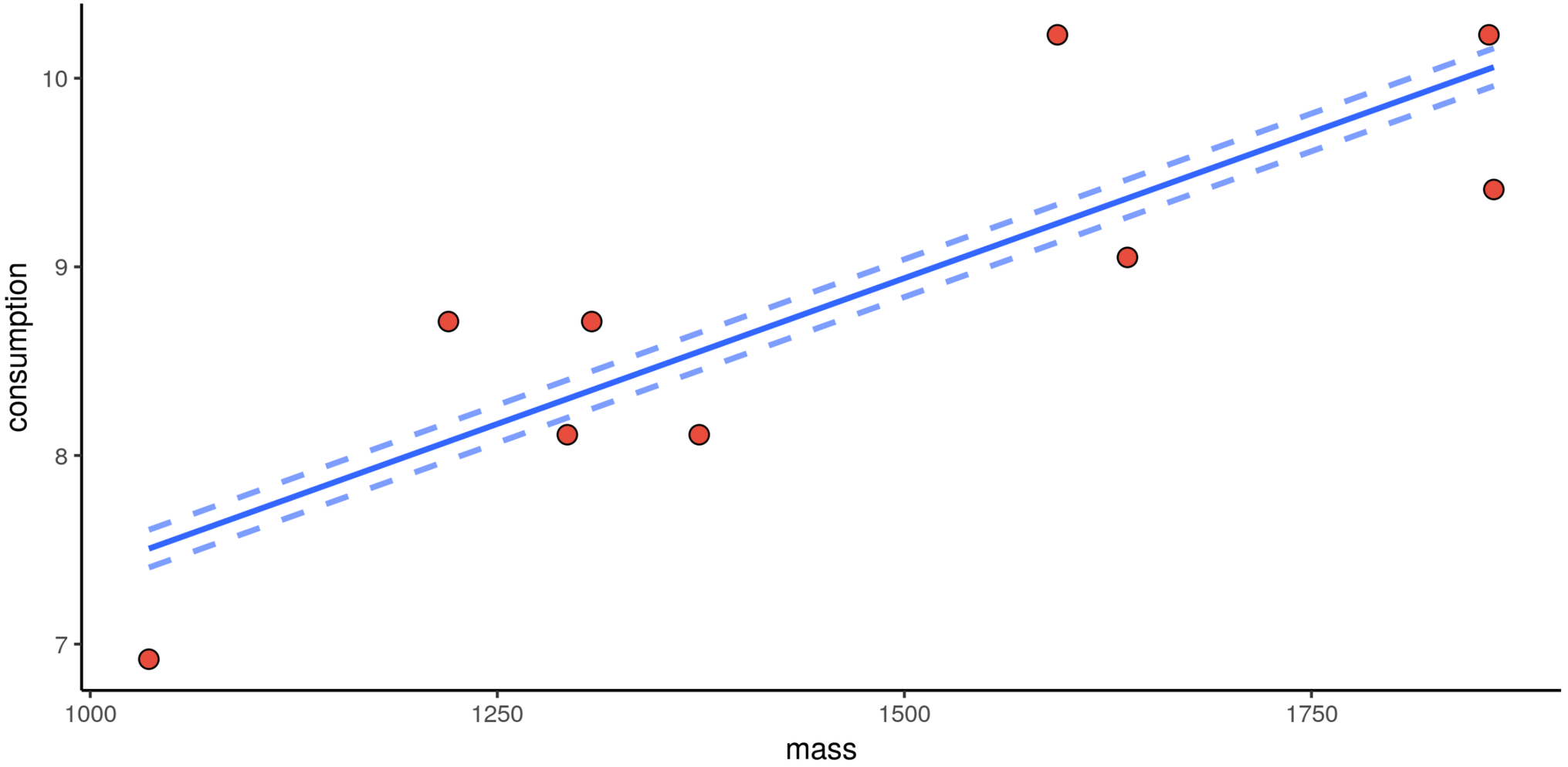
Consumption versus mass: scatter



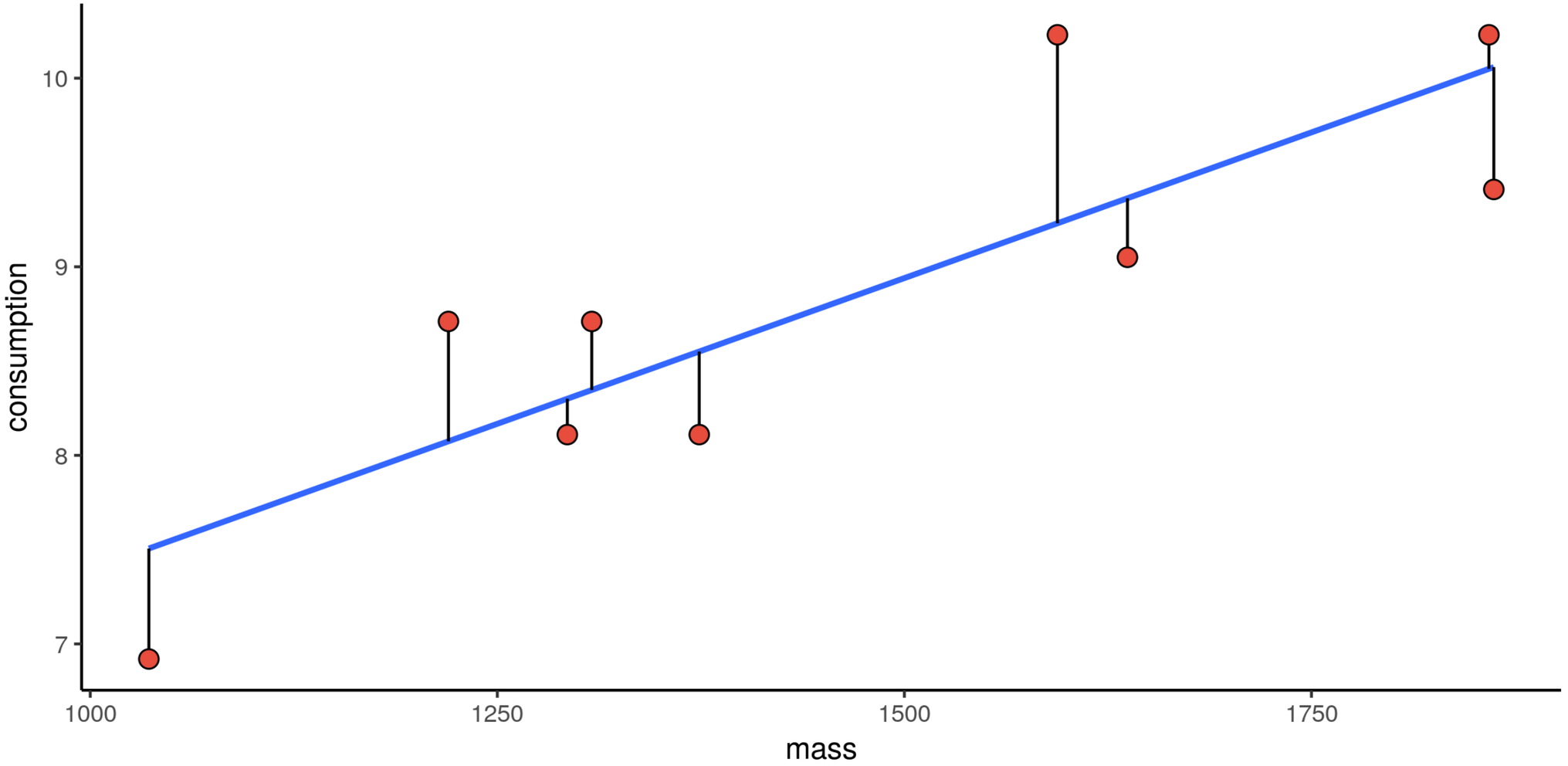
Consumption versus mass: fit



Consumption versus mass: alternative fits



Consumption versus mass: residuals



Loss function

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

MSE = "Mean Squared Error"

Loss function: Observed values

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

y_i — observed values

Loss function: Model values

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

y_i — observed values

$\hat{y}_i$ — model values

Loss function: Mean

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

y_i — observed values

$\hat{y}_i$ — model values

Assemble predictors

Predict `consumption` using `mass` , `cyl` and `type_dummy` .

Consolidate predictors into a single column.

```
+------+---+------------+-----------------------------+-----------+
|mass  |cyl|type_dummy  |features                     |consumption|
+------+---+------------+-----------------------------+-----------+
|1451.0|6  |(5,[0],[1.0])|(7,[0,1,2],[1451.0,6.0,1.0])|9.05       |
|1129.0|4  |(5,[2],[1.0])|(7,[0,1,4],[1129.0,4.0,1.0])|6.53       |
|1399.0|4  |(5,[2],[1.0])|(7,[0,1,4],[1399.0,4.0,1.0])|7.84       |
|1147.0|4  |(5,[1],[1.0])|(7,[0,1,3],[1147.0,4.0,1.0])|7.84       |
|1111.0|4  |(5,[3],[1.0])|(7,[0,1,5],[1111.0,4.0,1.0])|9.05       |
+------+---+------------+-----------------------------+-----------+
```

Build regression model

```
from pyspark.ml.regression import LinearRegression

regression = LinearRegression(labelCol='consumption')
```

Fit to `cars_train` (training data).

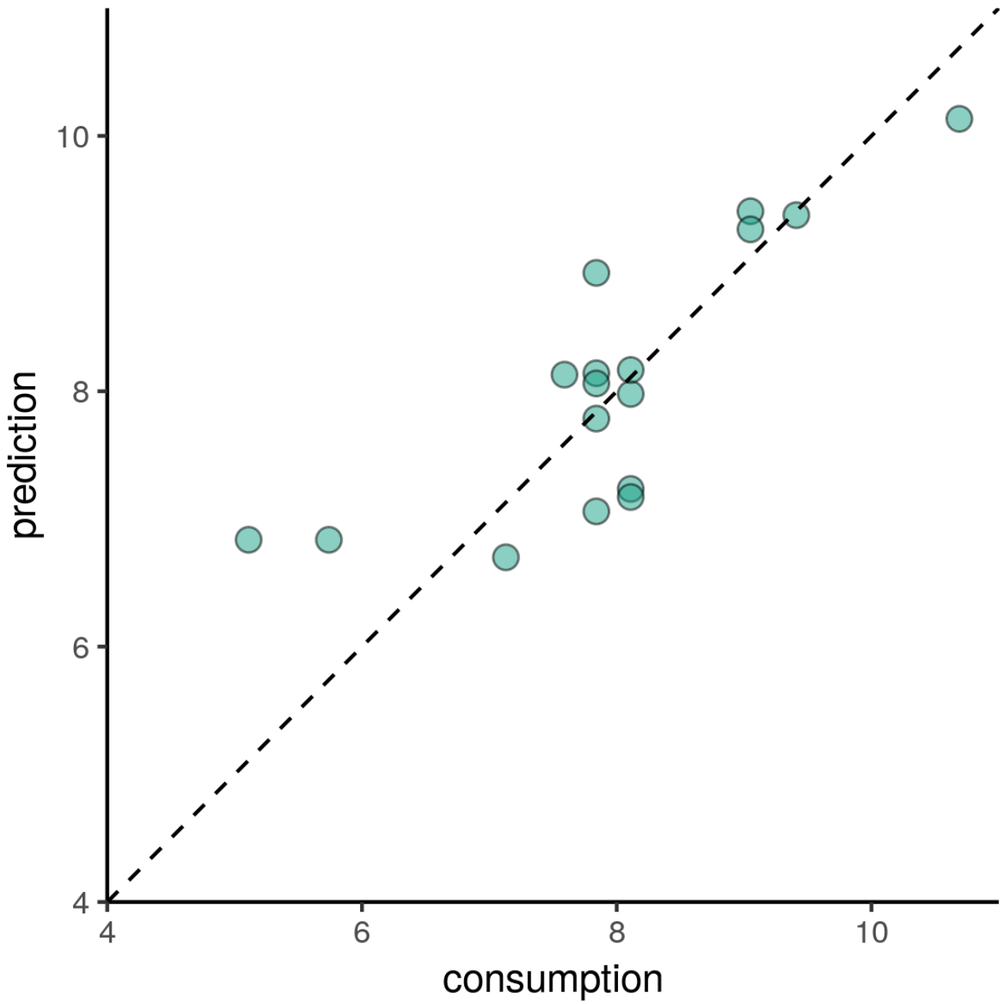
```
regression = regression.fit(cars_train)
```

Predict on `cars_test` (testing data).

```
predictions = regression.transform(cars_test)
```

Examine predictions

```
+-----------+------------------+
|consumption|prediction        |
+-----------+------------------+
|7.84       |8.92699470743403  |
|9.41       |9.379295891451353 |
|8.11       |7.23487264538364  |
|9.05       |9.409860194333735 |
|7.84       |7.059190923328711 |
|7.84       |7.785909738591766 |
|7.59       |8.129959405168547 |
|5.11       |6.836843743852942 |
|8.11       |7.17173702652015  |
+-----------+------------------+
```



Calculate RMSE

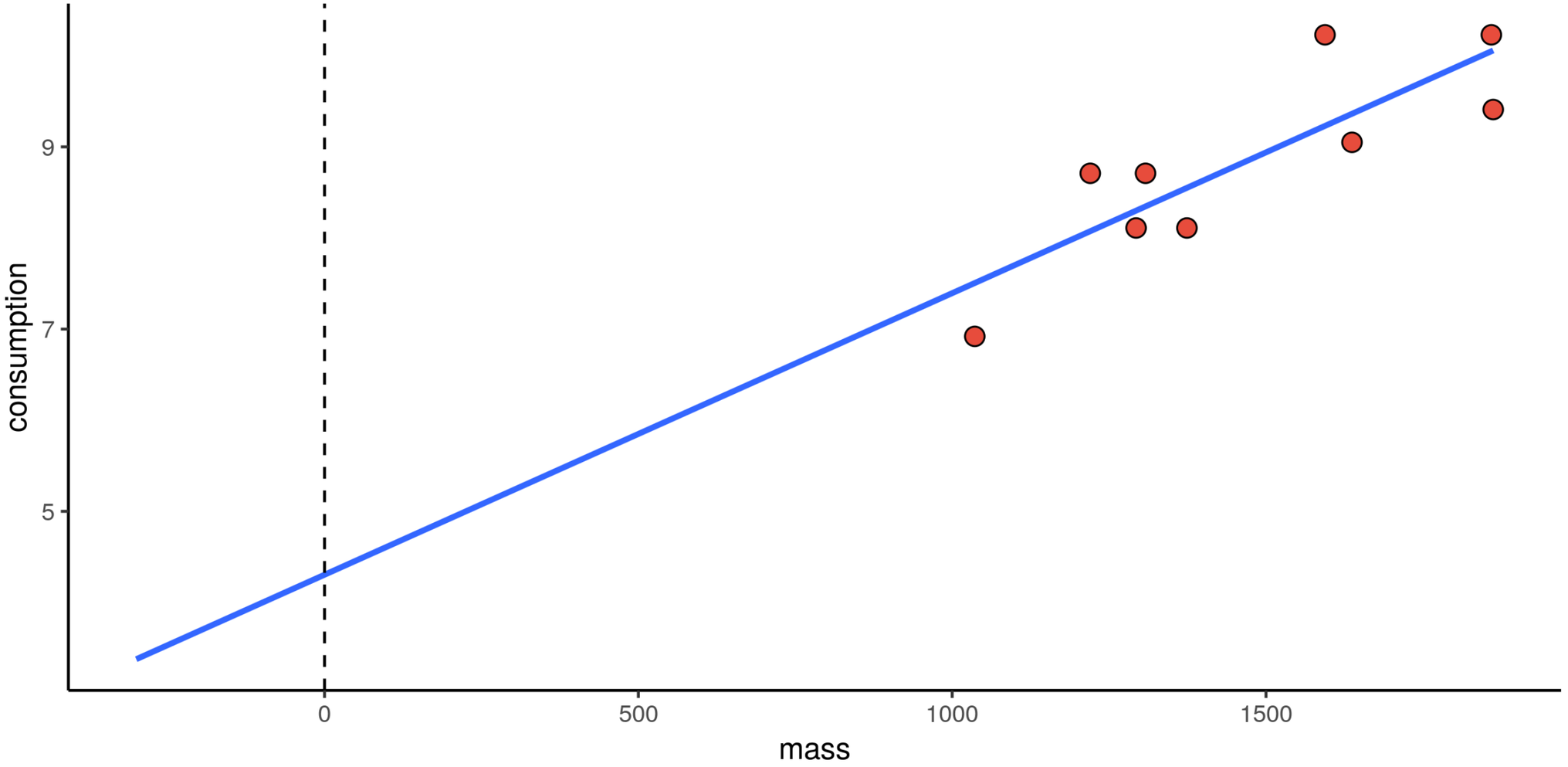
```
from pyspark.ml.evaluation import RegressionEvaluator

# Find RMSE (Root Mean Squared Error)
RegressionEvaluator(labelCol='consumption').evaluate(predictions)
```

```
0.708699086182001
```

A `RegressionEvaluator` can also calculate the following metrics:

- `mae` (Mean Absolute Error)
- `r2` (R^2)
- `mse` (Mean Squared Error).

Consumption versus mass: intercept



Examine intercept

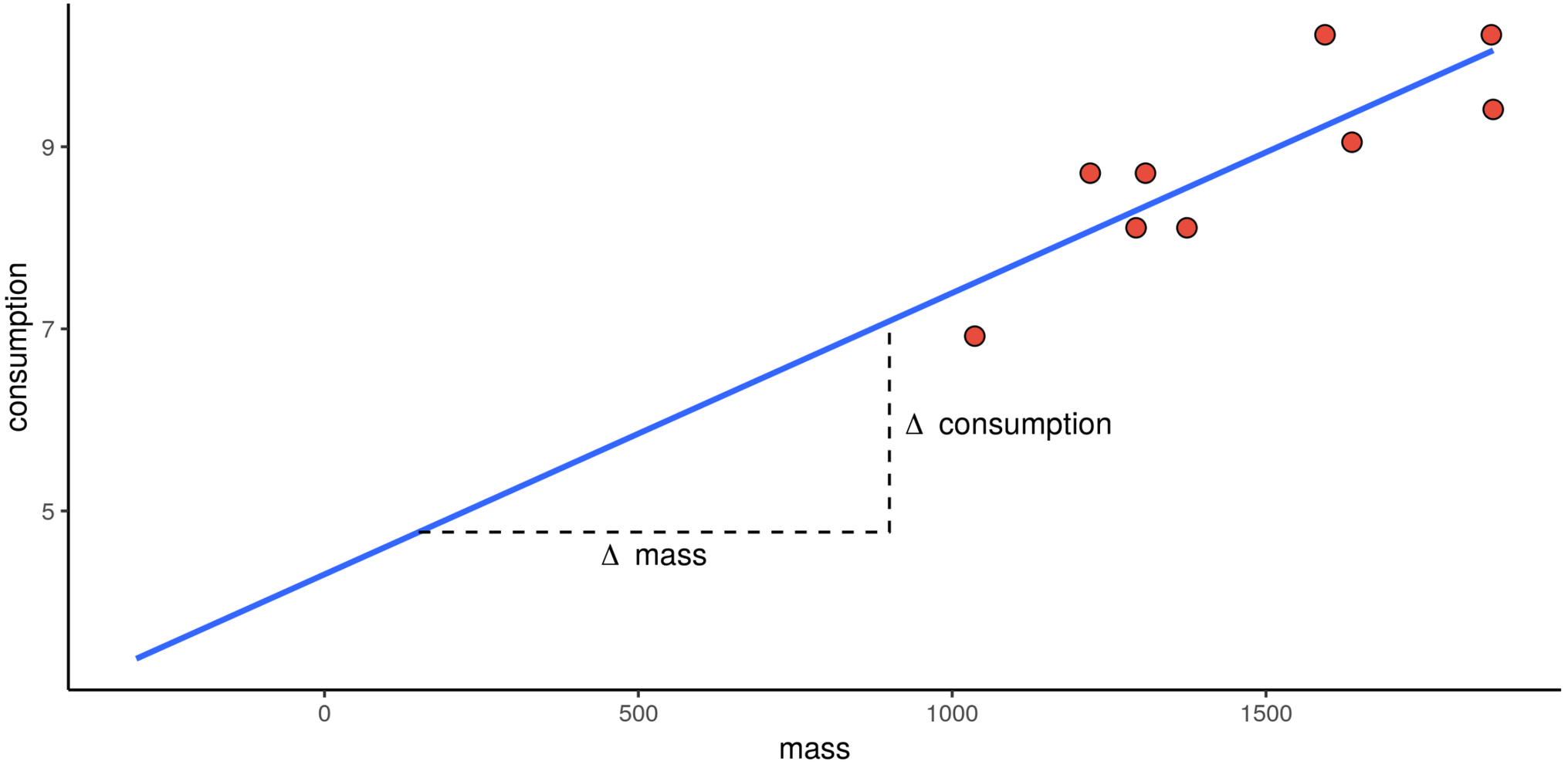
```
regression.intercept
```

```
4.94506168337270 95
```

This is the fuel consumption in the (hypothetical) case that:

- `mass` = 0
- `cyl` = 0 and
- vehicle type is 'Van'.

Consumption versus mass: slope



Examine Coefficients

```
regression.coefficients
```

```
DenseVector([0.0027, 0.1897, -1.309, -1.7933, -1.3594, -1.2917, -1.9693])
```

```
mass         0.0027
cyl          0.1897

Midsize     -1.3090
Small       -1.7933
Compact     -1.3594
Sporty      -1.2917
Large       -1.9693
```

Regression for numeric predictions

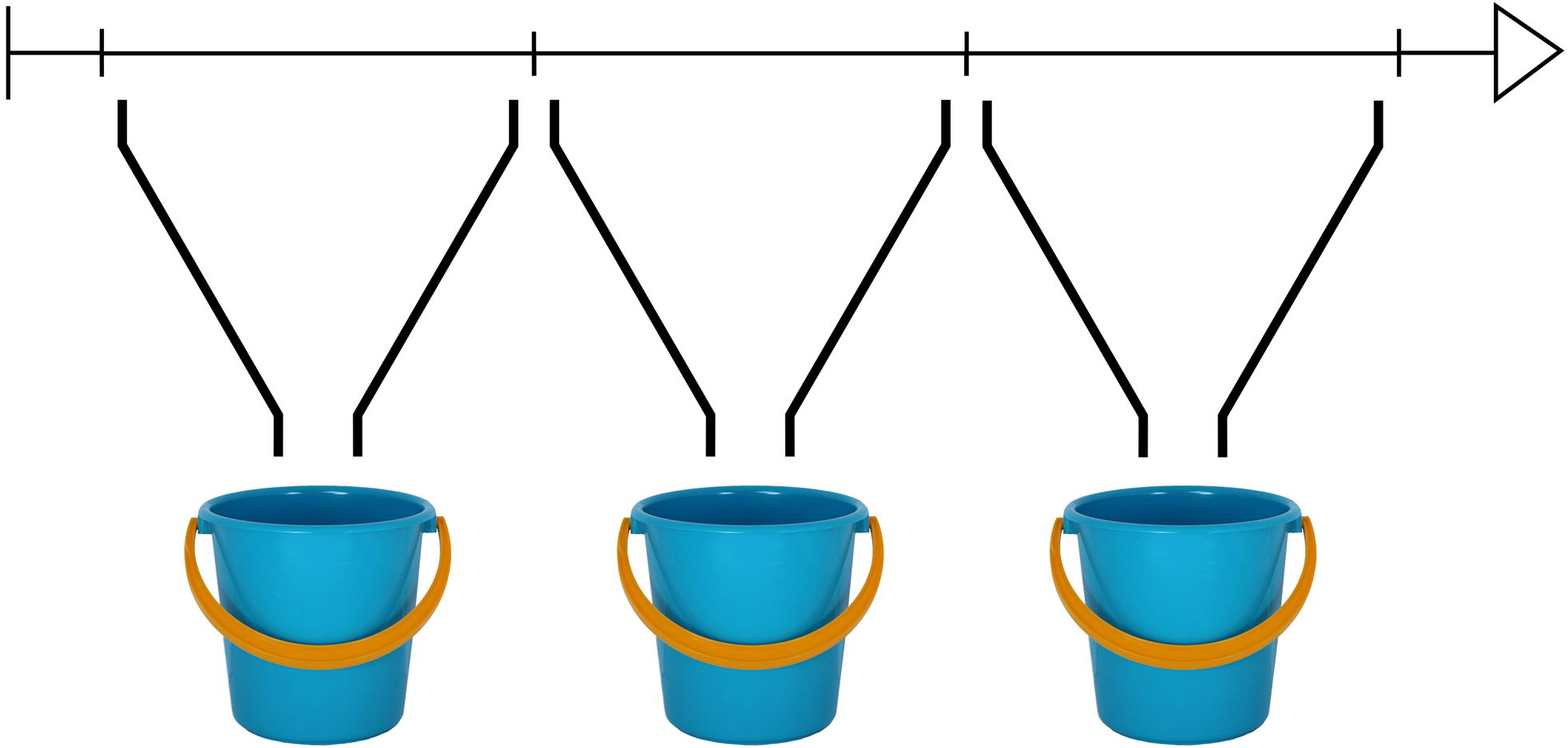
MACHINE LEARNING WITH PYSPARK

Bucketing & Engineering

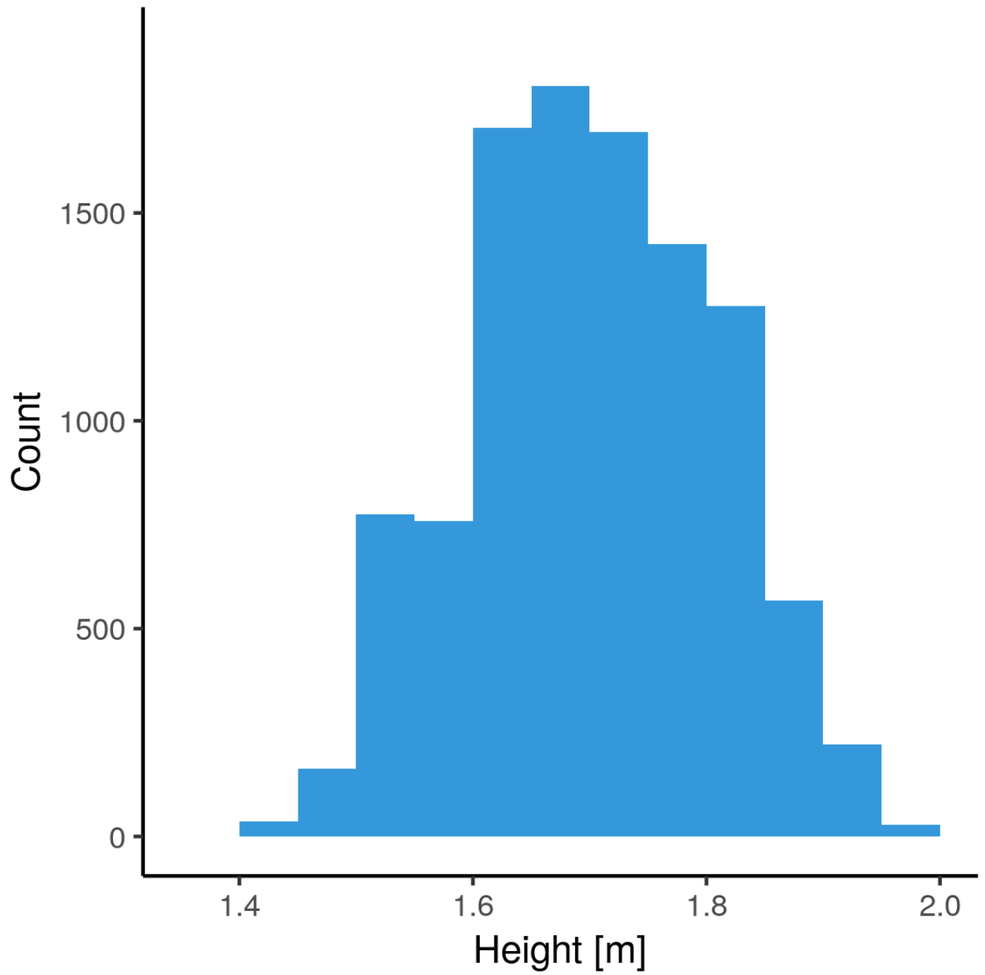
MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

Bucketing

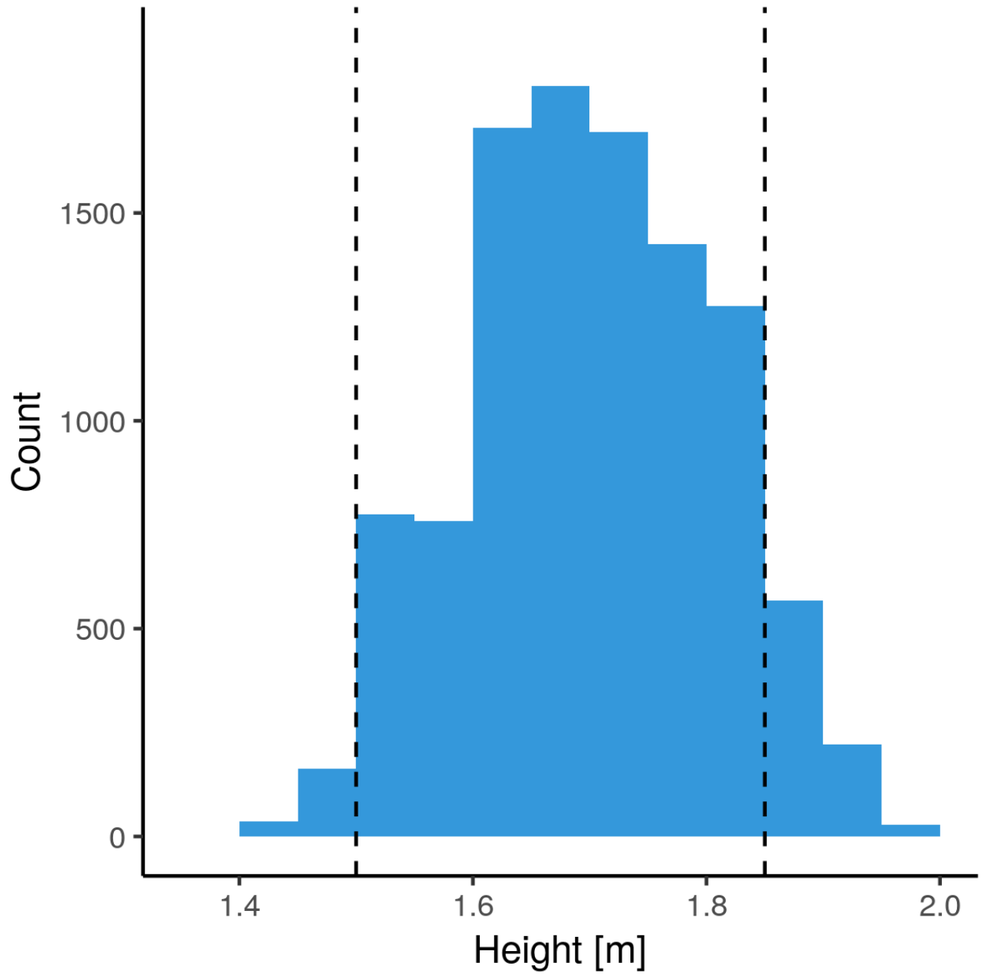


Bucketing heights



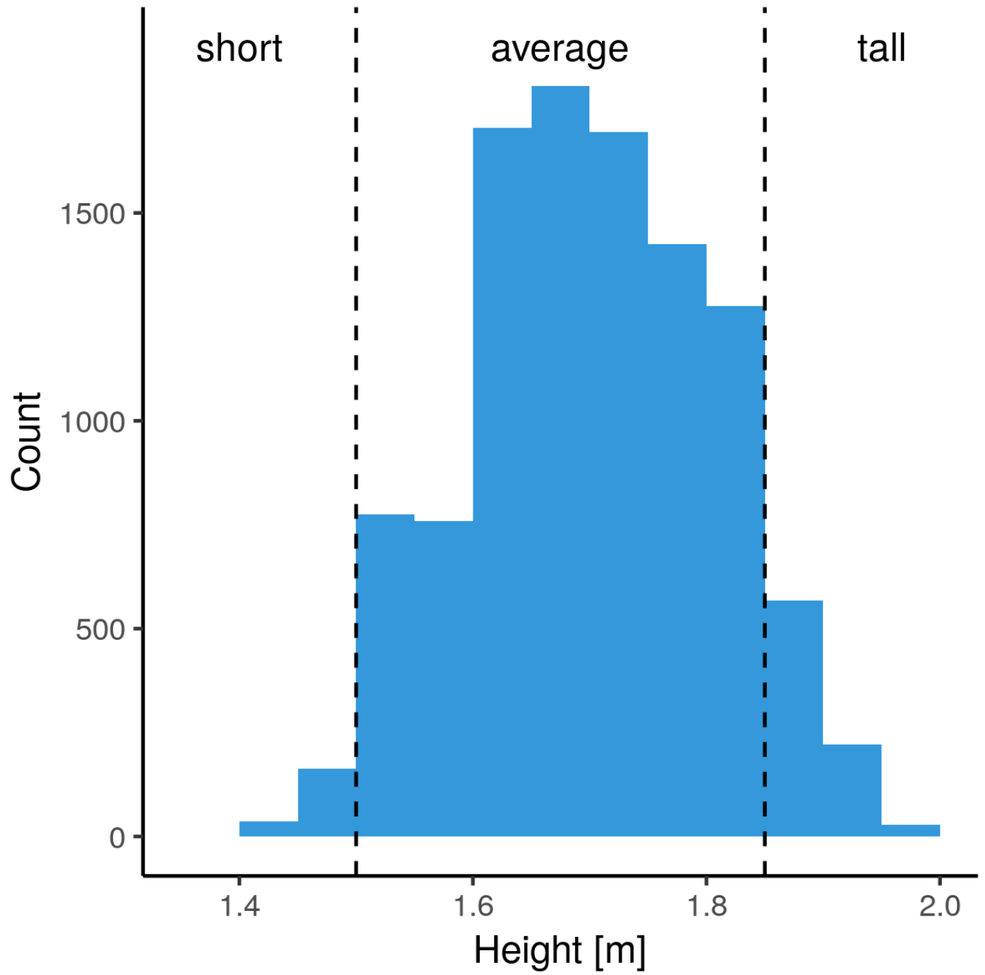

```
+------+
|height|
+------+
|  1.42|
|  1.45|
|  1.47|
|  1.50|
|  1.52|
|  1.57|
|  1.60|
|  1.75|
|  1.85|
|  1.88|
+------+
```

Bucketing heights



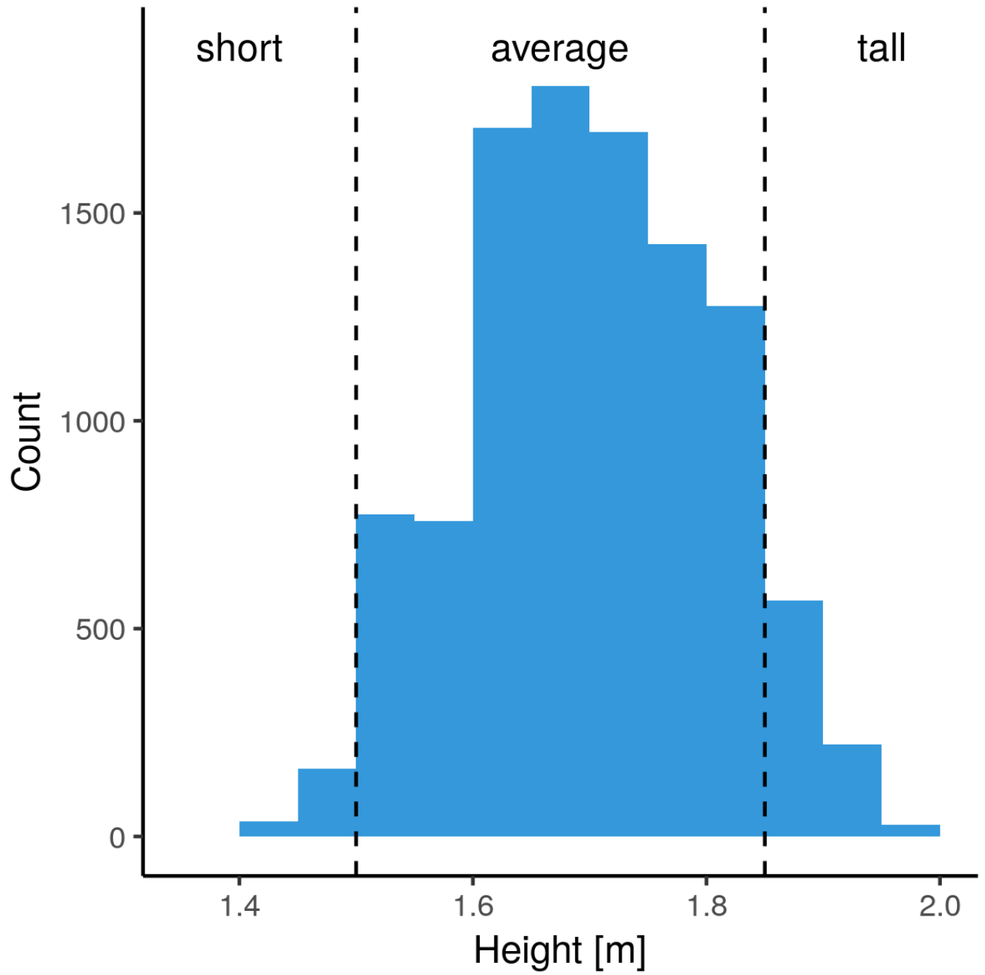

```
+------+
|height|
+------+
|  1.42|
|  1.45|
|  1.47|
|  1.50|
|  1.52|
|  1.57|
|  1.60|
|  1.75|
|  1.85|
|  1.88|
+------+
```

Bucketing heights




```
+------+
|height|
+------+
|  1.42|
|  1.45|
|  1.47|
|  1.50|
|  1.52|
|  1.57|
|  1.60|
|  1.75|
|  1.85|
|  1.88|
+------+
```

Bucketing heights

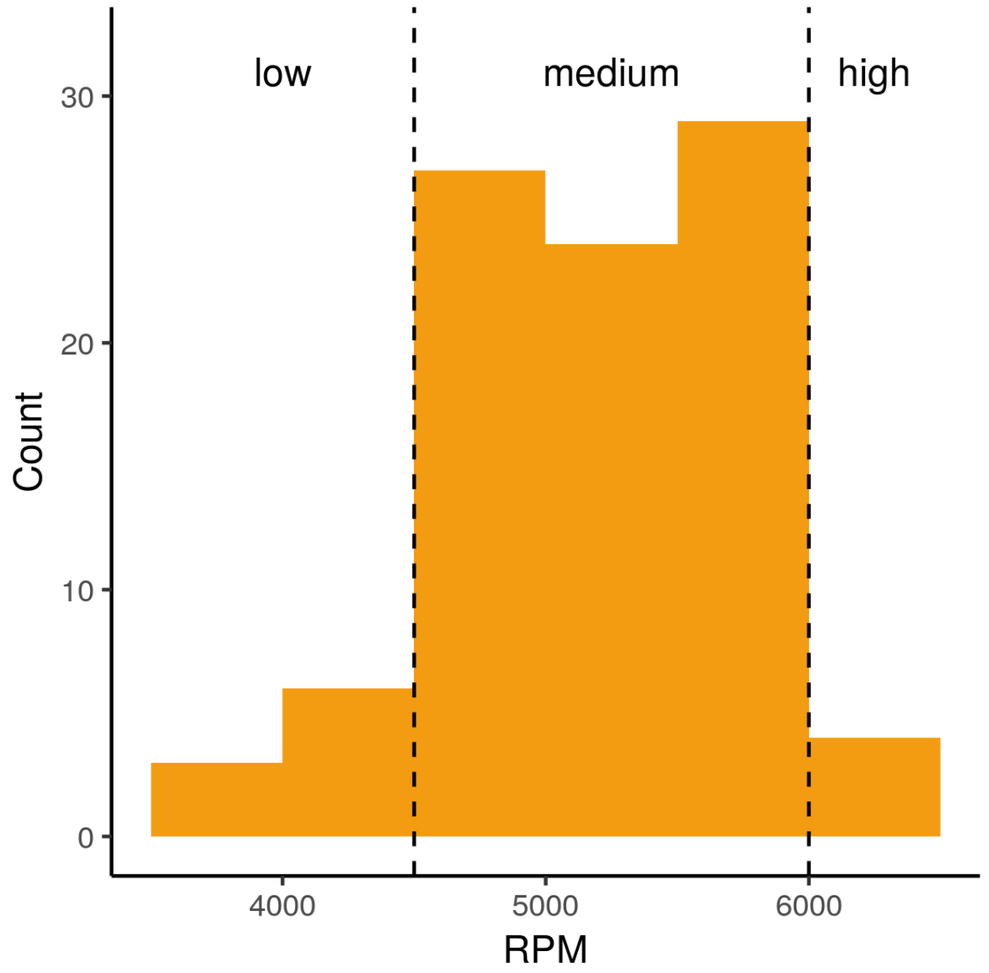



```
+------+----------+
|height|height_bin|
+------+----------+
|  1.42|     short|
|  1.45|     short|
|  1.47|     short|
|  1.50|     short|
|  1.52|   average|
|  1.57|   average|
|  1.60|   average|
|  1.75|   average|
|  1.85|      tall|
|  1.88|      tall|
+------+----------+
```

RPM histogram

Car RPM has "natural" breaks:

- $\text{RPM} < 4500$ — low
- $\text{RPM} > 6000$ — high
- otherwise — medium.



RPM buckets

```
from pyspark.ml.feature import Bucketizer

bucketizer = Bucketizer(splits=[3500, 4500, 6000, 6500],
                        inputCol="rpm",
                        outputCol="rpm_bin")
```

Apply buckets to `rpm` column.

```
cars = bucketizer.transform(cars)
```

RPM buckets

```
bucketed.select('rpm', 'rpm_bin').show(5)
```

```
+----+-------+
| rpm|rpm_bin|
+----+-------+
|3800|    0.0|
|4500|    1.0|
|5750|    1.0|
|5300|    1.0|
|6200|    2.0|
+----+-------+
```

```
cars.groupBy('rpm_bin').count().show()
```

```
+-------+-----+
|rpm_bin|count|
+-------+-----+
|    0.0|    8| <- low
|    1.0|   67| <- medium
|    2.0|   17| <- high
+-------+-----+
```

One-hot encoded RPM buckets

The RPM buckets are one-hot encoded to dummy variables.

```
+-------+-------------+
|rpm_bin|    rpm_dummy|
+-------+-------------+
|    0.0|(2,[0],[1.0])| <- low
|    1.0|(2,[1],[1.0])| <- medium
|    2.0|    (2,[],[])| <- high
+-------+-------------+
```

The 'high' RPM bucket is the reference level and doesn't get a dummy variable.

Model with bucketed RPM

```
regression.coefficients
```

```
DenseVector([1.3814, 0.1433])
```

```
+-------+-------------+
|rpm_bin|    rpm_dummy|
+-------+-------------+
|    0.0|(2,[0],[1.0])| <- low
|    1.0|(2,[1],[1.0])| <- medium
|    2.0|    (2,[],[])| <- high
+-------+-------------+
```

```
regression.intercept
```

```
8.1835
```

Consumption for 'low' RPM:

```
8.1835 + 1.3814 = 9.5649
```

Consumption for 'medium' RPM:

```
8.1835 + 0.1433 = 8.3268
```

More feature engineering

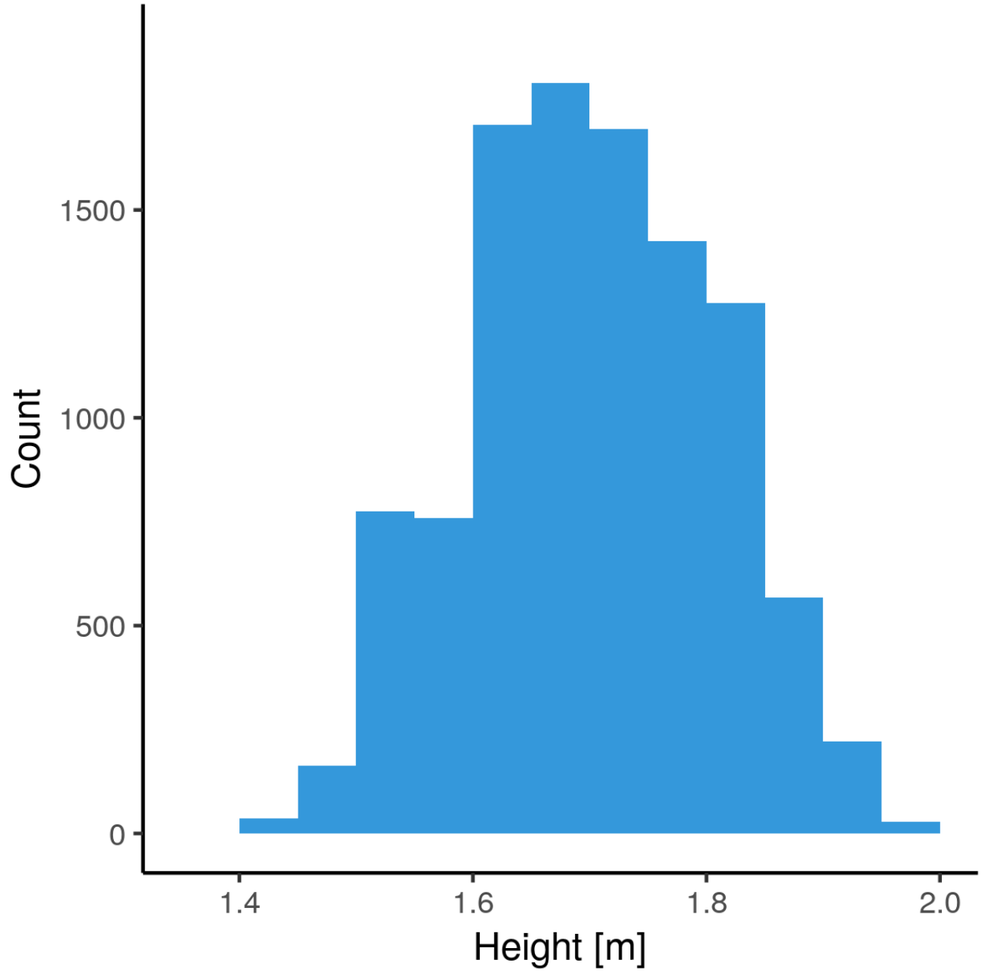
Operations on a single column:

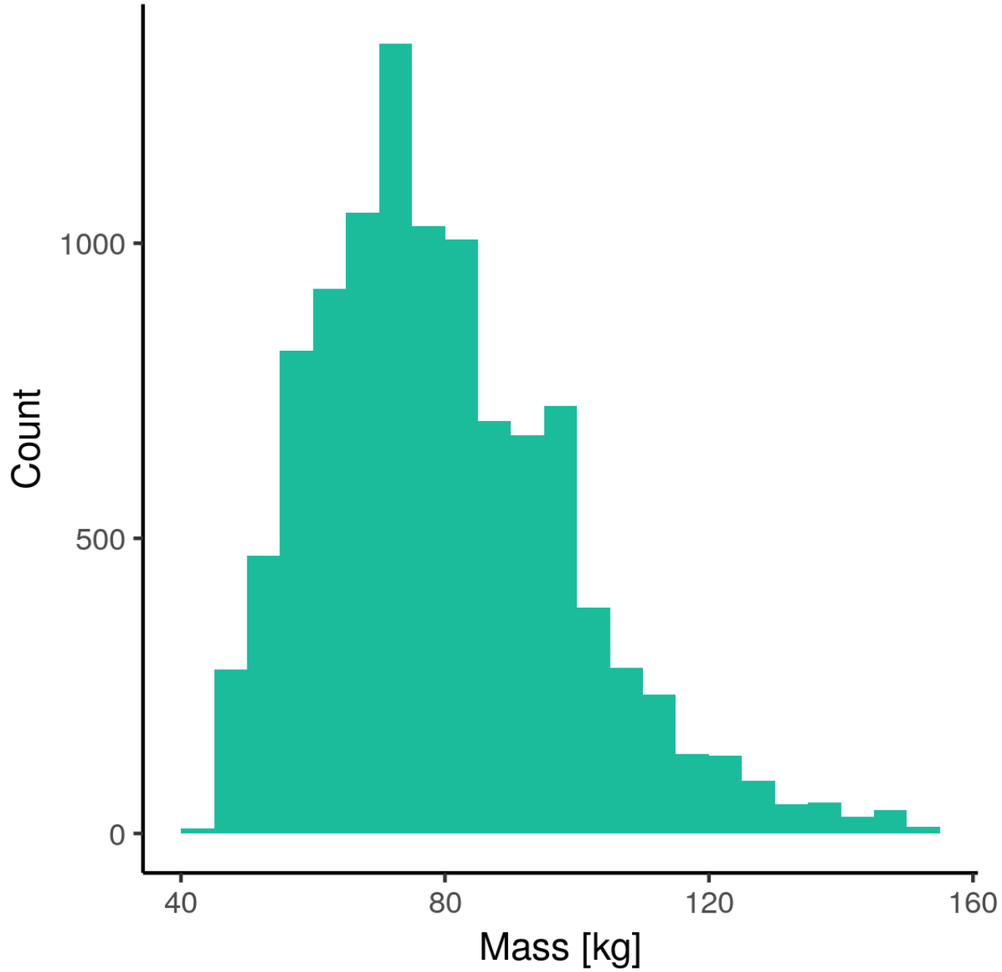
- `log()`
- `sqrt()`
- `pow()`

Operations on two columns:

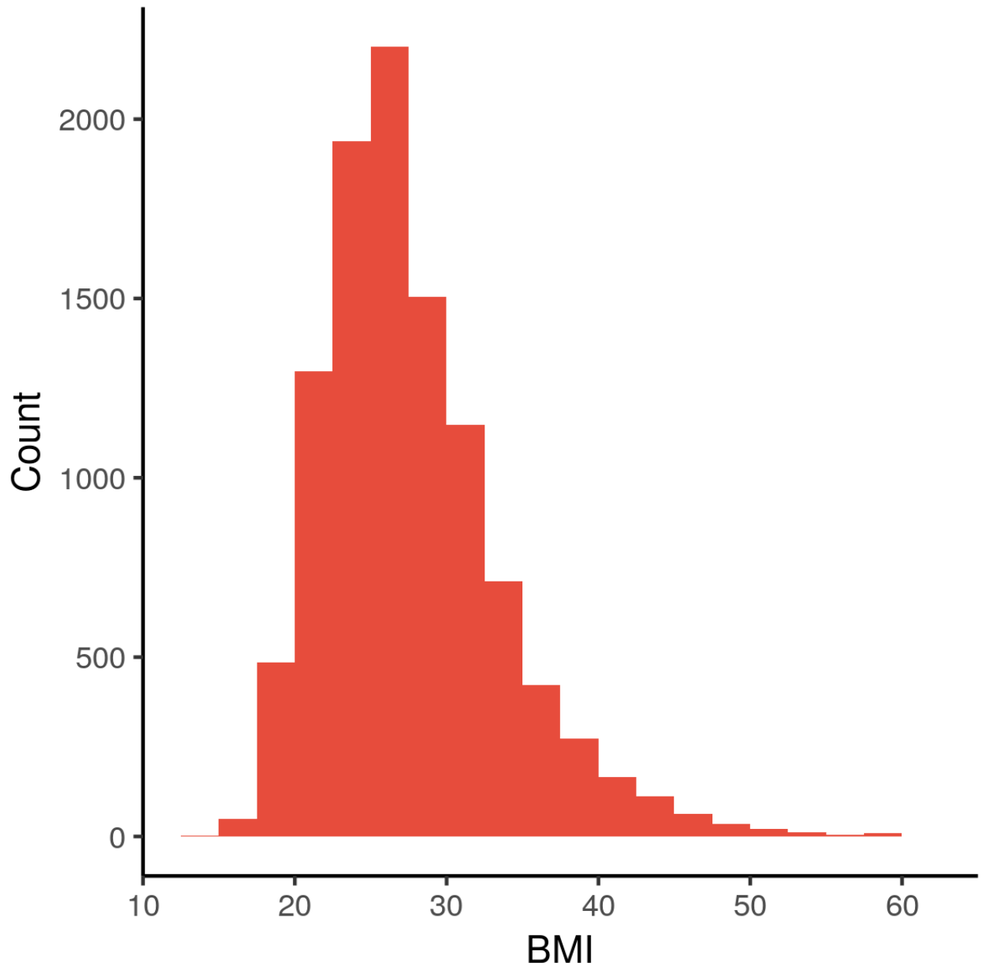
- product
- ratio.

Mass & Height to BMI





Mass & Height to BMI




```
+------+-----+----+
|height| mass| bmi|    bmi = mass / height^2
+------+-----+----+
|  1.52| 77.1|33.2|
|  1.60| 58.1|22.7|
|  1.57|122.0|49.4|
|  1.75| 95.3|31.0|
|  1.80| 99.8|30.7|
|  1.65| 90.7|33.3|
|  1.60| 70.3|27.5|
|  1.78| 81.6|25.8|
|  1.65| 77.1|28.3|
|  1.78|128.0|40.5|
+------+-----+----+
```

Engineering density

```
cars = cars.withColumn('density_line', cars.mass / cars.length)       # Linear density
cars = cars.withColumn('density_quad', cars.mass / cars.length**2)    # Area density
cars = cars.withColumn('density_cube', cars.mass / cars.length**3)    # Volume density
```

```
+------+------+------------+------------+------------+
|  mass|length|density_line|density_quad|density_cube|
+------+------+------------+------------+------------+
|1451.0| 4.775|303.87434554|63.638606397|13.327456837|
|1129.0| 4.623|244.21371403|52.825808790|11.426737787|
|1399.0| 4.547|307.67539036|67.665579583|14.881367843|
+------+------+------------+------------+------------+
```

Let's engineer some features!

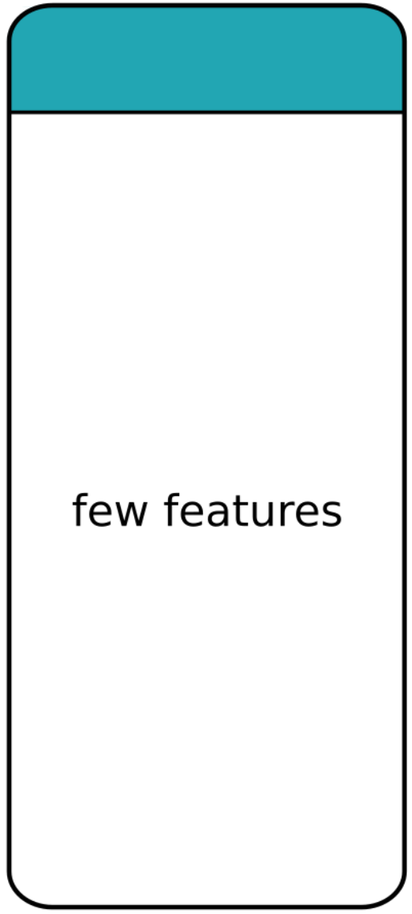
MACHINE LEARNING WITH PYSPARK

Regularization

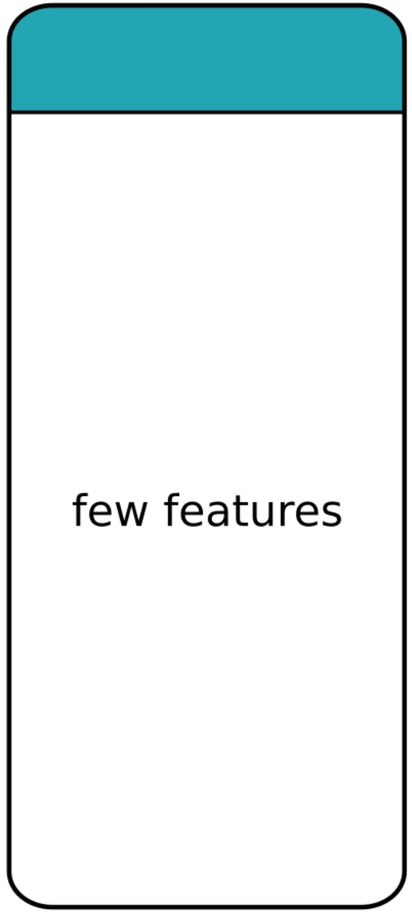
MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

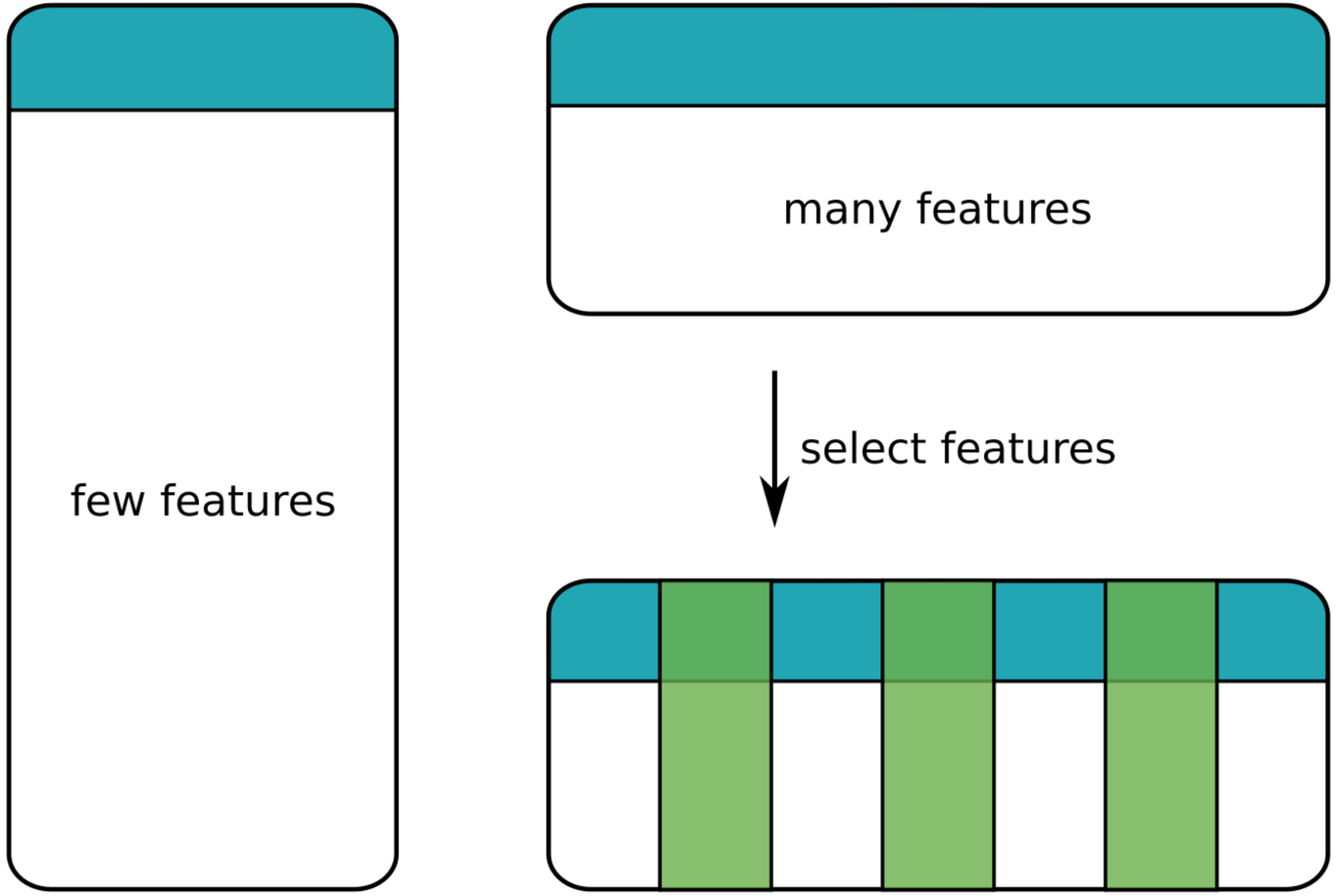
Features: Only a few



Features: Too many



Features: Selected



Loss function (revisited)

Linear regression aims to minimise the MSE.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Loss function with regularization

Linear regression aims to minimise the MSE.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \lambda f(\beta)$$

Add a *regularization* term which depends on coefficients.

Regularization term

An extra *regularization* term is added to the loss function.

The regularization term can be either

- *Lasso* — absolute value of the coefficients
- *Ridge* — square of the coefficients

It's also possible to have a blend of Lasso and Ridge regression.

Strength of regularization determined by parameter λ:

- $\lambda = 0$ — no regularization (standard regression)
- $\lambda = \infty$ — complete regularization (all coefficients zero)

Cars again

```
assembler = VectorAssembler(inputCols=[
    'mass', 'cyl', 'type_dummy', 'density_line', 'density_quad', 'density_cube'
], outputCol='features')
cars = assembler.transform(cars)
```

```
+-----------------------------------------------------------------------+-----------+
|features                                                               |consumption|
+-----------------------------------------------------------------------+-----------+
|[1451.0,6.0,1.0,0.0,0.0,0.0,0.0,303.8743455497,63.63860639785,13.32745683724]|9.05       |
|[1129.0,4.0,0.0,0.0,1.0,0.0,0.0,244.2137140385,52.82580879050,11.42673778726]|6.53       |
|[1399.0,4.0,0.0,0.0,1.0,0.0,0.0,307.6753903672,67.66557958374,14.88136784335]|7.84       |
|[1147.0,4.0,0.0,1.0,0.0,0.0,0.0,264.1031545014,60.81122599620,14.00212433714]|7.84       |
+-----------------------------------------------------------------------+-----------+
```

Cars: Linear regression

Fit a (standard) Linear Regression model to the training data.

```
regression = LinearRegression(labelCol='consumption').fit(cars_train)
```

```
# RMSE on testing data
0.708699086182001
```

Examine the coefficients:

```
regression.coefficients
```

```
DenseVector([-0.012, 0.174, -0.897, -1.445, -0.985, -1.071, -1.335, 0.189, -0.780, 1.160])
```

Cars: Ridge regression

```
# ? = 0.1 | ? = 0 -> Ridge
ridge = LinearRegression(labelCol='consumption', elasticNetParam=0, regParam=0.1)
ridge.fit(cars_train)
```

```
# RMSE
0.724535609745491
```

```
# Ridge coefficients
DenseVector([ 0.001, 0.137, -0.395, -0.822, -0.450, -0.582, -0.806, 0.008,  0.029, 0.001])
# Linear Regression coefficients
DenseVector([-0.012, 0.174, -0.897, -1.445, -0.985, -1.071, -1.335, 0.189, -0.780, 1.160])
```

Cars: Lasso regression

```
# ? = 0.1 | ? = 1 -> Lasso
lasso = LinearRegression(labelCol='consumption', elasticNetParam=1, regParam=0.1)
lasso.fit(cars_train)
```

```
# RMSE
0.771988667026998
```

```
# Lasso coefficients
DenseVector([   0.0,   0.0,    0.0, -0.056,    0.0,    0.0,    0.0, 0.026,    0.0,   0.0])
# Ridge coefficients
DenseVector([ 0.001, 0.137, -0.395, -0.822, -0.450, -0.582, -0.806, 0.008,  0.029, 0.001])
# Linear Regression coefficients
DenseVector([-0.012, 0.174, -0.897, -1.445, -0.985, -1.071, -1.335, 0.189, -0.780, 1.160])
```

Regularization ? simple model

MACHINE LEARNING WITH PYSPARK

Pipeline

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

Leakage?

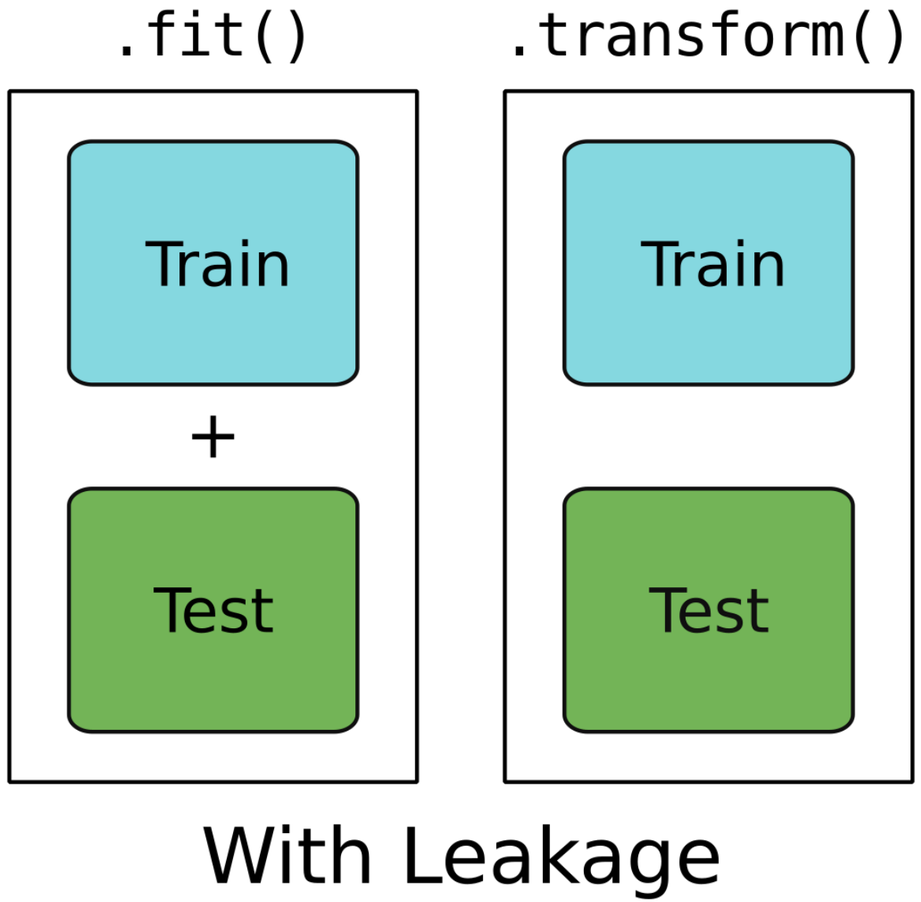
`.fit()`

Only for training data.

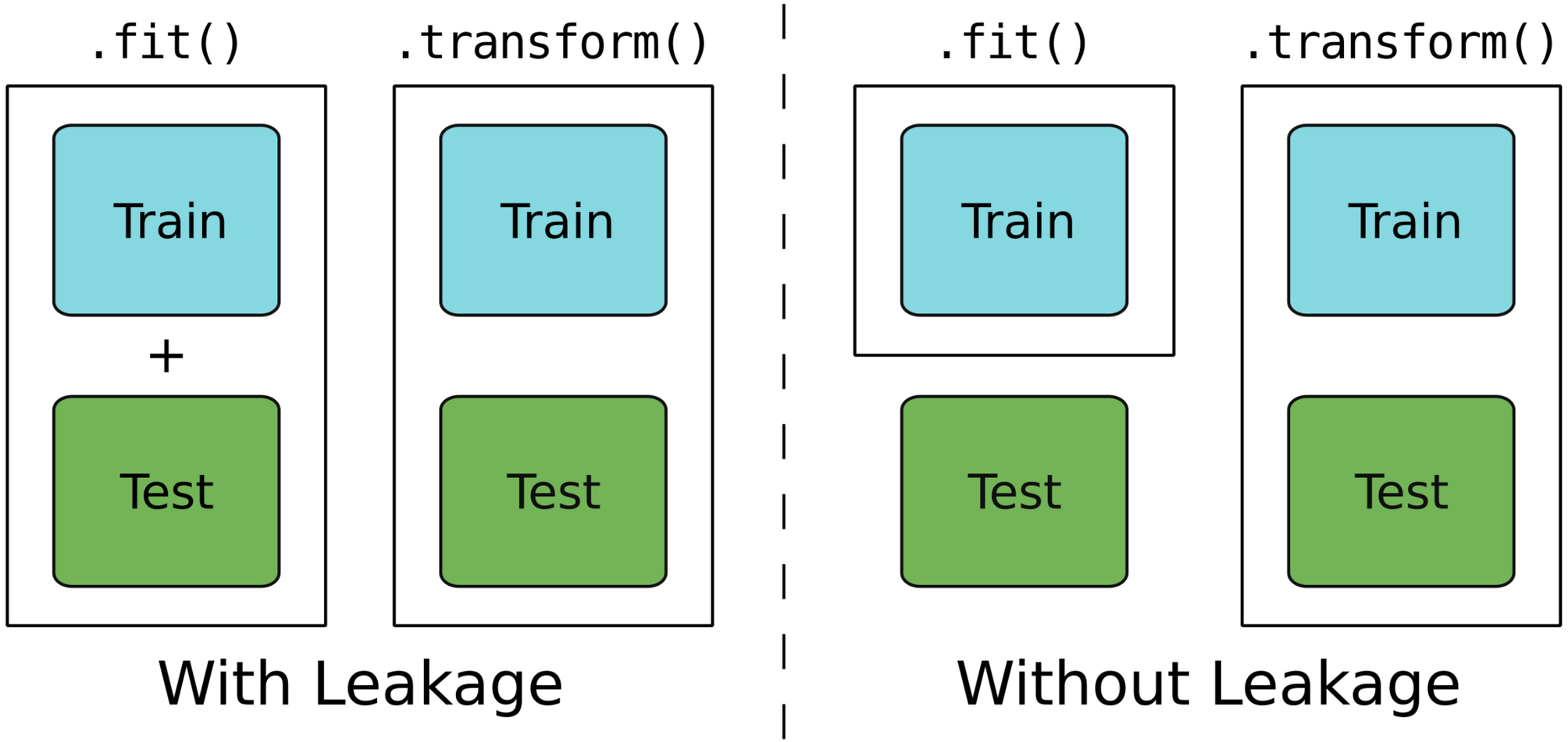
`.transform()`

For testing and training data.

A leaky model

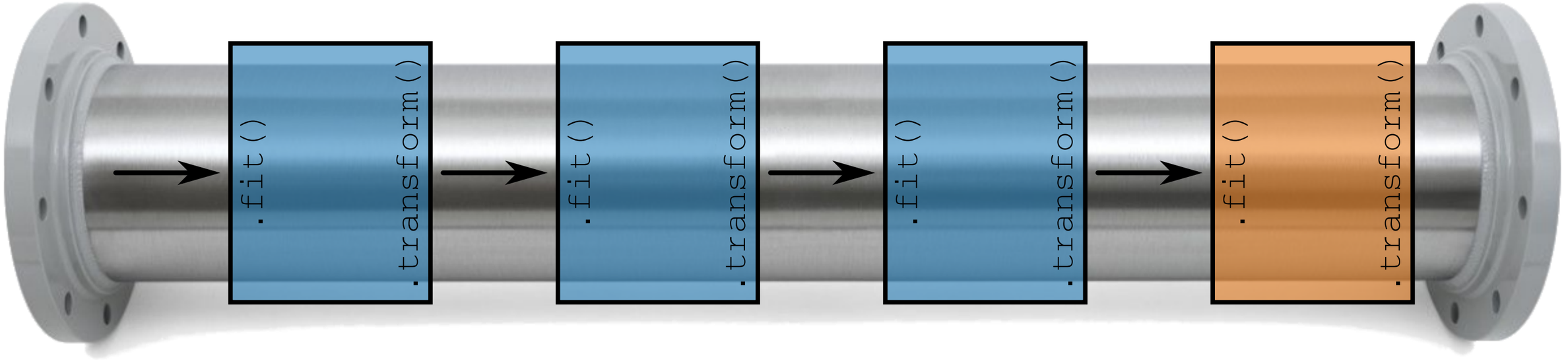


A watertight model



Pipeline

A pipeline consists of a series of operations.



You could apply each operation individually... or you could just apply the pipeline!

Cars model: Steps

```
indexer = StringIndexer(inputCol='type', outputCol='type_idx')

onehot = OneHotEncoderEstimator(inputCols=['type_idx'], outputCols=['type_dummy'])

assemble = VectorAssembler(
  inputCols=['mass', 'cyl', 'type_dummy'],
  outputCol='features'
)

regression = LinearRegression(labelCol='consumption')
```

Cars model: Applying steps

Training data

```
indexer = indexer.fit(cars_train)
cars_train = indexer.transform(cars_train)
```

```
onehot = onehot.fit(cars_train)
cars_train = onehot.transform(cars_train)
```

```
cars_train = assemble.transform(cars_train)
```

```
# Fit model to training data
regression = regression.fit(cars_train)
```

Testing data

```
#
cars_test  = indexer.transform(cars_test)
```

```
#
cars_test  = onehot.transform(cars_test)
```

```
cars_test  = assemble.transform(cars_test)
```

```
# Make predictions on testing data
predictions = regression.transform(cars_test)
```

Cars model: Pipeline

Combine steps into a pipeline.

```
from pyspark.ml import Pipeline

pipeline = Pipeline(stages=[indexer, onehot, assemble, regression])
```

Training data

```
pipeline = pipeline.fit(cars_train)
```

Testing data

```
predictions = pipeline.transform(cars_test)
```

Cars model: Stages

Access individual stages using the `.stages` attribute.

```
# The LinearRegression object (fourth stage -> index 3)
pipeline.stages[3]


print(pipeline.stages[3].intercept)
```

```
4.19433571782916
```

```
print(pipeline.stages[3].coefficients)
```

```
DenseVector([0.0028, 0.2705, -1.1813, -1.3696, -1.1751, -1.1553, -1.8894])
```

Pipelines streamline workflow!

MACHINE LEARNING WITH PYSPARK

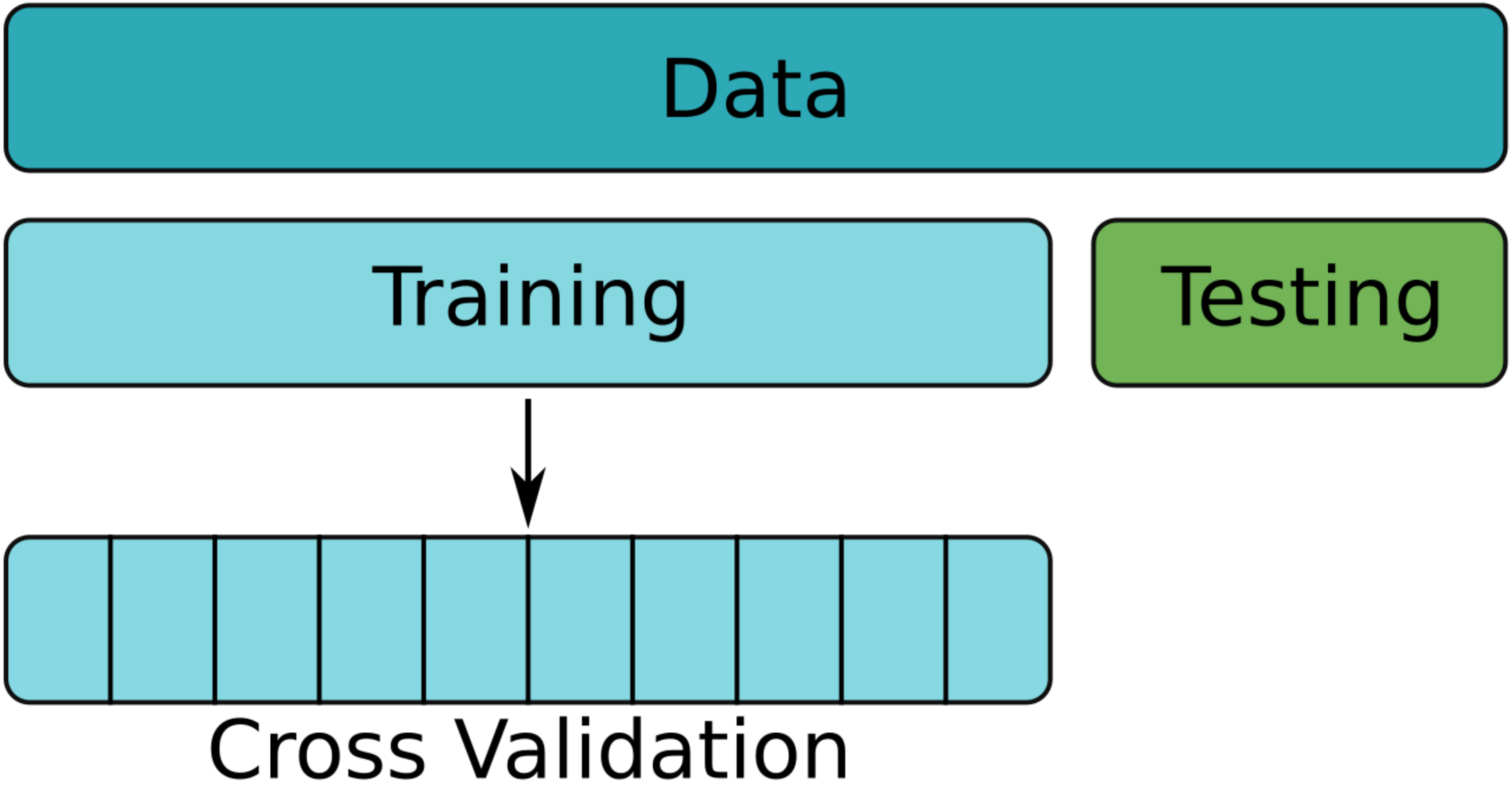
Cross-Validation

MACHINE LEARNING WITH PYSPARK

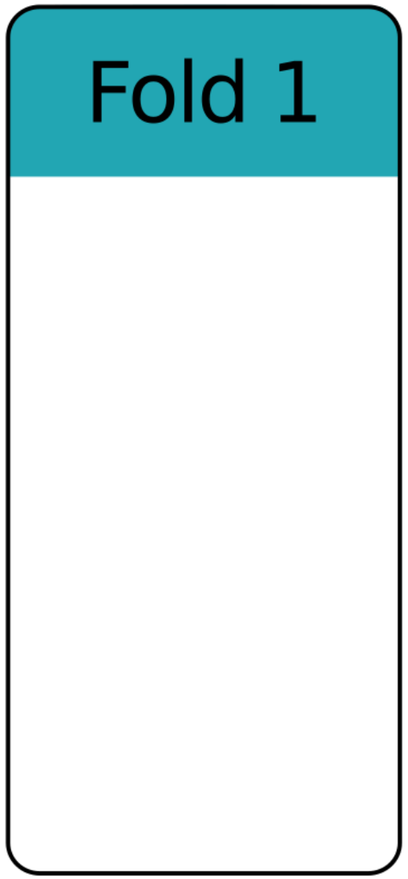
Andrew Collier
Data Scientist, Exegetic Analytics

Data

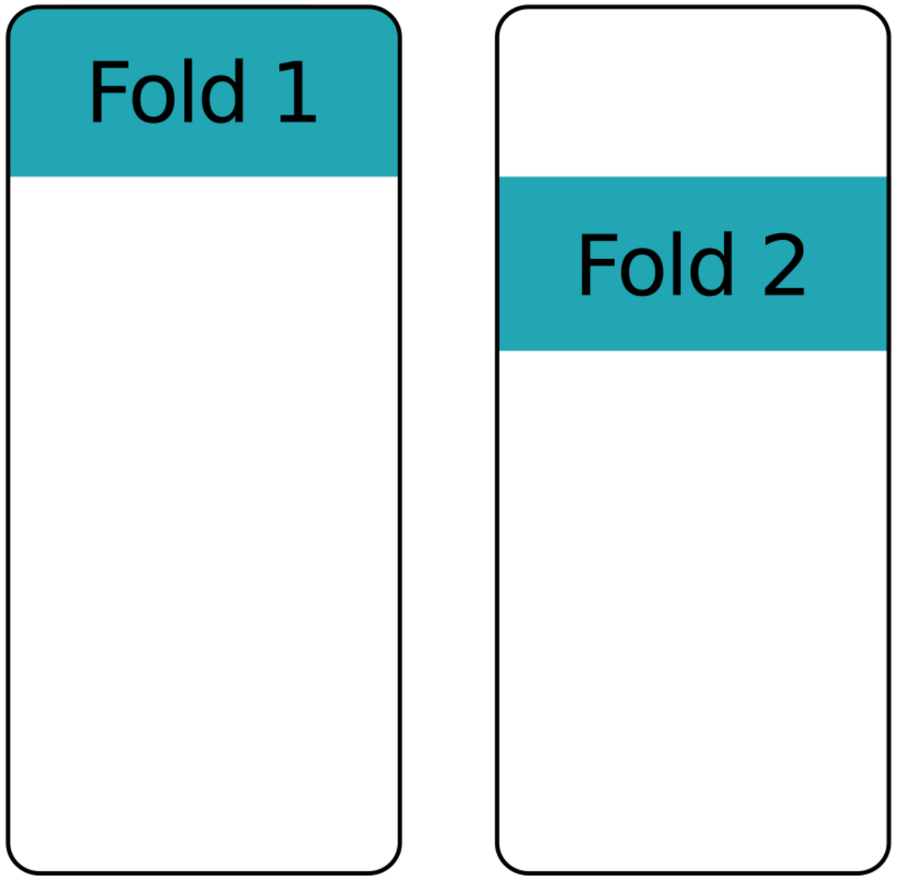

Data
Training
Testing


Data
Training
Testing
Cross Validation

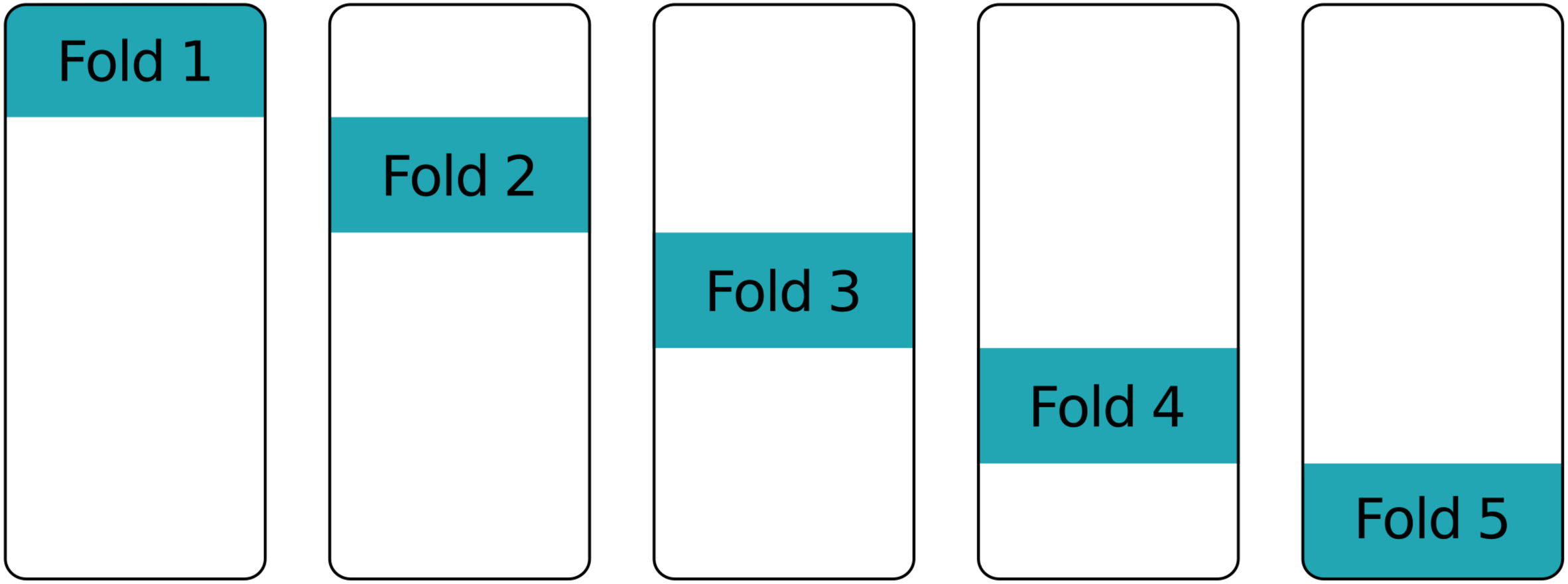
Fold upon fold - first fold



Fold upon fold - second fold



Fold upon fold - other folds



Cars revisited

```
cars.select('mass', 'cyl', 'consumption').show(5)
```

```
+------+---+-----------+
|  mass|cyl|consumption|
+------+---+-----------+
|1451.0|  6|       9.05|
|1129.0|  4|       6.53|
|1399.0|  4|       7.84|
|1147.0|  4|       7.84|
|1111.0|  4|       9.05|
+------+---+-----------+
```

Estimator and evaluator

An object to build the model. This can be a pipeline.

```
regression = LinearRegression(labelCol='consumption')
```

An object to evaluate model performance.

```
evaluator = RegressionEvaluator(labelCol='consumption')
```

Grid and cross-validator

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
```

A grid of parameter values (empty for the moment).

```
params = ParamGridBuilder().build()
```

The cross-validation object.

```
cv = CrossValidator(estimator=regression,
                    estimatorParamMaps=params,
                    evaluator=evaluator,
                    numFolds=10, seed=13)
```

Cross-validators need training too

Apply cross-validation to the training data.

```
cv = cv.fit(cars_train)
```

What's the average RMSE across the folds?

```
cv.avgMetrics
```

```
[0.800663722151572]
```

Cross-validators act like models

Make predictions on the original testing data.

```
evaluator.evaluate(cv.transform(cars_test))
```

```
# RMSE on testing data
0.745974203928479
```

Much smaller than the cross-validated RMSE.

```
# RMSE from cross-validation
0.800663722151572
```

A simple train-test split would have given an overly optimistic view on model performance.

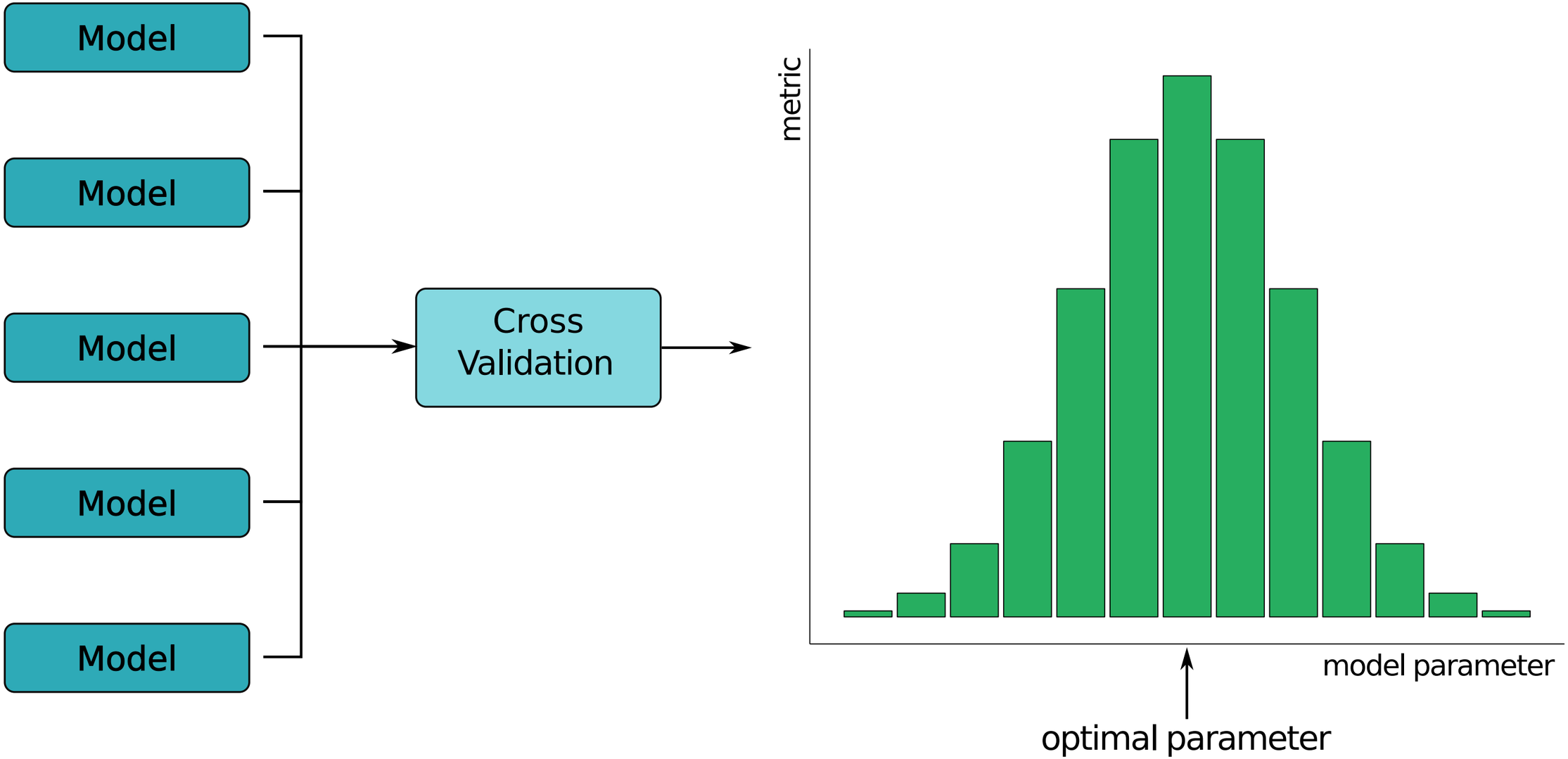
Cross-validate all the models!

MACHINE LEARNING WITH PYSPARK

Grid Search

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics


Model
Model
Model
Model
Model
Cross Validation
metric
model parameter
optimal parameter

Cars revisited (again)

```
cars.select('mass', 'cyl', 'consumption').show(5)
```

```
+------+---+-----------+
|  mass|cyl|consumption|
+------+---+-----------+
|1451.0|  6|       9.05|
|1129.0|  4|       6.53|
|1399.0|  4|       7.84|
|1147.0|  4|       7.84|
|1111.0|  4|       9.05|
+------+---+-----------+
```

Fuel consumption with intercept

Linear regression *with* an intercept. Fit to training data.

```
regression = LinearRegression(labelCol='consumption', fitIntercept=True)
regression = regression.fit(cars_train)
```

Calculate the RMSE on the testing data.

```
evaluator.evaluate(regression.transform(cars_test))
```

```
# RMSE for model with an intercept
0.745974203928479
```

Fuel consumption without intercept

Linear regression *without* an intercept. Fit to training data.

```
regression = LinearRegression(labelCol='consumption', fitIntercept=False)
regression = regression.fit(cars_train)
```

Calculate the RMSE on the testing data.

```
# RMSE for model without an intercept (second model)
0.852819012439
```

```
# RMSE for model with an intercept    (first model)
0.745974203928
```

Parameter grid

```
from pyspark.ml.tuning import ParamGridBuilder

# Create a parameter grid builder
params = ParamGridBuilder()

# Add grid points
params = params.addGrid(regression.fitIntercept, [True, False])

# Construct the grid
params = params.build()


# How many models?
print('Number of models to be tested: ', len(params))
```

```
Number of models to be tested:  2
```

Grid search with cross-validation

Create a cross-validator and fit to the training data.

```
cv = CrossValidator(estimator=regression,
                    estimatorParamMaps=params,
                    evaluator=evaluator)
cv = cv.setNumFolds(10).setSeed(13).fit(cars_train)
```

What's the cross-validated RMSE for each model?

```
cv.avgMetrics
```

```
[0.800663722151, 0.907977823182]
```

The best model & parameters

```
# Access the best model
cv.bestModel
```

Or just use the cross-validator object.

```
predictions = cv.transform(cars_test)
```

Retrieve the best parameter.

```
cv.bestModel.explainParam('fitIntercept')
```

```
'fitIntercept: whether to fit an intercept term (default: True, current: True)'
```

A more complicated grid

```
params = ParamGridBuilder() \
            .addGrid(regression.fitIntercept, [True, False]) \
            .addGrid(regression.regParam, [0.001, 0.01, 0.1, 1, 10]) \
            .addGrid(regression.elasticNetParam, [0, 0.25, 0.5, 0.75, 1]) \
            .build()
```

How many models now?

```
print ('Number of models to be tested: ', len(params))
```

```
Number of models to be tested:  50
```

Find the best parameters!

MACHINE LEARNING WITH PYSPARK

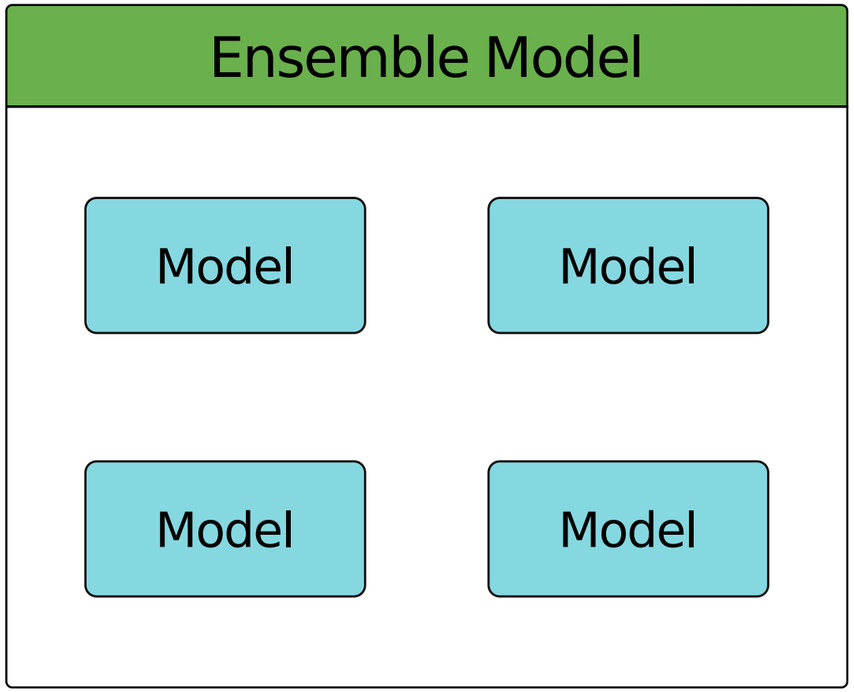
Ensemble

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

What's an ensemble?

It's a collection of models.



Wisdom of the Crowd — collective opinion of a group better than that of a single expert.

Ensemble diversity

> **Diversity** and **independence** are important because the best collective decisions are the product of disagreement and contest, not consensus or compromise.
>
> ? James Surowiecki, *The Wisdom of Crowds*

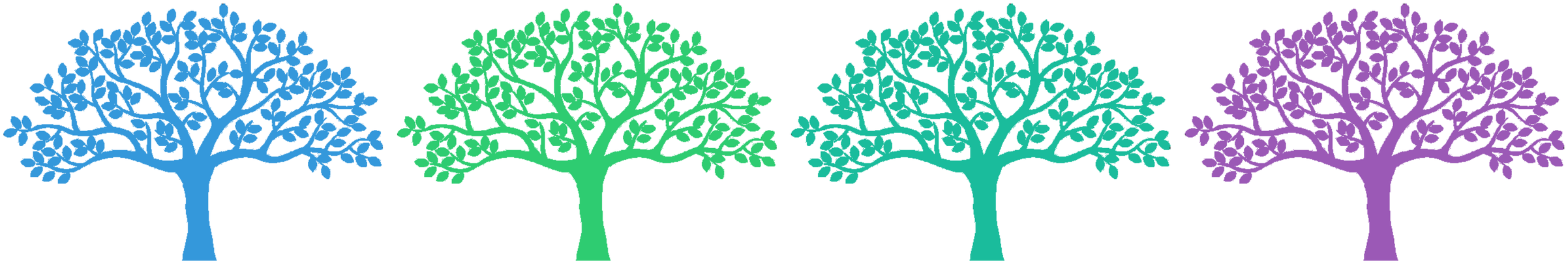
Random Forest

Random Forest — an ensemble of Decision Trees

Creating model diversity:

- each tree trained on *random subset* of data
- *random subset* of features used for splitting at each node

No two trees in the forest should be the same.



Create a forest of trees

Returning to cars data: manufactured in USA (`0.0`) or not (`1.0`).

Create Random Forest classifier.

```
from pyspark.ml.classification import RandomForestClassifier

forest = RandomForestClassifier(numTrees=5)
```

Fit to the training data.

```
forest = forest.fit(cars_train)
```

Seeing the trees

How to access trees within forest?

```
forest.trees
```

```
[DecisionTreeClassificationModel (uid=dtc_aa66702a4ce9) of depth 5 with 17 nodes,
 DecisionTreeClassificationModel (uid=dtc_99f7efedafe9) of depth 5 with 31 nodes,
 DecisionTreeClassificationModel (uid=dtc_9306e4a5fa1d) of depth 5 with 21 nodes,
 DecisionTreeClassificationModel (uid=dtc_d643bd48a8dd) of depth 5 with 23 nodes,
 DecisionTreeClassificationModel (uid=dtc_a2d5abd67969) of depth 5 with 27 nodes]
```

These can each be used to make individual predictions.

Predictions from individual trees

What predictions are generated by each tree?

```
+------+------+------+------+------+-----+
|tree 0|tree 1|tree 2|tree 3|tree 4|label|
+------+------+------+------+------+-----+
|   0.0|   0.0|   0.0|   0.0|   0.0|  0.0| <- perfect agreement
|   1.0|   1.0|   0.0|   1.0|   0.0|  0.0|
|   0.0|   0.0|   0.0|   1.0|   1.0|  1.0|
|   0.0|   0.0|   0.0|   1.0|   0.0|  0.0|
|   0.0|   1.0|   1.0|   1.0|   0.0|  1.0|
|   1.0|   1.0|   0.0|   1.0|   1.0|  1.0|
|   1.0|   1.0|   1.0|   1.0|   1.0|  1.0| <- perfect agreement
+------+------+------+------+------+-----+
```

Consensus predictions

Use the `.transform()` method to generate consensus predictions.

```
+-----+----------------------------------------+----------+
|label|probability                             |prediction|
+-----+----------------------------------------+----------+
|0.0  |[0.8,0.2]                               |0.0       |
|0.0  |[0.4,0.6]                               |1.0       |
|1.0  |[0.5333333333333333,0.4666666666666666] |0.0       |
|0.0  |[0.7177777777777778,0.2822222222222226]|0.0       |
|1.0  |[0.3936825396825397,0.606031746031746]  |1.0       |
|1.0  |[0.17660818713450294,0.823391812865497] |1.0       |
|1.0  |[0.05396825396825396,0.946031746031746]|1.0       |
+-----+----------------------------------------+----------+
```

Feature importances

The model uses these features: `cyl`, `size`, `mass`, `length`, `rpm` and `consumption`.

Which of these is most or least important?

```
forest.featureImportances
```

```
SparseVector(6, {0: 0.0205, 1: 0.2701, 2: 0.108, 3: 0.1895, 4: 0.2939, 5: 0.1181})
```

Looks like:

- `rpm` is most important
- `cyl` is least important.

Gradient-Boosted Trees

Iterative boosting algorithm:

1. Build a Decision Tree and add to ensemble.
2. Predict label for each training instance using ensemble.
3. Compare predictions with known labels.
4. Emphasize training instances with incorrect predictions.
5. Return to 1.

Model improves on each iteration.

Boosting trees

Create a Gradient-Boosted Tree classifier.

```
from pyspark.ml.classification import GBTClassifier

gbt = GBTClassifier(maxIter=10)
```

Fit to the training data.

```
gbt = gbt.fit(cars_train)
```

Comparing trees

Let's compare the three types of tree models on testing data.

```
# AUC for Decision Tree
0.5875

# AUC for Random Forest
0.65

# AUC for Gradient-Boosted Tree
0.65
```

Both of the ensemble methods perform better than a plain Decision Tree.

Ensemble all of the models!

MACHINE LEARNING WITH PYSPARK

Closing thoughts

MACHINE LEARNING WITH PYSPARK

Andrew Collier
Data Scientist, Exegetic Analytics

Things you've learned

- Load & prepare data
- Classifiers
  - Decision Tree
  - Logistic Regression
- Regression
  - Linear Regression
  - Penalized Regression
- Pipelines
- Cross-validation & grid search
- Ensembles

Learning more

Documentation at https://spark.apache.org/docs/latest/.

Congratulations!

MACHINE LEARNING WITH PYSPARK