STREAMLINE. FAST REACTIVE ANALYTICS

D3.2 - A High-Level Declarative Language for ML

Project Number	688191
Project Acronym	STREAMLINE
Nature	R: Report
Dissemination Level	Public
Work Package	WP3
Due Delivery Date	November 2017
Actual Delivery Date	November 2017
Lead Beneficiary	SICS
Authors	Alireza Rezaei Mahdiraji Bonaventura Del Monte Jeyhun Karimov Behrouz Derakhshan Tilmann Rabl Seif Haridi

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 688191.

Executive Summary

This deliverable describes a high-level declarative language for scalable machine learning on big data which decreases the effort and complexity of implementing machine learning algorithms on big data frameworks. The high-level declarative language is based on Emma language. Emma is a domain specific language embedded in Scala, which abstracts system-specific aspects of the big data frameworks and thus makes programming in the data processing frameworks easier. We build a library of scalable machine learning algorithms based on the abstractions offered by Emma language. We demonstrate how Emma can provide an easy access to new machine learning algorithms such as the one developed in the work package 2 of the Streamline project. In particular, we show how XGBoost algorithm from the work package 2 can be easily reused by Emma. In comparison to Apache Flink, the user needs much fewer lines of code to implement algorithms such as XGBoost in Emma. This is due to the high-level abstractions provided by Emma that facilitate expressing machine learning algorithms.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations and Acronyms

API	Application programming interface
AST	Abstract syntax tree
IR	Intermediate representation
UDF	User defined function
ANF	Administrative Normal Form
ML	Machine Learning
LNF	Let Normal Form
DSL	Domain Specific Language

1 Introduction

In this deliverable, we provide a high-level declarative language for Machine Learning. First, we provide an overview of the requirements of advanced data analysis on large datasets. Then, we introduce a programming model and easy to use declarative language for expressing different machine learning algorithms and use-cases. The latest version of the high-level language can be found in the Streamline Github repository:
https://github.com/streamline-eu/streamline-ml-language

Big Data drastically changed the data analytics business. They introduced new challenges to traditional analysis such as the need of advanced analytics to extract meaningful insights as well as the handling of large, complex, and unstructured data. Whereas Machine Learning (ML) is a solution to perform advanced analytics tasks, frameworks such Apache Flink and Apache Spark emerged as the de-facto solution to perform Big Data processing in the Cloud.
Nowadays, data scientists are expected to master both ML and Big Data frameworks to get actionable insights from the business data. However, those requirements are difficult to meet because: 1) ML and related analytics frameworks require a specific skill set (e.g., bayesian statistics, knowledge of tools like R or Matlab) and 2) to obtain an actual performance gain, a deep knowledge of low-level details of the selected Big Data framework are required, e.g., selecting the best join method. This results in an exponential increase of complexity that data scientists have to face because they have to match algorithms expertise with a deep knowledge of the chosen Big Data framework. As a result, considering both aspects hinder daily productivity of data scientists.
To overcome abovementioned problems, we propose a library called *Emma Machine Learning*, i.e., a library for scalable data analysis, which we build on top of the *Emma* language. Emma is quotation-based embedded Domain Specific Language (eDSL) [8], which is shipped as a Scala library. Our library benefits from Emma eDSL as it provides the toolbox to build an high-level declarative language for Machine Learning, which abstracts system-specific aspects of the Big Data Framework (e.g., choosing the best join strategy). In our prototype, we provide implementation for a range of well-known machine learning algorithms such as linear regression, naive bayes, and k-means. Furthermore, we provide an implementation of the XGBoost algorithm from WP2, which we reimplement using Emma high-level abstractions for ML. The *key insight* regarding the use of Emma is the ease of programming of ML algorithms on top of dataflow engines (e.g., Apache Flink).

This document is structured as follows: we first provide a brief description of Emma (Section 2), its internal components (Section 3), and its API (Section 4). Then, we describe Emma Machine Learning library (see Section 5) and we show how we implement one of the algorithms described in D2.1 (see Section 6). Finally, we summarize the main contribution of this deliverable in the conclusion section (see Section 7).

2 Brief description of Emma

As reported on its main website[1], Emma aims to improve productivity of data scientists by hiding parallelism aspects behind a high-level, declarative API as well as through a deep reuse of native Scala syntax and constructs. Emma supports state-of-the-art dataflow engines (e.g., Apache Flink and Apache Spark) as backend co-processors.

The first observation that leads to Emma development deals with the acknowledgment of known drawbacks in the programming interfaces of state-of-the-art dataflow engines. Those drawbacks are: 1) the requirement of a deep knowledge of those execution engines in order to obtain a performance boost, 2) the APIs of those executions engines are hard to read due to abstraction leaks (e.g., native support to iterations in Apache Flink), and 3) the lack of a holistic dataflow optimizer, which leads to manual, error-prone hand-tuning of any execution plan.

To overcome above pitfalls, Emma provides a declarative API for parallel collection processing. Programs written in Emma leverage on linguistic re-use of native Scala features (e.g., for-comprehension, case-classes, and pattern matching) [1]. Emma analyzes and holistically optimizes input programs for data-parallel execution on a co-processor engine such as Apache Flink or Apache Spark [1].

Emma APIs differs from the high-level APIs of those engines because they are deeply embedded in the host language. In contrast, the runtime-centric evolution of the high-level APIs of those engines leads to their shallow embedding into the host language, which results in more complexity. The Emma choice of deep embedding of its APIs instead results in a natural reuse of native constructs of the native language and it enables the access to the program AST on which Emma compiler can perform non-trivial optimization.

In the following two sections, we provide insights about the internals of Emma and its high-level declarative APIs.

[1] http://emma-language.org/

3 Emma Internal Components

In this section, we describe the internal components of *Emma*, i.e., the Emma Compiler Infrastructure.
Emma's users write their programs in the Emma Source language and compile them using the Emma Compiler. The Emma Compiler performs two main operations on the source code. First, it translates (lifts) the input code written in the *Emma Source language* into the *Emma Core language*, which is an *Intermediate Representation* (IR) of the program. Finally, Emma optimizes and translates (lowers) such IR into the abstractions of the targeted Runtime Execution Engine (e.g., Apache Flink or Apache Spark). Figure 1 summarizes the above steps. In the rest of this Section, we explains different parts of Emma compiler.

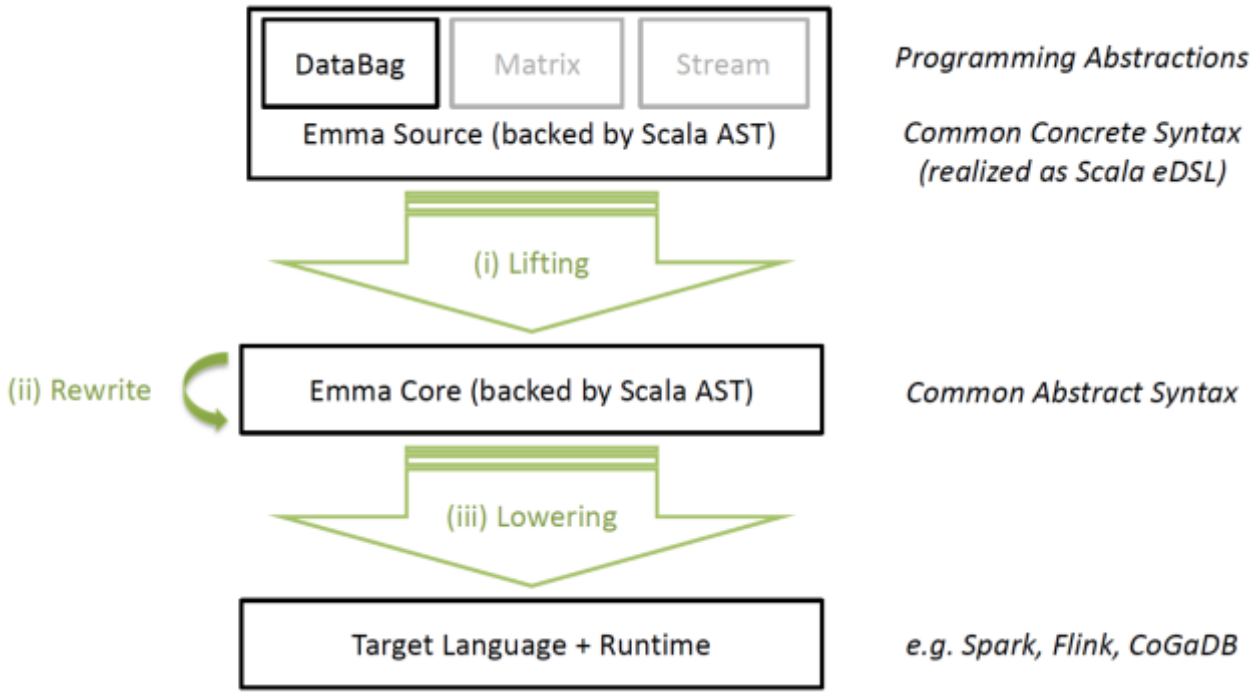

Figure 1. Emma Compiler Infrastructure.

Emma Source Language

The source language of Emma consists of a subset of the Scala programming language. The source language enforces the following restrictions to ensure that the quoted source code can be translated and efficiently executed on dataflow engines:

1. Object mutation is not allowed to prevent race conditions.
2. Closure are modified to ensure serializability of lambda functions.
3. Class and object definitions may not be externally defined.
4. Local method definitions are required to disallow general recursion.

The Emma Source code does not allow for non-local control flow (e.g., throwing and handling exceptions, refutable pattern matching) because it involves more complex transformations.

Furthermore, the Emma Source Language requires the following constraints to semantically support laziness and side effects due to Administrative Normal Form (ANF) of the Emma Core Language:

1. The only side-effecting operations allowed are the ones defined by the Emma API. (The Emma compiler might eliminate other side effects through code reordering.)
2. The source code should not contain side effects in call-by-name code because intermediate terms are assigned to Scala “vals” and eagerly computed.

Emma Core Language

Emma uses an intermediate representation format, which we call *Emma Core language*. The major advantage of using such an IR instead of only the Emma Source language is that it enables a wider range of optimizations. The Emma Core language is indeed more suitable for reasoning about non-trivial code optimizations because it hides the complexity of pure Scala Abstract Syntax Trees (ASTs). The Emma Core language is an IR based on Single Static Assignment (SSA) [4, 5]. As a result, Emma IR makes the dataflow of the program more explicit because it contains expressions in administrative normal form (ANF) [6] or let-normal form (LNF) [7].
In the following, we describe how the Emma Compiler lifts Emma Source ASTs to the Emma Core language and which optimizations it performs on the intermediate representation.

Emma Compiler

Emma Compiler performs three main tasks:

1. it lifts the Abstract Syntax Tree (AST) of the code written in Emma Source language to Emma Core language (i.e., the Emma Intermediate Representation),
2. it optimizes such IR through banana-split and fold-fusion optimization techniques [1], and
3. it translates the optimized IR into a physical execution plan of the target execution engine (e.g., Flink and Spark).

The Emma Compiler is based on a pipeline of transformations, which operate on the Scala Abstract Syntax Tree. The Emma Compiler provides a set of transformations, which can be chained together to build a pipeline. These transformations lift the code written in the Emma Source language to the Emma Core language, which is the Emma intermediate representation format.

Table 1 summarizes the transformations that Emma performs to lift the AST of Emma Source to Emma Core representation.

Table 1. Transformations bundle available in Emma

Language Transformation	Name of the Transformation	Goal
Source -> $Source_{ANF}$	Administrative Normal Form	Destruct composite terms such that arguments are simple literals or identifiers.
Source -> LNF	Direct-Style control flow	Eliminate while and do-while loops, if-else branches, and local variables. Replace them with nested, mutually recursive local methods and calls.
Source -> LNF	Let-Normal Form	Composition of Administrative Normal Form and Direct-Style.
LNF -> Core	Resugar	Add comprehension syntax as a first-class citizen in by re-sugaring monad operators for a monad type M.
Core -> LNF	Desugar	Opposite of Resugar.
LNF -> LNF' LNF' -> LNF Core -> Core' Core' -> Core	Reference (Un)Inlining	Expand / reduce the type of terms that can appear in the expr position of let blocks by inlining references.
Core' -> Core'	Normalize	Normalizes comprehensions for a monad type M by iteratively unnesting nested comprehensions appearing in the right term of a generator or in the head position.
Source -> Core	Lift	The canonical way to lift Emma expressions from the Source to Core language.

The lift transformation is based on quotation. This means that a Scala macro accesses the Scala AST of the quoted source code and then Emma compiler performs several transformations on this AST to generate a suitable intermediate representation, i.e., the Emma Core Language.

In Figure 2, we summarize the succession of steps that are involved in the lift transformation.

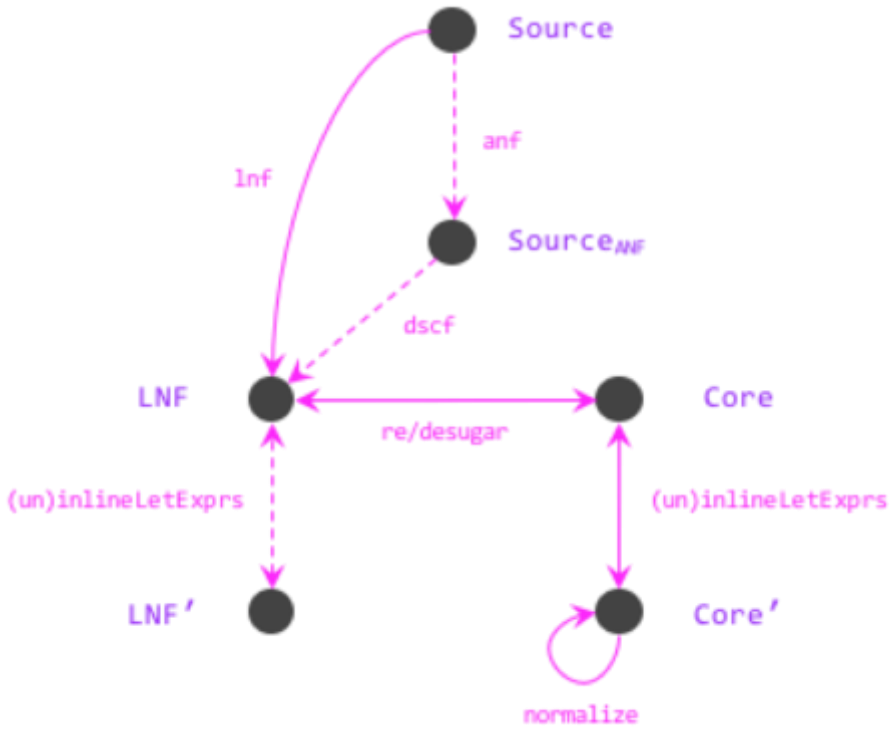

Figure 2. Lifting from Source to Core Emma language

The Emma Compiler further optimizes and lowers the IR to the desired runtime, e.g., Apache Spark or Apache Flink. The lowering step on the Emma Core Language transparently inserts partial aggregates when possible and it removes expensive group materializations.
Candidates for this rewrite are **groupBy** term, which meet the two following constraints:

1. all occurrences of the group values are consumed by a fold
2. a consuming fold does not have data dependencies

Upon the triggering of an optimization, Emma compiler replaces the groupBy with an **aggregateBy** operator whose task is to fuse the group construction performed by the **groupBy** and the subsequent fold applications on the group values. Those rewrites are the implementations of the following optimizations: banana split and fold-build fusion.
The banana split abstracts loop fusion for arbitrary structural recursion. In a nutshell, it states that a pair of folds can be rewritten as a fold over pairs.
The fold-build fusion enables a rewrite called cheap deforestation in functional programming languages. Briefly, fold-build fusion states that an operation that constructs a bag can be pipelined with a subsequent fold over the constructed value.

4 Emma APIs

In this section, we explain the basic APIs of Emma which based on *DataBag*, the core abstraction of Emma. DataBag models bags in union representation. It represents a distributed collection of elements of particular type that do not have particular order and may contain duplicates [1].

The main difference between DataBag and Spark's RDD or Flink's DataSet is that DataBag type is a proper monad. Thus, we can use for-comprehension syntax for joins and cross products in Emma.

In the following, we dive into the details of different parts of the DataBag abstraction.

4.1 Structural recursion

Listing 1 shows the structural recursion methods of DataBag. To understand the meaning behind structural recursion in Emma, we assume that instances of DataBag[A] are constructed exclusively by the following constructors:

- *emp* denotes the empty bag,
- *sng(x)* denotes a singleton bag with exactly one element *x*,
- *uni(xs, ys)* denotes the union of two existing bags *xs* and *ys*.

```
/** DataBag Class **/

def fold[B: Meta](agg: Alg[A, B]): B
  def fold[B: Meta](zero: B)(init: A => B, plus: (B, B) => B): B =
    fold(Fold(zero, init, plus))

/** DataBag Class **/
```

Listing 1. Structural recursion methods of DataBag API

Consider a case where we have a bag xs = {{3, 5, 7}} and we want to compute the sum of the elements in the bag. We can define this operation with a higher-order function called fold. Union constructor is used for the inner nodes of the tree applications and Empty and Singleton constructors are used for the leafs.

The main intuition is that we can represent each bag *xs* as a binary tree of constructor applications. We elaborate on the example above to highlight the importance of this view on bag computations from a data management perspective. Let *xs* be partitioned over two nodes: xs_1 =

{{3,5}} and xs_2 = {{7}} (Figure 3). Conceptually, the result is still *xs* = uni xs_1 xs_2. However, the *uni* operator is evaluated only if we have to materialize xs in a single node.

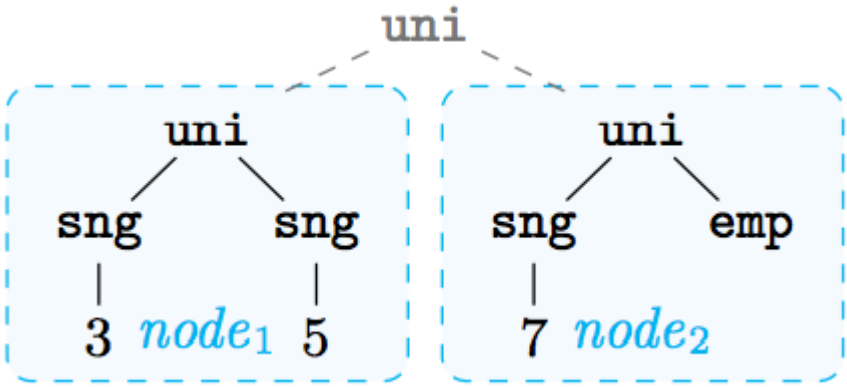

Figure 3. Partitioned bag over two nodes.

We push fold argument functions to the nodes containing *xs1* and *xs2* and apply the fold locally. Then, we ship the resulted values instead of *xs*. Figure 4 shows the overall intuition behind this.

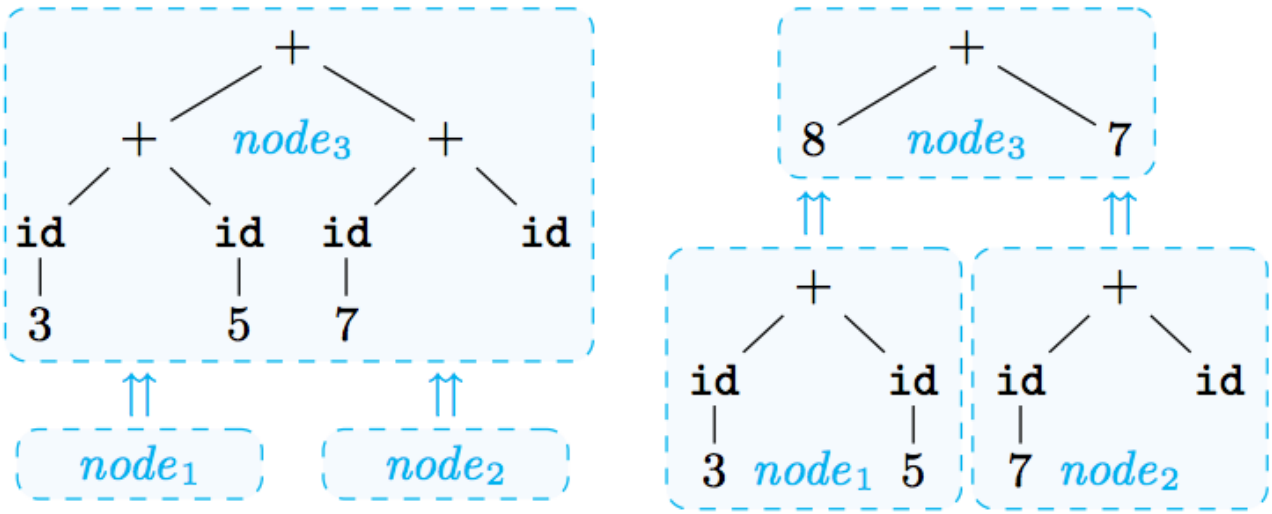

Figure 4. Applying fold function in each node and gathering intermediate results.

Structural recursion on bags in union representation works in several steps. First, it deconstructs the input DataBag[A] instance. Second, Emma replaces the constructors with the corresponding UDFs. Finally, Emma evaluates the resulting expression.

4.2 Monads

Listing 2 shows the monads methods of DataBag API. Instead of including the binary operators such as join and cross in the API level, we propose a layered intermediate representation where

dataflow expressions found in the original AST are converted and transformed into a declarative, calculus-like representation called monad comprehensions. One advantage of monad comprehension is that it combines functional programming and database systems in one interface. Moreover, it is exploited as IR for database queries [2, 3]. This allows great opportunities for us to utilize dataflow engines with a declarative syntax similar to the Select-Project-Join style.

```
/** DataBag Class **/

 def map[B: Meta](f: (A) => B): DataBag[B]
  def flatMap[B: Meta](f: (A) => DataBag[B]): DataBag[B]
  def withFilter(p: (A) => Boolean): DataBag[A]

/** DataBag Class **/
```

Listing 2. Monads methods of DataBag API

4.3 Grouping and Union

Listing 3 and 4 show the grouping and union methods of DataBag API. Besides from union operators, Emma provides duplicate elimination feature as well (distinct method).

```
/** DataBag Class **/

  def groupBy[K: Meta](k: (A) => K): DataBag[Group[K, DataBag[A]]]

/** DataBag Class **/
```

Listing 3. Grouping method of DataBag API.

```
/** DataBag Class **/

def union(that: DataBag[A]): DataBag[A]
  def distinct: DataBag[A]

/** DataBag Class **/
```

Listing 4. Set operations of DataBag API.

4.4 Partition based operators

Listing 5 shows the partition based operators of the DataBag API. The sample method creates a sample of up to `k` elements using reservoir sampling initialized with the given `seed`. If the collection represented by the DataBag instance contains less than `k` elements, the resulting collection is trimmed to a smaller size. The method should be deterministic for a fixed DataBag instance with a materialized result. In other words, calling `xs.sample(n)(seed)` two times in succession will return the same result. The result, however, might vary between program runs and DataBag implementations.

The zipWithIndex method zips the elements of this collection with a unique dense index. The method should be deterministic for a fixed DataBag instance with a materialized result. In other words, calling `xs.zipWithIndex()` two times in succession will return the same result. The result, however, might vary between program runs and DataBag implementations.

```
/** DataBag Class **/

def sample(k: Int, seed: Long = 5394826801L): Vector[A]
def zipWithIndex(): DataBag[(A, Long)]

/** DataBag Class **/
```

Listing 5. Partition based operations of DataBag API.

4.5 Sinks

Listing 6 shows the sink methods of DataBag API:

- *writeCSV* method writes a DataBag into the specified `path` in a CSV format,
- *writeText* method Writes a DataBag into the specified `path` as plain text,
- *collect* converts the DataBag back into a scala Seq, and
- *as* converts this bag into a distributed collection of type `DColl[A]`.

```
/** DataBag Class **/

  def writeCSV(path: String, format: CSV)(implicit converter: CSVConverter[A]): Unit
  def writeText(path: String): Unit
  def writeParquet(path: String, format: Parquet)(implicit converter: ParquetConverter[A]): Unit
  def collect(): Seq[A]
  def as[DColl[_]](implicit conv: DataBag[A] => DColl[A]): DColl[A] =
    conv(this)

/** DataBag Class **/
```

Listing 6. Sink methods of DataBag API.

4.6 Predefined fold operations

Listing 7 shows the predefined fold operations for DataBag API:

- **isEmpty** method tests the collection for emptiness,
- **nonEmpty** is the opposite of **isEmpty**,
- **reduce** method aggregates the input elements based on given criteria,
- **min** finds the smallest element in the collection with respect to the natural ordering of the elements' type,
- **max** finds the largest element in the collection with respect to the natural ordering of the elements' type,
- **sum** calculates the sum over all elements in the collection,
- **product** calculates the product over all elements in the collection
- **size** returns the number of elements in the collection,
- **count** counts the number of elements in the collection that satisfy a predicate,
- **exists** tests if at least one element of the collection satisfies `p`,
- **forall** tests if all elements of the collection satisfy `p`,
- **find** finds some element in the collection that satisfies a given predicate,
- **bottom** finds the bottom `n` elements in the collection with respect to the natural ordering of the elements' type, and
- **top** finds the top `n` elements in the collection with respect to the natural ordering of the elements' type.

```
/** DataBag Class **/

def isEmpty: Boolean =
  fold(IsEmpty)
def nonEmpty: Boolean =
  fold(NonEmpty)
 def reduce[B >: A : Meta](zero: B)(plus: (B, B) => B): B =
  fold(Reduce(zero, plus))
 def reduceOption(plus: (A, A) => A): Option[A] =
  fold(ReduceOpt(plus))
 def min(implicit o: Ordering[A]): A =
  fold(Min(o)).get
 def max(implicit o: Ordering[A]): A =
  fold(Max(o)).get
 def sum(implicit n: Numeric[A]): A =
  fold(Sum(n))
 def product(implicit n: Numeric[A]): A =
  fold(Product(n))
```

```
  def size: Long =
    fold(Size)
  def count(p: A => Boolean): Long =
    fold(Count(p))
  def exists(p: A => Boolean): Boolean =
    fold(Exists(p))
  def forall(p: A => Boolean): Boolean =
    fold(Forall(p))
  def find(p: A => Boolean): Option[A] =
    fold(Find(p))
  def bottom(n: Int)(implicit o: Ordering[A]): List[A] =
    fold(Bottom(n, o))
  def top(n: Int)(implicit o: Ordering[A]): List[A] =
    fold(Top(n, o))

/** DataBag Class **/
```

Listing 7. Predefined fold operations of DataBag API.

Emma[2] is shipped as a Maven[3] dependency, which is a build automation tool for software projects. Therefore, it can be easily adopted into JVM-based projects by a simple maven import statement[4].

4.7 Executing Emma on Flink and Spark

Emma is a backend independent language. Currently, it supports Flink and Spark distributed dataflow engines as backend. The user specifies which engine to adopt and Emma automatically compiles the user code to the particular engine. We need to follow two steps to make it running:

- Choose a parallel dataflow framework (Flink or Spark)
- Enclose the code fragment with *emma.onSpark* quote to run on Spark or with *emma.onFlink* quote to run on Flink

Listing 8 and 9 show a simple selection query, which selects airports in Berlin based on specific latitude and longitude values. We change the backend engine with modifying the implicits without touching the core part of the actual query.

[2] https://github.com/emmalanguage
[3] https://mvnrepository.com/artifact/org.emmalanguage/emma-language
[4] http://emma-language.org/project-setup.html

```
private[snippets] def `emma on Spark`(implicit spark: SparkSession) = emma.onSpark {
  val berlinAirports = for {
    a <- airports
    if a.latitude > 52.3
    if a.latitude < 52.6
    if a.longitude > 13.2
    if a.longitude < 13.7
  } yield Location(
    a.name,
    a.latitude,
    a.longitude)

  berlinAirports.collect()
}
```

Listing 8. Emma code snippet to be executed on Spark.

```
private[snippets] def `emma on Flink`(implicit flink: ExecutionEnvironment) = emma.onFlink {
  val berlinAirports = for {
    a <- airports
    if a.latitude > 52.3
    if a.latitude < 52.6
    if a.longitude > 13.2
    if a.longitude < 13.7
  } yield Location(
    a.name,
    a.latitude,
    a.longitude)

  berlinAirports.collect()
}
```

Listing 9. Emma code snippet to be executed on Flink.

5 Emma Machine Learning Library

Emma features a machine learning library, which utilizes the DataBag concept explained in the previous section to implement ML algorithms such as Naive Bayes, *K*-means, and Linear Regression. The implementations are system-agnostic, i.e., the user can run on different backends.

5.1 Implementation

Listings 10, 11, and 12 show the implementations of Naive Bayes, *k*-means, and linear regression algorithms in Emma, respectively. The implementations show the expressiveness of Emma domain specific language which can be used to implement more complex scenarios.

In all implementations, the core part of execution is performed inside **apply** method. Naive Bayes implementation (Listing 10) accepts hyper parameters and the actual data which is abstracted in DataBag. Naive Bayes is based on Bayes' theorem with strong (naive) independence assumptions between the features. One reason for its wide adoption is that it is highly scalable. We support two types of Naive Bayes model being Bernoulli and Multinomial. The implementation aggregates the input tuples into the triples (label, lCnt, lSum). Based on aggregated triples, we build our model based on original formula [10].

```
object naiveBayes {
  type ModelType = ModelType.Value
  case class Model[L](label: L, pi: Double, theta: DVector)
  def apply[ID: Meta, L: Meta](D: Int,
    lambda: Double, modelType: ModelType // hyper-parameters
  )(
    data: DataBag[LDPoint[ID, L]] // data-parameters
  ): DataBag[Model[L]] = {
    val aggregated = for (Group(label, values) <- data.groupBy(_.label)) yield {
      val lCnt = values.size
      val lSum = stat.sum(D)(values.map(_.pos))
      (label, lCnt, lSum)
    }
    val numPoints = data.size
    val numLabels = aggregated.size
    val priorDenom = math.log(numPoints + numLabels * lambda)

    val model = for ((label, lCnt, lSum) <- aggregated) yield {
      val prior = math.log(lCnt + lambda) - priorDenom
      val evidenceDenom =
```

```
      if (modelType == ModelType.Multinomial) math.log(sum(lSum) + lambda * D)
      else /* bernoulli */ math.log(lCnt + 2.0 * lambda)
    val evidence = dense(for {
      x <- lSum.values
    } yield math.log(x + lambda) - evidenceDenom)

    Model(label, prior, evidence)
  }
  model
 }
 object ModelType extends Enumeration {
  val Multinomial = Value("multinomial")
  val Bernoulli = Value("bernoulli")
 }
}
```

Listing 10. Naive Bayes implementation in Emma

K-means clustering originated from signal processing. *K*-means is a method of vector quantization. k-means is frequently used in data analysis. The aim of k-means algorithm is to partition n observations into k clusters. K-Means discovers clusters centroids and assigns input . K-means may not give an optimal result.

Listing 11 shows the implementation of *k*-means [9] in Emma. Firstly, it orders the input points based on their distance to reference point. Then the iteration starts. The number of maximum iterations is provided as an input to the implementation. At the end of each iteration, we update the *solution* variable, which are the mean of the existing clusters. The computation continues for a given number of iterations and the resulting set of centroids is returned.

```
object kMeans {
 def apply[PID: Meta](
   D: Int, k: Int, epsilon: Double, iterations: Int // hyper-parameters )(
   points: DataBag[DPoint[PID]] // data-parameters
 ): DataBag[Solution[PID]] = {
   // helper method: orders points `x` based on their distance to `pos`
   val distanceTo = (pos: DVector) => Ordering.by { x: DPoint[PID] =>
     sqdist(pos, x.pos)
   }
   var optSolution = DataBag.empty[Solution[PID]]
   var minSqrDist = 0.0
   for (i <- 1 to iterations) {
     // initialize forgy cluster means
```

```
    var delta = 0.0
    var ctrds = DataBag(points.sample(k))
    // initialize solution: label points with themselves
    var solution = for (p <- points) yield LDPoint(p.id, p.pos, p)
    do {
      // update solution: label each point with its nearest cluster
      solution = for (s <- solution) yield {
        val closest = ctrds.min(distanceTo(s.pos))
        s.copy(label = closest)
      }
      // update centroid positions as mean of associated points
      val newCtrds = for {
        Group(cid, ps) <- solution.groupBy(_.label.id)
      } yield {
        val sum = stat.sum(D)(ps.map(_.pos))
        val cnt = ps.size.toDouble
        val avg = sum * (1 / cnt)
        DPoint(cid, avg)
      }
      // update delta as the sum of squared distances between the old and the new means
      delta = (for {
        cOld <- ctrds
        cNew <- newCtrds
        if cOld.id == cNew.id
      } yield sqdist(cOld.pos, cNew.pos)).sum
      // use new means for the next iteration
      ctrds = newCtrds
    } while (delta > epsilon)
    val sumSqrDist = (for (p <- solution) yield { sqdist(p.label.pos, p.pos) }).sum
    if (i <= 1 || sumSqrDist < minSqrDist) {
      minSqrDist = sumSqrDist
      optSolution = solution
    }
  }
  optSolution
 }
 type Solution[PID] = LDPoint[PID, DPoint[PID]]
}
```

Listing 11. *K*-means implementation in Emma.

It is important to stress the importance of the for-comprehension optimization used in Emma and in particular in the examples we demonstrate. Comprehensions generalize SQL and are available as first-class syntax in modern general-purpose programming languages such as Scala, Python, and SQL. Emma implements monad operations instead of binary operators like join, which enables the user to express Select-From-Where queries in a declarative way. Listing 12 shows the overall intuition behind this.

```
for {
  email <- emails
  from <- people
  to <- people
  if from.email == email.from
  if to.email == email.to
} yield (from, to, email)
```

Listing 12. Example Select-From-Where expression through for comprehension

Linear regression is used to model a relationship between a two continuous (quantitative) variables. One variable is the one we predicting (Y). We also call this as *criterion variable.* The other variable is called the predictor (X). The goal of simple linear regression is to find a function, which forms a straight line when predicting the variable Y.

Listing 13 shows the implementation of linear regression in Emma. It consists of two main parts, namely, train and predict.

The prependBias method is a helper module to *train* and *predict*. The train method uses Stochastic Gradient Descent as optimization method. After the training phase, we can use the the trained model to predict input unlabeled samples by calling the predict method.

```
object linreg {
  type Instance[ID] = LDPoint[ID, Double]
  type Solver[ID] = DataBag[Instance[ID]] => LinearModel
  def train[ID: Meta](instances: DataBag[Instance[ID]], solve: Solver[ID]): LinearModel =
    solve(prependBias(instances))
  def predict[ID: Meta](
    weights: DVector, err: Error
  )(
    instances: DataBag[Instance[ID]]
  ): Double = err(weights, prependBias(instances))
  def prependBias[ID: Meta](instances: DataBag[Instance[ID]]): DataBag[Instance[ID]] =
    instances.map(x => x.copy(pos = dense(1.0 +: x.pos.values)))
}
```

Listing 13. Linear regression implementation in Emma.

5.2 Examples of the Emma ML library

Listing 14 shows the *k*-means usage in Emma language.We pass dimensionality, the number of clusters, epsilon, and the number of iterations as argument to the kMeans object.

```
val points = for (line <- DataBag.readText(input)) yield {
  val record = line.split("\t")
  DPoint(record.head.toLong, dense(record.tail.map(_.toDouble)))
}
// do the clustering
val result = kMeans(2, k, runs, iterations)(points)
// return the solution as a local set
result.collect().toSet[kMeans.Solution[Long]]
```

Listing 14. Example of performing k-means clustering in Emma.

Listing 15 shows Linear Regression usage in Emma. Firstly, we initialize the input instances. Secondly, we initialize the solver, which is SGD, and provide necessary parameters to it such as learning rate, max iterations, batch size, and etc. Lastly, we call the train function which decouples all the implementation details from the end user.

```
val insts = DataBag(instances.map(x => x.copy(pos = dense(x.pos.values.drop(1)))))
  val solve = solver.sgd[Int](
    learningRate = learningRate,
    iterations = iterations,
    miniBatchSize = miniBatchSize,
    lambda = lambda,
    seed = seed
  )(
    error.rmse,
    regularization.l2
  )(
    weights
  )(_)
  linreg.train(insts, solve)
}
```

Listing 15. Example of performing Linear Regression in Emma

Listing 16 shows the Naive Bayes classifier usage in Emma. First, we iterate though the data points and construct LDPoints which are input data points with their corresponding labels. We pass the resulting set with other meta arguments for the algorithm, such as lambda function, model type, and dimensionality.

```
val data = for ((line, index) <- DataBag.readText(input).zipWithIndex()) yield {
    val record = line.split(",").map(_.toDouble)
    val label = record.head
    val dVector = dense(record.slice(1, record.length))
    LDPoint(index, dVector, label)
  }
  // classification
  val result = naiveBayes(16, lambda, modelType)(data)
  // collect the result locally
  result.collect().toSet[naiveBayes.Model[Double]]
```

Listing 16. Example of performing Naive Bayes in Emma

6 Implementing XGBoost from WP2 in Emma

To demonstrate the applicability of Emma to new machine learning algorithms, we implement a machine learning algorithm from Work Package 2 (XGBoost [11]) on Flink using Emma. XGBoost is a distributed, efficient, and high performant library that implements the gradient boosting machine learning technique for regression and classification tasks. In gradient boosting, a model is built by using an ensemble of weaker models in a stage-wise fashion, where in every stage a new model is trained and added to the list of the existing models [12].
The original Flink version was developed by SZTAKI, one of the Streamline partners, to be used in test applications and proof of concept of Streamline partners. The XGBoost implementation in Apache Flink on Github has more than 360 lines of code. Thus, we skip showing it in this section, but it can found on our Github repository (see links below). In comparison to the Flink implementation, the XGBoost implementation in Emma, shown in Listing 17, is much more concise. The reason is that the higher-level ***DataBag*** abstraction and the for-comprehensions offered by Emma hide many of the implementation details that otherwise have to be explicitly specified when using Apache Flink. As a result, the high-level abstractions in Emma enable the user to implement the same algorithm using fewer lines of code (only 24 lines of code rather than 369 lines) and to easily use either Flink or Spark as co-processor engine. This in turn means that the implementation in Emma has less complexity and thus it is easier to use.

The full implementation of XGBoost in both Flink and Emma and sample codes for executing them can be found on Streamline github:
Flink Application: https://github.com/streamline-eu/xgboost-application
Flink Library: https://github.com/streamline-eu/xgboost-jvm-packages/blob/master/jvm-packages/xgboost4j-flink/src/main/scala/ml/dmlc/xgboost4j/scala/flink/stream/XGBoost.scala
Emma: https://github.com/streamline-eu/streamline-ml-language/blob/master/emma-lib/src/main/scala/org/emmalanguage/lib/ml/xgboost/XGBoost.scala

```
@emma.lib
object XGBoost {
 def train(
   instances: DataBag[(LabeledPoint, Long)],
   tracker: IRabitTracker,
   round: Int,
   params: Map[String, Any]
   obj: ObjectiveTrait = null,
   eval: EvalTrait = null,
 ): Booster = {
   (
    for (Group(pid, partition: DataBag[(LabeledPoint, Long)]) <- instances groupBy { _._2 } ) yield
{
     tracker.getWorkerEnvs.put("DMLC_TASK_ID", pid.toString)  // XGBoost specific call
```

```
    val trainMat: DMatrix = new DMatrix(partition.map(_._1).collect().toIterator) // get
partition

    Rabit.init(tracker.getWorkerEnvs)
    // perform the training with XGBoost
    val booster = XGBoostScala.train(
      trainMat, params, round, watches = List("train" -> trainMat).toMap, obj, eval)
    // cleanup resources
    Rabit.shutdown()
    trainMat.delete()
    booster // return model
   }
  ).collect().head // retrieve the latest version of the model
 }
```

Listing 17. XGBoost Boosted Gradient implementation in Emma

In Listing 17, we show how XGBoost is reused within Emma. The key insights of the implementation are the use of 1) Scala native for-comprehension (to process items of the distributed DataBag representing the training set) and 2) Emma groupBy construct to access each parallel shard of the distributed DataBag. Scala native for-comprehension helps the user because she does not require to know system-specific APIs to scan the input training set. Emma groupBy is required because XGBoost expects a set of labelled points as input rather than only one point. *RabitTracker*, *ObjectiveTrait*, *EvalTrait*, and *DMatrix* are XGBoost components, which we need to keep in Emma code to support XGBoost.

```
object XGBoostFlink extends FlinkAware {

 def run(input: String, dimension: Int, rounds: Int, params: Map[String, Any]) = {
  withDefaultFlinkEnv(implicit flink => emma.onFlink {
   // XGBoost tracker init
   val trackerConf = TrackerUtils.getDefaultTackerConf
   val tracker = TrackerUtils.startTracker(flink.getParallelism, trackerConf)

   val input = DataBag.readText(input) map doParse // perform pre-processing
   // call the Emma XGBoost routine
   val model = XGBoost.train(input, tracker, rounds, params)
   // do something with model here
  })
 }
}
```

Listing 18. XGBoost Boosted Gradient execution in Emma with Flink as backend

In Listing 18, we show how to train a XGBoost model using Apache Flink as backend. First, we implicitly define Flink as backend and we initialize the XGBoost tracker (which controls the XGBoost internal environment). Then, we use Emma API to load the input from a local or remote location (e.g., local file system, distributed file system, raw socket, distributed messaging system). We parse the input into a suitable format, i.e., labelled vectors. Then, we provide to the train method (see Listing 16) with the input DataBag, XGBoost tracker, number of rounds for the training process, and some extra parameters for XGBoost[5].

[5] https://github.com/dmlc/xgboost/blob/master/doc/parameter.md

7 Conclusion

This deliverable presents a high-level declarative language for machine learning. The high-level abstractions of the proposed ML language are based on Emma language, a domain specific language, which is deeply embedded in Scala. Emma abstracts away several details of the underlying big data frameworks and thus facilitates programming in such frameworks. We build a machine learning library called Emma ML using the abstractions provided by Emma. We also show how the abstractions of Emma provide easy access to the ML algorithms such as XGBoost developed in the work package 2 of the Streamline project.

8 References

[1] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. 2015. Implicit Parallelism through Deep Language Embedding. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). ACM, New York, NY, USA, 47-61. DOI: https://doi.org/10.1145/2723372.2750543

[2] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. SIGMOD Record, 1994.

[3] T. Grust. Comprehending Queries (PhD Thesis). PhD thesis, Universit¨at Konstanz, 1999.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89, pages 25{35, New York, NY, USA, 1989. ACM.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Trans. Program. Lang. Syst.,, October 1991.

[6] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93, pages 237{247, New York, NY, USA, 1993. ACM. 87

[7] Andrew W. Appel. SSA is Functional Programming. SIGPLAN Not., 33(4):17{20, April 1998. 87

[8] Emma Language, available at http://emma-language.org/ (last accessed: 15/11/2017)

[9] MacQueen, James. "Some methods for classification and analysis of multivariate observations." Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. Vol. 1. No. 14. 1967.

[10] Lewis, David D. "Naive (Bayes) at forty: The independence assumption in information retrieval." European conference on machine learning. Springer, Berlin, Heidelberg, 1998.

[11] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016

[12] Freund, Yoav, Robert Schapire, and Naoki Abe. "A short introduction to boosting." *Journal-Japanese Society For Artificial Intelligence* 14.771-780 (1999): 1612.