
THE FUTURE IS NOW

Scalable Predictive Pipelines with Spark and Scala

Tom van der Weide
Based on a presentation from Dimitris Papadopoulos

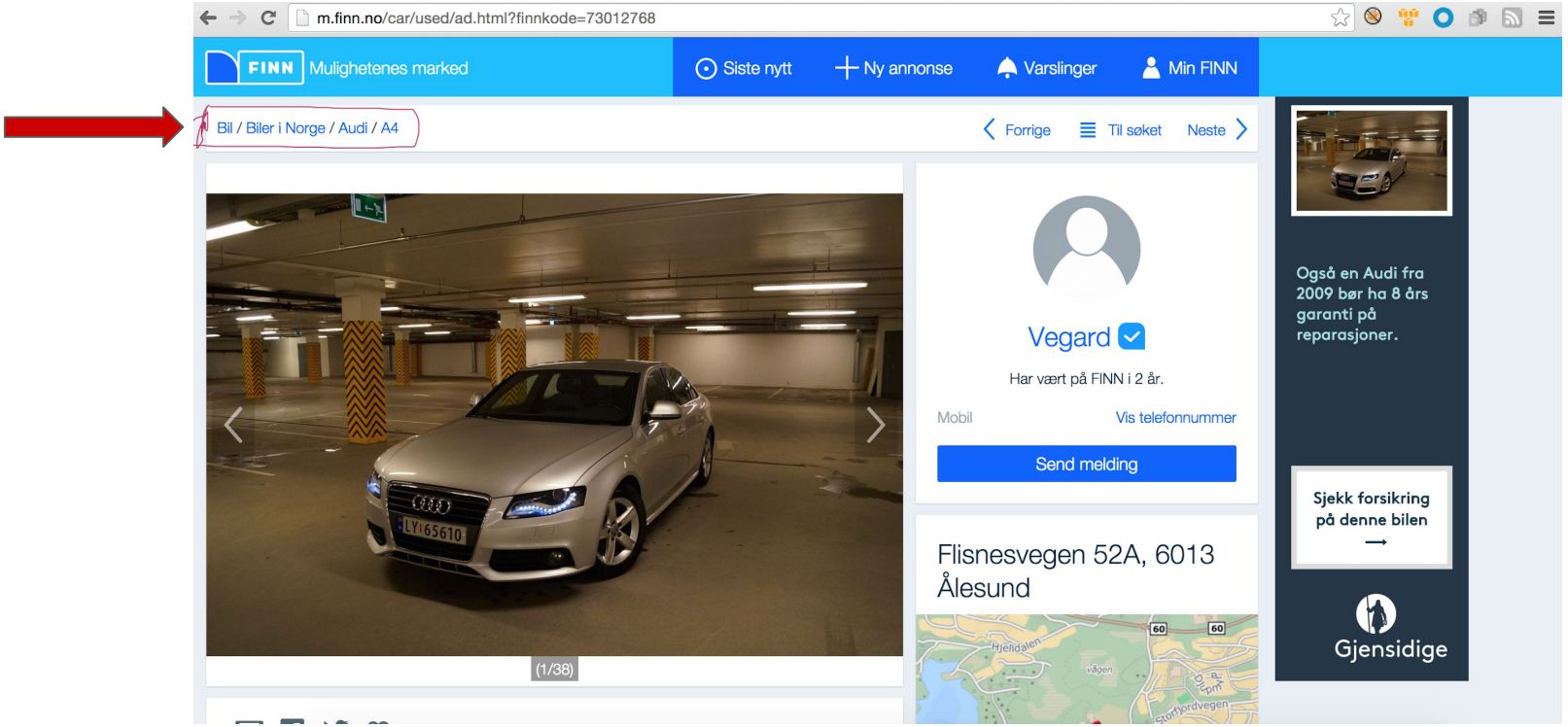
SCHIBSTED
MEDIA GROUP

About Schibsted

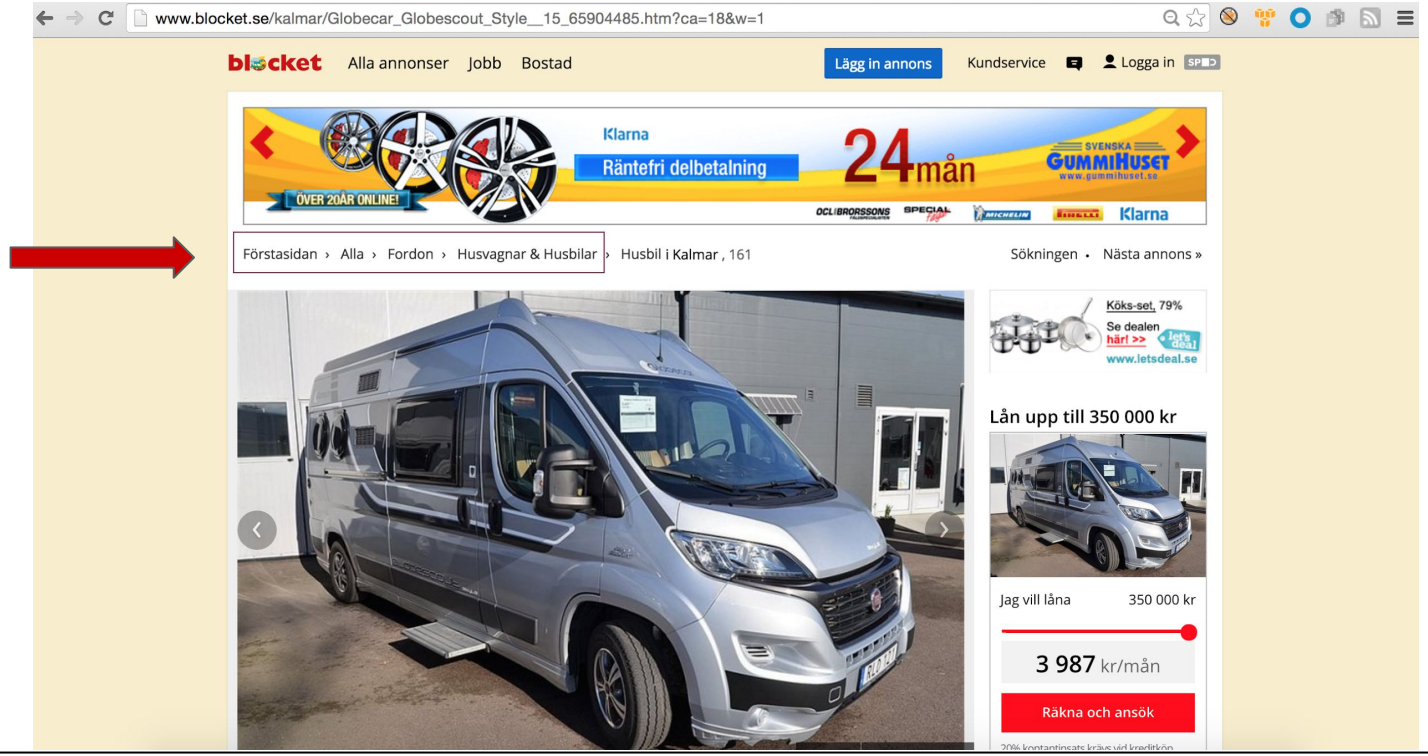
About Schibsted

About Schibsted

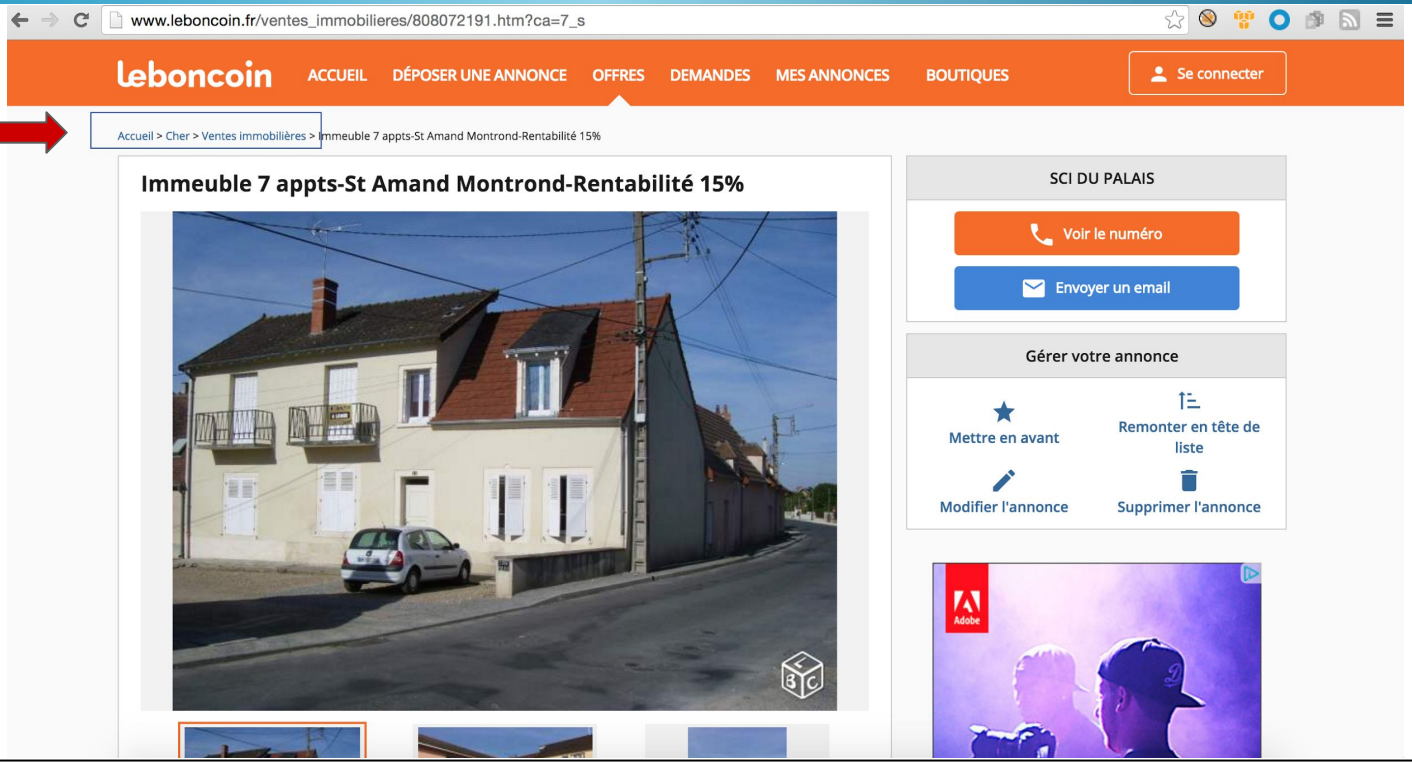
Event Tracking Data



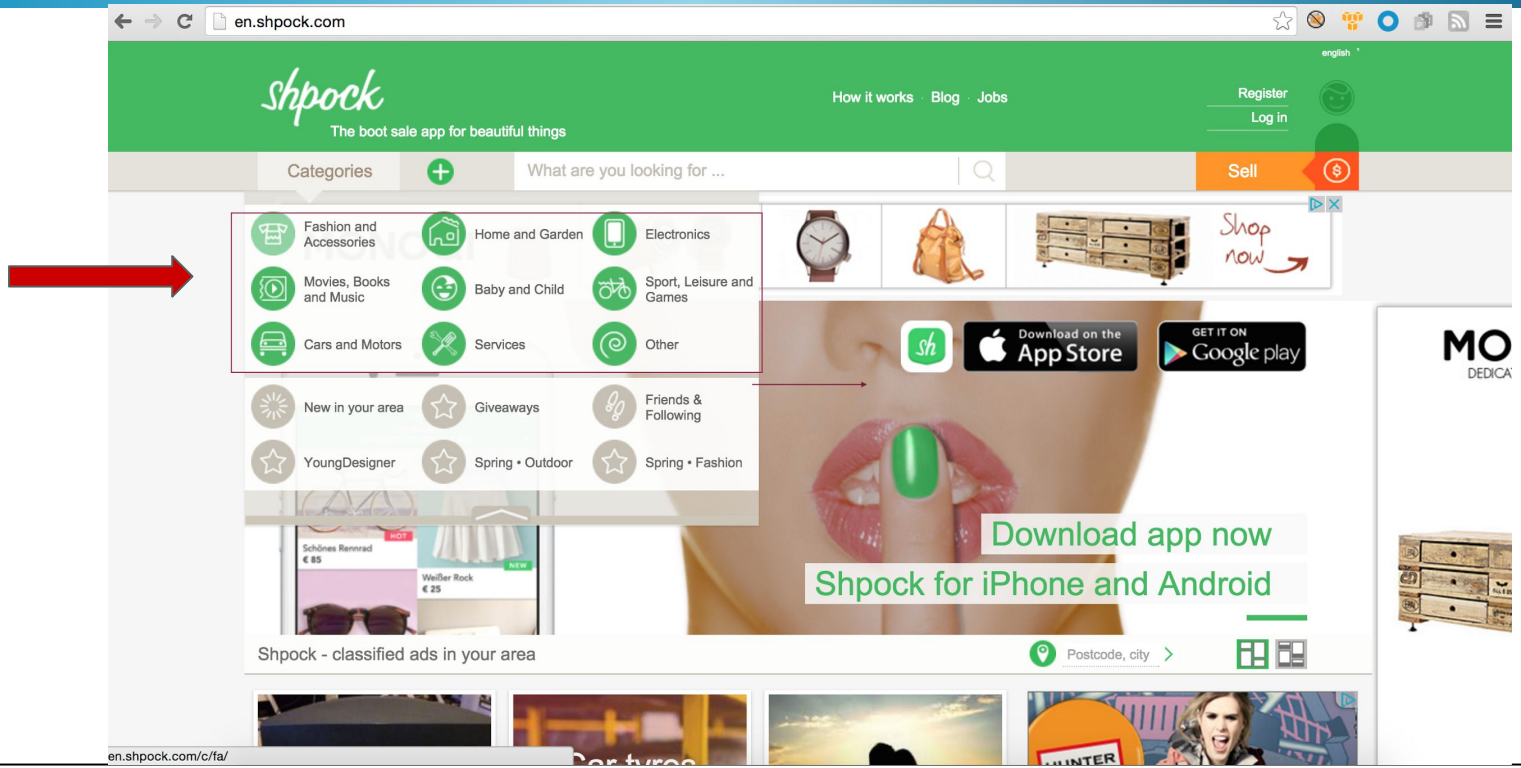
Event Tracking Data



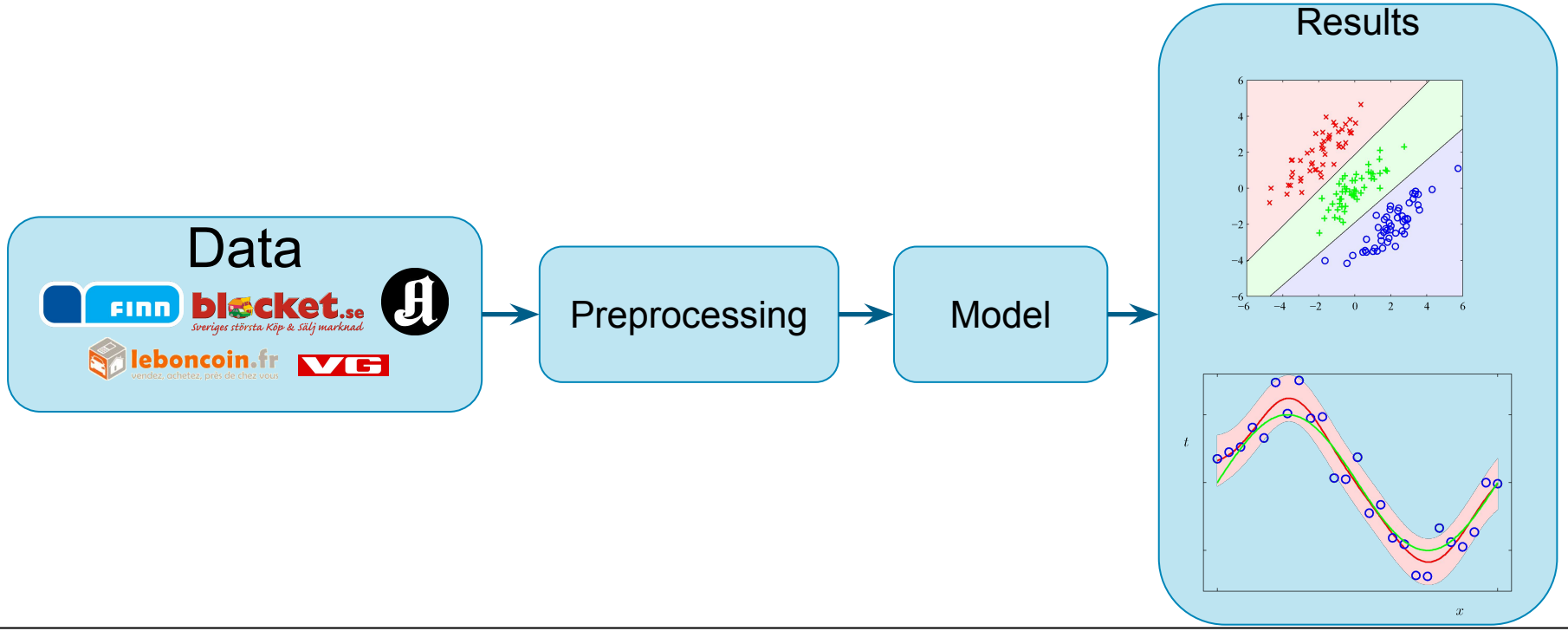
Event Tracking Data



Event Tracking Data



Data Science Tasks

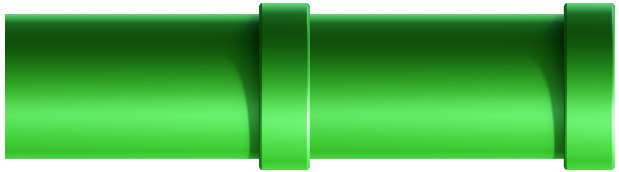


Outline

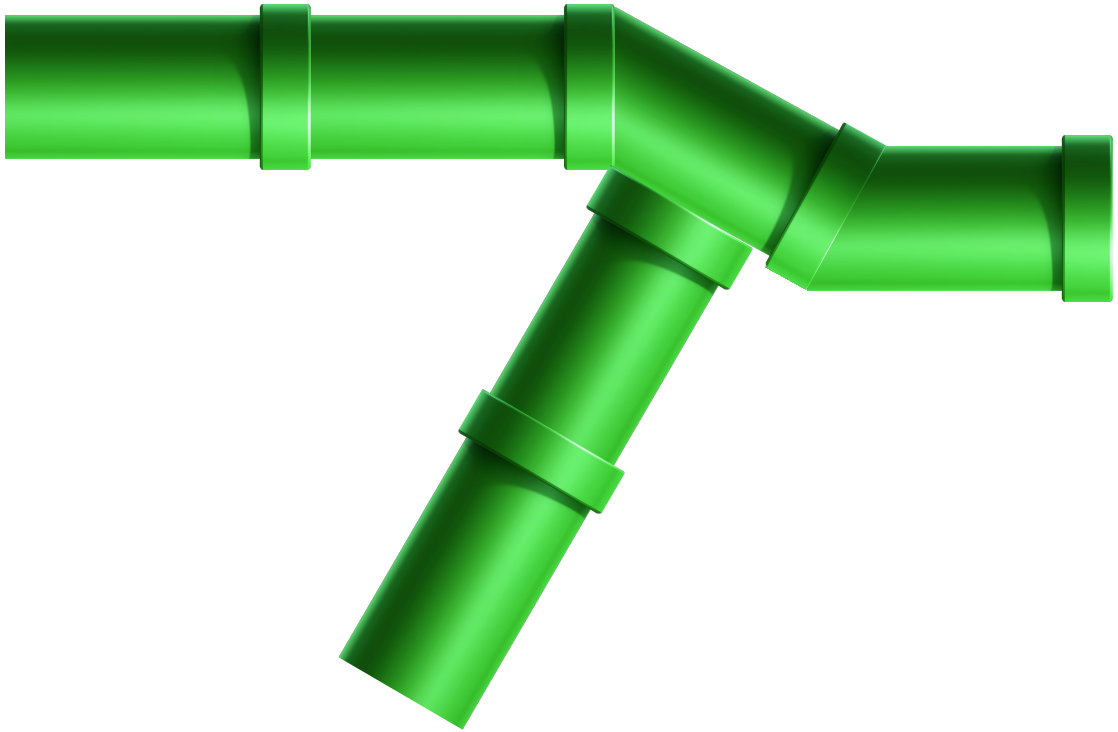
1. Using Spark ML Pipelines

2. Scalable Pipelines

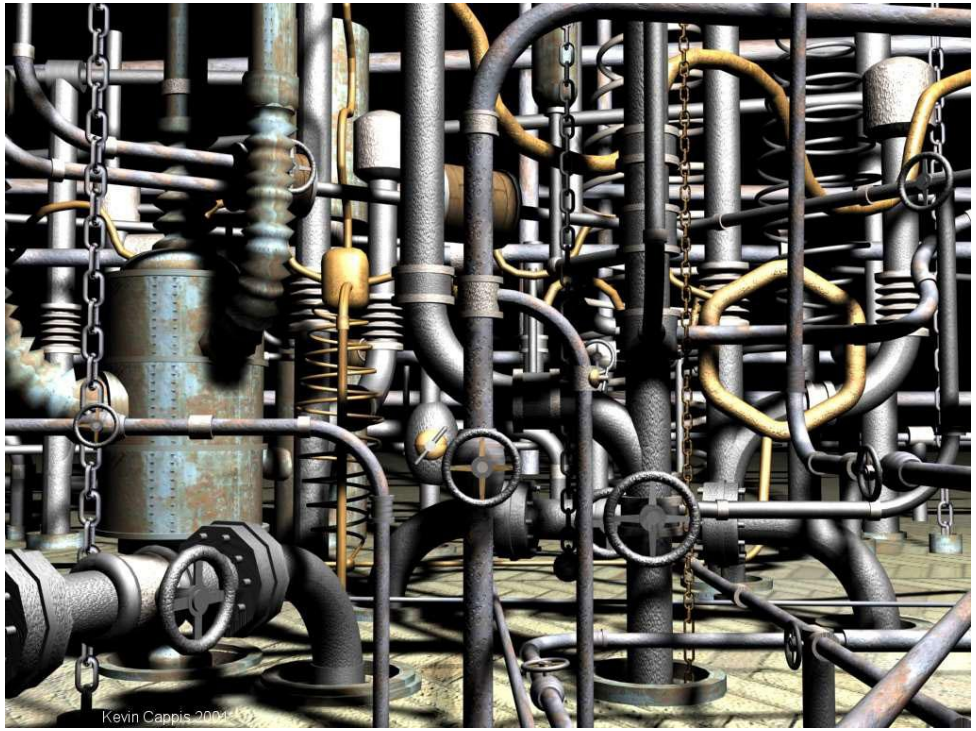
Pipeline



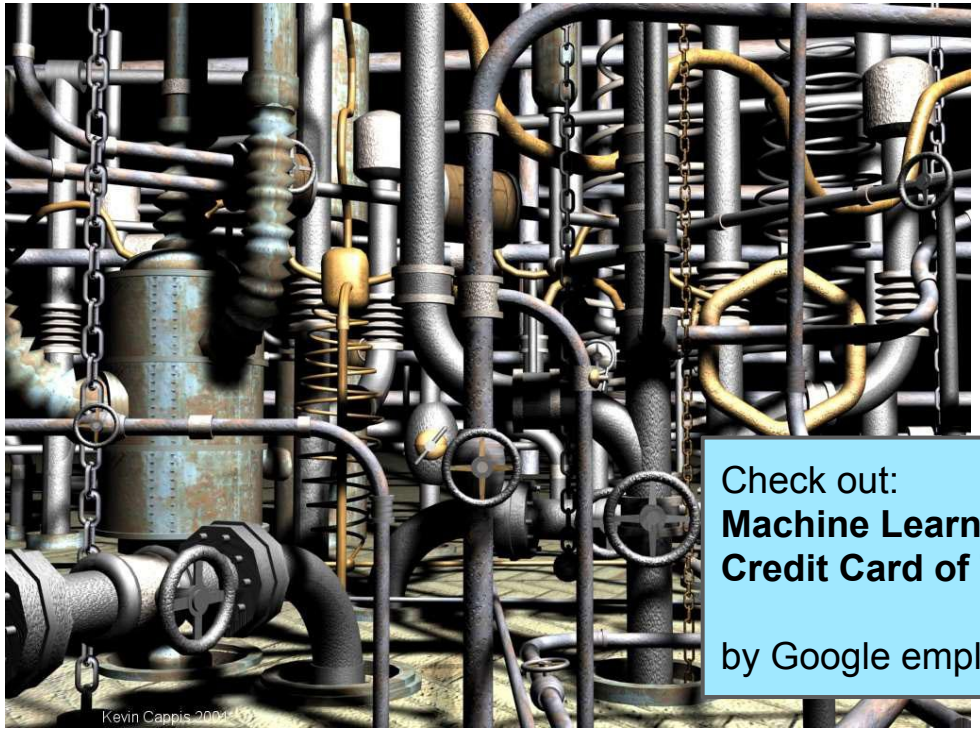
Pipeline



Pipeline



Pipeline



Check out:
Machine Learning: The High-Interest Credit Card of Technical Debt

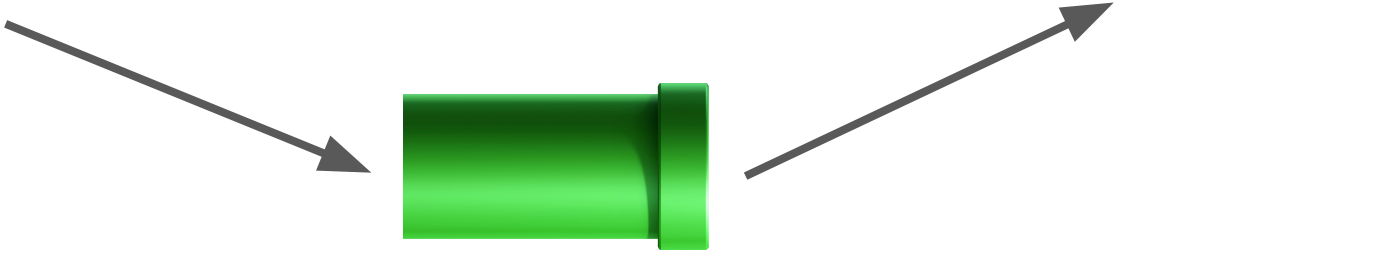
by Google employees

Not a pipe

Pipeline Stage

One or more inputs

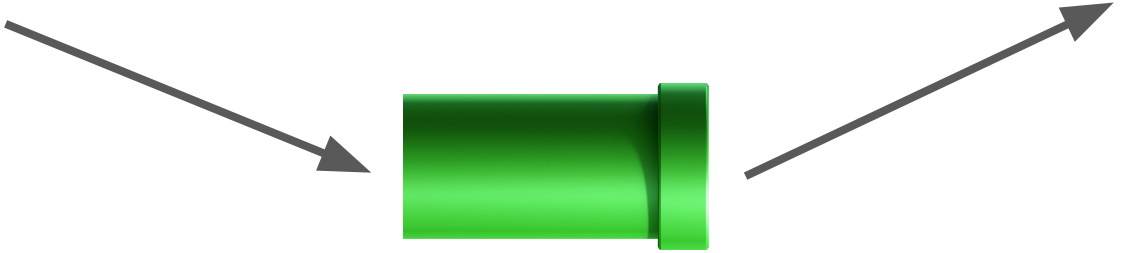
Strictly one output



Pipeline Stage

One or more inputs

Strictly one output

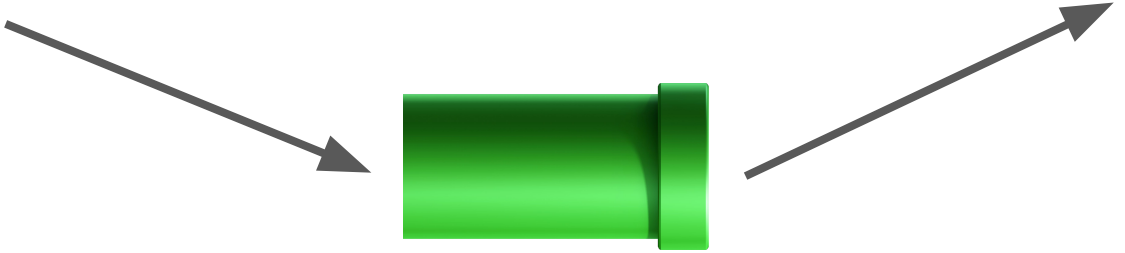


- Closed under concatenation

Pipeline Stage

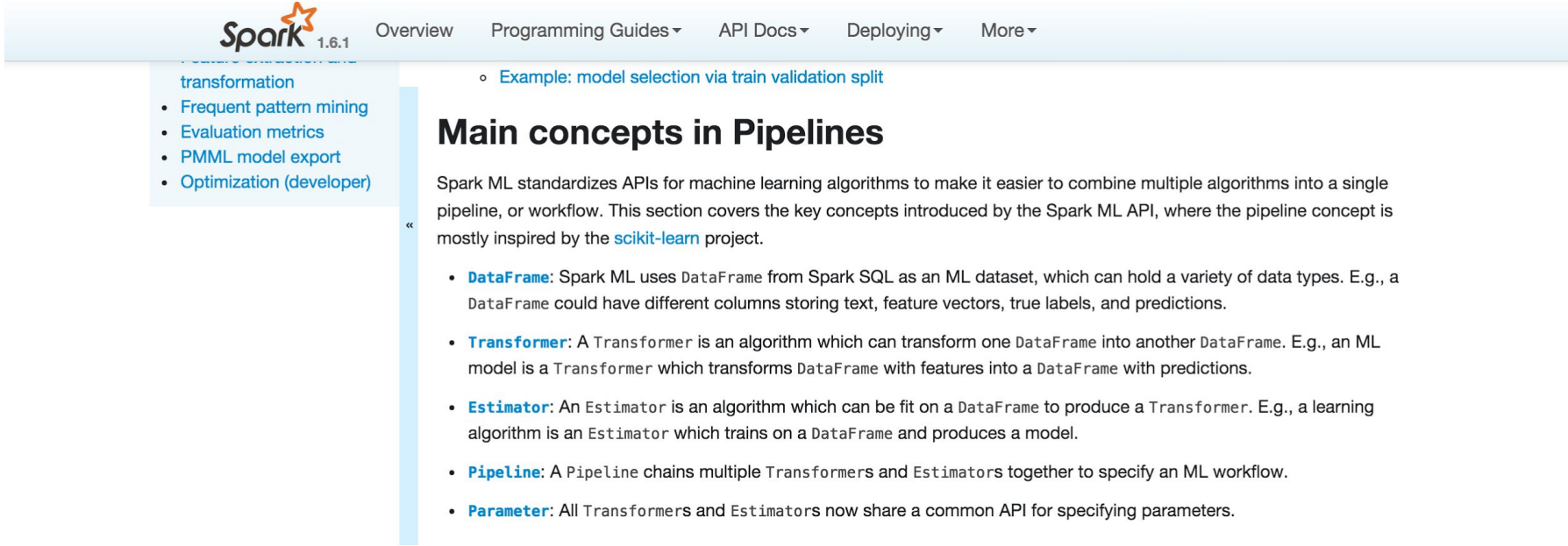
One or more inputs

Strictly one output



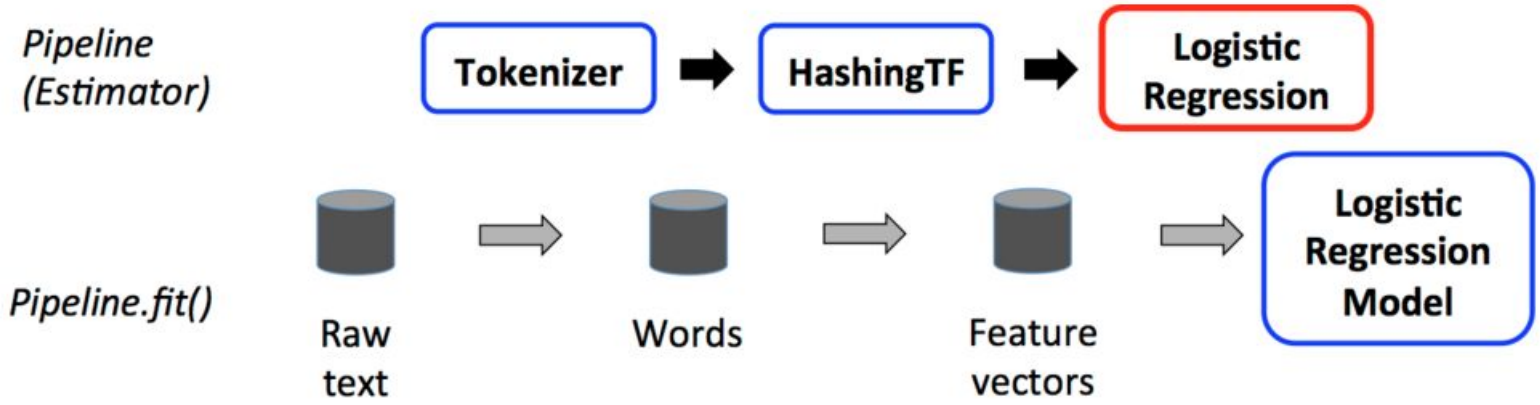
- Closed under concatenation
- Standalone and runnable
- Spark™ ML inside

Spark ML Pipelines



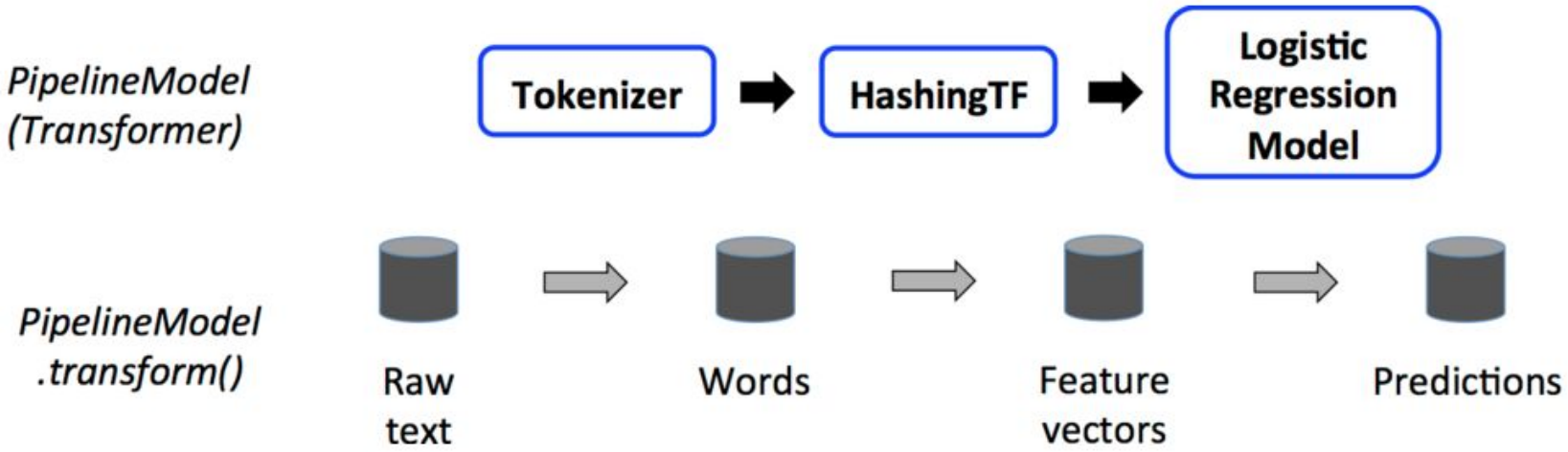
Spark ML Pipelines

Using a Pipeline to train a model

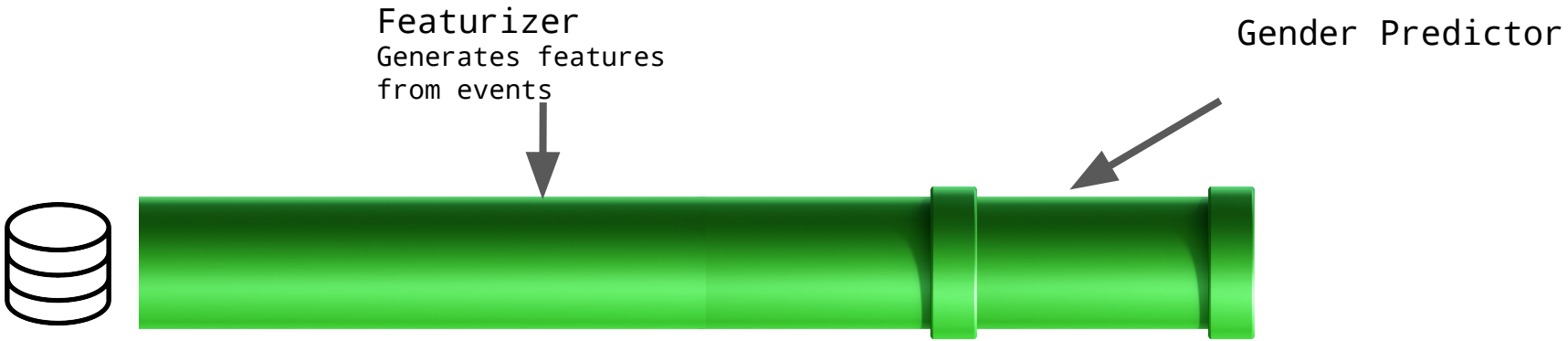


Spark ML Pipelines

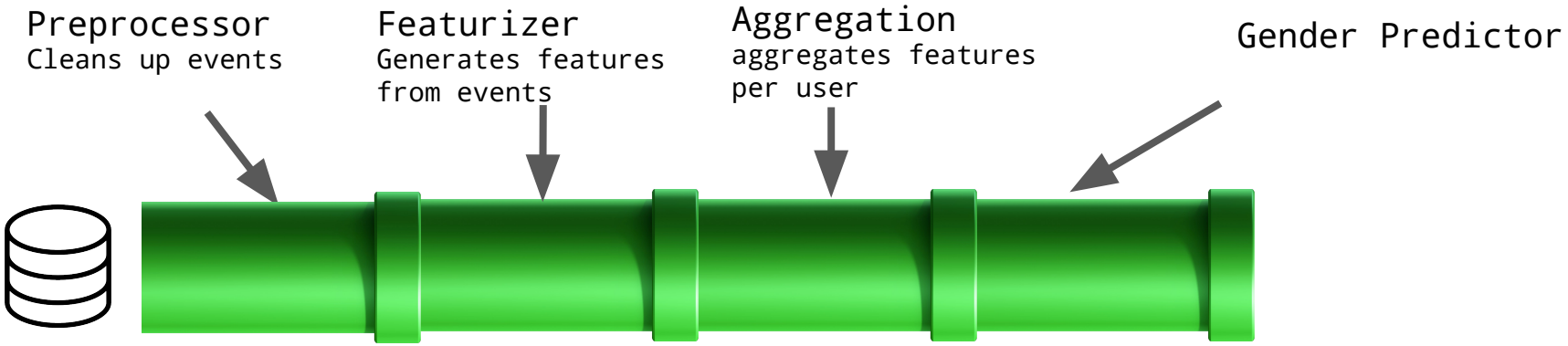
Using a PipelineModel to get predictions



Example: Gender Prediction Pipeline



Example: Gender Prediction Pipeline



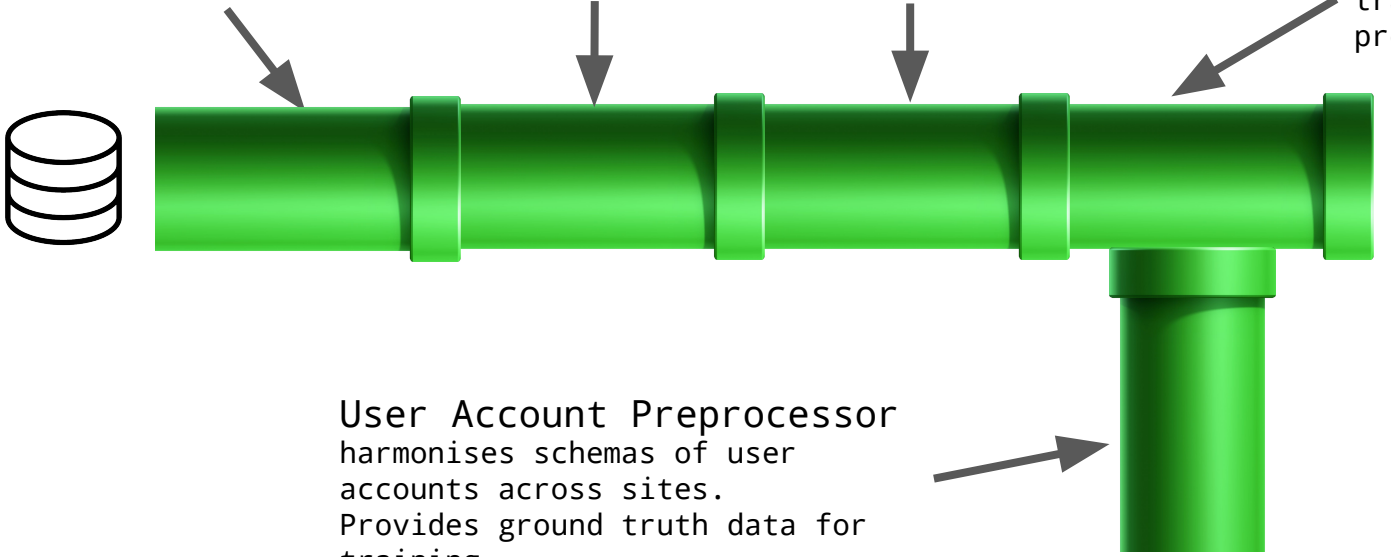
Example: Gender Prediction Pipeline

Preprocessor
Cleans up events

Featurizer
Generates features from events

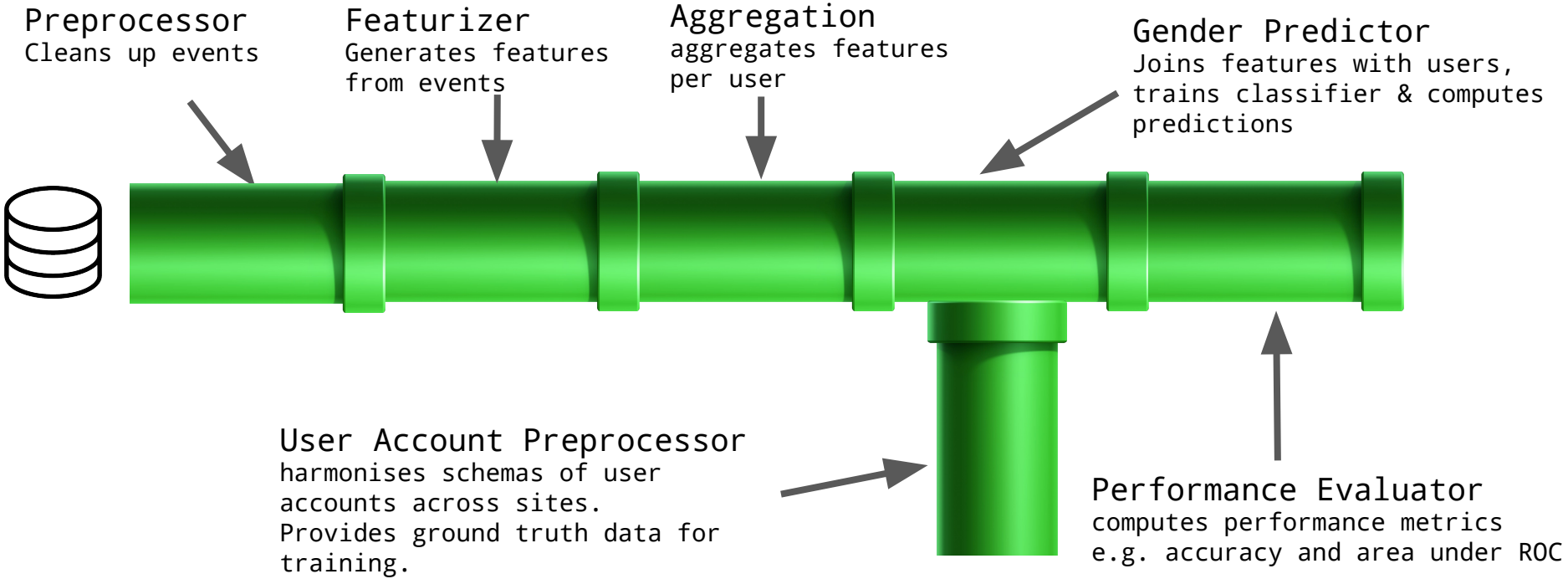
Aggregation
aggregates features per user

Gender Predictor
Joins features with users, trains classifier & computes predictions

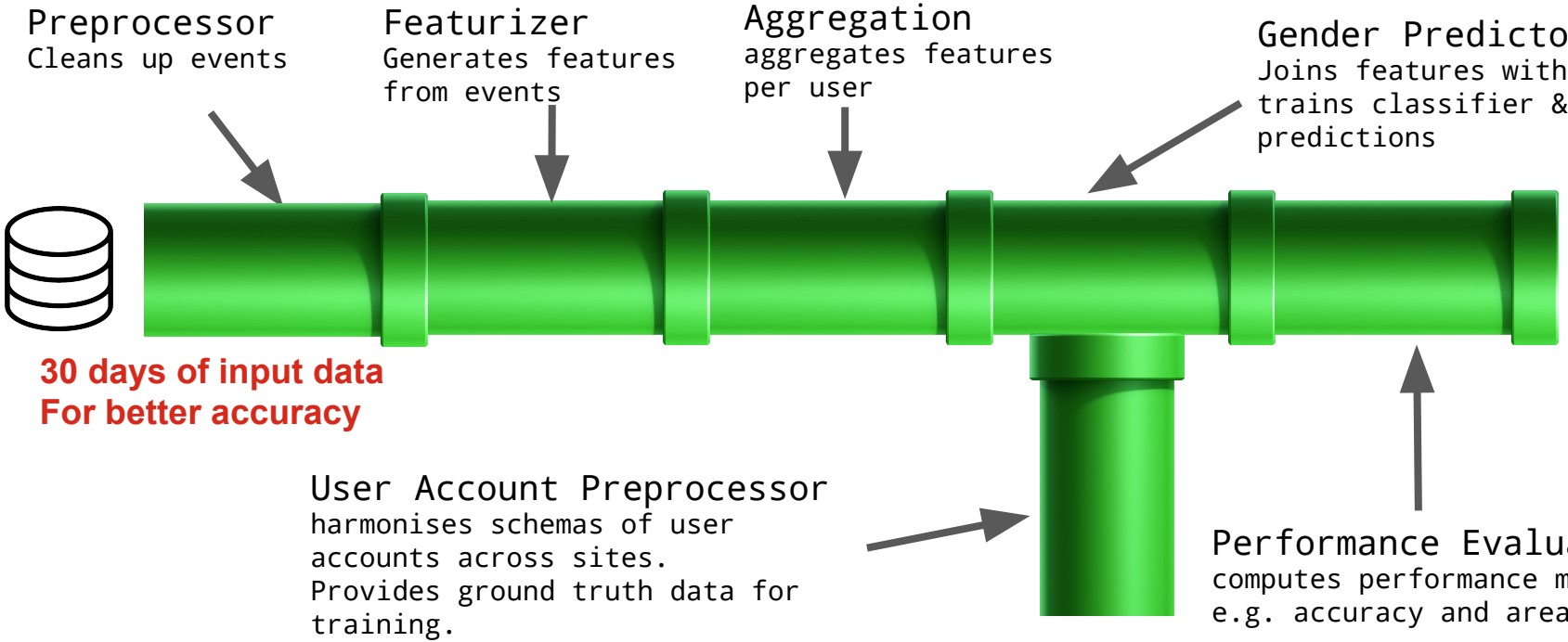


User Account Preprocessor
harmonises schemas of user accounts across sites.
Provides ground truth data for training.

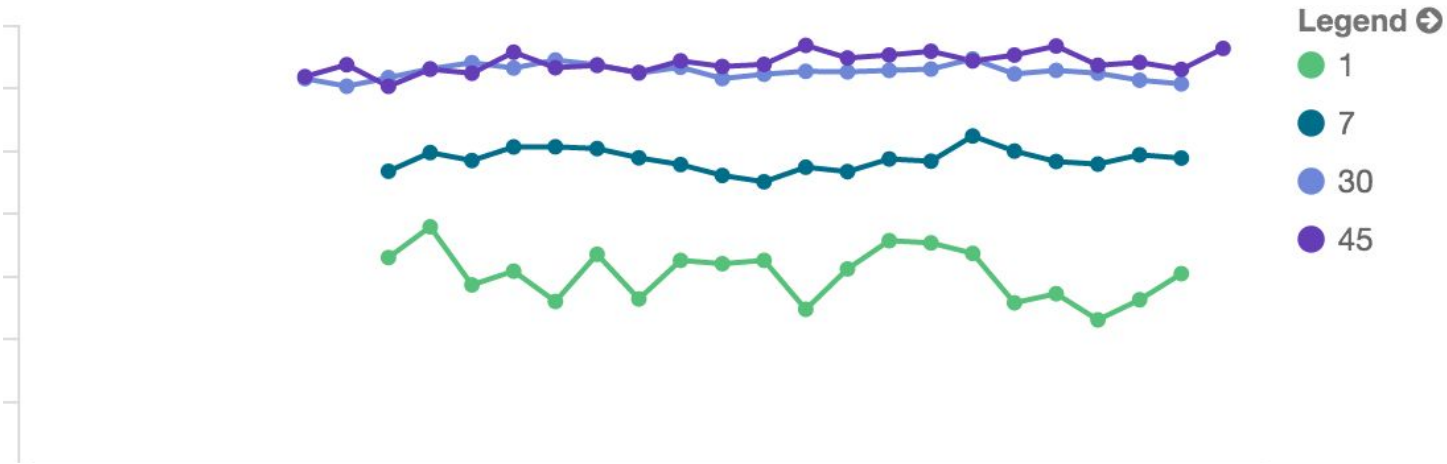
Example: Gender Prediction Pipeline



Scalability pain points

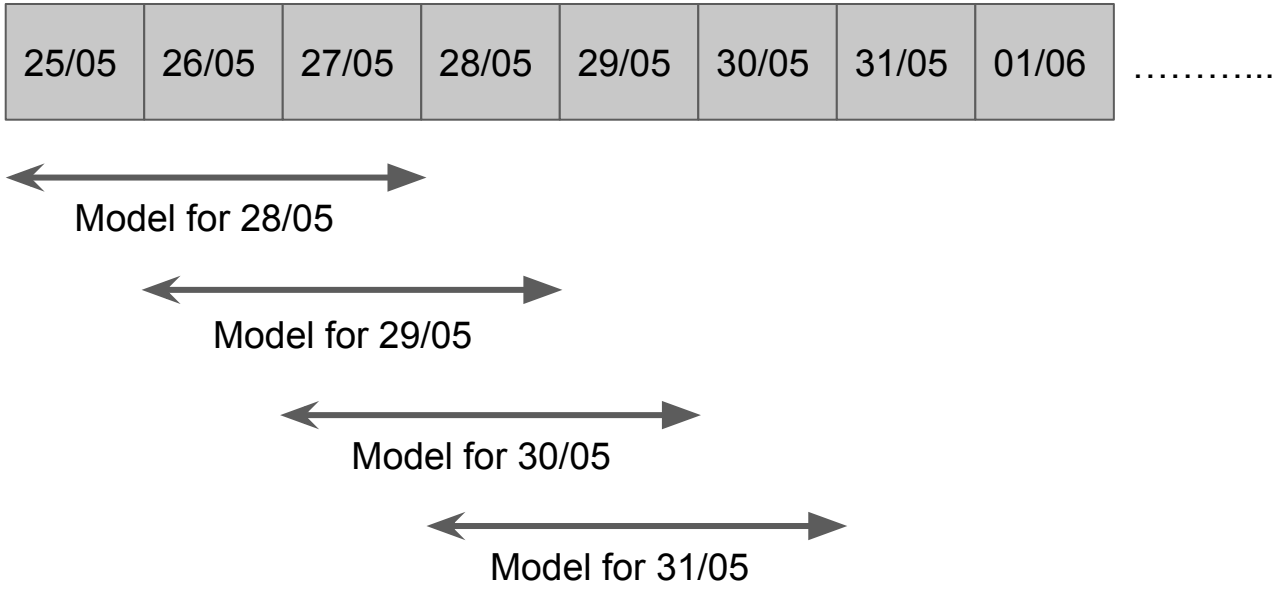


More data for better performance



Redundant processing

Input data per day



Scalable Pipelines: pain points

"What will happen if we try to process
*30 days worth of data (e.g. **6B events**) ???"*

Memory and processing heavy

- In one use-case, for 7 days lookback (~7 x 200M events)
  we used to need 20 Spark executors with 22G of memory each.

Not easily scalable

- As the lookback increases
- As more and more sites are incorporated into our pipelines

Redundant processing

- For K days of lookback, we are repeating processing of K-1 days worth of data,
  when we run the pipeline every day, in a rolling window fashion.

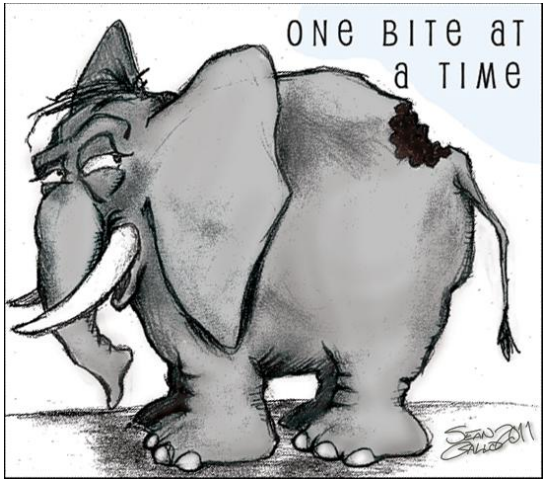
Saved by Algebra

- The operations (*op*) along with the corresponding data structures (S) that we are interested in are ***monoids***.
  - Associative:
    - for all A,B,C in S, (A *op* B) *op* C = A *op* (B *op* C)
  - Identity element:
    - there exists E in S such that for each A in S, E *op* A = A *op* E = A
- Examples:
  - Summation: 1 + 2 + 3 + 4 = (1 + 2) + (3 + 4) = 1 + ((2 + 3) + 4)
  - String array concatenation: [“foo”] + [“bar”] + [“baz”] = [“foo”, “bar”] + [“baz”]

Scalable Pipelines: in monoids fashion

Split work into smaller chunks, cache intermediate results

- Split the aggregations into smaller chunks
  - Persist aggregations per day
- Combine partial aggregations into final feature vectors



Scalable Pipelines: building blocks

User Defined Functions (UDF)

compute a value looking at a single row

e.g. SELECT **concat**('user', user_id) FROM events

User Defined **Aggregate** Functions (UDAF) since Spark 1.5

computation over groups of rows

e.g. SELECT user_id, **AVG**(time_spent) FROM events **GROUP BY** user_id

Example: map with term frequencies

For some user, we already aggregated events per day:

```
val day1 = Map("cars" -> 2.0, "kitchen" -> 1.0, "house" -> 3.0)
val day2 = Map("kitchen" -> 4.0, "house" -> 1.0, "furniture" -> 5.0)
```

Monoid operation: MapAggregator ⊕

i.e. m1 ⊕ m2 ⊕ m3 = (m1 ⊕ m2) ⊕ m3 = m1 ⊕ (m2 ⊕ m3)

m0 = Map()

Result:

```
val together = Map("cars" -> 2.0, "kitchen" -> 5.0, "house" -> 4.0, "furniture" -> 5.0)
```

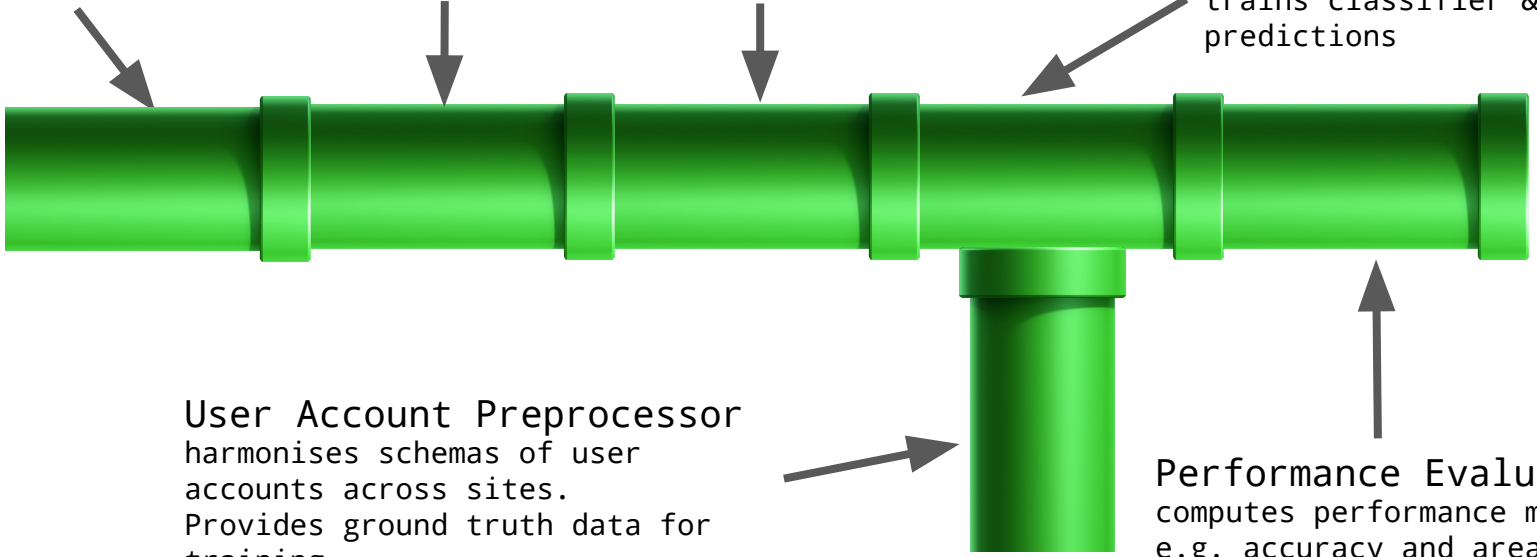
```scala
object MapAggregator extends UserDefinedAggregateFunction {
  type K = String
  type V = Double

  override def initialize(buffer: MutableAggregationBuffer): Unit =
    buffer.update(0, new HashMap[K, V])

  override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
      buffer.update(0, sum(evaluate(buffer), evaluate(input)))
    }
  }

  override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit =
    buffer1.update(0, sum(evaluate(buffer1), evaluate(buffer2)))

  override def evaluate(buffer: Row): Map[K, V] =
    buffer.getAs[Map[K, V]](0)

  private def sum(map1: Map[K, V], map2: Map[K, V]): Map[K, V] =
    map1 ++ (for ((k, v) <- map2) yield k -> (v + map1.getOrElse(k, 0.0)))

  override def dataType = DataTypes.createMapType(StringType, DoubleType, true)
  override def inputSchema = new StructType().add("input", dataType)
  override def bufferSchema = new StructType().add("input", dataType)
  override def deterministic = true
}
```

Buffer for intermediate results

Merge buffers of two executors

Get content from buffer

Aggregation pipeline stage



Preprocessor
Cleans up events

Featurizer
Generates features from events

Aggregation
aggregates features per user

Gender Predictor
Joins features with users, trains classifier & computes predictions

User Account Preprocessor
harmonises schemas of user accounts across sites.
Provides ground truth data for training.

Performance Evaluator
computes performance metrics e.g. accuracy and area under ROC

Scalable Pipelines: Aggregating Events

```
class EventAggregator(override val uid: String) extends Transformer {

  def this() = this(Identifiable.randomUID("eventAggregator"))

  override def transform(dataset: DataFrame): DataFrame = {
    val aggregates = dataset.schema.fields
      .map { field: StructField => field.dataType match {
        case ArrayType(StringType, _) => concat_array_string(col(field.name)).as(field.name)
        case IntegerType | LongType | ShortType | FloatType | DoubleType => sum(col(field.name)).as(field.name)
        case VectorType => SparseVectorAggregator(col(field.name)).as(field.name)
        case MapType(StringType, DoubleType, _) => MapAggregator(col(field.name)).as(field.name)
        case _ => col(field.name).as(field.name)
      }
    }

    dataset.groupBy(col(userId), col(isAnonymous))
      .agg(aggregates.head, aggregates.tail: _*)
  }

  override def transformSchema(schema: StructType): StructType = schema

  override def copy(extra: ParamMap): Transformer = defaultCopy(extra)
}
```

Scalable Pipelines: Aggregating Events

```
class EventAggregator(override val uid: String) extends Transformer {

  def this() = this(Identifiable.randomUID("eventAggregator"))

  override def transform(dataset: DataFrame): DataFrame = {
    val aggregates = dataset.schema.fields
      .map { field: StructField => field.dataType match {
        case ArrayType(StringType, _) => concat_array_string(col(field.name)).as(field.name)
        case IntegerType | LongType | ShortType | FloatType | DoubleType => sum(col(field.name)).as(field.name)
        case VectorType => SparseVectorAggregator(col(field.name)).as(field.name)
        case MapType(StringType, DoubleType, _) => MapAggregator(col(field.name)).as(field.name)
        case _ => col(field.name).as(field.name)
      }
    }

    dataset.groupBy(col(userId), col(isAnonymous))
      .agg(aggregates.head, aggregates.tail: _*)
  }

  override def transformSchema(schema: StructType): StructType = schema

  override def copy(extra: ParamMap): Transformer = defaultCopy(extra)
}
```

It's a Transformer

Scalable Pipelines: Aggregating Events

```
class EventAggregator(override val uid: String) extends Transformer {

  def this() = this(Identifiable.randomUID("eventAggregator"))

  override def transform(dataset: DataFrame): DataFrame = {
    val aggregates = dataset.schema.fields
      .map { field: StructField => field.dataType match {
        case ArrayType(StringType, _) => concat_array_string(col(field.name)).as(field.name)
        case IntegerType | LongType | ShortType | FloatType | DoubleType => sum(col(field.name)).as(field.name)
        case VectorType => SparseVectorAggregator(col(field.name)).as(field.name)
        case MapType(StringType, DoubleType, _) => MapAggregator(col(field.name)).as(field.name)
        case _ => col(field.name).as(field.name)
      }
    }

    dataset.groupBy(col(userId), col(isAnonymous))
      .agg(aggregates.head, aggregates.tail: _*)
  }

  override def transformSchema(schema: StructType): StructType = schema

  override def copy(extra: ParamMap): Transformer = defaultCopy(extra)
}
```

It's a Transformer

DataFrame in , DataFrame out

Scalable Pipelines: Aggregating Events

```
class EventAggregator(override val uid: String) extends Transformer {

  def this() = this(Identifiable.randomUID("eventAggregator"))

  override def transform(dataset: DataFrame): DataFrame = {
    val aggregates = dataset.schema.fields
      .map { field: StructField => field.dataType match {
        case ArrayType(StringType, _) => concat_array_string(col(field.name)).as(field.name)
        case IntegerType | LongType | ShortType | FloatType | DoubleType => sum(col(field.name)).as(field.name)
        case VectorType => SparseVectorAggregator(col(field.name)).as(field.name)
        case MapType(StringType, DoubleType, _) => MapAggregator(col(field.name)).as(field.name)
        case _ => col(field.name).as(field.name)
      }
    }

    dataset.groupBy(col(userId), col(isAnonymous))
      .agg(aggregates.head, aggregates.tail: _*)
  }

  override def transformSchema(schema: StructType): StructType = schema

  override def copy(extra: ParamMap): Transformer = defaultCopy(extra)
}
```

It's a Transformer

DataFrame in , DataFrame out

Aggregating maps of feature frequency counts

Scalable Pipelines: closing remarks

- With User Defined Aggregate Functions, we have reduced the workload of our pipelines by a factor of 20!

Scalable Pipelines: closing remarks

- With User Defined Aggregate Functions, we have reduced the workload of our pipelines by a factor of 20!
- Obvious gains: freeing up resources that can be used for running even more pipelines, faster, over even more input data

Scalable Pipelines: closing remarks

- Needless to say, more factors contribute towards a scalable pipeline:
  - Performance tuning of the Spark cluster
  - Use of a workflow manager (e.g. Luigi) for pipeline orchestration

Scalable Pipelines: closing remarks

- Needless to say, more factors contribute towards a scalable pipeline:
  - Performance tuning of the Spark cluster
  - Use of a workflow manager (e.g. Luigi) for pipeline orchestration

- But each one of these is a topic for a separate talk

Shameless plug

We are hiring!

Across all our hubs

in London, Oslo, Stockholm, Barcelona

for Data Science, Engineering, UX and Product roles

https://jobs.lever.co/schibsted

spt-recruiters@schibsted.com

Q/A

Thank you!