KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics

Lorand Madai-Tahy, Khaled Mansour

Outline

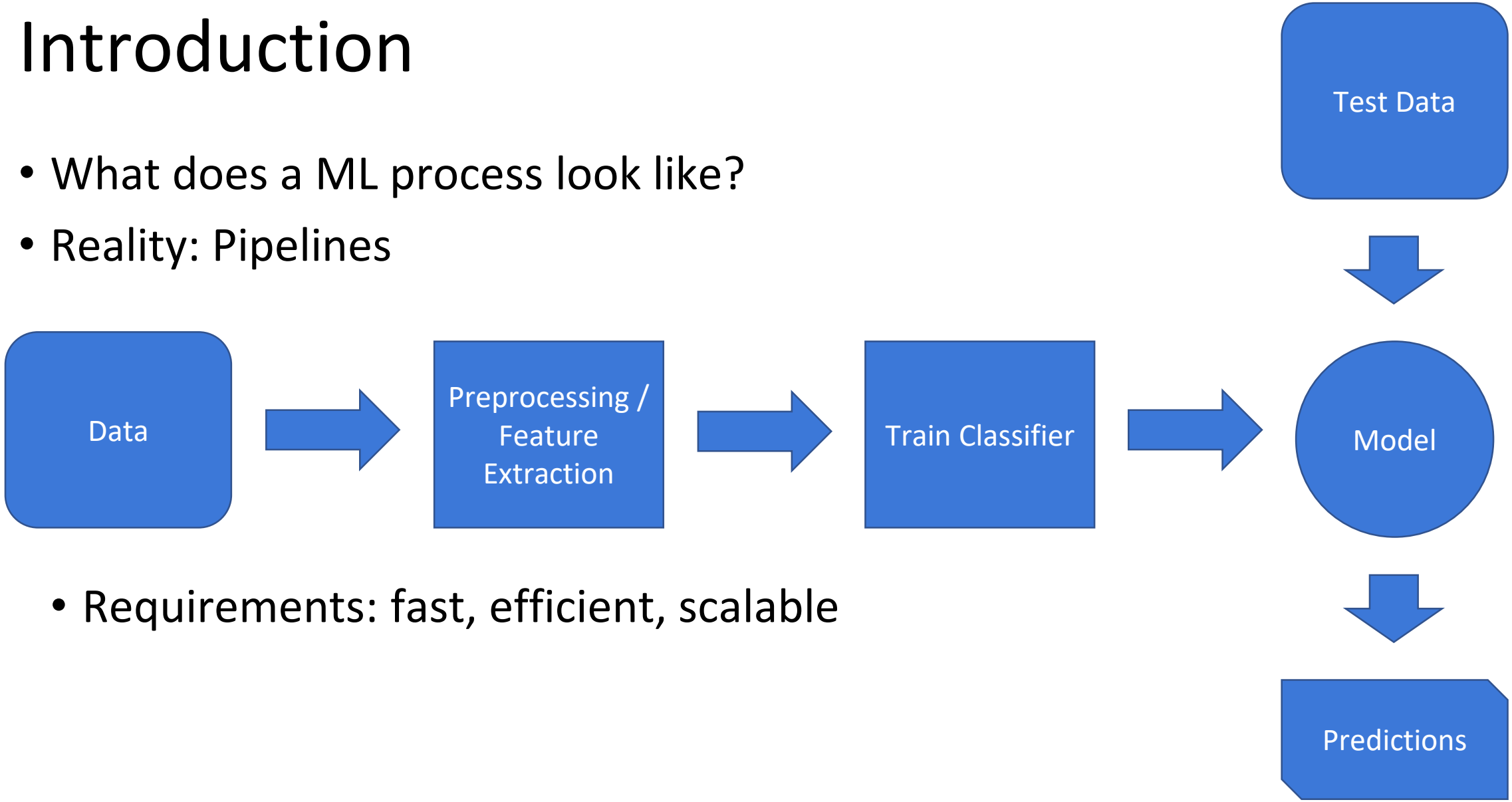
- Introduction
- KeystoneML Structure and Core APIs
- Example Pipelines
- Optimization
- PCA
- Scalability / Experiment Results
- Further Research

Introduction

- What does a ML process look like?
- Wish

Introduction

- What does a ML process look like?
- Reality: Pipelines

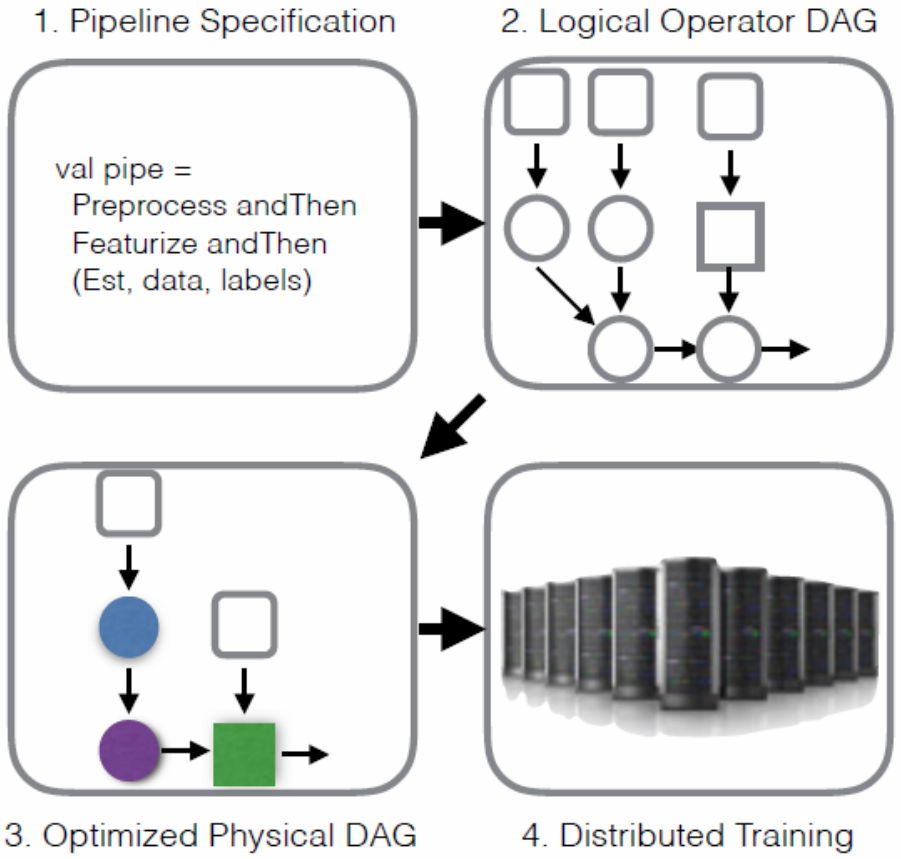


- Requirements: fast, efficient, scalable

Introduction

KeystoneML

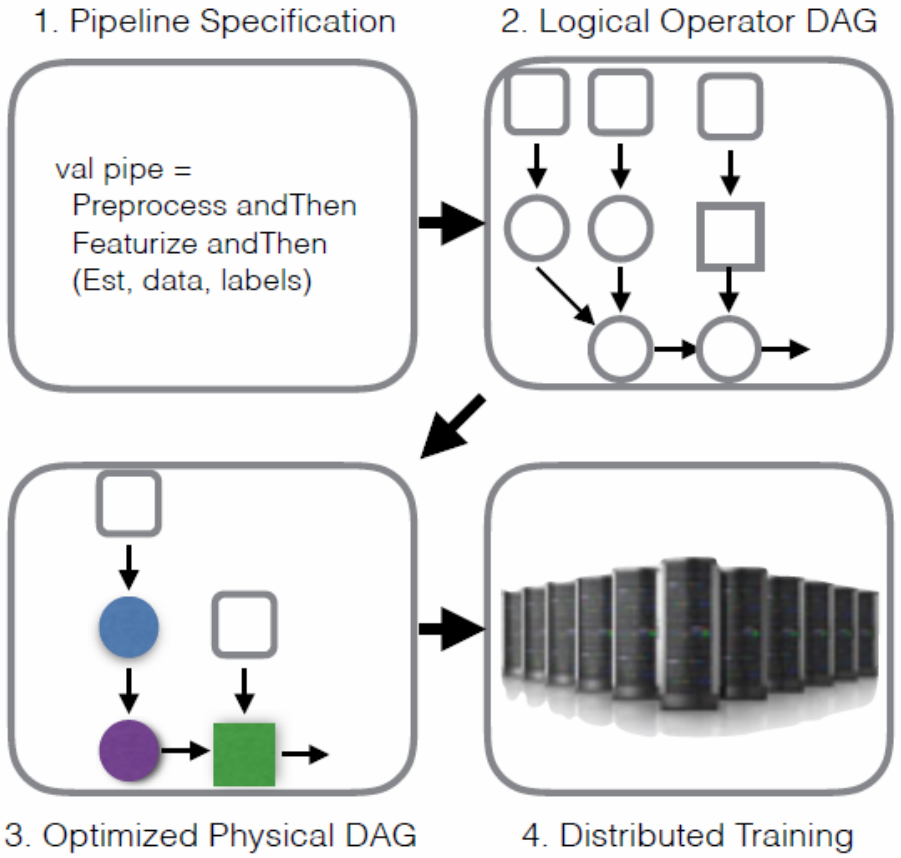
- Evan Sparks PhD Thesis, UC Berkeley
- Published at 33rd IEEE International Conference on Data Engineering, April 2017
- Open source ML pipelining framework



Introduction

KeystoneML Goals

- Framework for pipelining end to end
- Feature extraction, model building and testing in one
- Scalability using Apache Spark
- Fast and efficient due to optimization

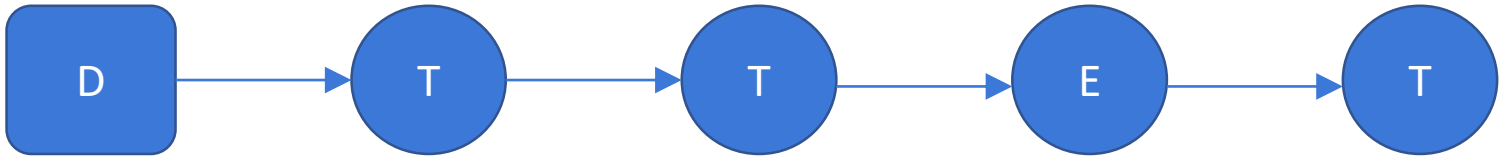


Sparks et al., KeystoneML, ICDE 2017

KeystoneML Structure and Core API

Pipeline

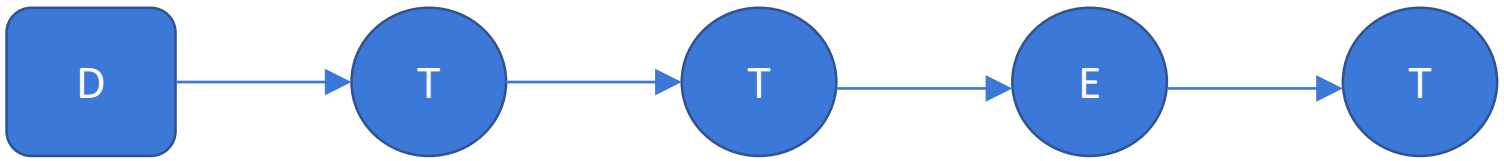
- Chain of operators
- Operator: Function with zero or more inputs to produce some output
- Two kinds of operators
  - Transformer
  - Estimator



KeystoneML Structure and Core API

Transformer

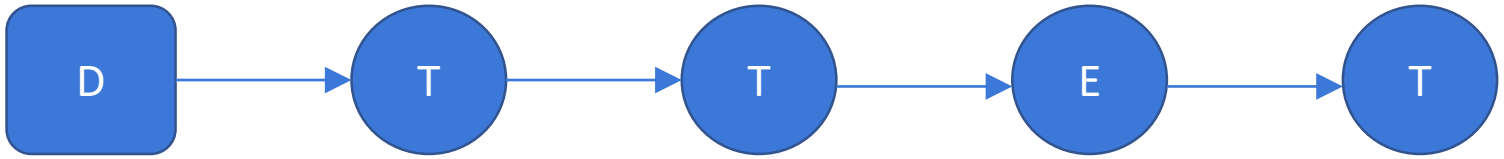
- Operator on a data item or collection of items
- Unary function without side effects
- Returns new data item or collection of items
- E.g. data transformation, feature extraction, model application (!)



KeystoneML Structure and Core API

Transformer

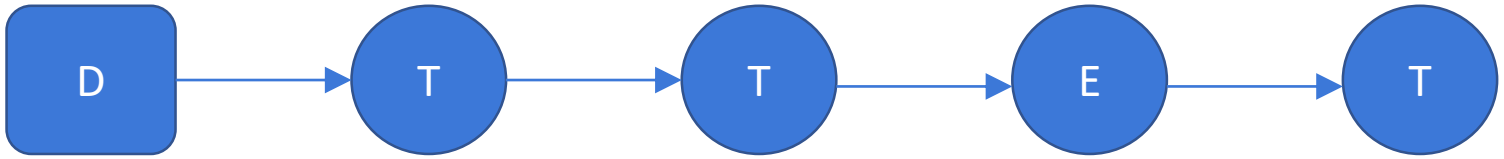
```
trait Transformer[A, B] extends Pipeline[A, B] {
  def apply(in: Dataset[A]): Dataset[B] = in.map(apply)
  def apply(in: A): B
}
```



KeystoneML Structure and Core API

Estimator

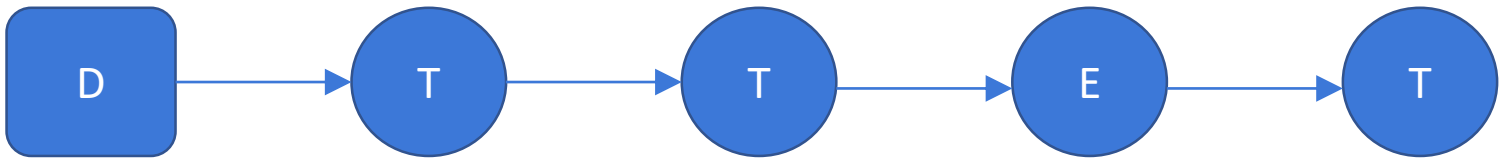
- Applied to a collection of data items
- Generates a transformer => function generating function
- E.g. Linear Solver
  - Input: data item set and labels
  - Find linear model, return transformer that can apply model to new data



KeystoneML Structure and Core API

Estimator

```
trait Estimator[A, B] {
    def fit(data: Dataset[A]): Transformer[A, B]
}
```



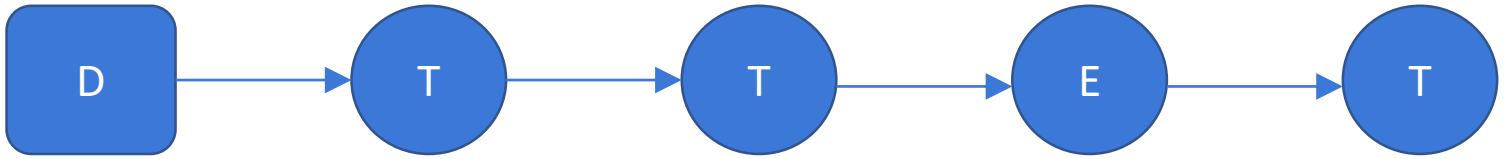
KeystoneML Structure and Core API

`andThen`

- Concatenate operators
- Branch Pipelines

`gather`

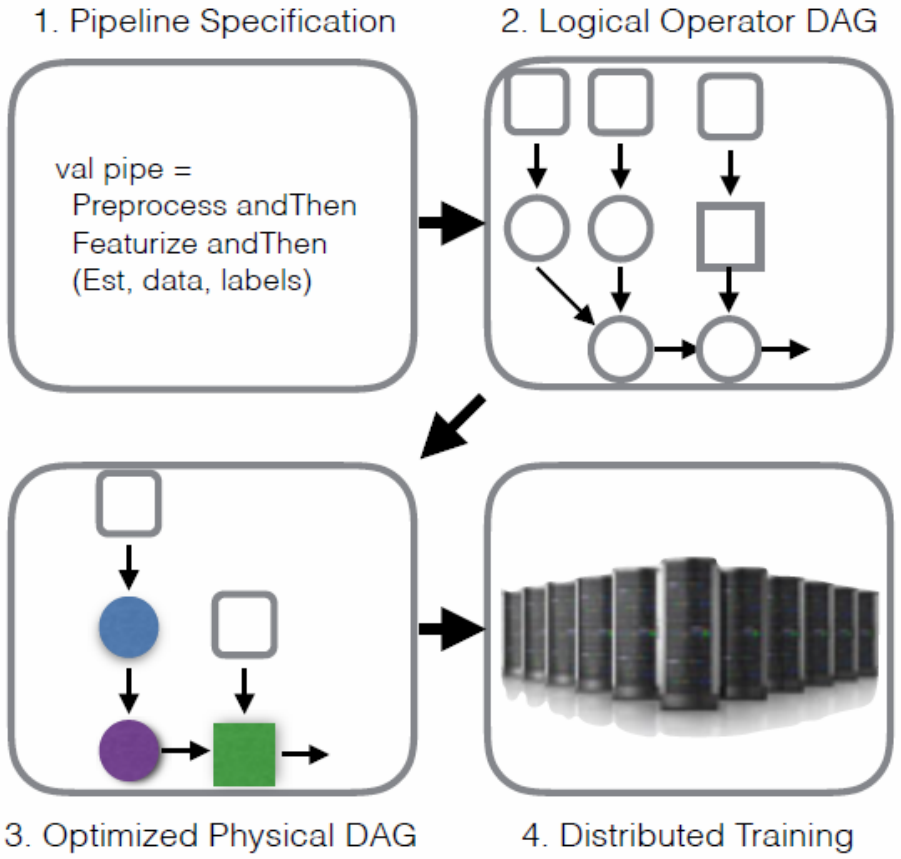
- Concatenate Output



KeystoneML Structure and Core API

Pipeline Execution Recap

1. Define pipeline using API
2. KeystoneML builds directed acyclic graph
3. **Optimization** when applied to data
4. Distributed execution

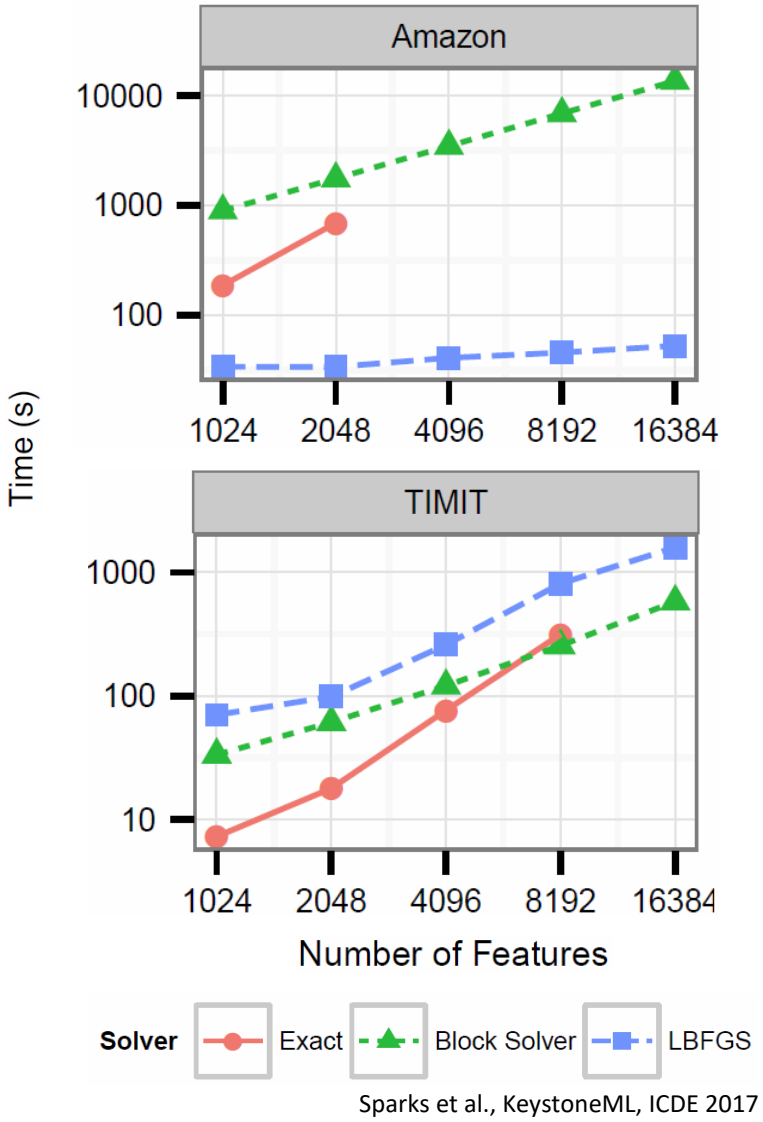


Sparks et al., KeystoneML, ICDE 2017

Optimization

Motivation

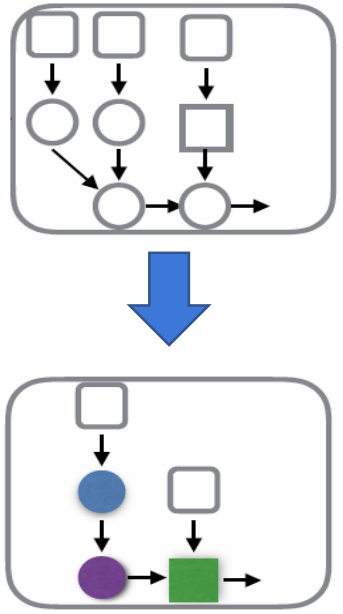
- Different solvers work well in different settings
- Pipelines can have iterative stages
- Intermediate results might be reusable
- Stages might be reorderable
- Optimal execution might depend on workload (e.g. data)



Sparks et al., KeystoneML, ICDE 2017

Optimization

- Unique feature of KeystoneML
- Logical and physical optimization
- Two-fold optimization
  - Operator-level optimization
  - Whole-pipeline optimization
- Determining the best implementation and machine
- Reordering and merging of operators

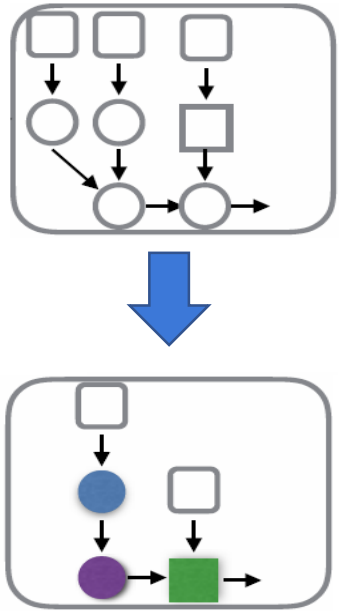


Sparks et al., KeystoneML, ICDE 2017

Optimization

Operator-Level

- Goal: choose the best physical implementation
- Inspired by database query optimizers, but more complex
- Cost-based optimizers
- Two parts
  - Operator-specific
  - Cluster-specific



Sparks et al., KeystoneML, ICDE 2017

Optimization

$$c(f, A_s, R) = R_{exec} c_{exec}(f, A_s, R_w) + R_{coord} c_{coord}(f, A_s, R_w)$$

f operator

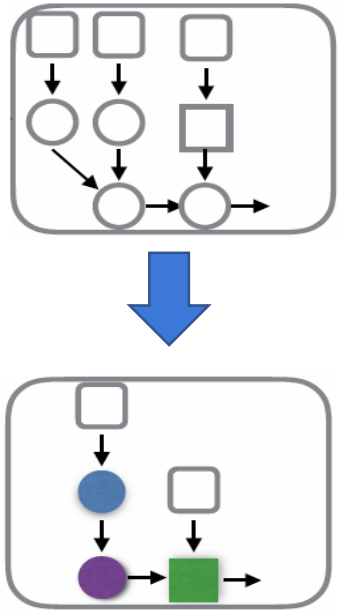
A_s dataset statistics

R cluster resource descriptor

R_w number of available cluster nodes

exec execution

coord coordination



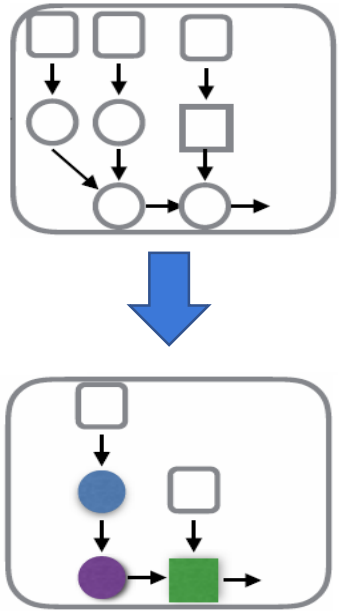
Sparks et al., KeystoneML, ICDE 2017

Optimization

$$c(f, A_s, R) = R_{exec}c_{exec}(f, A_s, R_w) + R_{coord}c_{coord}(f, A_s, R_w)$$

R: cluster resource descriptor

- Automatically defined
- Represents available computing, memory and networking resources
- Defined via configuration data and microbenchmarks
  - Per-node CPU throughput (GFLOP/s)
  - Memory Bandwidth (GB/s)
  - Network Speed (GB/s)



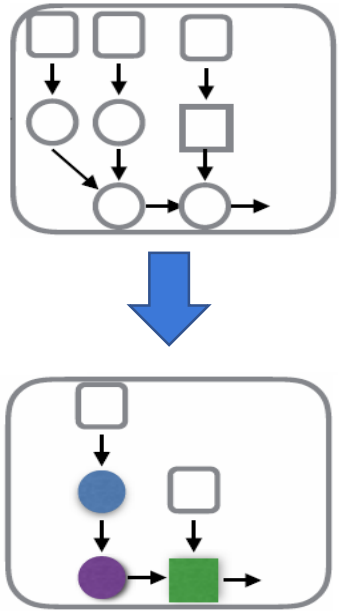
Sparks et al., KeystoneML, ICDE 2017

Optimization

$$c(f, A_s, R) = R_{exec}c_{exec}(f, A_s, R_w) + R_{coord}c_{coord}(f, A_s, R_w)$$

c_{exec}, c_{coord}: cost functions

- User-defined
- Part of the `CostModel` API
- Cost of the longest critical path in the execution graph
  - i.e. max FLOPS used in node
  - i.e. max data sent through a link

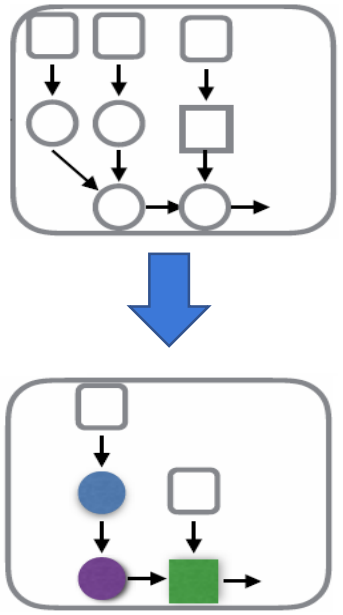


Sparks et al., KeystoneML, ICDE 2017

Optimization

Algorithm	Compute	Network	Memory
Local QR	$O(nd(d+k))$	$O(n(d+k))$	$O(d(n+k))$
Dist. QR	$O(\frac{nd(d+k)}{w})$	$O(d(d+k))$	$O(\frac{nd}{w}+d^2)$
L-BFGS	$O(\frac{insk}{w})$	$O(idk)$	$O(\frac{ns}{w}+dk)$
Block Solve	$O(\frac{ind(b+k)}{w})$	$O(id(b+k))$	$O(\frac{nb}{w}+dk)$

Sparks et al., KeystoneML, ICDE 2017



Sparks et al., KeystoneML, ICDE 2017

w number of workers
i number of passes over the dataset
s average number of non-zero values for sparse solvers
b block size

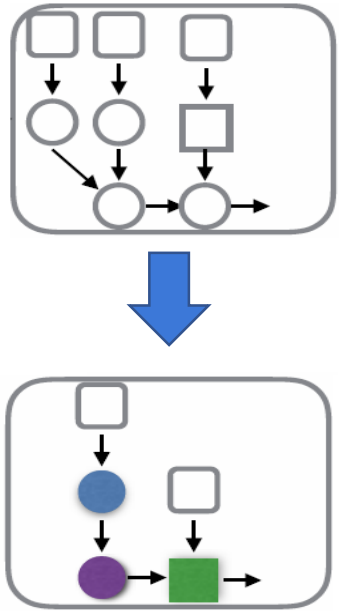
Compute/Memory requirements per node
Network data sent over the most loaded link

KeystoneML chooses the operator with the best overall performance

Optimization

Whole-Pipeline optimization

- Execution subsampling
  Analyze initial input dataset, optimize first operator, then optimize subsequent operators
- Merge common sub-expressions of a pipeline
- Automatic materialization
  - Avoid recalculation of results
  - Caching Algorithm
  - Materialize most promising intermediate results with regards to estimated runtime, output size and system memory size until memory is full



Sparks et al., KeystoneML, ICDE 2017

Optimization

Caching Algorithm

- Greedy algorithm
- Maximize time saved

`G`	unoptimized pipeline DAG
`v`	node
`t`	time per iteration in v
`size`	size of v's output
`memSize`	memory constraint

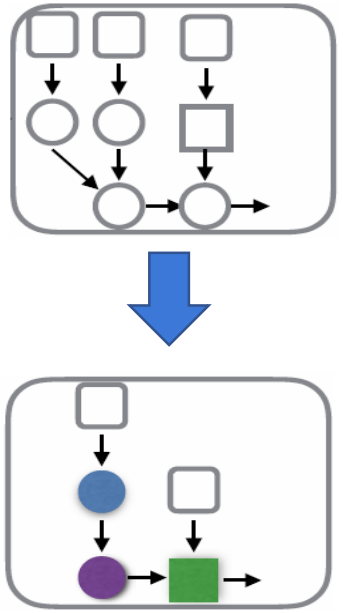
```
1  Algorithm GreedyOptimizer
       input  : G, t, size, memSize
       output: cache
2      cache ← ∅;
3      memLeft ← memSize;
4      next ← pickNext (G, cache, size, memLeft, t);
5      while nextNode ≠ ∅ do
6          cache ← cache ∪ next;
7          memLeft ← memLeft - size(next);
8          next ← pickNext (G, cache, size, memLeft, t);
9      end
10     return cache;
1  Procedure pickNext()
       input  : G, cache, size, memLeft, t
       output: next
2      minTime ← ∞;
3      next ← ∅;
4      for v ∈ nodes(G) do
5          runtime ← estRuntime (G, cache ∪ v, t);
6          if runtime < minTime & size(v) < memLeft then
7              next ← v;
8              minTime ← runtime;
9          end
10     end
11     return next;
```

Sparks et al., KeystoneML, ICDE 2017

Optimization

Easy usage:

- Define cost model according to (new) hardware
- Automatic optimization (e.g. choosing the most efficient solver for the problem)



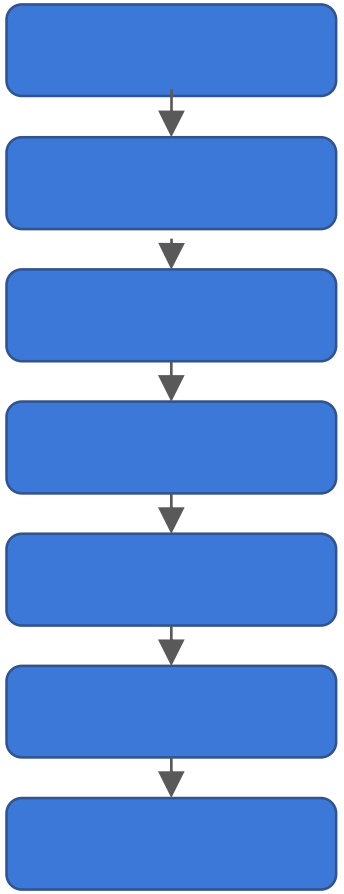
Sparks et al., KeystoneML, ICDE 2017

- Goal is to avoid bad decisions
- Rough cost models generally efficient

Example: Text Classification

News articles classification

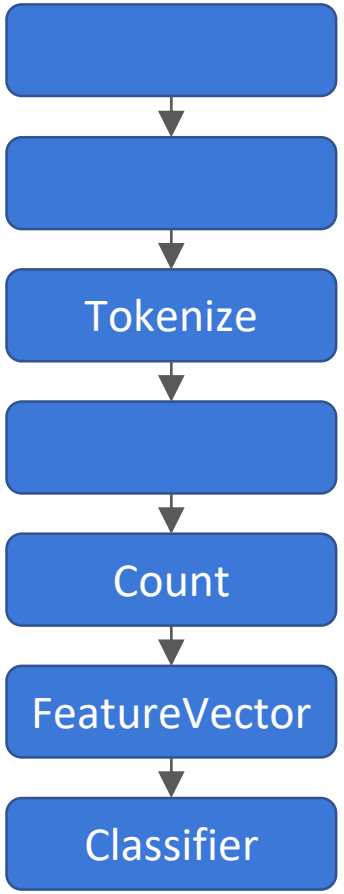
- Example pipeline provided with KeystoneML
- Bag-of-words-Model
- Task: Classify news articles according to topic (atheism, autos, mideast politics, …)
- Training set: about 11000 articles
- Test set: about 7500 articles



Example: Text Classification

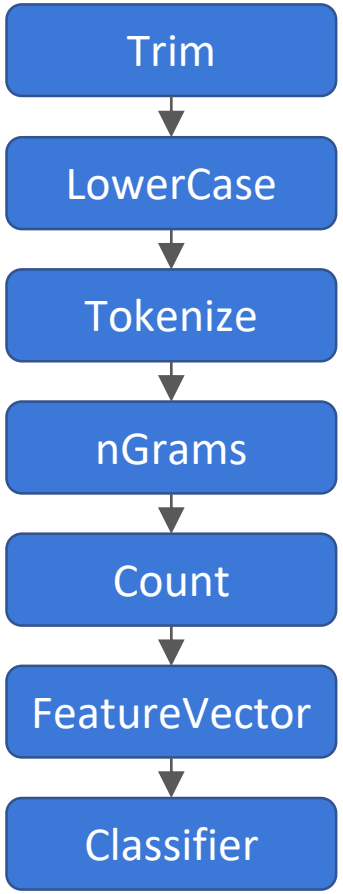
Bag of words in a minute

- Represent Texts as sets (bag) of words
- Create sparse vector with entry for every distinct word representing frequency of words in a given text
- Use frequency of words in given training data to determine the probability of a given new text to belong to a class
- Example: Bayesian Spam Filtering



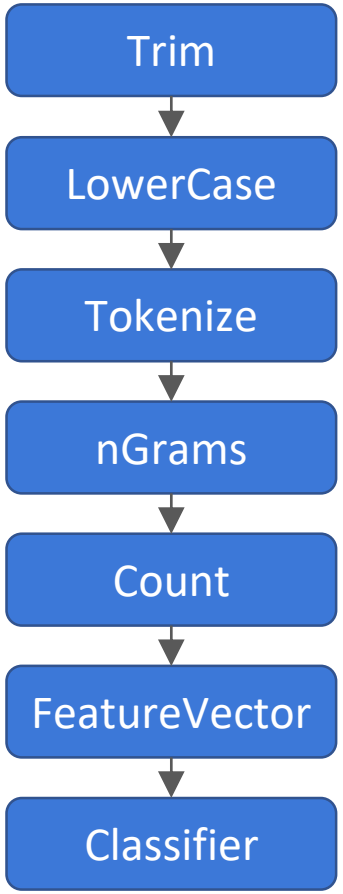
Example: Text Classification

```
def run(sc: SparkContext, conf: NewsgroupsConfig): Pipeline[String, Int] = {
   val trainData = NewsgroupsDataLoader(sc, conf.trainLocation)
   val numClasses = NewsgroupsDataLoader.classes.length

   val predictor = Trim andThen
       LowerCase() andThen
       Tokenizer() andThen
       NGramsFeaturizer(1 to conf.nGrams) andThen
       TermFrequency(x => 1) andThen
       (CommonSparseFeatures[Seq[String]](conf.commonFeatures), trainData.data) andThen
       (NaiveBayesEstimator[SparseVector[Double]](numClasses), trainData.data, trainData.labels) andThen
       MaxClassifier

   val testData = NewsgroupsDataLoader(sc, conf.testLocation)
   val testLabels = testData.labels
   val testResults = predictor(testData.data)
   val eval = new MulticlassClassifierEvaluator(numClasses).evaluate(testResults, testLabels)

   predictor
 }
```



Example: Text Classification

```
a    b    c    d    e    f    g    h    i    j    k    l    m    n    o    p    q    r    s    t    <-- Classified As
--   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
190  28   21   16   32   2    10   3    0    13   16   7    11   30   2    0    0    1    5    2    a = comp.graphics
56   167  50   12   34   3    7    6    3    7    4    3    4    12   8    9    1    4    4    0    b = comp.os.ms-windows.misc
43   47   145  60   20   9    8    1    0    11   17   2    0    26   2    0    0    0    1    0    c = comp.sys.ibm.pc.hardware
26   22   51   165  8    20   12   10   1    4    18   8    6    28   2    2    1    0    1    0    d = comp.sys.mac.hardware
78   46   15   10   182  3    5    0    1    10   3    3    6    25   3    0    2    0    0    3    e = comp.windows.x
14   7    9    26   3    175  46   16   3    3    25   15   6    25   6    11   4    0    2    0    f = rec.autos
7    3    4    8    3    25   280  9    6    1    11   4    3    12   3    8    1    1    8    1    g = rec.motorcycles
14   7    4    6    0    7    7    187  63   2    6    6    6    26   18   8    3    4    10   13   h = rec.sport.baseball
.
.
.
3    2    1    6    3    3    6    16   22   1    1    10   15   5    20   16   178  17   36   15   q = talk.politics.mideast
3    1    3    3    4    3    14   7    7    2    1    13   4    11   7    5    14   53   59   37   r = talk.religion.misc
7    5    2    3    3    5    17   17   2    6    5    14   7    2    4    11   10   13   155  31   s = alt.atheism
9    4    3    3    1    2    4    7    4    1    4    15   7    7    1    0    10   10   10   296  t = soc.religion.christian
--   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --   --
Avg Accuracy: 0.948
Macro Precision: 0.479
Macro Recall: 0.474
Macro F1: 0.468
Total Accuracy:0.481
Micro Precision: 0.481
Micro Recall: 0.481
Micro F1:     0.481
```

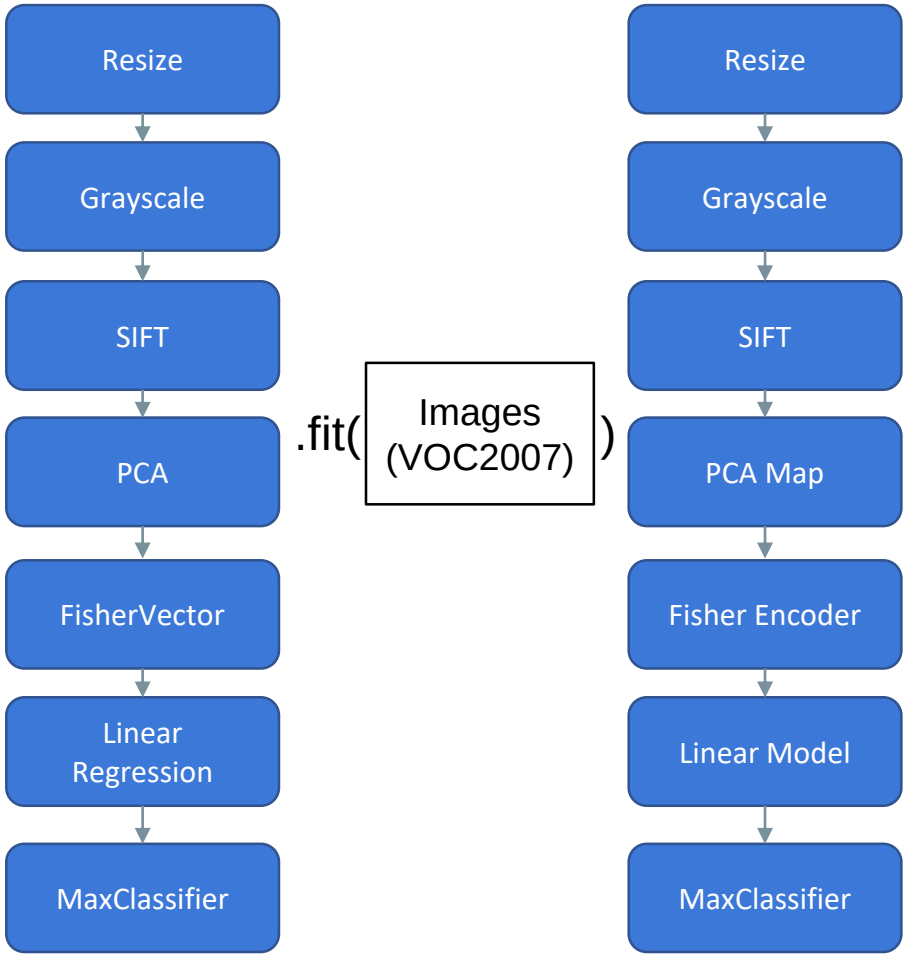


A less simple example: Image Classification

- Dataset : VOC 2007
- Training set of 5000 full sized images, 20 classes
- Early implementation of an example pipeline provided by KeystoneML
- Task: multiclass image classification

Stage	n	d	size (GB)
Input	5000	1 m pixel JPEG	0.4
Resize	5000	260k pixels	3.6
Grayscale	5000	260k pixels	1.2
SIFT	5000	65000x128	309
PCA	5000	65000x64	154
FisherVector	5000	256x64x2	1.2
Linear Regression	5000	20	0.0007
Max Classifier	5000	1	0.00009

Image Classification



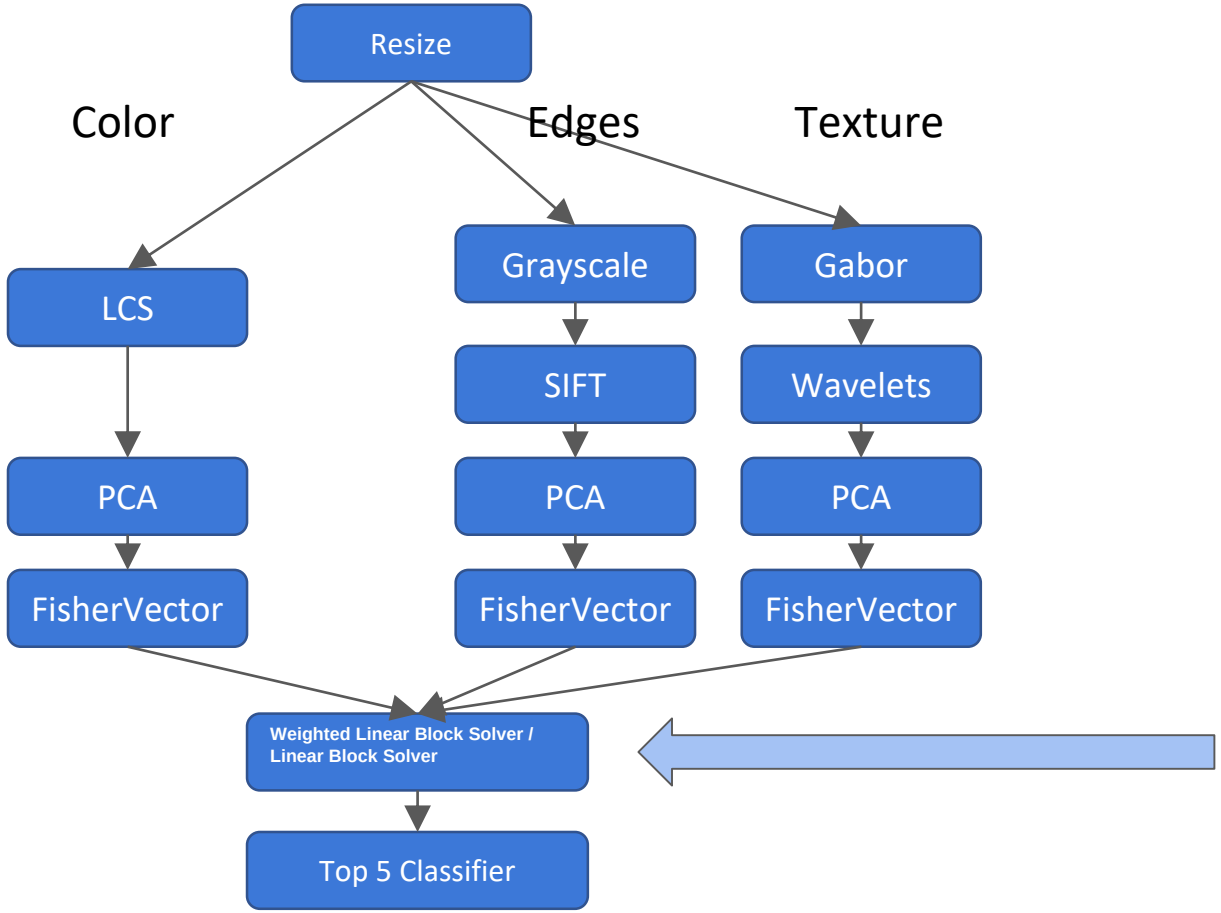

```
val pipeline = (PixelScaler then
  GrayScaler then
  SIFTExtractor() then
  new BatchPCATransformer(pcaMat) then
  new FisherVector(gmm) then
  FloatToDouble then
  MatrixVectorizer then
  NormalizeRows then
  SignedHellingerMapper thenLabelEstimator
  new BlockLeastSquaresEstimator(blockSize, numPasses, lambda))
    .fit(trainImages, trainingLabels)
```

Outdated code example, see at 25:10
https://www.oreilly.com/learning/keystoneml-optimized-large-scale-machine-learning-pipelines-on-apache-spark

Image classification Pipeline

- SIFT
- Fisher Vector
- Linear Regeression
- PCA

Even more complex: ImageNet

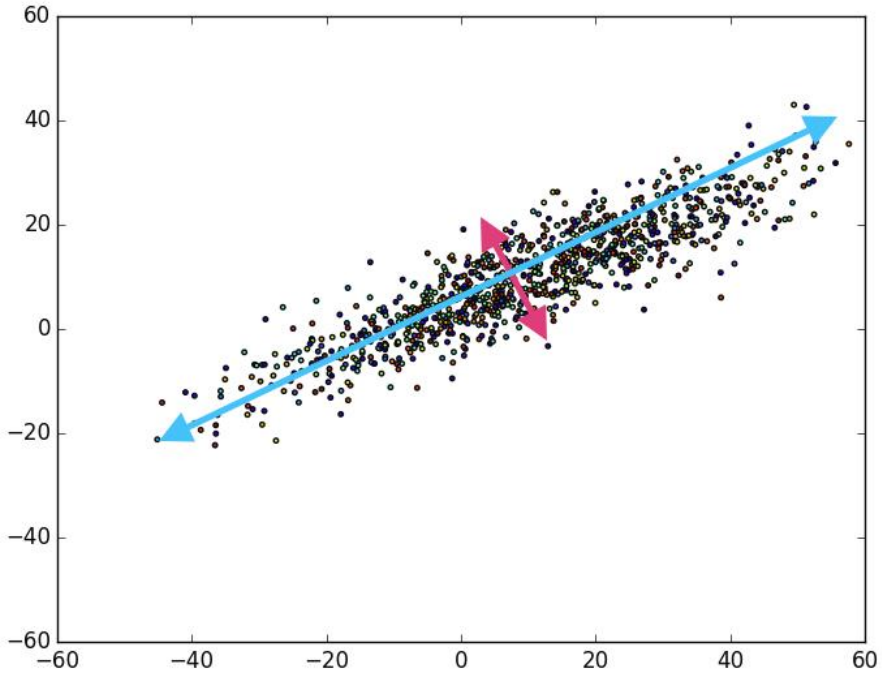


- 1000 classes
- 1.2M images
- 64k features
- Different parallel feature extraction pipelines
- Changing a solver is changing one line of code

Principal Component Analysis (PCA)

- Motivation: We assume higher variance means more information
- Find the axis of highest variance (1. PC)
- Subsequent PCs have highest possible variance and are orthogonal to preceding PCs
- Classical approach: solve the eigenvalue problem for the covariance matrix

$$Ce_a = \lambda e_a$$



http://austingwalters.com/pca-principal-component-analysis/

Principal Component Analysis (PCA)

Algorithm	Compute	Network	Memory
SVD	$O(nd^2)$	$O(nd)$	$O(nd + d^2)$
TSVD	$O(ndk + ink^2)$	$O(nd)$	$O(nd + dk)$
Dist. SVD	$O(\frac{nd^2}{w})$	$O(d^2)$	$O(\frac{nd}{w} + d^2)$
Dist. TSVD	$O(\frac{ndk+ink^2}{w})$	$O(i(nk + dk))$	$O(\frac{nd}{w} + dk)$

Sparks et al., KeystoneML, ICDE 2017

- As for the solver, KeystoneML offers multiple implementations of PCA with according cost functions
- Eigenvalues and Eigenvectors are found using Singular Value Decomposition (SVD) (exact algorithm) or via an approximate algorithm, Truncated SVD
- We see that the fastest implementation depends number of samples n, number of dimensions d, and number of requested eigenvalues k

	$d = 256$			$d = 4096$		
	$k = 1$	16	64	$k = 16$	64	1024
$n = 10^4$						
SVD	**0.1**	**0.1**	**0.1**	26	26	**26**
TSVD	0.2	0.3	0.4	**3**	**6**	34
Dist. SVD	1.7	1.7	1.7	106	106	106
Dist. TSVD	4.9	3.8	5.3	6	22	104
$n = 10^6$						
SVD	11	11	11	x	x	x
TSVD	14	30	65	x	x	x
Dist. SVD	**2**	**2**	**2**	260	**260**	**260**
Dist. TSVD	16	59	262	**75**	1,326	8,310

Sparks et al., KeystoneML, ICDE 2017

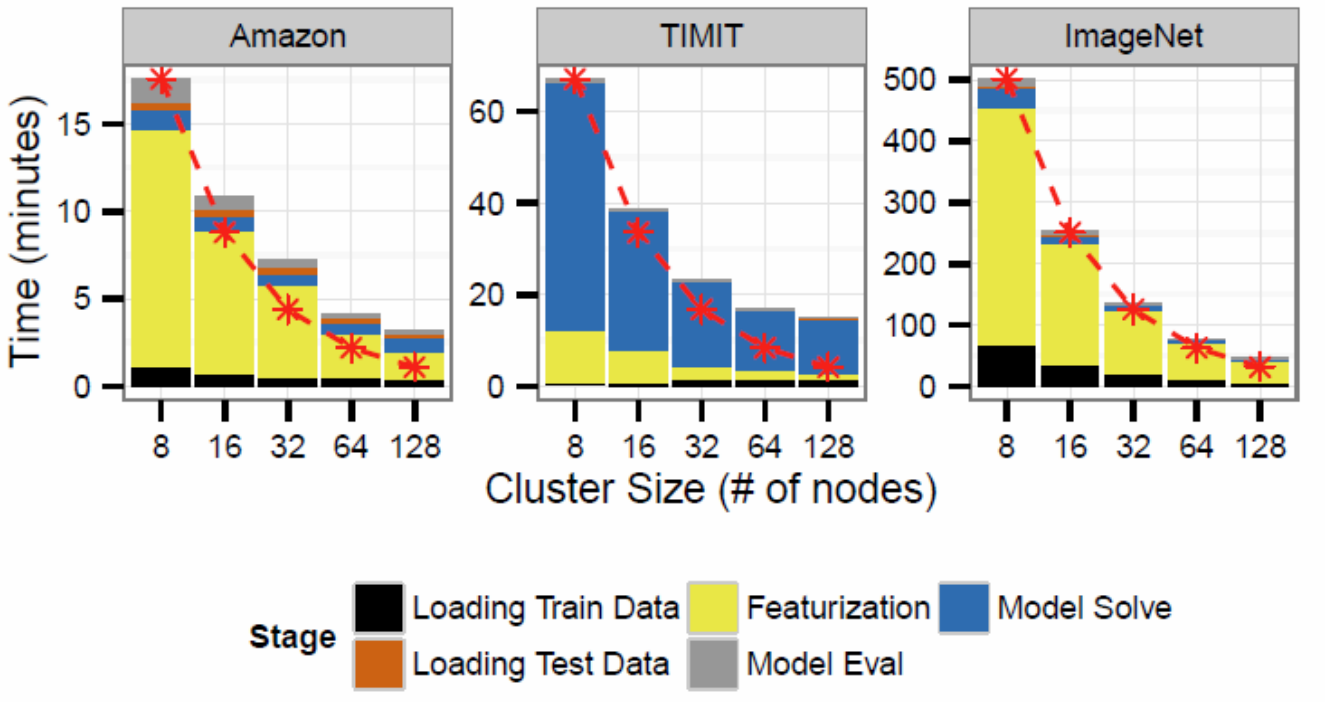
Results

Dataset	KeystoneML		Reported	
	Accuracy	Time (m)	Accuracy	Time (m)
Amazon [1]	91.6%	3.3	-	-
TIMIT [2]	66.06%	138	66.33%	120
ImageNet [3][2]	67.43%	270	66.58%	5760
VOC 2007 [18]	57.2%	7	59.2%	87
CIFAR-10 [45]	84.0%	28.7	84.0%	50.0

Sparks et al., KeystoneML, ICDE 2017

- Comparison VOC pipeline executes end-to-end on 32 nodes using KeystoneML in just 7 minutes compared to 1 hour and 27 minutes to execute on a single 16-core machine with 256 GB of RAM using the original source code
- 12.4× speedup with 16× the cores
- KeystoneML mostly achieves significant speedups compared to original code / reported time (if code not available)

Evaluation of KeystoneML: Scalability



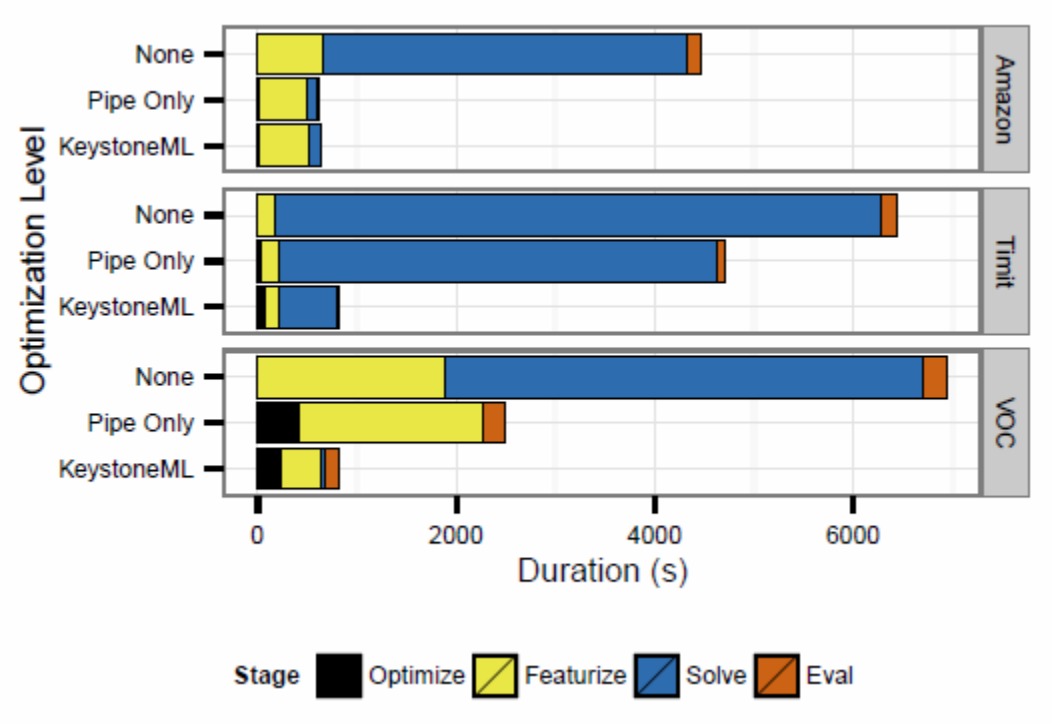
Sparks et al., KeystoneML, ICDE 2017

- Here we demonstrate the scaling properties from 8 to 128 nodes of the text, image, and Kernel SVM pipelines on the Amazon, ImageNet (with 16k features) and TIMIT datasets (with 65k features) respectively
- The ImageNet pipeline exhibits near perfect horizontal scalability up to 128 nodes, while the Amazon and TIMIT pipeline scale well up to 64 nodes
- => KeystoneML is scalable! Why?

Evaluation of KeystoneML: Scalability

Optimization

- Optimization shows a performance gain of up to 1500%
- Gains depend on dataset, solvers and hardware



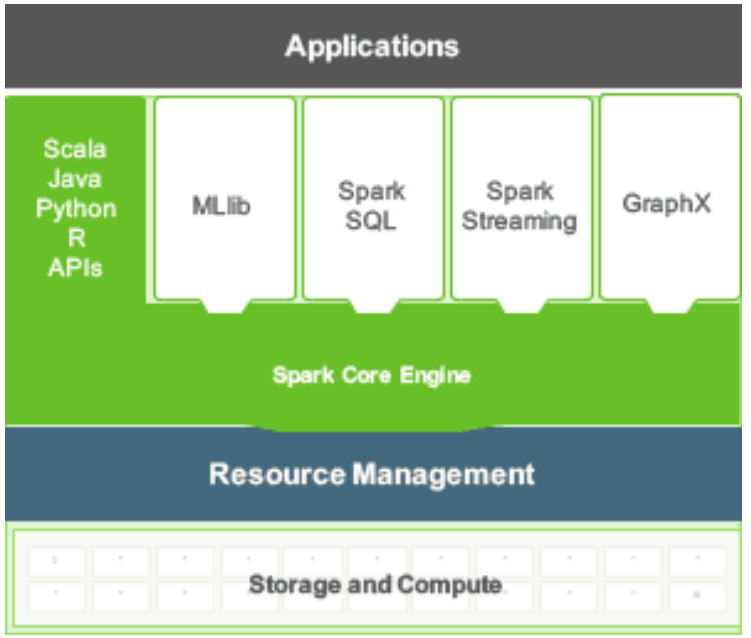
Sparks et al., KeystoneML, ICDE 2017

Apache Spark

- KeystoneML is implemented on top of Apache Spark, a cluster computing engine that has been shown to have good scalability and performance for many iterative ML algorithms

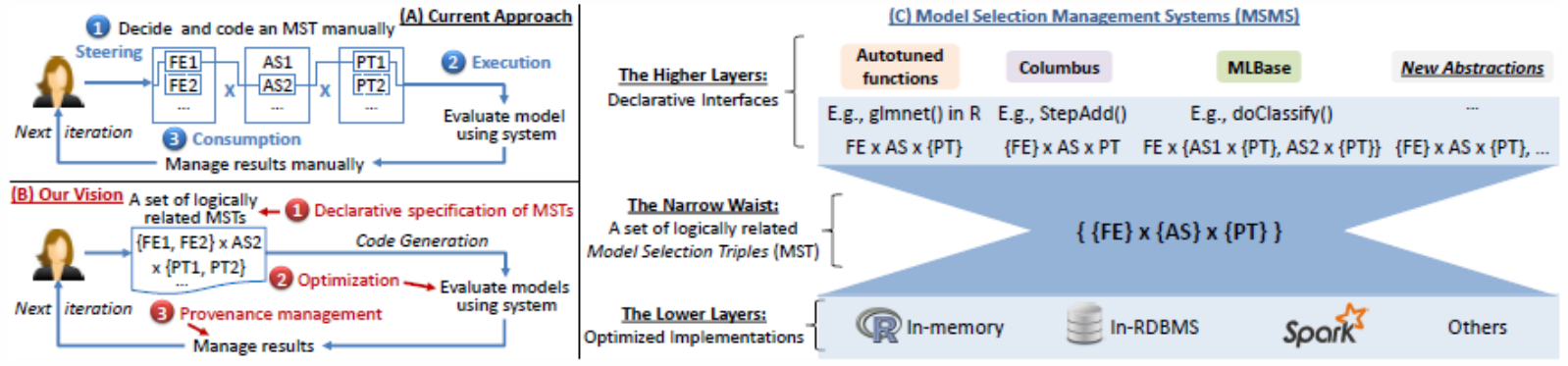
- „Porting KeystoneML to work with other distributed computing systems would require that the presence of a distributed collections API that supports the MapReduce paradigm as well as predictable communication via broadcast, all-to-one reduce operations, and aggregation trees.” (Sparks et al., KeystoneML, ICDE 2017)

https://spark.apache.org/



https://de.hortonworks.com/apache/spark/

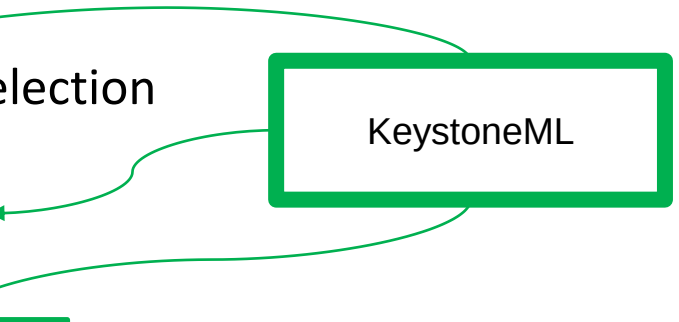
KeystoneML as part of a MSMSs?



Kumar et al, Model Selection Management Systems, SIGMOD 2015

Model Selection Management System

- Help the engineer
- Automate pipeline selection
  - Feature engineering
  - Algorithm selection
  - Parameter tuning
- Test multiple optimized configurations at once to avoid too many iterative trials



KeystoneML

- Pipelining
- Chooses operators depending on
- Optimization

Further Research

- How can algorithms like asynchronous SGD or back-propagation be integrated with the robustness and scalability that KeystoneML provides
- Further research can investigate pipeline optimizations like node reordering to reduce data transfers and also look at how hyperparameter tuning can be integrated into the system
- Future of KeystoneML?
  - Most open issues on Github older than March 2016
  - One active branch
  - Unclear

KeystoneML Summary

Is:

- Open Source ML Pipelining framework
- Easy to use API
- Fast, efficient, scalable
  - Operator and whole-pipeline optimization
  - Distributed execution
- Maximum automation
- Research project

Is not (yet):

- Framework for NN training
- Replacement for specialized statistics programs or libraries, e.g. R
- General-purpose optimizing dataflow system
- Framework for practical use

References

- KeystoneML: http://keystone-ml.org
- Evan Sparks PhD Thesis, UC Berkeley: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-200.pdf
- Sparks, Venkataraman, Kaftan, Franklin, Recht, “KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics”, ICDE, 2017: https://amplab.cs.berkeley.edu/wp-content/uploads/2017/01/ICDE_2017_CameraReady_475.pdf
- Talk by Evan Sparks about KeystoneML: https://www.oreilly.com/learning/keystoneml-optimized-large-scale-machine-learning-pipelines-on-apache-spark
- Apache Spark: https://spark.apache.org/
- PCA overview: https://en.wikipedia.org/wiki/Principal_component_analysis
- Kumar, McCann, Naughton, Patel, “Model Selection Management Systems: The Next Frontier of Advanced Analytics”, SIGMOD 2015: http://cseweb.ucsd.edu/~arunkk/vision/SIGMODRecord15.pdf

Thank you for your attention!

Questions?