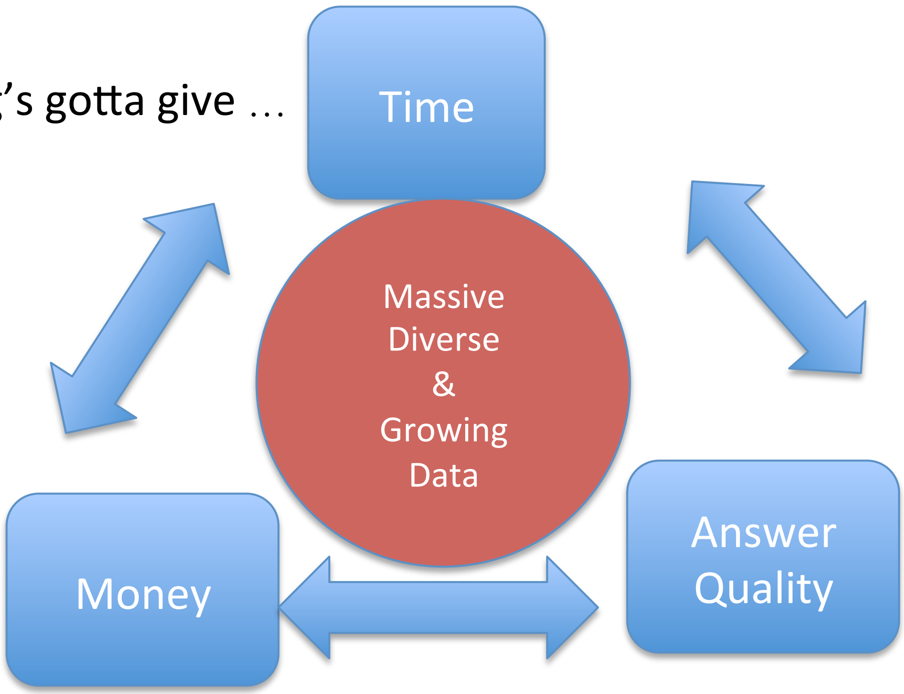
Introduction to Spark ML

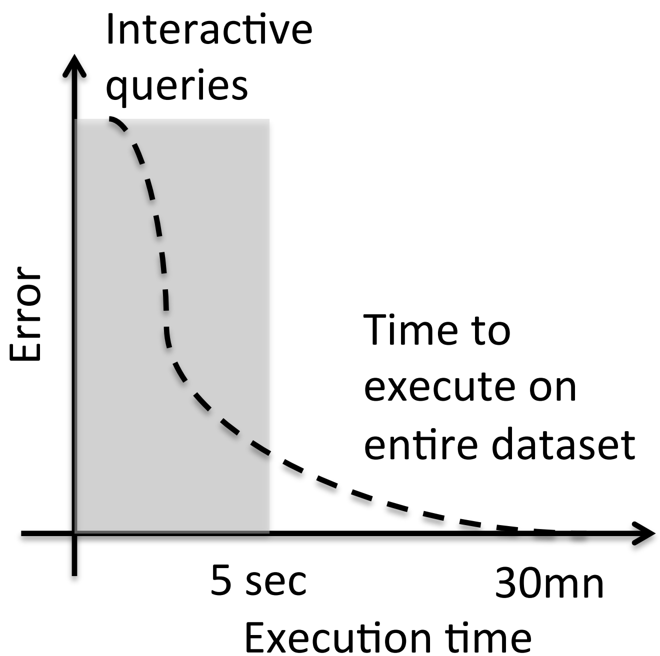
Adam S.Z Belloum
Software and Network engineering group
University of Amsterdam

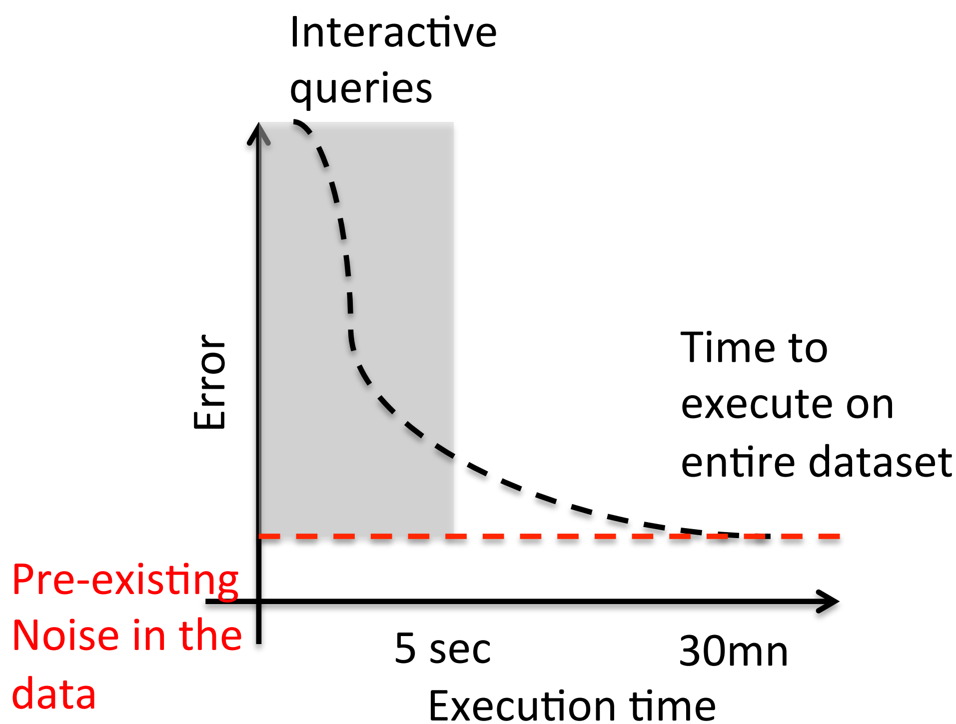
AmpLab view on BigData

Something's gotta give …

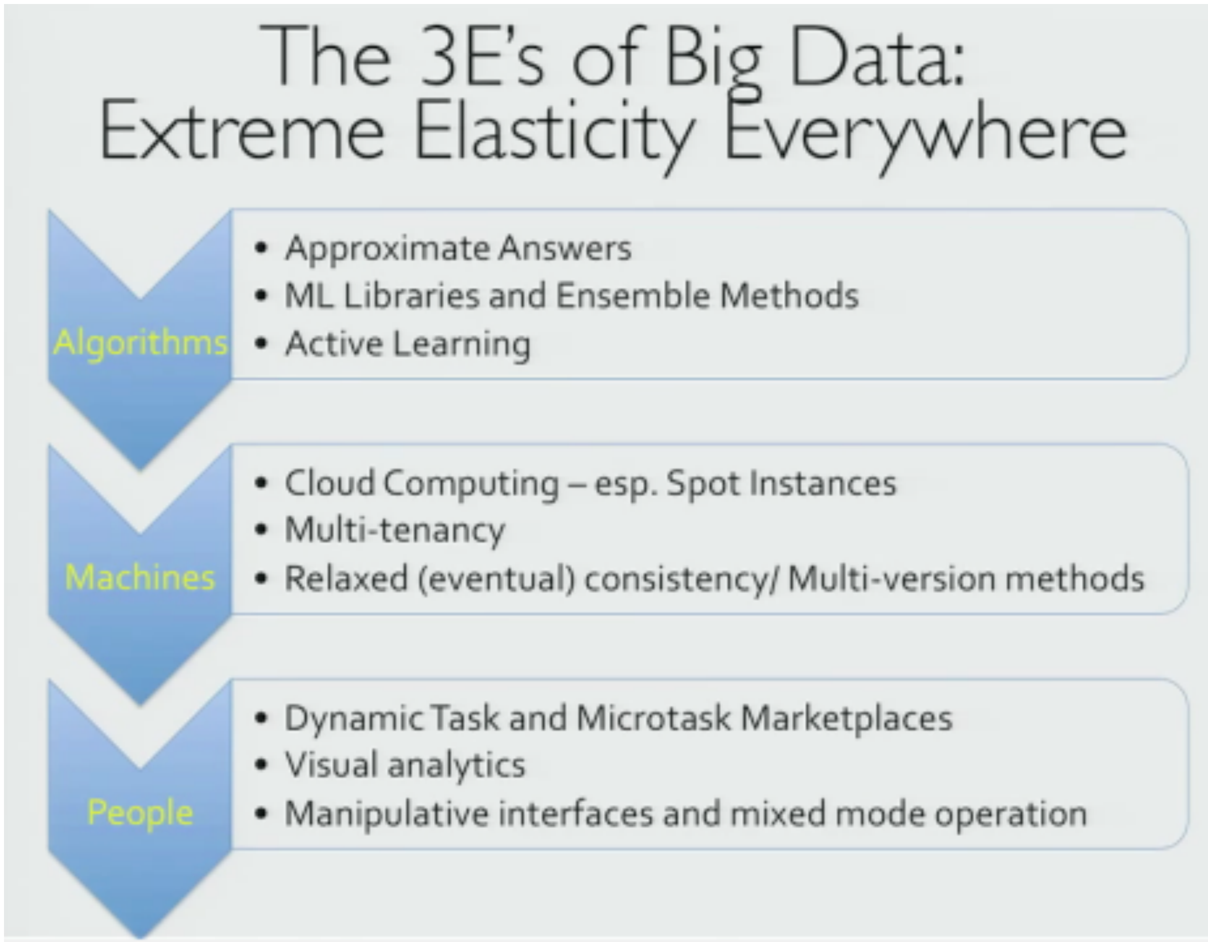


AmpLab view on BigData

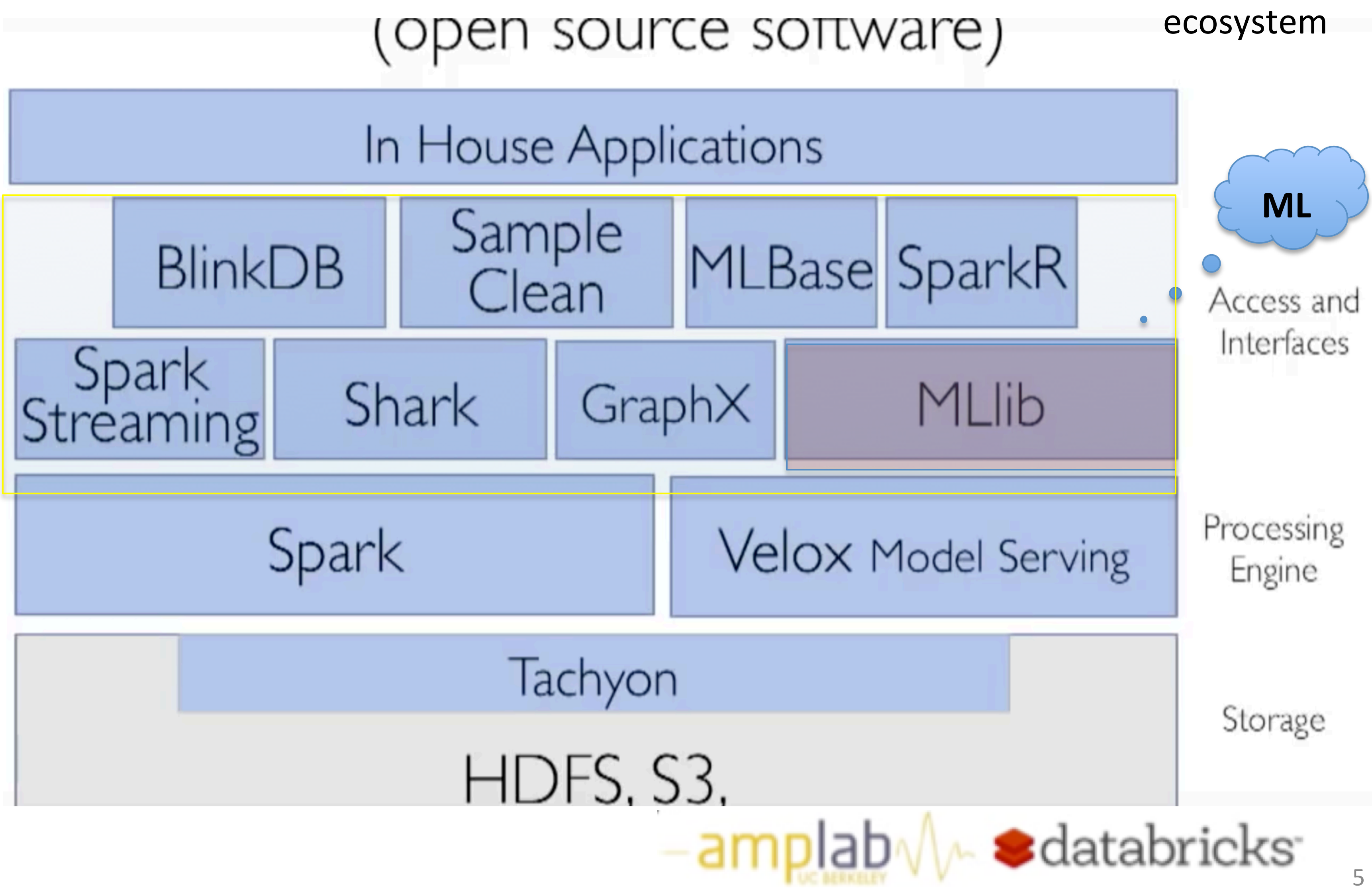




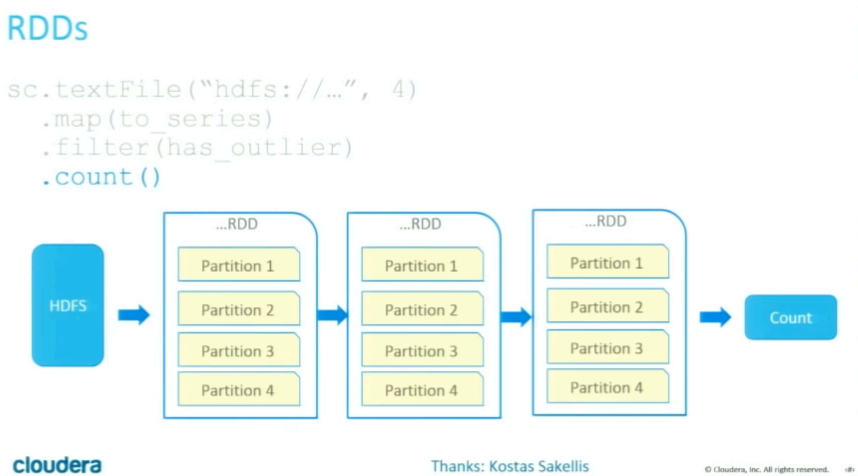
AMP Key resource

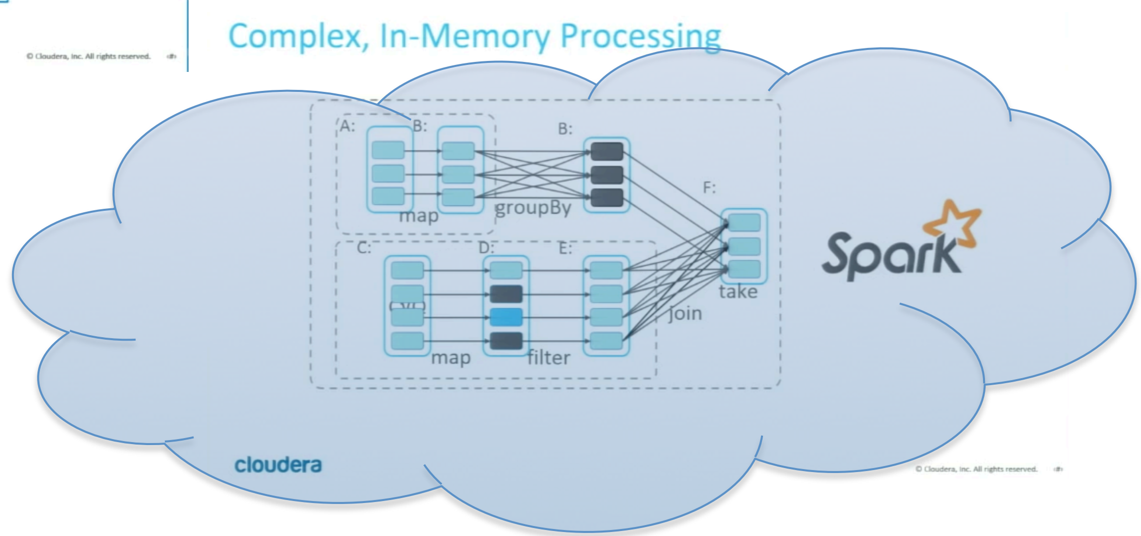


Berkeley Data Analytics Stack



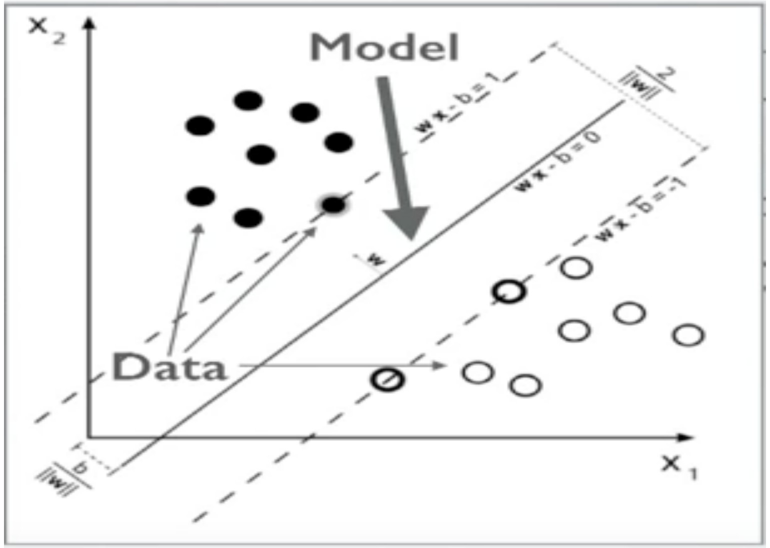
Conceptually, how Spark works &
What really happens inside Spark





Modelling Lifecycle

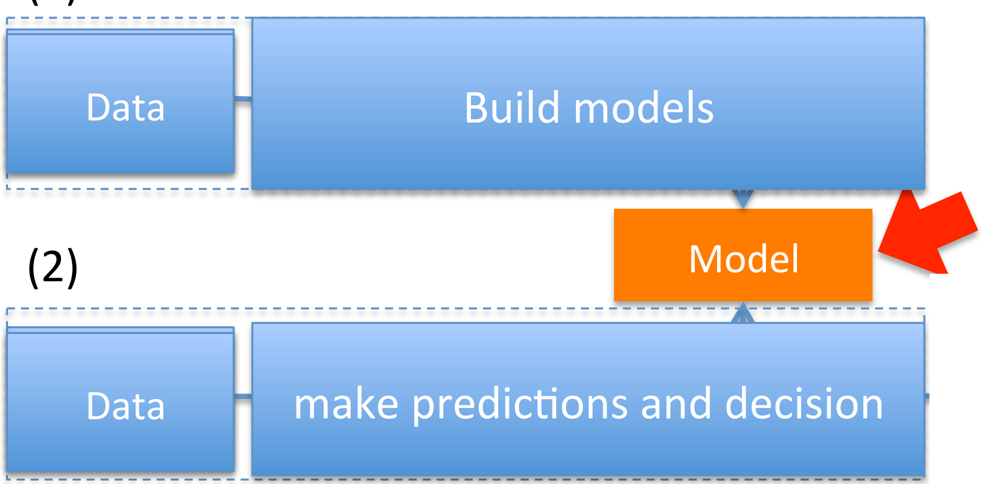
- "ML is a scientific discipline that deals with the construction and study of algorithms that can **lean form data**. Such Algorithms operate:
  1. by building a **model** based on inputs
  2. and using that make **predictions** and **decision** rather that following explicitly programmed instructions "

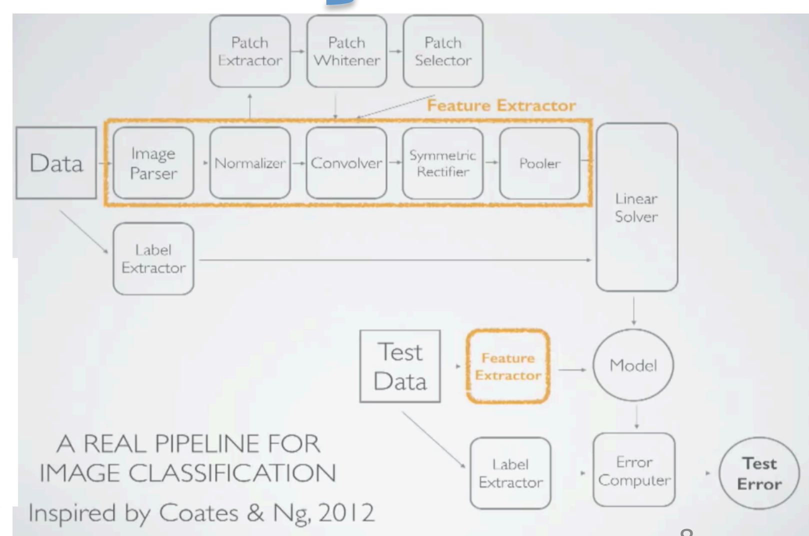


ML Problems

- Real data often not Real number
- Real data not well-behaved according to algorithms
  - **Features** need to be engineered (**extracted**)
  - **Transformations** need to be applied
  - Hyper parameters need to be tuned

ML Pipeline (1)

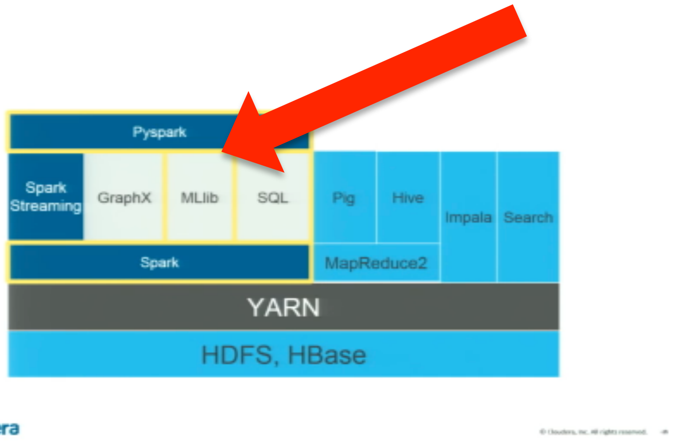




Source: Evan Sparks from AMPLab, Amp camp 5

Spark ML

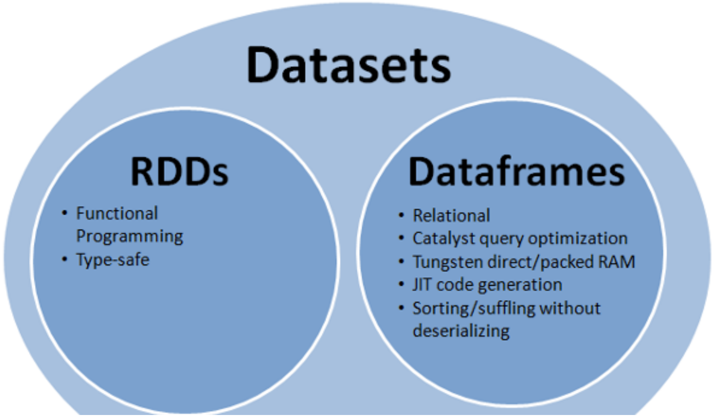
- **Basic statistics:** summaries, correlation, sampling, testing, …
- **Classification and regression**: linear models , trees, ensembles, …
- **Clustering**: k-mean, Gaussian mixture models, …
- **Dimensionality reduction**: PCA, SVD
- Feature **extraction** and **transformation**
- **Optimization**: gradient descent, and L-BFGS



MLlib to ML

- Proposed in 2014 & included in Spark in 2015
- High-level and more flexible
- Use processing ideas from scikit-learn

- Use DataFrames (from R and Pandas) instead of RDD used in MLlib



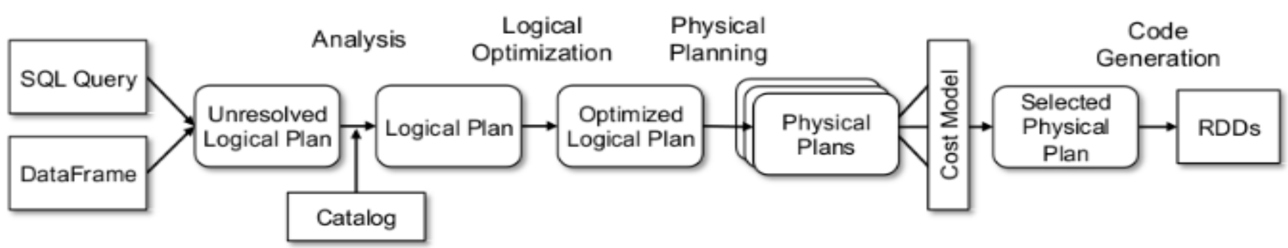
History of Spark API

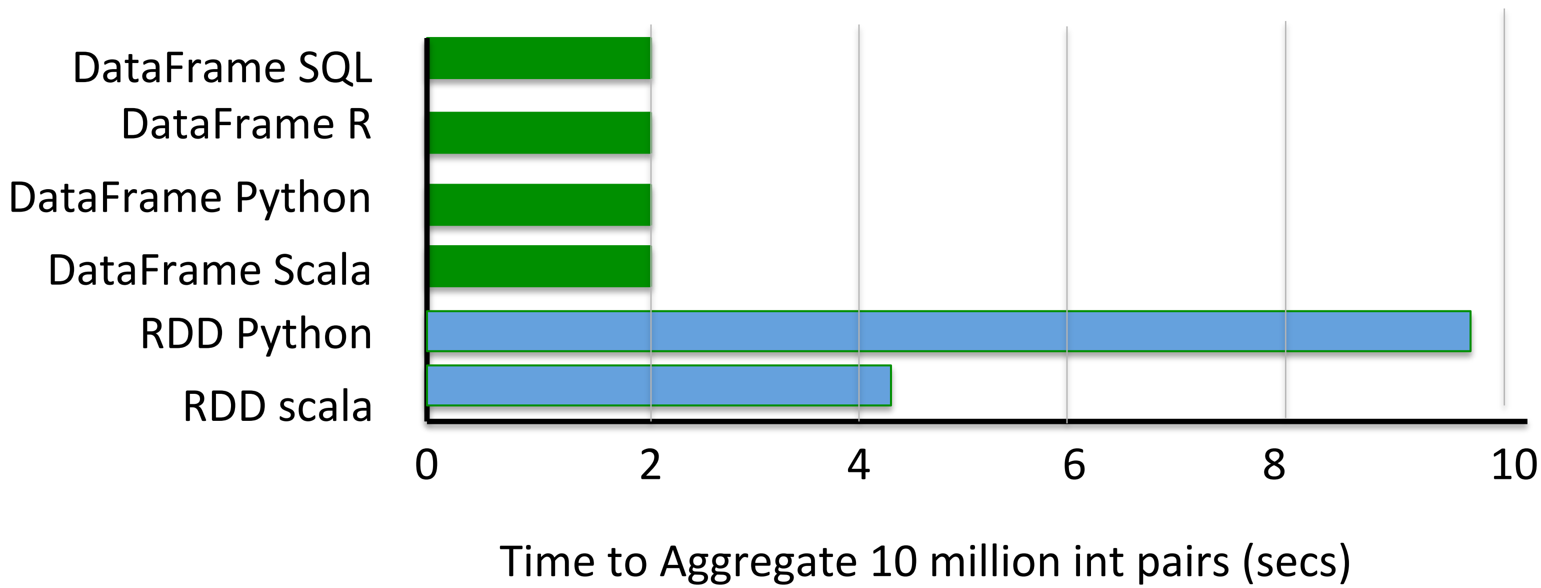
RDD (2011)	DataFrame (2013)	DataSet (2015)
Distribute collection of JVM objects	Distribute collection of Row objects	Internally rows, externally JVM objects
Functional Operators (map, filter, etc.)	Expression-based operations and UDFs	Almost the "Best of both worlds": type safe + fast
	Logical plans and optimizer	But slower than DF Not as good for interactive analysis, especially Python
	Fast/efficient internal representations	

databricks

Catalyst optimizer

- Typical DB optimizers across SQL and DF
  - Extensibility via optimization Rule written in scala
  - Open source optimizer development
  - Code generation for inner loops, iterator removal
- Extensible data sources: CSV, Avro, Parquet, JDBC, …
  - via tableScan (all cols), PrunedScan (project), FiltredPrunedScan (push advisory selection and projects) catalystScan (push advisory full catalyst expression trees)
- Extensible (user Defined) Types



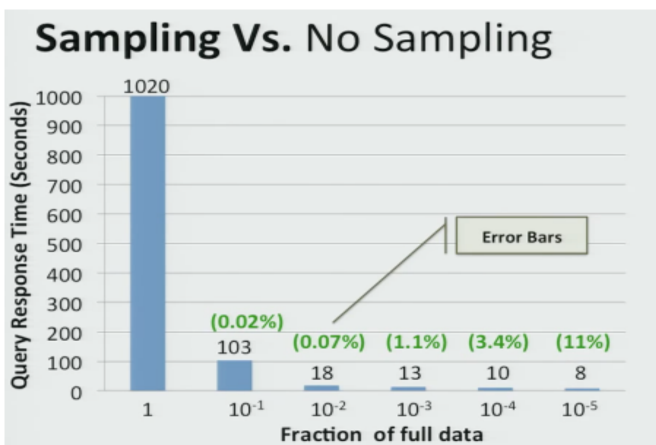

DataFrame SQL
DataFrame R
DataFrame Python
DataFrame Scala
RDD Python
RDD scala
0
2
4
6
8
10
Time to Aggregate 10 million int pairs (secs)

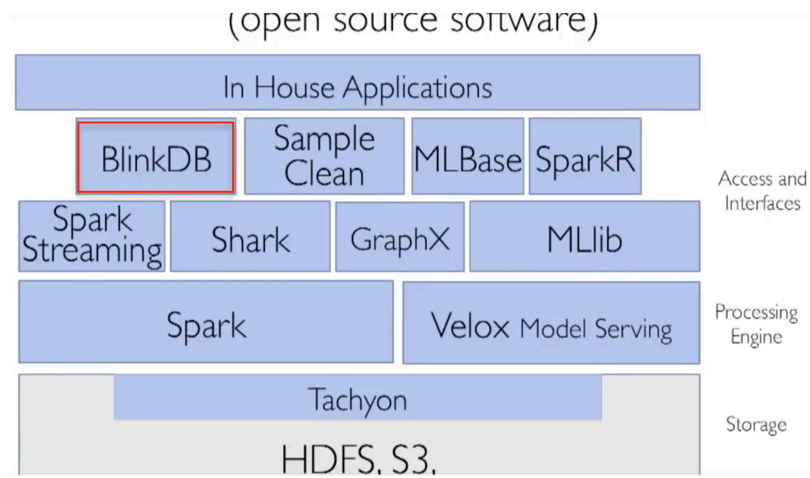
Approximation

- DBAS user Approximation in two main ways:
- BlinkDb
  - Run queries on a sample of the data
  - Return answers and confidence intervals
  - Can adjust time vs confidence
- Sample Clean
  - Clean sample of the data rather than whole date set
    - Run query on the Clean sample (get error bars) OR
    - Run query on dirty data and correct the answers

BlinkDB

- A data analysis (warehouse) systems that …
  - Build on Shark and Spark
  - Returns fast, approximate answers with error bars by executing queries on a small sample of data
  - Trading precision for speed





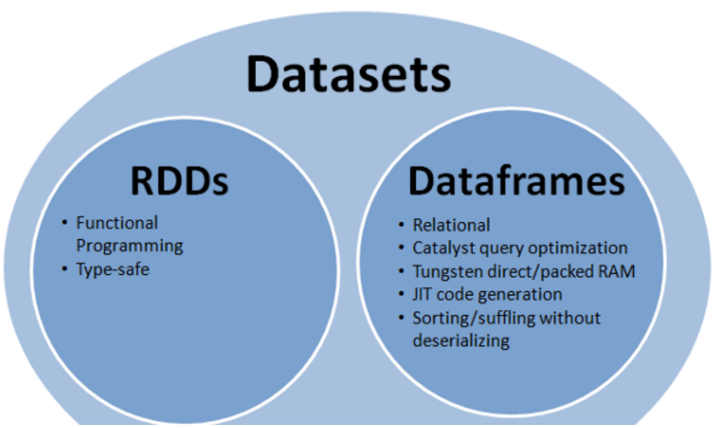
RDD, DataFrame, and DataSets

- RDD contain anything

  ➔ VERY flexible (could be counter productive)

  – Nested objects (can slow the execution)
    - memory management for creating these objects and Garbage collection
    - solution flatten out the data structure looks like going back to table structure (then why not let spark do this by defining schema)

  – From Python process to JVM: open a pipe between Process Python and the JVM

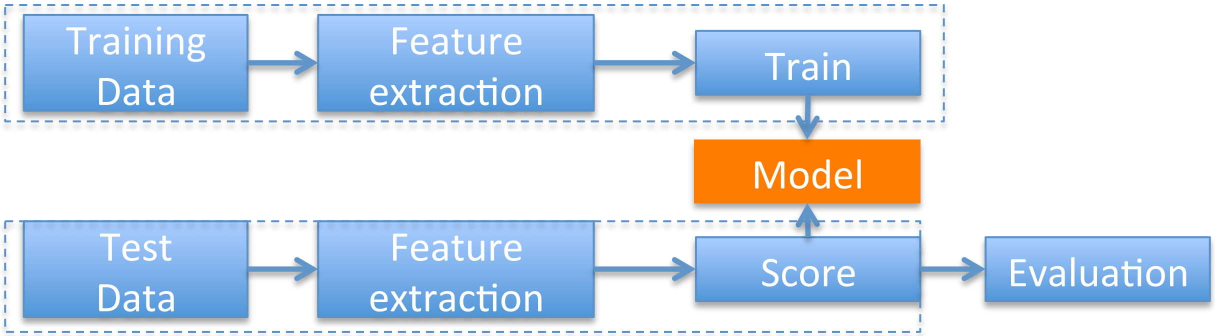


Main concepts

Part of the ML Spark API

- **DataFrame**: flexible data type from Spark SQL allowing parallelism

- **Transformer**: algorithm which transform one DataFrame to another
- **Estimator**: algorithm which is fitted on the DataFrame returning a model
- **Parameters**: uniform structures for Estimators and transformers
- **Pipeline**: chain of transformers and estimators



DataFrame

System concept Spark API

- A distributed collection of rows organized into named columns
  - Similar to tables in a RDB (R and Pandas)
  - Created from file, regular RDD, or other sources
  - Supports a variety of data types: vectors, text, images, and structured data
  - Columns can be named using names as "features" and "Label"

Transformer

Abstraction of the Spark API

A Transformer is an **abstraction** that includes

- **Feature transformers**: tokenisation, hashing, normalisation
- **Learned models**: result form estimation, eg. Outputting prediction

Implements the method transform(), which converts one DataFrame into another

Estimator

Abstraction of the Spark API

An Estimator abstracts the concept of

- a learning algorithm or any algorithm that fits or trains on data

Implements the method **fit()**, which:

- takes a DataFrame
- returns a learning **model**, which is a transformer

Pipeline

Abstraction of the Spark API

- **Sequence of stages** of Transformers/estimators.

  (inspired by scikit-learn)

  – Estimators are fitted on DataFrame turning them into transformers to keep the chain going

- A pipeline **itself** is an **estimator**

  – it is fitted on the DataFrame

  – turning it into a PipelineModel (transformer)

Example: Classifying Reuters articles

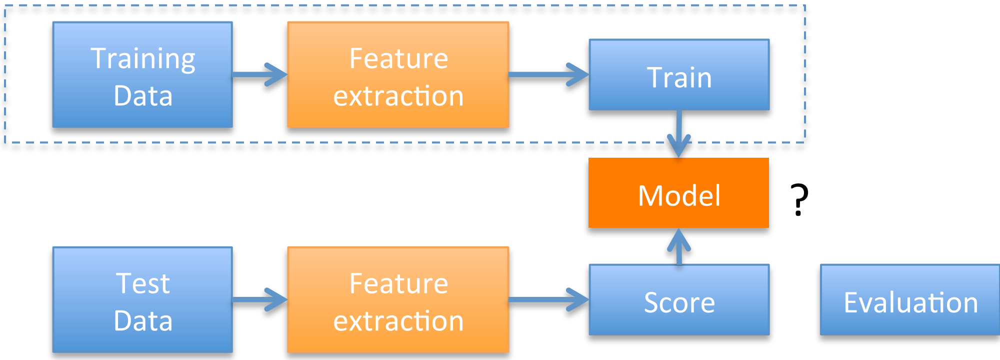
- Data:
  – 21578 articles with metadata divide in 22 JSON files

- problem:
  – based on the <u>words in the body </u>in an article, *determine whether the article has "<u>earn</u>" as one it of its <u>topics</u>*

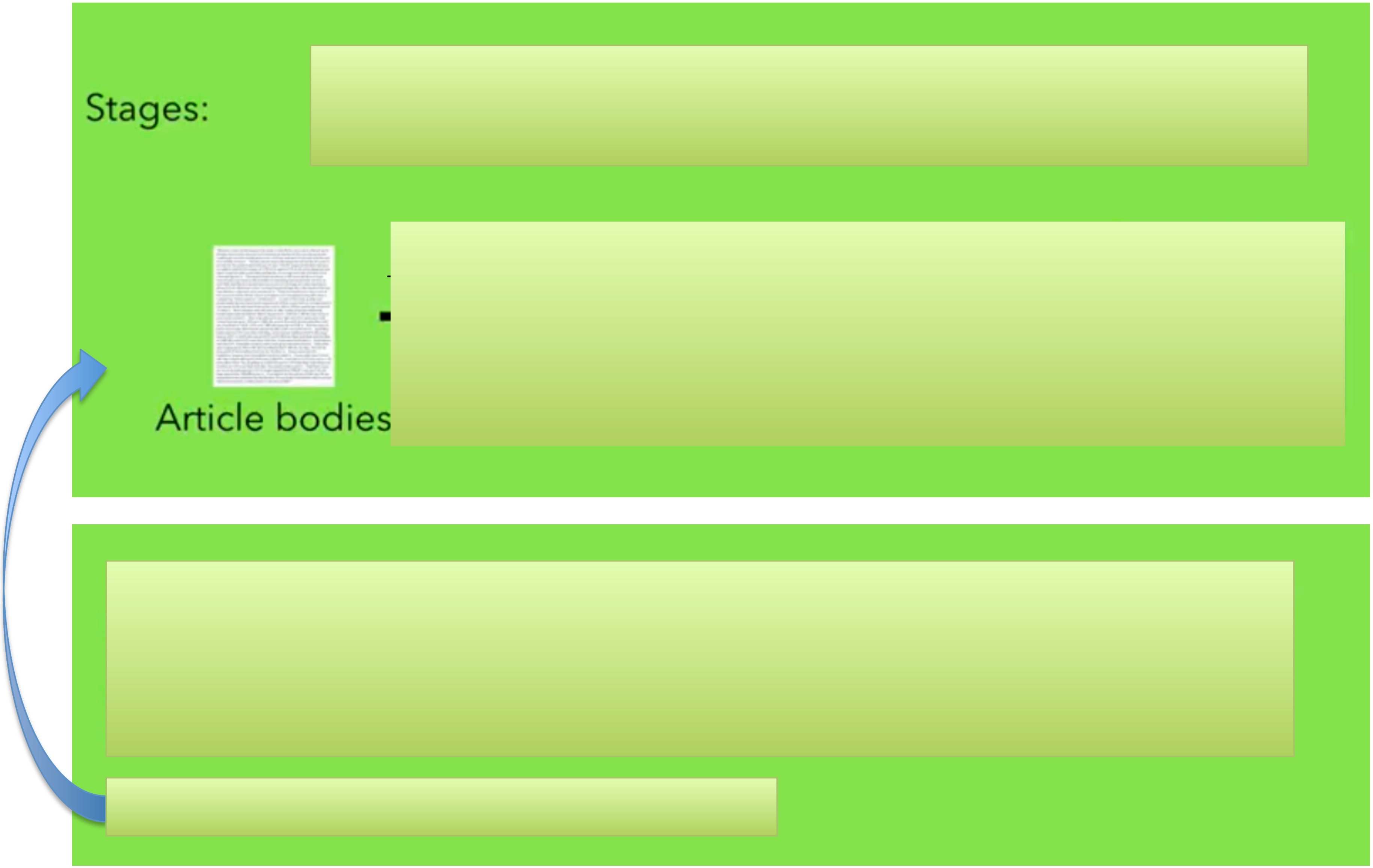
Example: Classifying Reuters articles

- Logistic Regression: is a regression model where the dependent variable is categorical

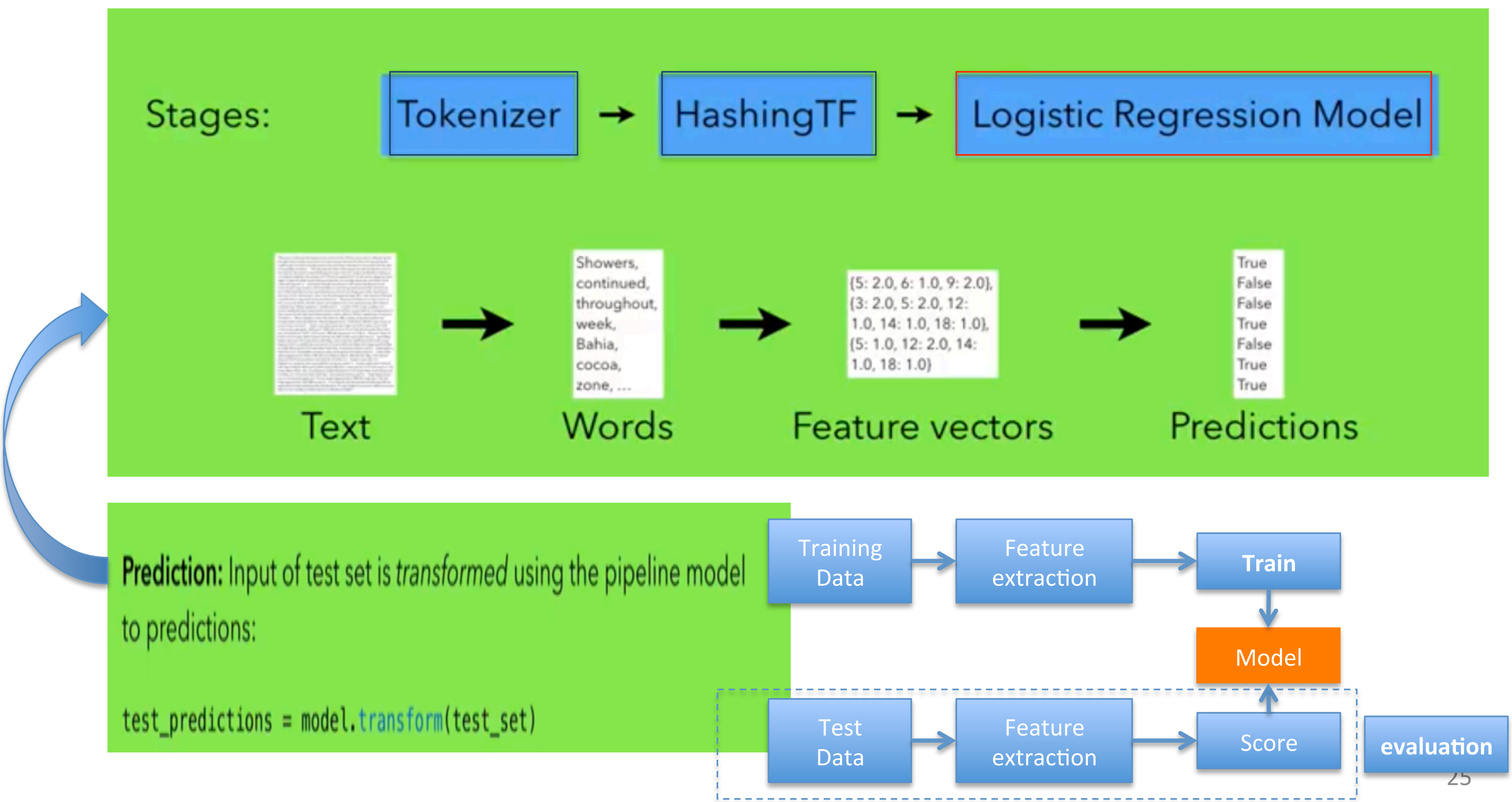
- Feature hashing: turning arbitrary features into indices in a vector or matrix
  - ➢ applying a hash function to the features and using their hash values as indices directly



Pipeline



PipelineModel → Prediction



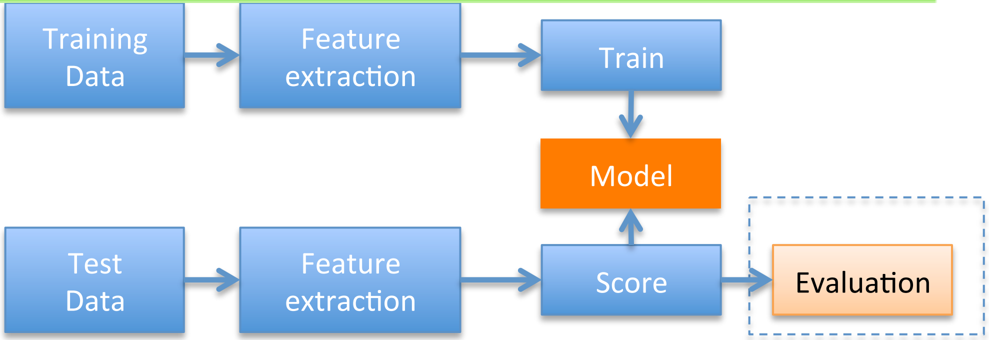
evaluation

- Classification models
- Regression models

https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html

Evaluation: A binary classification evaluator is used:

```
evaluator = BinaryClassificationEvaluator()
test_accuracy = evaluator.evaluate(test_predictions)
```



Test Set Accuracy = 0.975
(with an execution time 27s)

Model selection using cross-validation

Estimator: our pipeline.

Parameter grid: grid of values for the regularisation parameter and maximum number of iteration for the logistic regression algorithm.

Evaluator: The binary classification evaluator.

```
# Evaluator
evaluator = BinaryClassificationEvaluator()

# Cross validation
grid = ParamGridBuilder() \
    .addGrid(classifier.regParam, [0.001, 0.01, 0.1]) \
    .addGrid(classifier.maxIter, [5, 10, 15]) \
    .build()

validator = CrossValidator(estimator = pipeline,
    estimatorParamMaps = grid, evaluator = evaluator, numFolds = 3)

# Pipeline model with cross-validation
model = validator.fit(training_set)

print("Pipeline with cross-validation trained on training dataset.")

# Predictions
training_predictions = model.transform(training_set)
test_predictions = model.transform(test_set)
```

Test Set Accuracy = 0.979
(with an execution time 3mn)

Example: churn prediction model

- Data:
  - Dataset coming from UC Irvine Machine Learning Repository
  - Input data is in CSV format "structured data"

- problem: study the risk of a customer to go to another company.

- Objective: building a churn prediction model

  - http://blog.cloudera.com/blog/2016/02/how-to-predict-telco-churn-with-apache-spark-mllib/

Create DataFrame

The full set of fields,
from the data subscription
In CSV format

state
account length
area code
phone number
international plan
voice mail plan
number vmail messages
total day minutes
total day calls
total day charge
total eve minutes
total eve calls
total eve charge
total night minutes
total night calls
total night charge
total intl minutes
total intl calls
total intl charge
number customer service calls
churned

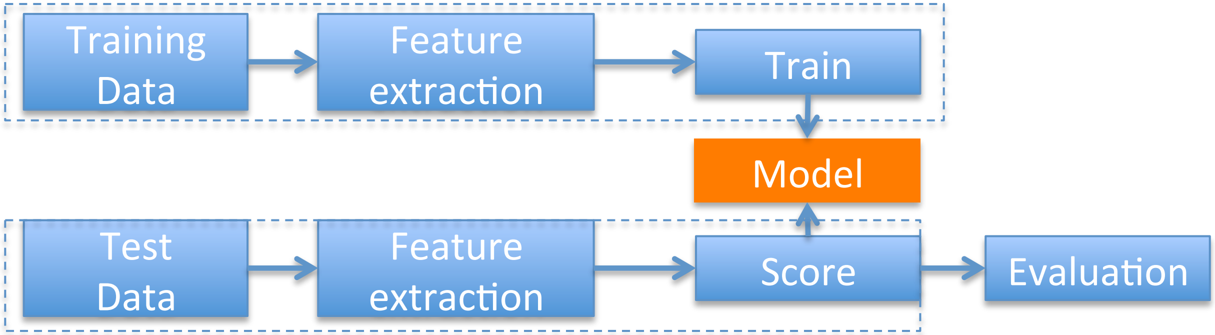
```
from pyspark.sql import SQLContext
from pyspark.sql.types import *

sqlContext = SQLContext(sc)
schema = StructType([ \
    StructField("state", StringType(), True), \
    StructField("account_length", DoubleType(), True), \
    StructField("area_code", StringType(), True), \
    StructField("phone_number", StringType(), True), \
    StructField("intl_plan", StringType(), True), \
    StructField("voice_mail_plan", StringType(), True), \
    StructField("number_vmail_messages", DoubleType(), True), \
    StructField("total_day_minutes", DoubleType(), True), \
    StructField("total_day_calls", DoubleType(), True), \
    StructField("total_day_charge", DoubleType(), True), \
    StructField("total_eve_minutes", DoubleType(), True), \
    StructField("total_eve_calls", DoubleType(), True), \
    StructField("total_eve_charge", DoubleType(), True), \
    StructField("total_night_minutes", DoubleType(), True), \
    StructField("total_night_calls", DoubleType(), True), \
    StructField("total_night_charge", DoubleType(), True), \
    StructField("total_intl_minutes", DoubleType(), True), \
    StructField("total_intl_calls", DoubleType(), True), \
    StructField("total_intl_charge", DoubleType(), True), \
    StructField("number_customer_service_calls", DoubleType(),
True), \
    StructField("churned", StringType(), True)])

churn_data = sqlContext.read \
    .format('com.databricks.spark.csv') \
    .load('churn.all', schema = schema)
```

Specify Feature Extraction

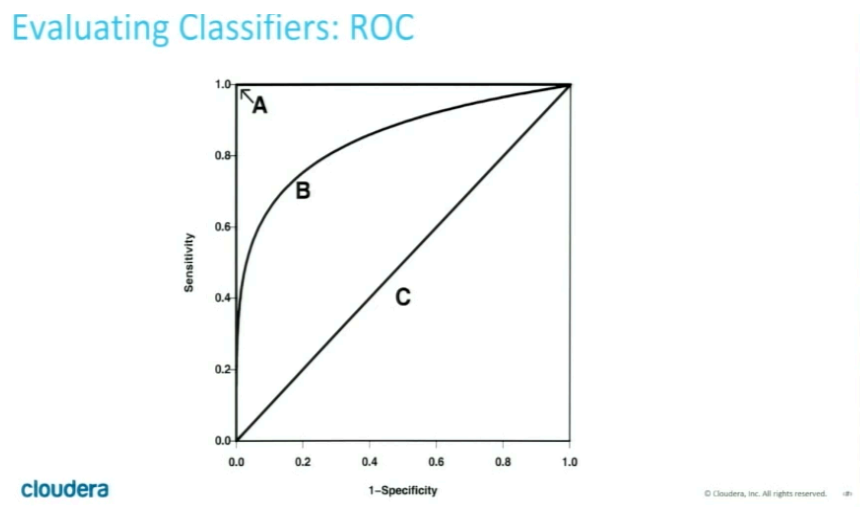
```
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler

label_indexer = StringIndexer(inputCol = 'churned', outputCol = 'label')
plan_indexer = StringIndexer(inputCol = 'intl_plan', outputCol = 'intl_plan_indexed')

reduced_numeric_cols = ["account_length", "number_vmail_messages", "total_day_calls",
                        "total_day_charge", "total_eve_calls", "total_eve_charge",
                        "total_night_calls", "total_intl_calls", "total_intl_charge"]

assembler = VectorAssembler( inputCols = ['intl_plan_indexed'] + reduced_numeric_cols,outputCol = 'features')
```

Model training

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier

classifier = RandomForestClassifier(labelCol = 'label', featuresCol = 'features')
pipeline  = Pipeline(stages=[plan_indexer, label_indexer, assembler, classifier])
model     = pipeline.fit(train)
```



Model Evaluation

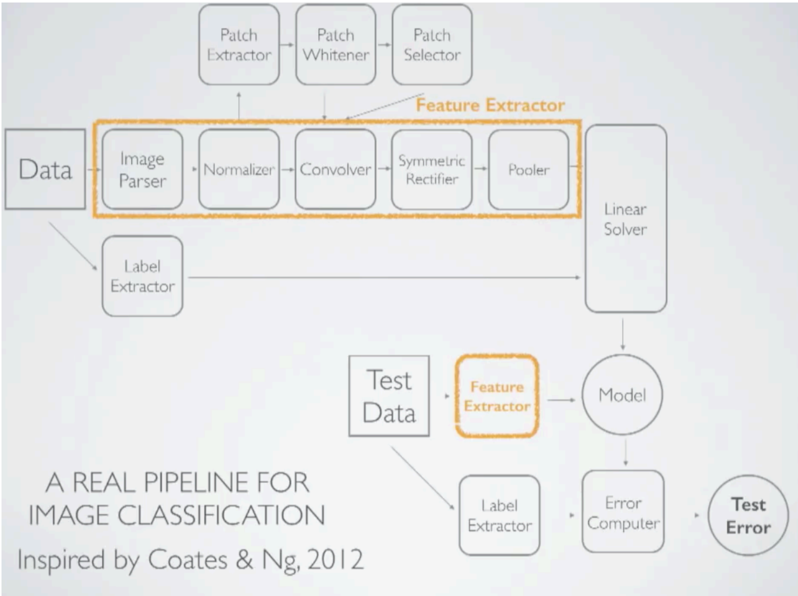



```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

predictions = model.transform(test)
evaluator = BinaryClassificationEvaluator()
auroc = evaluator.evaluate(predictions, {evaluator.metricName: "areaUnderROC"})
```

The AUROC is 0.494987527228

Example: Image Classifier

- Data:
  - Input data is in images
- problem:



Conclusion

Advantages

- General purpose big-data package
- Good scalability with parallelisation
- High flexibility/class standardisation
- Actively developed

Disadvantages

- Inferior to others in single-processor performance
- in early development teething troubles