D4.5 Scalable Online Algorithms in Flink

Lead Author: Jeyhun Karimov
With contributions from: Wenjuan Wang,
Waqas Jamil, Ngoc Canh Duong,
Bonaventura Del Monte, Alireza Rezaei Mahdiraji
Reviewer: Hamid Bouchachia

<table>
<thead>
<tr>
<th>Deliverable nature:</th>
<th>Demonstrator (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dissemination level:</td>
<td>Public (PU)</td>
</tr>
<tr>
<td>Contractual delivery date:</td>
<td>31.08.2018</td>
</tr>
<tr>
<td>Actual delivery date:</td>
<td>19.08.2018</td>
</tr>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
<tr>
<td>Total number of pages:</td>
<td>44</td>
</tr>
<tr>
<td>Keywords:</td>
<td>SOLMA, scalable machine learning</td>
</tr>
</tbody>
</table>
Abstract

Online machine learning algorithms play crucial role in modern industries, especially for ArcelorMittal use-case of PROTEUS. In this case, the use-case requires a short-time response from the machine learning model, as waiting for offline machine learning results might have severe negative impacts. For example, if the flatness of steel is not predicted on time and if the flatness quality is low as a result, the steel is not sent to the customer. The goal of this deliverable is to provide a set of online scalable algorithms and their implementation in scalable dataflow engine Apache Flink. The main contribution of this deliverable is twofold. Firstly, we provide an abstraction, SOLMA, on top of Apache Flink. This layer is designed to implement machine learning algorithms in an easy way. We adopt several approaches from state-of-the-art machine learning libraries, such as python scikit, and implement them on top of Apache Flink. Secondly, we develop a set of online machine learning algorithms with SOLMA abstraction.
Executive summary

The goal of this deliverable is to provide a library with a set of online machine learning algorithms implemented on top of distributed dataflow engine Apache Flink. We call the library SOLMA, Scalable Online Machine Learning Algorithms. The provided algorithms are general purpose algorithms to be used in distributed machine learning context and they are not tightly coupled with PROTEUS use-case. However, SOLMA also possesses PROTEUS use-cases related algorithms.

SOLMA aims to extend FlinkML’s online machine learning capabilities and it is built with the same intuition behind this latter, which is providing scalable ML algorithms, an intuitive API, and tools that help minimize glue code in end-to-end ML systems. As a matter of fact, ML systems developers are usually concerned about the huge quantity of glue code needed in building end-to-end ML systems. SOLMA abstraction lets end users build, test and deploy their own ML pipelines and seamlessly chain them with any kind of hybrid ETL pipeline (namely streaming and historical data) defined using PROTEUS and/or Apache Flink API.

Although FlinkML provides a nice set of traditional machine learning algorithms on top of Flink Batch API, there is no support for data streams. In the context of stream data processing, also in PROTEUS use-case, data analytics must satisfy new requirements as stated in [9]:

- Low memory usage
- Low processing time
- Items can be processed at most once
- Prediction should be possible at any stage

Here, online learning plays a prominent role; the way machine learning models are trained slightly changes, indeed prequential evaluation and holdout are the two mainstream methods known in literature.

Therefore, FlinkML should be extended such that the algorithms described in deliverables D4.1-4.4 can be expressed using pipeline abstractions (i.e., Estimator, Transformer, and Predictor pattern) and executed on data streams.

We also provide information about a set of algorithms implemented on top of SOLMA abstraction.
Online machine learning algorithms play crucial role in modern industries, especially for ArcelorMittal use-case of PROTEUS. In this case, the use-case requires a short-time response from the machine learning model, as waiting for offline machine learning results might have severe negative impacts. For example, if the flatness of steel is not predicted on time and if the flatness quality is low as a result, the steel is not sent to the customer. The goal of this deliverable is to provide a set of online scalable algorithms and their implementation in scalable dataflow engine Apache Flink. The main contribution of this deliverable is twofold. Firstly, we provide an abstraction, SOLMA, on top of Apache Flink. This layer is designed to implement machine learning algorithms in an easy way. We adopt several approaches from state-of-the-art machine learning libraries, such as python scikit, and implement them on top of Apache Flink. Secondly, we develop a set of online machine learning algorithms with SOLMA abstraction.

Keywords
SOLMA, scalable machine learning

<table>
<thead>
<tr>
<th>Issue Date</th>
<th>Rev. No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.07.2018</td>
<td>0.1</td>
<td>Javier De Matias Bejarano</td>
<td>Contribution of algorithms from TreeLogic</td>
</tr>
<tr>
<td>17.07.2018</td>
<td>0.2</td>
<td>Wenjuan Wang</td>
<td>Discussing contribution of BU</td>
</tr>
<tr>
<td>24.07.2018</td>
<td>0.3</td>
<td>Wenjuan Wang</td>
<td>Discussing contribution of BU</td>
</tr>
<tr>
<td>26.07.2018</td>
<td>0.4</td>
<td>Wenjuan Wang</td>
<td>Discussing contribution of BU</td>
</tr>
<tr>
<td>30.07.2018</td>
<td>0.5</td>
<td>Wenjuan Wang</td>
<td>First contribution of algorithms</td>
</tr>
<tr>
<td>06.08.2018</td>
<td>0.6</td>
<td>Wenjuan Wang</td>
<td>Address DFKI’s comments</td>
</tr>
<tr>
<td>10.08.2018</td>
<td>0.7</td>
<td>Wenjuan Wang</td>
<td>Address DFKI’s comments</td>
</tr>
<tr>
<td>24.08.2018</td>
<td>0.9</td>
<td>Jeyhun Karimov</td>
<td>Address BU’s comments</td>
</tr>
<tr>
<td>29.08.2018</td>
<td>1.0</td>
<td>Jeyhun Karimov</td>
<td>Final revision</td>
</tr>
</tbody>
</table>
# Table of Contents

*Executive summary* ................................................................................................................. 3  
*Document Information* ............................................................................................................. 4  
*Table of Contents* ...................................................................................................................... 6  
*List of listings* ............................................................................................................................. 7  
*List of Figures* ............................................................................................................................ 8  
*Abbreviations* .............................................................................................................................. 9  

1 *Introduction* .......................................................................................................................... 10  
2 *SOLMA Pipeline* ..................................................................................................................... 11  
   2.1 State-of-the-art libraries ....................................................................................................... 11  
   2.2 SOLMA Abstraction ........................................................................................................... 17  
3 *SOLMA Library* ....................................................................................................................... 21  
   3.1 SAX .................................................................................................................................... 21  
      3.1.1 Description ................................................................................................................... 21  
      3.1.2 Implementation ........................................................................................................... 22  
      3.1.3 Example code snippets ............................................................................................... 23  
   3.2 LASSO ................................................................................................................................ 24  
      3.2.1 Description ................................................................................................................... 24  
      3.2.2 Implementation ........................................................................................................... 25  
      3.2.3 Example code snippets ............................................................................................... 27  
   3.3 Regression based ML algorithms ......................................................................................... 29  
      3.3.1 Notations ..................................................................................................................... 29  
      3.3.2 Online Ridge Regression (ORR) .................................................................................. 29  
      3.3.3 Aggregation Algorithm for Regression (AAR) .............................................................. 29  
      3.3.4 Online Shrinkage via Limit Of Gibbs sampling (OSLOG) ........................................... 30  
      3.3.5 Competitive Online Iterative Ridge Regression (COIRR) .......................................... 30  
      3.3.6 Competitive Online Normalised Least Mean Squares Regression (TNLMS) .................. 31  
   3.4 Classification based ML algorithms ...................................................................................... 31  
      3.4.1 Online SVM ................................................................................................................. 31  
      3.4.2 Online Bilevel Stochastic Gradient for Support Vector Machine (OBSG_SVM) ......... 33  
      3.4.3 Passive-Aggressive Learning (PA) ................................................................................. 35  
      3.4.4 Online WAPA .............................................................................................................. 36  
   3.5 Sampling algorithms ............................................................................................................ 38  
      3.5.1 Simple reservoir sampling ............................................................................................ 38  
      3.5.2 Adaptive reservoir sampling ....................................................................................... 39  
      3.5.3 Weighted reservoir sampling ....................................................................................... 39  
   3.6 Moments .............................................................................................................................. 40  
   3.7 Frequent Directions .............................................................................................................. 40  
4 *Conclusions* ............................................................................................................................ 42  
*References* ................................................................................................................................. 43
List of listings

Listing 1. Pipeline analysis with FlinkML ................................................................. 12
Listing 2. Matrix class in FlinkML ........................................................................... 13
Listing 3. Vector class in FlinkML ............................................................................ 14
Listing 4. Estimator class in FlinkML ...................................................................... 15
Listing 5. TransformOperation and PredictOperation classes in FlinkML ............. 17
Listing 6. StreamTransformer class in SOLMA abstraction .................................... 18
Listing 7. StreamEstimator class in SOLMA abstraction ......................................... 19
Listing 8. StreamPredictor class in SOLMA abstraction ........................................ 20
Listing 13. OSVM Prequential training ..................................................................... 33
Listing 14. OSVM algorithm .................................................................................... 33
Listing 9. Simple reservoir sampling class ............................................................ 38
Listing 10 Main transformation function for simple reservoir sampling .................. 39
Listing 11. Frequent Directions class ....................................................................... 41
Listing 12. Main transformation for Frequent Directions ........................................ 41
List of Figures

Figure 1. SAX data pipeline................................................................. 22
Figure 2. Figure 2: SAX class diagram............................................. 23
Figure 3. SAXDictionary class diagram............................................ 23
Figure 4. Flink – PS interaction......................................................... 25
Figure 5. Worker – PS interaction (training procedure)...................... 25
Figure 6. Worker – PS interaction (prediction procedure).................... 26
Figure 7. Lasso data pipeline............................................................ 26
Figure 8. Lasso UML diagram (I)...................................................... 26
Figure 9. Lasso UML diagram (II)..................................................... 27
Abbreviations

**SOLMA**: Scalable Online Machine Learning Algorithms
**ML**: Machine Learning
**API**: Application Programming Interface
**ETL**: Extract Transform Load
**BLAS**: Basic Linear Algebra Subprograms
**PS**: Parameter Server
**SVD**: Singular Value Decomposition
**SVM**: Support Vector Machine
**OSVM**: One-class Support Vector Machine
1 Introduction

One of the main targets of machine learning is to extract knowledge from data with the help of statistics, probability and optimization techniques. Machine learning has a wide usage area from everyday tasks (e.g., product recommendations) to spam filtering and self-driving cars. In the ‘age of big data’ performing machine learning tasks is challenging as the size of data sets grow fast. In PROTEUS, we aim the next generation machine learning techniques, which are both distributed and online to provide real-time response to customers instead of training models periodically overnight.

The goal of this deliverable is to provide a library with a set of online machine learning algorithms implemented on top of distributed dataflow engine Apache Flink. We call the library SOLMA, Scalable Online Machine Learning Algorithms. The provided algorithms are general purpose algorithms to be used in distributed machine learning context and they are not tightly coupled with PROTEUS use-case. However, SOLMA also possesses PROTEUS use-cases related algorithms.

The main problem is that implementing machine learning algorithms of SOLMA library is cumbersome as those algorithms share common stages. As a result, most widely accepted libraries such as python scikit-learn introduce interfaces Estimator, Transformer, and Predictor to easily develop machine ML pipelines.

Our solution is to build a common layer on top of Apache Flink like python scikit-learn library does for machine learning algorithms. The Estimator interface performs the actual training of the model. As evident by the naming, classes that implement Transformer are transform operations like scaling the input and Predictor implementations are learning algorithms, such as Multiple Linear Regression. We provide the details of our implementations of SOLMA abstraction in Section 2. First, we analyze existing state-of-the-art techniques in this area and provide our abstraction to be used on top of Apache Flink.

We use SOLMA abstraction to implement a set of algorithms on top of Apache Flink. We provide theoretical information about the algorithms, their distributed working semantics, their implementation on top of SOLMA abstraction and example code snippets. As this library is open source [12], machine learning, distributed systems, or language development communities can benefit and extend the library.

We provide high level description of algorithms. Firstly, we describe LASSO and SAX, which are the most relevant ML algorithms to PROTEUS. We elaborate the main design and implementation decisions and provide examples. Then, we categorize remaining algorithms into regression, classification and other categories and provide a high level information.

The deliverable is organized as follows. In Section 2, we provide SOLMA abstraction. We show state-of-the-art library interfaces and based on that we build our solution for online environment. In Section 3, we provide list of ML algorithms in SOLMA library. We show their parallelisation semantics and implementation details on top of SOLMA abstraction. We conclude in Section 4.
2 SOLMA Pipeline

In this section, we present the SOLMA abstraction, built on PROTEUS engine, an overhauled Apache Flink version supporting hybrid computation between data-in-motion (data streams) and data-in-rest (historical batch datasets).

SOLMA aims to extend FlinkML’s online machine learning capabilities and it is built with the same intuition behind this latter, which is providing scalable ML algorithms, an intuitive API, and tools that help minimize glue code in end-to-end ML systems. As a matter of fact, ML systems developers are usually concerned about the huge quantity of glue code needed in building end-to-end ML systems [1]. SOLMA abstraction lets end users build, test and deploy their own ML pipelines and seamlessly chain them with any kind of hybrid ETL pipeline (namely streaming and historical data) defined using PROTEUS and/or Apache Flink API.

2.1 State-of-the-art libraries

As described in FlinkML programming guide, pipelines in the ML context can be thought of as chains of operations that have some data as input, perform a set of transformations to that data, and then output the transformed data, either to be used as the input (features) of a predictor function, such as a learning model, or just output the transformed data themselves, to be used in some other task. The end learner can of course be a part of the pipeline as well. ML pipelines can often be complicated sets of operations and can become sources of errors for end-to-end learning systems.

The purpose of ML pipelines is then to create a framework that can be used to manage the complexity introduced by these chains of operations. Pipelines should make it easy for developers to define chained transformations that can be applied to the training data, in order to create the end features that will be used to train a learning model, and then perform the same set of transformations just as easily to unlabelled (test) data. Pipelines should also simplify cross-validation and model selection on these chains of operations.

Finally, by ensuring that the consecutive links in the pipeline chain “fit together” we also avoid costly type errors. Since each step in a pipeline can be a computationally-heavy operation, we want to avoid running a pipelined job, unless we are sure that all the input/output pairs in a pipeline “fit”. FlinkML follows an API inspired by sklearn [13], which means that we have Estimator, Transformer, and Predictor interfaces. For an in-depth look at the design of the sklearn API the interested reader is referred [10].

In short, the Estimator is the base class from which Transformer and Predictor inherit. Estimator defines a fit method, and Transformer also defines a transform method and Predictor defines a predict method.

The fit method of the Estimator performs the actual training of the model, for example finding the correct weights in a linear regression task, or the mean and standard deviation of the data in a feature scaler. As evident by the naming, classes that implement Transformer are transform operations like scaling the input and Predictor implementations are learning algorithms, such as Multiple Linear Regression. Pipelines can be created by chaining together one or more Transformers, and the final link in a pipeline can be a Predictor or another Transformer. Pipelines that end with Predictor cannot be chained any further.

FlinkML also supports optimized routines for handling sparse and dense matrix as well as BLAS (Basic Linear Algebra Subprograms)-compliant operations. Support for distributed large matrix is also guaranteed.

Currently FlinkML supports the following algorithms:

Supervised Learning
- SVM using Communication efficient distributed dual coordinate ascent (CoCoA)
- Multiple linear regression
- Optimization Framework
Unsupervised Learning

- k-Nearest neighbors join

Data Preprocessing

- Polynomial Features
- Standard Scaler
- MinMax Scaler

Recommendation

- Alternating Least Squares (ALS)

Outlier selection

- Stochastic Outlier Selection (SOS)

Utilities

- Distance Metrics
- Cross Validation

Listing 1 shows how easy it is to set up an analysis pipeline with FlinkML.

```scala
val trainingData: DataSet[LabeledVector] = ...
val testingData: DataSet[Vector] = ...

val scaler = StandardScaler()
val polyFeatures = PolynomialFeatures().setDegree(3)
val mlr = MultipleLinearRegression()

// Construct pipeline of standard scaler, polynomial features and multiple linear regression
val pipeline = scaler.chainTransformer(polyFeatures).chainPredictor(mlr)

// Train pipeline
pipeline.fit(trainingData)

// Calculate predictions
val predictions: DataSet[LabeledVector] = pipeline.predict(testingData)
```

Listing 1. Pipeline analysis with FlinkML

As explained above, FlinkML library consists mainly of a matrix and vector abstraction, three building blocks for algorithms (e.g. Estimator, Transformer and Predictor) and some ready to use batch machine learning algorithms.

The matrix and vector base classes are defined in Listing 2 and Listing 3.
/** Base trait for a matrix representation */

trait Matrix {

/** Number of rows */
  @return number of rows in the matrix */
  def numRows: Int

/** Number of columns */
  @return number of columns in the matrix */
  def numCols: Int

/** Element wise access function */
  @param row row index
  @param col column index
  @return matrix entry at (row, col) */
  def apply(row: Int, col: Int): Double

/** Element wise update function */
  @param row row index
  @param col column index
  @param value value to set at (row, col) */
  def update(row: Int, col: Int, value: Double): Unit

/** Copies the matrix instance */
  @return Copy of itself */
  def copy: Matrix

  def equalsMatrix(matrix: Matrix): Boolean = {
    if(numRows == matrix.numRows && numCols == matrix.numCols) {
      val coordinates = for(row <- 0 until numRows;
        col <- 0 until numCols)
        yield (row, col)
      coordinates forall { case (row, col) =>
        this.apply(row, col) == matrix(row, col)}
    } else {
      false
    }
  }
}

Listing 2. Matrix class in FlinkML
trait Vector extends Serializable {

  /** Number of elements in a vector
     * @return The number of elements of the vector */
  def size: Int

  /** Element wise access function
     * @param index index of the accessed element
     * @return value of the associated with the index */
  def apply(index: Int): Double

  /** Updates the element at the given index with the provided value
     * @param index The index of the element to be updated
     * @param value The new value */
  def update(index: Int, value: Double): Unit

  /** Copies the vector instance
     * @return Copy of the vector instance */
  def copy: Vector

  /** Returns the dot product of the recipient and the argument
     * @param other a Vector
     * @return a scalar double of dot product */
  def dot(other: Vector): Double

  /** Returns the outer product of the recipient and the argument
     * @param other a Vector
     * @return a matrix */
  def outer(other: Vector): Matrix

  /** Magnitude of a vector
     * @return The length of the vector */
  def magnitude: Double

  def equalsVector(vector: Vector): Boolean = {
    if(size == vector.size) {
      (0 until size) forall { idx =>
        this(idx) == vector(idx)
      }
    } else {
      false
    }
  }
}

Listing 3. Vector class in FlinkML
Specialized DenseMatrix and SparseMatrix as well as DenseVector and SparseVector are available with extended operations support (e.g. BLAS). All those matrix operations are executed by using Scala Breeze library, a Scala numerical processing library\(^1\).

The signature of an Estimator is shown in Listing 4.

```scala
trait Estimator[Self] extends WithParameters with Serializable {
  def fit[Training](
    training: DataSet[Training],
    fitParameters: ParameterMap = ParameterMap.Empty
  ) (implicit fitOperation: FitOperation[Self, Training]): Unit = {
    FlinkMLTools.registerFlinkMLTypes(training.getExecutionEnvironment)
    fitOperation.fit(this, fitParameters, training)
  }
}
```

Listing 4. Estimator class in FlinkML

Within the fit operation, a pipeline component is trained with respect to the given training data. The algorithm is, however, not implemented by overriding the fit method but by providing an implementation of a corresponding FitOperation for the correct type. Looking at the definition of the fit method in Estimator, which is the parent class of Transformer, reveals why this is the case.

We see that the fit method is called with an input data set of type Training, an optional parameter list and in the second parameter list with an implicit parameter of type FitOperation. Within the body of the function, first some machine learning types are registered and then the fit method of the FitOperation parameter is called. The instance gives itself, the parameter map and the training data set as parameters to the method. Thus, all the program logic takes place within the FitOperation.

The FitOperation has two type parameters. The first defines the pipeline operator type for which this FitOperation shall work and the second type parameter defines the type of the data set elements.

The same happens for Transformer and Predictor traits: they respectively have a TransformerOperation and a PredictorOperation in their companion objects. The signatures of these operations are shown in Listing 5.

\(^1\) https://github.com/scalanlp/breeze
### TransformOperation

```scala
trait TransformOperation[Instance, Model, Input, Output] extends Serializable {
  /** Retrieves the model of the [[Transformer]] for which this operation has been defined. *
   * @param instance *
   * @param transformParameters *
   * @return */
  def getModel(instance: Instance, transformParameters: ParameterMap): DataSet[Model]
}
```

### PredictOperation

```scala
trait PredictOperation[Instance, Model, Testing, Prediction] extends Serializable {
  /** Defines how to retrieve the model of the type for which this operation was defined *
   * @param instance The Predictor instance that we will use for predictions *
   * @param predictParameters The parameters for the prediction *
   * @return A DataSet with the model representation as its only element */
  def getModel(instance: Instance, predictParameters: ParameterMap): DataSet[Model]
}
```
2.2 SOLMA Abstraction

Although FlinkML provides a nice set of traditional machine learning algorithms on top of Flink Batch API, there is no support for data streams. In the context of stream data processing, also in PROTEUS use-case, data analytics must satisfy new requirements as stated in [9]:

- Low memory usage
- Low processing time
- Items can be processed at most once
- Prediction should be possible at any stage

Here, online learning plays a prominent role; the way machine learning models are trained slightly changes, indeed prequential evaluation and holdout are the two mainstream methods known in literature [8]. Therefore, FlinkML should be extended such that the algorithms described in deliverables D4.1-4.4 can be expressed using pipeline abstractions (i.e., Estimator, Transformer, and Predictor pattern) and executed on data streams.

Examples of Transformers are:

- Reservoir samplers (transform: transform a data stream in a windowed stream by sampling the former and assigning items to windows according to their timestamps)
- Online PCA (fit: incrementally learns the principal features; transform: applies the PCA to input stream)
- Random Projection (fit: determine a sketch of the stream; transform: reduce input dimensions)
- SVD (usage in previous deliverable is unclear, I can guess but need confirmation from BU)

Example of Estimators:

- Heavy hitters: (fit: figures out the most frequent items in the stream through FD algorithm)
- Moments: (fit: calculates the desired moment)

Although no predictor has already been discussed in past deliverables yet, an example could be:

- VHT (fit: prequential evaluation, predict: only predict)

All the algorithms listed above will have some hyper-parameters that the end-user is able to tweak.

Moreover, they could work per-window or per-item and for those that need to update a model, they will be connected to the parameter server developed in WP3. By taking advantage of the modularity of FlinkML, we can write the actual ML operation directly in Scala.

Listing 6, Listing 7, and Listing 8 show the main interfaces of SOLMA abstraction. StreamEstimator contains fitting operation which fits the estimator to the given input data. StreamTransformer extends from StreamEstimator having transform operations. It can also chain together several transformations or predictions. StreamPredictor also inherits from StreamEstimator. A user should provide predicting function and the rest is translated to Apache Flink DataStream API automatically.
trait StreamTransformer[Self <: StreamTransformer[Self]] extends StreamEstimator[Self]{
    that: Self =>

def transform[Input, Output](
    input: DataStream[Input],
    transformParameters: ParameterMap = ParameterMap.Empty
)(implicit transformOperation: TransformDataStreamOperation[Self, Input, Output]) :
DataStream[Output] = {
    FlinkSolmaUtils.registerFlinkMLTypes(input.executionEnvironment)
    transformOperation.transformDataStream(that, transformParameters, input)
}

/** Chains two [[StreamTransformer]] to form a [[ChainedStreamTransformer]].
 * @param transformer Right side transformer of the resulting pipeline
 * @tparam T Type of the [[StreamTransformer]]
 * @return */
def chainTransformer[T <: StreamTransformer[T]](transformer: T)
    : ChainedStreamTransformer[Self, T] = {
        new ChainedStreamTransformer(this, transformer)
    }

/** Chains a [[StreamPredictor]] with a [[StreamPredictor]] to
 * form a [[ChainedStreamPredictor]].
 * @param predictor Trailing [[StreamPredictor]] of the resulting pipeline
 * @tparam P Type of the [[StreamPredictor]]
 * @return */
def chainPredictor[P <: StreamPredictor[P]](predictor: P):
    ChainedStreamPredictor[Self, P] = {
        ChainedStreamPredictor(this, predictor)
    }
}

Listing 6. StreamTransformer class in SOLMA abstraction.
trait StreamEstimator[Self] extends WithParameters with Serializable {

  that: Self =>

  def setPartitioning[Training](fun: (DataStream[Any]) => KeyedStream[(Any, Long), Long]): Self = {
    parameters.add(StreamEstimator.PartitioningOperation, fun)
    this
  }

  /** Fits the estimator to the given input data. The fitting logic is contained in the [[StreamFitOperation]]. The computed state will be stored in the implementing class.
   * @param training Training data stream
   * @param fitParameters Additional parameters for the [[StreamFitOperation]]
   * @param fitOperation [[StreamFitOperation]] which encapsulates the algorithm logic
   * @tparam Training Type of the training data
   * @return */
  def train[Training](
    training: DataStream[Training],
    fitParameters: ParameterMap = ParameterMap.Empty)(implicit
    fitOperation: StreamFitOperation[Self, Training]): Unit = {
    FlinkSolmaUtils.registerFlinkMLTypes(training.executionEnvironment)
    fitOperation.fit(this, fitParameters, training)
  }
}

Listing 7. StreamEstimator class in SOLMA abstraction.
trait StreamPredictor[Self] extends StreamEstimator[Self] {
    that: Self =>

    def predict[Testing, Prediction](
        testing: DataStream[Testing],
        predictParameters: ParameterMap = ParameterMap.Empty)(implicit
        predictor: PredictDataStreamOperation[Self, Testing, Prediction])
    : DataStream[Prediction] = {
        FlinkSolmaUtils.registerFlinkMLTypes(testing.getExecutionEnvironment)
        predictor.predictDataStream(this, predictParameters, testing)
    }
}

object StreamPredictor {
    implicit def defaultPredictDataStreamOperation[
        Instance <: StreamEstimator[Instance],
        Model,
        Testing,
        PredictionValue]
    (implicit predictOperation: StreamPredictOperation[Instance, Model,
        Testing, PredictionValue],
        testingTypeInformation: TypeInformation[Testing],
        predictionValueTypeInformation: TypeInformation[PredictionValue])
    : PredictDataStreamOperation[Instance, Testing, (Testing, PredictionValue)]
    = {
        new PredictDataStreamOperation[Instance, Testing, (Testing,
            PredictionValue)] {
            override def predictDataStream(
                instance: Instance,
                predictParameters: ParameterMap,
                input: DataStream[Testing])
            : DataStream[(Testing, PredictionValue)] = {
                val resultingParameters = instance.parameters ++ predictParameters
                val model = predictOperation.getModel(instance, resultingParameters)
                implicit val resultTypeInformation = createTypeInformation[(Testing,
                    PredictionValue)]
                input.map(element => {
                    (element, predictOperation.predict(element, model))
                })
            }
        }
    }
}
3 SOLMA Library

In this section, we provide a set of algorithms implemented using SOLMA abstraction. We divide each algorithm or set of algorithms into three sections: 1) description, where we explain high level details of the algorithm, 2) implementation, where we describe how the algorithm was implemented using SOLMA abstractions, and 3) examples, where we provide some example code snippets.

First, we provide SAX and LASSO, which are more related to PROTEUS use-case, then we elaborate on other set of machine learning algorithms in SOLMA library.

3.1 SAX

3.1.1 Description

SAX-SVM is a method proposed by P. Senin and S. Malinchick. SAX-VSM is based on two well-known techniques[14]. The first technique is Symbolic Aggregate approXimation (SAX), which is a high-level symbolic representation of time series. The second technique is the classic Vector Space Model (VSM) based on tf*idf weighting scheme.

By using SAX, the algorithm transforms real-valued time series of a single input class into a combined collection of SAX words, which is called “bag of words”. Next, by using tf*idf weighting, the algorithm transforms these collections (one collection for each of the input classes) into class-characteristic weight vectors, which, in turn, are used in classification built upon Cosine similarity.

Symbolic Aggregate approXimation (SAX). Symbolic representation of time series, once introduced, has attracted much attention by enabling the application of numerous string-processing algorithms, bioinformatics tools, and text mining techniques to time series. The method provides a significant reduction of the time series dimensionality and a low-bounding to Euclidean distance metrics, which guarantees no false dismissal. These properties are often leveraged by other techniques that embed SAX representation for indexing and approximation. Configured by two parameters, a desired word size w and an alphabet size α, SAX produces a symbolic approximation of a time series T of a length n by compressing it into a string of the length w (usually w << n), whose letters are taken from the alphabet α. At the first step of the algorithm, T is z-normalized (to unit of standard deviation). At the second step, a dimensionality of the normalized time series is reduced from n to w by obtaining its Piecewise Aggregate Approximation (PAA). For this, the normalized time series is divided into w equal-sized segments and mean values for points within each segment are computed. The sequence of these values forms PAA approximation of T. Finally, each of w PAA coefficients is converted into a letter of an alphabet α using the lookup table which defines a set of breakpoints that divide the normalized time series values distribution space into α equal-sized regions (as in the original SAX work, it is assumed Gaussian distribution).

Following its introduction, SAX was shown to be an efficient tool for solving problems of finding time series motifs and discords. The authors employed a sliding window-based subsequence extraction technique and augmented data structures in order to build SAX words “vocabularies”. By analyzing words frequencies, they were able to capture frequent and rare SAX words representing motif and discord subsequences. The same technique, based on the combination of sliding window and SAX, was used in numerous works, most notably in Shapelet and BOP-based classifiers. SAX-SVM also use this sliding window technique to convert a time series T of a length n into the set of m SAX words, where m = (n − l) + 1 and l is the sliding window length. By sliding a window of length l across time series T, extracting overlapping subsequences, converting them to SAX words, and placing these words into an unordered collection, it is obtained the bag of words representation of the original time series T.
Vector Space Model adaptation. SAX-SVM use the vector space model exactly as it is known in Information Retrieval. Similarly, it is defined and used the following expressions: term - a single SAX word, bag of words - an unordered collection of SAX words, corpus - a set of bags, and weight matrix - a matrix defining weights of all words in a corpus. Given a training set, SAX-VSM builds a bag of SAX words for each of the classes by processing each time series with a sliding window and SAX. Bags are combined into a corpus, which is built as a term frequency matrix, whose rows correspond to the set of all SAX words (terms) found in all classes, whereas each column denotes a class of the training set. Each element of this matrix is an observed frequency of a term in a class. Because SAX words extracted from the time series of one class are often not found in others this matrix is usually sparse.

Next, SAX-VSM applies tf-idf weighting scheme for each element of this matrix to transform a frequency value into a weight coefficient. The tf*idf weight for a term t is defined as a product of two factors: term frequency (tf) and inverse document frequency (idf).

Once all frequency values are computed, term frequency matrix becomes the term weight matrix, whose columns used as class’ term weight vectors that facilitate the classification using Cosine similarity.

3.1.2 Implementation

As it is expected from PROTEUS project implementations, Apache Flink framework has been the base of all the proposed scalable online machine learning algorithms. To process streaming data is one of the main requirements to this kind of algorithms. Apache Flink is conceived for it and make available mechanisms to parallelise this kind of jobs. SAX has been implemented as a streaming transformation pipeline as it is shown in Figure 1. The data pipeline input is a stream of sensor measurements of one of the variables of the problem and the output will be a stream of SAX results. A SAX result object contains:

- coilId: Coil identifier Error! Reference source not found.
- varName: Name of the selected variable
- x1 and x2: Interval of X where is calculated this SAX result
- classId: Name of the calculated class between the predefined ones (normally with a capital letter: A, B, C, ...)
- similarity: A measure of how this classId fits with the predefined classes.

The SAX pipeline has two important transformation operations:

- SAX: The SAX transform operation performs two operations for a given stream: PAA and SAX. For the PAA algorithm, the stream is divided into windows of the user selected PAA size. On each window, a Z-normalization operation is applied first. Then, we average the values of the window. With the averaged values, the algorithm converts each value into a symbol of the alphabet. Several symbols are grouped as to form a word of a user-selected size
- SAXDictionary: Calculates the predictions for all elements in the input data stream.

As it has been explained above, SAX algorithm is implemented as pipeline of transform operation with a input data of sensor measurements. The first step, implemented by SAX class, is a pure transformation,
therefore **SAX** class inherits from **StreamTransformer** and **Estimator** classes (Figure 2). These two classes need implicit implementations. Concretely it has been necessary to implement the transform operation in the class **SAXStreamTransformOperation** and estimation operation in the class **SAXFitOperation**.

![Figure 2. SAX class diagram](image)

Respect to **SAXDictionary**, it implements the prediction operation strictly speaking, therefore **SAXDictionary** class inherits from **StreamPredictor** and **Estimator** classes (Figure 3). These two classes need implicit implementations. In this case, it has been necessary to implement the predict operation in the class **SAXDictionaryPredictOperation** and the estimation operation in the class **SAXDictionaryFitOperation**.

![Figure 3. SAXDictionary class diagram](image)

### 3.1.3 Example code snippets

As it has been explained above, the signature of SAX class is:

```scala
class SAX extends StreamTransformer[SAX] with Estimator[SAX] {
  ...
}
```

SAX class inherits from StreamTransformer class and Estimator class. The transforming operation and the fit operation are implemented in **SAXStreamTransformOperation** class and **SAXFitOperation** respectively. The first one has a signature like this:

```scala
class SAXFitOperation[T] extends FitOperation[SAX, T] {
  override def fit(instance: SAX, fitParameters: ParameterMap, input: DataSet[T]): Unit = {
    [...] Unit = {
  }
}
```

and the second one like this other:
**class** SAXStreamTransformOperation[T <: Double]
**extends** TransformDataStreamOperation[SAX, (T, Int), (String, Int)]{

**override def** transformDataStream(instance: SAX,
transformParameters: ParameterMap, input: DataStream[(T, Int)])
: DataStream[(String, Int)] = {

[...]
}
}

SAXDictionary class inherits from StreamPredictor class and Estimator class. The transforming operation and the fit operation are implemented in SAXDictionaryPredictOperation class and SAXDictionaryFitOperation respectively. The first one has a signature like this:

class SAXDictionaryPredictOperation[T <: String]
**extends** PredictDataStreamOperation[SAXDictionary, (T, Int),
SAXPrediction] {

**override def** predictDataStream(
instance: SAXDictionary,
predictParameters: ParameterMap,
input: DataStream[(T, Int)]): DataStream[SAXPrediction] = {

[...]
}
}

and the second one like this other:

class SAXDictionaryFitOperation[T] **extends** FitOperation[SAXDictionary, T]{

**override def** fit(
instance: SAXDictionary,
fitParameters: ParameterMap,
input: DataSet[T]): Unit = {

[...]
}
}

### 3.2 LASSO

#### 3.2.1 Description

The acronym “LASSO” stands for Least Absolute Shrinkage and Selection Operator. Lasso regression is a type of linear regression that uses shrinkage. Shrinkage is where data values are shrunk towards a central point, like the mean. The lasso procedure encourages simple, sparse models (i.e. models with fewer parameters). This particular type of regression is well-suited for models showing high levels of multicollinearity or when you want to automate certain parts of model selection, like variable selection/parameter elimination.

Give a set of input measurements \(x_1, x_2 \ldots x_p\) and an outcome measurement \(y\), the lasso fits a linear model

\[
\hat{y} = b_0 + b_1 \cdot x_1 + b_2 \cdot x_2 + \ldots b_p \cdot x_p
\]
The criterion it uses is to minimize \( \sum (y - \hat{y})^2 \) subject to \( \sum |b_j| \leq s \).

The first sum is taken over observations (cases) in the dataset. The bound "s" is a tuning parameter. When "s" is large enough, the constraint has no effect and the solution is just the usual multiple linear least squares regression of \( y \) on \( x_1, x_2 \ldots x_p \).

However, when for smaller values of \( s \) \((s \geq 0)\) the solutions are shrunken versions of the least squares estimates. Often, some of the coefficients \( b_j \) are zero. Choosing "s" is like choosing the number of predictors to use in a regression model, and cross-validation is a good tool for estimating the best value for "s".

### 3.2.2 Implementation

As it was defined in PROTEUS proposal, Apache Flink framework has been the base of SOLMA library and therefore the tool which allow the implementation of a set of scalable online machine learning algorithms. To process streaming data is one of the main requirements to this kind of algorithms. Apache Flink is conceived for it and make available mechanisms to parallelise this kind of jobs. Unfortunately, it is not trivial to parallelise online machine learning. If all workers (Task managers in Flink language) process in parallel input data (to train the model or to make predictions) they need to “share” a unique updated model (or at least most updated as it is possible). The parameter server (PS) is a tool to address these issues efficiently. The overall intuition can be seen from Figure 4.

![Figure 4. Flink – PS interaction](image)

PS has been included in SOLMA implementation to accomplish this essential mission. The PS working is quite simple. PS store the most updated version of the model. Workers will need the model in two kind of procedures:

- Training procedure: As can be seen from Figure 5, the worker receives a labelled input data, make a “pull” operation (in an asynchronous way), wait until PS response it, receive the shared model, update it with the labelled input data and make a “push” operation to update the model in PS.

![Figure 5. Worker – PS interaction (training procedure)](image)
• Prediction procedure: As can be seen from Figure 6, the worker receives a unlabelled input data, make a “pull” operation (in an asynchronous way), wait until PS response it, receive the shared model, make a prediction from the unlabelled data and requested model and produce an output with the prediction.

![Worker – PS interaction (prediction procedure)](image1)

As it was explained above, to process streaming data is one of the most important requirements to scalable online machine learning algorithms. Apache Flink provides several abstractions to streaming processing. The Figure 7 shows an abstraction of how Lasso implementation works. From a bird's eye view, Lasso algorithm is implemented as transform operation with two kinds of input data, sensor measurements (from all the sensors distributed along factory line) and flatness measurements (measured with a delay after finish coil production) and an output data, predictions of coil flatness (before to know the real flatness value).

![Lasso data pipeline](image2)

SOLMA provide us some abstractions for this kind of algorithms. A combination of classes StreamTransformer and StreamPredictor define the behavior of this Lasso implementation. The UML class diagram is as it is shown in Figure 8.

![Lasso UML diagram (I)](image3)

The inheritance from StreamTransformer class has implicit to implement stream transformation operation in class LassoDFStreamTransformOperation. This class uses LassoParameterServer which it is the entry point
to Parameter Server use. As it is explained in Figure 9, it is necessary two important classes for this, LassoBasicAlgorithm, with the basic details of Lasso algorithm, and LassoWorkerLogic, where it is defined how will be the interaction with Parameter Server.

![Figure 9. Lasso UML diagram (II)](image)

### 3.2.3 Example code snippets

SOLMA provide us some abstractions for this kind of algorithm. A combination of classes StreamTransformer and StreamPredictor define the behavior of this Lasso implementation. Our implementation extends from these two classes which make easy the implementation of the expected behavior:

```scala
class LassoDelayedFeedbacks extends StreamTransformer[LassoDelayedFeedbacks] with StreamPredictor[LassoDelayedFeedbacks]
```

To extend StreamTransformer class it was necessary to implement a class where transformation operation is defined. This class is LassoDFStreamTransformOperation and extends TransformDataStreamOperation (defined in SOLMA). LassoDFStreamTransformOperation override method transformDataStream:

```scala
override def transformDataStream(instance: LassoDelayedFeedbacks, transformParameters: ParameterMap, rawInput: DataStream[InputData]): DataStream[OutputData] = {

    val workerLogic: LassoWorkerLogic = new LassoWorkerLogic(...)  
    val output = LassoParameterServer.transformLasso(None)(rawInput, workerLogic)

    output
}
```

This method implements the tranformation logic and it is here where is defined the interaction with PS. That is the entry point to PS working. It has been necessary to implement previously:

- The class LassoParameterServer, with the logic of stream transformation.
- The class LassoWorkerLogic, with the behavior of the workers respect to PS.

As it was explained above, PS provide a simple interface with three operations: pull, push and output. The ParameterServerClient class has next signature:

```scala
trait ParameterServerClient[P, WorkerOut] extends Serializable {
    def pull(id: Int): Unit
    def push(id: Int, deltaUpdate: P): Unit
    def output(out: WorkerOut): Unit
}
```
and the class WorkerLogic has this other signature:

```scala
trait WorkerLogic[T, P, WorkerOut] extends Serializable {
  def onRecv(data: T, ps: ParameterServerClient[P, WorkerOut]): Unit
  def onPullRecv(paramId: Int, paramValue: P, ps: ParameterServerClient[P, WorkerOut]): Unit
}
```

Therefore the LassoWorkerLogic class implements these two methods:

- **Method onRecv**: it implements the behavior of the algorithm when a new input data is received. The input data can be a feature value (a value from one of the set of variables available to train the model) and a target value (a value from the target variable). When a new input data is received several operation could be performed:

  - If there are labelled data it is necessary to request the updated model from PS (using “pull” operation)
  - If there are unlabelled data it is necessary to request the updated model from PS (using “pull” operation)
  - If there are values from input features, but it is not available the related target value, it is necessary to store this input features values to label in the future (when target variable arrives). In addition, this values from input features are store as “pending for predict”
  - If there are values from input features stored waiting the related target value and it is arrived, it is necessary to label this input features values and store this data as “pending for training”

- **Method onPullRecv**: it implements the behavior of the algorithm when PS send to workers the updated model. Two operations could be performed here:

  - If there are a set of unlabelled vectors, stored as “pending for predict”, a set of predictions will be calculated.
  - If there are a set of labelled vectors, stored as “pending for training”, they will be used to train the model and perform a “push” operation to update the model in PS.

The algorithm basis are implemented in **LassoBasicAlgorithm** class. Two operations has been implemented:

```scala
class LassoBasicAlgorithm {
  override def delta(dataPoint: LabeledVector, model: LassoModel, label: Double, lastPrediction: Double): LassoModel
  override def predict(dataPoint: LabeledVector, model: LassoModel): Double
}
```

- **Method delta** implements how the model is trained.
- **Method predict** implements how to make a prediction using the model.
3.3 Regression based ML algorithms

3.3.1 Notations

- \( \mathbf{x}_t \) = An \( n \)-dimensional row vector input.
- \( \mathbf{y}_t \) = The label or the outcome
- \( \mathbf{A} = A_{n \times n} \) matrix calculated using \( \mathbf{x} \)
- \( \mathbf{D} = A_{n \times n} \) diagonal matrix calculated using \( \mathbf{A} \) and \( \mathbf{b} \)
- \( \mathbf{M} = A_{n \times n} \) matrix calculated using \( \mathbf{x} \)
- \( \mathbf{b} = A_{n \times 1} \) dimensional column vector calculated using \( \mathbf{x} \) and \( \mathbf{y} \)
- \( \mathbf{\hat{y}}_t = \mathbf{\hat{y}}_t \) = prediction given by the algorithm
- \( \hat{\alpha} = \) difference between in the prediction and label divided by the learning rate and the plus the Euclidean norm of \( \mathbf{x} \).
- \( \mathbf{w} = \) weight vector calculated by using either \( \mathbf{A} \) and \( \mathbf{b} \) or \( \hat{\alpha} \) and \( \mathbf{x} \).

3.3.2 Online Ridge Regression (ORR)

The algorithm performs the well-studied ridge regression algorithm [15] in online mode. Ridge Regression adds a \( L_2 \) norm penalty term to ordinary least squares regression to deal with multicollinearity amongst regression predictor variables. The pseudo-code is as follows:

Initialise: \( \mathbf{A} = aI^{n \times n}, \mathbf{b} = 0^{n \times 1} \& a > 0 \)
FOR \( t = 1, 2, \ldots \)
\hspace{1cm} Read input \( \mathbf{x}_t \in \mathbb{R}^n \)
\hspace{1cm} output \( \mathbf{y}_t = b' \mathbf{A}^{-1} \mathbf{x}_t \in \mathbb{R} \)
\hspace{1cm} \( \mathbf{A} = \mathbf{A} + \mathbf{x}_t \mathbf{x}_t' \)
\hspace{1cm} Read outcome \( \mathbf{y}_t \in \mathbb{R} \)
\hspace{1cm} \( \mathbf{b} = \mathbf{b} + \mathbf{x}_t \mathbf{y}_t \)
END FOR

3.3.3 Aggregation Algorithm for Regression (AAR)

This algorithm can be thought of as a game-theoretic version of ORR [16]. AAR algorithm is last-step min-max optimal [17], which allows AAR to shrink the predictions. Also, AAR has a better upper bound on the cumulative loss in comparison to ORR.
3.3.4 Online Shrinkage via Limit Of Gibbs sampling (OSLOG)

The algorithm is an online version of shrinkage via limit of Gibbs sampling (SLOG) [18] OSLOG uses an $\ell_1$ norm penalty resulting in a difficult problem to bound because $\ell_1$ norm is non-differentiable but is convex. An approximation has been used to obtain sparsity in the solution. The pseudo-code of the algorithm is as follows:

Initialise: $M = 0^{n \times n}, b = 0^{n \times 1}, w = 1^{n \times 1}$ & $a > 0$

FOR $t = 1, 2, ...$

Read input $x_t \in \mathbb{R}^n$

$M = M + x_t x_t^T$

$M^{-1} = \sqrt{D_{w_{t-1}} \left( aI + \sqrt{D_{w_{t-1}} M \sqrt{D_{w_{t-1}}}} \right)^{-1} D_{w_{t-1}}}$

Read outcome $y_t \in \mathbb{R}$

$b = b + x_t y_t$

$w = M^{-1} b$

END FOR

3.3.5 Competitive Online Iterative Ridge Regression (COIRR)

COIRR can be thought of as a game-theoretic version of OSLOG. The algorithm has the best upper bound on cumulative loss under certain conditions. The pseudo-code of the algorithm is as follows:

Initialise: $M = 0^{n \times n}, b = 0^{n \times 1}, w = 1^{n \times 1}$ & $a > 0$

FOR $t = 1, 2, ...$

Read input $x_t \in \mathbb{R}^n$

$D_{w_{t-1}} = \text{diag}(w^1, ..., w^n)$

$y_t = \frac{w x_t}{1 + x_t D_{w_{t-1}}^{-1/2} D_{w_{t-1}}} \in \mathbb{R}$

$M = M + x_t x_t^T$

Read outcome $y_t \in \mathbb{R}$

$b = b + x_t y_t$

$w = M^{-1} b$

END FOR
3.3.6 Competitive Online Normalised Least Mean Squares Regression (TNLMS)

This algorithm is a competitive online regression algorithm, which is inspired from the work done in [19]. It can handle drift based on first order information and is computationally the most efficient regression algorithm in SOLMA [20]. The pseudo-code of the algorithm is as follows:

\[
\begin{align*}
\text{Initialise}: & \quad w = 0^{n \times 1} \land -\infty < \eta < \infty \\
\text{FOR } t = 1, 2, \ldots & \\
\quad & \text{Read input } x_t \in \mathbb{R}^n \\
\quad & \text{output } y_t = w'x_t \in \mathbb{R} \\
\quad & \text{Read } y_t \in \mathbb{R} \\
\quad & \text{Normalise loss } \lambda = \frac{y_t - y_r}{\eta + ||x||^2} \\
\quad & w = w + \lambda x_t \\
\text{END FOR}
\end{align*}
\]

3.4 Classification based ML algorithms

3.4.1 Online SVM

One of the important aspects of Support Vector Machine is that it generalizes easily to a broader set of problems and it has a fast convergence performance. However, traditional SVM methods are not well-suited for online scenarios. Online version of SVM [29], OSVM, is a solution for realtime scenarios, such as the use-case for PROTEUS.

The basic idea behind OSVM is as follows. Initially, an OSVM operator gets data point \( x_i \). Then, the operator pulls global parameters from the master node. Prediction is performed based on the information received. Finally, local parameters are updated and global parameters with respect to the step size.

We implement OSVM on top of SOLMA.

```java
class OSVM extends StreamTransformer[OSVM]
```

The OSVM object features stream events, OSVM model, label and unlabeled vector:

```java
object OSVM extends WithParameters with Serializable {
  type OSVMStreamEvent = Either[(Long, StreamEvent), Label]
  type OSVMModel = (DenseVector[Double], Double)
  type UnlabeledVector = Vector[Double]
  type Label = (Long, Double)
}
```

One essential part of OSVM is prequential training (Listing 9). We provide necessary properties to the stream transformer method, such as global parameters map, OSVM instance. Based on the current OSVM
model and gradient, the update module is built. Because OSVM uses parameter server, we also provide worker and server logic.

```scala
override def transformDataStream(
  instance: OSVM,
  transformParameters: ParameterMap,
  input: DataStream[OSVMStreamEvent]
): DataStream[Either[(Long, OSVM.UnlabeledVector, Double), (Int, OSVM.OSVMModel)]] = {

  val workerParallelism: Int = instance.getWorkerParallelism
  val psParallelism: Int = instance.getPSParallelism
  val pullLimit: Int = instance.getPullLimit
  val featureCount: Int = instance.getFeaturesCount
  val iterationWaitTime: Long = instance.getIterationWaitTime
  val allowedLateness: Long = instance.getAllowedLateness
  val updater = (currModel: OSVM.OSVMModel, gradient: OSVM.OSVMModel) => {
    (currModel._1 - gradient._1, currModel._2 - gradient._2)
  }
  val rnd = new XORShiftRandom()
  val initializer = init(featureCount, -8.0, 4.0, -1.0, 1.0)
  val workerLogic = WorkerLogic.addPullLimiter(
    new OSVMWorkerLogic(
      new OSVMAlgorithm(instance),
      pullLimit)
    )
  val serverLogic = new RangePSLogicWithClose[OSVM.OSVMModel](featureCount, initializer, updater)
  val paramPartitioner: WorkerToPS[OSVM.OSVMModel] => Int = rangePartitionerPS(featureCount)(psParallelism)
  val winPartition: PSToWorker[OSVM.OSVMModel] => Int = {
    case PSToWorker(workerPartitionIndex, _) => workerPartitionIndex
  }
  implicit val inputTypeInfo = createTypeInformation[OSVMStreamEvent]

  val partitionedInput = input.partitionCustom(
    new Partitioner[Long] {
      override def partition(k: Long, total: Int): Int = {
        (k % total).toInt
      }
    },
    event: OSVM.OSVMStreamEvent) => event match {
    case Left(v) => v._1
    case Right(v) => v._1
  })

  FlinkParameterServer.transform(
    partitionedInput,
    workerLogic,
    serverLogic,
    workerParallelism,
    psParallelism,
    }

```
Listing 9. OSVM Prequential training

Listing 10 shows the main intuition behind OSVM algorithm. So, as we explained above, for every data point, the model checks current and global model and performs necessary predictions based on the results of previous step.

class OSVMAlgorithm(instance: OSVM) extends BaseOSVMAlgorithm[UnlabeledVector, Double, OSVMModel] {

  override def delta(
    dataPoint: UnlabeledVector,
    model: OSVMModel,
    label: Double,
    t: Long
  ): (DenseVector[Double], Double) = {

    val c = instance.getCParam()
    var sign = 0.0
    if (label * (dataPoint dot model._1 + model._2) < 1){
      sign = 1.0
    }
    val dirw = model._1 - c * label * dataPoint * sign
    val dirb = - label * sign
    (dirw * (1.0 / t), dirb * (1.0 / t))
  }

  override def predict(
    dataPoint: UnlabeledVector,
    model: OSVMModel): Double = {
    signum(dataPoint dot model._1 + model._2)
  }
}

Listing 10. OSVM algorithm

3.4.2 Online Bilevel Stochastic Gradient for Support Vector Machine (OBSG_SVM)

Online bi-level stochastic gradient algorithm for support vector machine (OBSG-SVM) learns the hyperplane and adjust the hyperparameter sequentially. In this setting, the algorithm goes through the data one by one as the data comes sequentially. We denote $x_t, t = 1, 2, ..., T$ the data sequence. When the dataset is too large, going through the data points is more convenient. A competitive convergence guarantee is given in [21] showing that with the stochastic algorithm going through it sequentially with a small number of
passes, the convergence rate will not be significantly worse than with replacement sampling. In [22], the authors stated that going through one dataset multiple times can be regarded as a bias-variance trade-off.

In OBSG-SVM, instead of splitting the training set into a training fold and validation fold, we adjust the model with the data that has been seen. At time t, the algorithm receives one data point \( x_t \). OBSG-SVM firstly checks if the data point is an error vector (defined as \( 1 - y_t(w^T x_t) > 0 \) in hinge loss). Then, it uses the current error vector to train the lower optimisation problem and use the current and the previous error vector to adjust the hyperparameter. The OBSG-SVM algorithm is summarized in below algorithm.

### Algorithm

1. Initialise \( w^1 = 0 \), \( C^1 > 0, C_{\text{min}}, C_{\text{max}} \)
2. Initialise \( x_p = x_1, y_p = y_1 \)
3. While True:
   - Read data point \( x_t \) \( \in R^N \)
   - Predict \( \hat{y}_t = \text{sign}(w_t \cdot x_t) \)
   - Receive true label \( y_t \in \{-1,1\} \)
   - If \( 1 - y_t(w^T x_t) > 0 \)
     - Calculate \( \nabla_w G_t(C, w) = w - C y_t x_t \)
     - Calculate \( \nabla_w F_p(C, \bar{w}(C)) = -y_p x_p^T y_t x_t \)
     - Perform the move \( w^{t+1} = w^t - \alpha^t w \nabla_w G_t(C, w) \)
     - Perform the move \( C^{t+1} = C^t - \alpha^t C \nabla_C F_p(C, \bar{w}(C)) \)
   - If \( C^{t+1} < C_{\text{min}}, C^{t+1} = C_{\text{min}} \)
   - If \( C^{t+1} > C_{\text{max}}, C^{t+1} = C_{\text{max}} \)
   - \( x_p = x_t, y_p = y_t \)

In the algorithm, we take \( G_t \) as the inner objective function with the data point \( x_t \) and \( F_p \) as the outer objective function with the data point \( x_p \) which is the previous error vector.

To explain the learning process more explicitly, we assume that \( x_{i-1}, x_i, x_{i+1} \) are three error vectors in the learning sequence. For the error vector \( x_i \), OBSG-SVM uses \( x_i \) to learn the hyperplane (inner level problem), and uses both \( x_{i-1} \) and \( x_i \) to adjust the hyperparameter \( C \) (outer level problem). For the error vector \( x_{i+1} \), OBSG-SVM uses \( x_{i+1} \) to learn the hyperplane (inner level problem) and both \( x_i \) and \( x_{i+1} \) to adjust the hyperparameter \( C \) (outer level problem). Note that each error vector is used once for learning the hyperplane and used twice for adjusting the hyperparameter \( C \). This is interpreted as "double insurance" of the hyperparameter \( C \) by using twice of the error vectors.

To implement OBSG-SVM, the starting point \( w^1 \) is randomly selected. The value \( C^1 > 0 \) is chosen according to experience or a small grid search. The same as BSG-SVM, the values of \( C_{\text{min}} \) and \( C_{\text{max}} \) are used to avoid too small and too large values of \( C \) which might cause numerical scaling problems in practice. The diminishing step size rule is still applicable since we are using stochastic gradient descent. Instead of learning with the randomly chosen data points, we do online passes on the training set. Practical choices of the step size are \( O(1/t) \) and \( O(1/\sqrt{t}) \).

Distributed version of the algorithm is provided below.

### Worker

1. Initialise \( x_p = x_1, y_p = y_1 \)
2. While True:
   - Read data point \( x_t \) \( \in R^N \)
   - Predict \( \hat{y}_t = \text{sign}(w_t \cdot x_t) \)
   - Receive true label \( y_t \in \{-1,1\} \)
If $1 - y_t(w^T x_t) > 0$
Calculate $\nabla_w G_t(C, w) = w - C y_t x_t$
Calculate $\nabla_w F_p(C, \bar{w}(C)) = -y_p x_p^T y_t x_t$
Perform the move $C^{t+1} = C^t - \alpha C \nabla_c F_p(C, \bar{w}(C))$
If $C^{t+1} < C_{min}$, $C^{t+1} = C_{min}$
If $C^{t+1} > C_{max}$, $C^{t+1} = C_{max}$
$x_p = x_t, y_p = y_t$
$dev^t_i = \alpha^t C \nabla_w G_t(C, w)$
Push local parameters $dev^t_i$ to the master

Master

Initialize: $w_1 = 0, C > 0, 0 < \rho \leq 1$
While True:
Receive $m$ local parameters $dev^t_i, i = 1, \ldots, m$ from the $n$ workers $m \leq n$
Compute the global parameter $\bar{w}_{t+1}$ from the collected local information
$\bar{w}_{t+1} = \bar{w}_t - \sum_{i=1}^m dev^t_i$
t=t+1

3.4.3 Passive-Aggressive Learning (PA)

PA [23] algorithms solve an optimisation problem in each round which is formulated as:

$$w_{t+1} = \text{argmin}_w \frac{1}{2} ||w - w_t||_2 \quad \text{s.t.} \quad l_t(w) = 0.$$  

where $l_t(w)$ is the hinge loss on instance $(x_t, y_t)$, i.e. $l_t(w) = \max\{0, 1 - y_t (w_t \cdot x_t)\}$. In each round, PA learning aggressively forces the loss to be zero and passively makes sure that the new updated weight vector will be not too far from the previous weight vector. For detailed PA derivation, please refer to deliverable D4.4 [24].

To cope with noise and be more robust, Crammer et. al. introduced a slack variable $\xi$ into the optimisation problem using two types of penalty, namely, linear and quadratic. This results in two different variants of the PA algorithm, i.e. PA-I and PA-II. The algorithms are as follows:

Initialise $w_1 = 0, C > 0, 0 < \rho \leq 1$
While True:
Read data point $x_t \in \mathbb{R}^n$
Predict $\hat{y}_t = \text{sign}(w_t \cdot x_t)$
Receive true label $y_t \in \{-1, 1\}$
Compute coefficient $\alpha_t$
$\alpha_t = \frac{l_t(w_t)}{||x_t||^2}$ (PA)
\[
\begin{align*}
\alpha_t &= \min \left\{ C, \frac{i_t(w_t)}{||x_t||^2} \right\} \quad (PA - I) \\
\alpha_t &= \frac{i_t(w_t)}{||x_t||^2 + \frac{1}{2C}} \quad (PA - II) \\
\text{Update } w & \\
\quad w_{t+1} &= w_t + \alpha_t y_t x_t \\
\text{t} &= t+1
\end{align*}
\]

We implement distributed PA method for online binary classification as follows. To design a distributed version of PA, we opt for the Master/Slave architecture. The structure of the Master and Worker is given as follows. At round \( t \), each worker receives a new data point, predicts the label with the weight vector pulled from Master. After receiving the true label, each worker computes a local parameter and sends them to the Master. Then, the master aggregates the local parameter and updates the model.

**Worker**

```
While True:
    Read data point
    Pull global parameter \( w_t \) from the Master
    Predict \( \hat{y}_t = \text{sign}(w_t \cdot x_t) \)
    Receive true label \( y_t \in \{-1,1\} \)
    Compute coefficient \( \alpha_t \)
    Compute local parameter \( dev_t^i = \alpha_t y_t x_t \)
    Push local parameters \( dev_t^i \) to the master
```

**Master**

```
Initialize: \( w_1 = 0, C > 0, 0 < \rho \leq 1 \),
While True:
    Receive m local parameters \( dev_t^i, i = 1, \ldots, m \) from the n workers \( m \leq n \)
    Compute the global parameter \( w_{t+1} \) from the collected local information
    \[
    w_{t+1} = w_t + \sum_{i=1}^{m} dev_t^i
    \]
    \( t = t+1 \)
```

### 3.4.4 Online WAPA

Being motivated by the advantages of averaging methods, Weighted Averaging PA (WAPA) learns to enhance the learning ability when there exist fluctuations such as label noise. WAPA passively retains the weighted average \( \bar{w} \) of the previous weight vectors when the hinge loss is zero; otherwise it reduces the hinge loss suffered on the current data point less aggressively than PA updates. In the passive step, WAPA employs the Exponential Weighted Moving Average (EWMA) \( \bar{w}_{t+1} = (1 - \rho)w_{t+1} + \rho \bar{w}_t \) which gives higher weight to the more recent weight vectors. The parameter \( \rho \) is a smoothing factor in EWMA. Combining with the PA optimisation in each round, the update for WAPA is \( \bar{w}_{t+1} = \bar{w}_t + \rho \alpha_t y_t x_t \) with the coefficient \( \alpha_t \) as shown in the following algorithm. WAPA provides extra robustness since the latest data might contain up to date information. For detailed WAPA derivation, please refer to deliverable D4.4 [4]. The algorithm are presented as follows:
Initialise $w_1 = 0$, $C > 0$, $0 < \rho \leq 1$

While True:

Read data point $x_t \in R^n$

Predict the true label $\hat{y}_t = \text{sign}(w_t \cdot x_t)$

Receive true label $y_t \in \{-1,1\}$

Compute coefficient $\alpha_t$

$$\alpha_t = \frac{l_t(\tilde{w}_t)}{||x_t||^2}$$ \hspace{1cm} (WAPA)

$$\alpha_t = \min \left\{ C, \frac{l_t(\tilde{w}_t)}{||x_t||^2} \right\}$$ \hspace{1cm} (WAPA - I)

$$\alpha_t = \frac{l_t(\tilde{w}_t)}{||x_t||^2 + \frac{1}{2C}}$$ \hspace{1cm} (WAPA - II)

Update $\tilde{w}$

$$\tilde{w}_{t+1} = \tilde{w}_t + \rho \alpha_t y_t x_t$$

$t = t+1$

Distributed Online WAPA is implemented with the same structure as Distributed PA.

**Worker**

While True:

Read data point $x_t \in R^n$

Pull global parameter $\tilde{w}$ from the Master

Predict $\hat{y}_t = \text{sign}(\tilde{w} \cdot x_t)$

Receive true label $y_t \in \{-1,1\}$

Compute coefficient $\alpha_t$

Compute local parameter $dev^i_t = \rho \alpha_t y_t x_t$

Push local parameters $dev^i_t$ to the master

**Master**

Initialize: $w_1 = 0$, $C > 0$, $0 < \rho \leq 1$

While True:

Receive $m$ local parameters $dev^i_t, i = 1, \ldots, m$ from the $n$ workers $m \leq n$

Compute the global parameter $\tilde{w}_{t+1}$ from the collected local information

The below algorithm is executed in master machine. In each step $t$ is increase by one: $t=t+1$

$$\tilde{w}_{t+1} = \tilde{w}_t + \sum_{i=1}^{m} dev^i_t$$
3.5 Sampling algorithms

3.5.1 Simple reservoir sampling

Reservoir sampling belongs to a family of randomized algorithms. The main idea behind randomized algorithms is to pose some degree of randomness as part of its logic. The degree should be configurable by user specified parameters.

We implement reservoir sampling in SOLMA pipeline developed on top of Apache Flink [11]. We can see the base class from Listing 11. The main class extends from StreamTransformer class in SOLMA pipeline. We provide the parameter, which is reservoir size.

```scala
@Proteus
class SimpleReservoirSampling extends StreamTransformer[SimpleReservoirSampling] {

def setReservoirSize(size: Int): SimpleReservoirSampling = {
  parameters.add(ReservoirSize, size)
  this
}
}
```

**Listing 11. Simple reservoir sampling class**

**Error! Not a valid bookmark self-reference.** shows the main logic behind simple reservoir sampling function. This function is implemented as a black-box by a user and provided to SOLMA pipeline. It is the job of SOLMA pipeline to utilize this implicit transformation in the correct places of distributed streaming pipeline. Here we implement `TransformDataStreamOperation` trait, from SOLMA library. After deducing the type info of record tuples, we transform our transformation into user defined function inside a flatmap operation. Flatmap operator takes one record as an input and may produce zero or more output records. We adopt stateful flatmap operator, as we need to hold state for reservoirs. If the reservoir size is less than a given threshold, then we add incoming tuple to the state of operator. Otherwise, we generate an integer number and based on that, we decide whether to hold new record in reservoir to not.

```scala
implicit def transformSimpleReservoirSampling[T <: Vector : TypeInformation : ClassTag] = {
  new TransformDataStreamOperation[SimpleReservoirSampling, T, T] {
    override def transformDataStream(
      instance: SimpleReservoirSampling,
      transformParameters: ParameterMap,
      input: DataStream[T]) :
      DataStream[T] = {
      val resultingParameters = instance.parameters ++ transformParameters
      val statefulStream = FlinkSolmaUtils.ensureKeyedStream[T](input,
        resultingParameters.get(PartitioningOperation))
      val k = resultingParameters(ReservoirSize)
      val gen = new XORShiftRandom()
      implicit val typeInfo = createTypeInformation[(Long, Array[T])]
      statefulStream.flatMapWithState((in, state: Option[(Long, Array[T])]) => {
        val (element, _) = in
        state match {
          case Some(curr) => {
            ```
val (streamCounter, reservoir) = curr
val data = new mutable.ListBuffer[T]()
if (streamCounter < k) {
  reservoir(streamCounter.toInt) = element
  data += element
} else {
  val j = gen.nextInt(streamCounter.toInt + 1)
  if (j < k) {
    reservoir(j) = element
    data += element
  }
}(data, Some((streamCounter + 1, reservoir)))
}
case None => {
  val reservoir = Array.ofDim[T](k)
  reservoir(0) = element
  (Seq(element), Some((1L, reservoir)))
}
}
}

Listing 12 Main transformation function for simple reservoir sampling

3.5.2 Adaptive reservoir sampling

Adaptive reservoir sampling maintains the reservoir sample after the size is adjusted. It is proven that, when the reservoir size decreases, the algorithm generates a sample in the reduced reservoir with a 100% uniformity confidence (UC), defined in [25]. This means each item in the reduced reservoir has an equal probability of being selected from the stream. In contrast, when the reservoir size is increased, the enlarged reservoir cannot be maintained with a 100% uniformity confidence.

The adaptive reservoir sampling algorithm runs as follows. If the reservoir size does not change, reservoir sampling is used. If the reservoir size decreases by $\delta$, the algorithm discards $\delta$ items from the original reservoir and then continues. In contrast, if the reservoir size increases by $\delta$, the algorithm computes the minimum value of $m$ (defined as the number of incoming items used to fill the enlarged reservoir) that causes the uniformity confidence to exceed a given threshold $\zeta$. Afterwards, it flips a biased coin to decide on how many items $x$ are retained among the $k$ items in the original reservoir. $k-x$ items are randomly discarded from the original reservoir. The enlarged reservoir is refilled with $k+\delta-x$ items from the arriving $m$ items. For the detailed algorithm please see deliverable D4.2 [27].

3.5.3 Weighted reservoir sampling

Weighted random sampling is used in cases where items are assigned with weights. The probability of each item being selected is determined by its weight. One interpretation is that the probability of being selected is
determined by the weight of each item. We implement here the algorithm A-Res [26]. The key $k_i$ of stream item $S_i$ in the population is defined as $k_i = u_i^{1/w_i}$ ($w_i$ is the weight of the $i$th streaming data) with a uniform random number $u_i = \text{random}(0,1)$. Firstly, the algorithm keeps the first $m$ items in the reservoir and calculates their key. If the key of the new arriving item is larger than the minimum key in the reservoir, the minimum key item is replaced by the new arriving one. This step is repeated until the data stream is exhausted. For the detailed algorithm please see deliverable D4.2 [27].

### 3.6 Moments

The moving average updates as new values become available, the oldest data points must be dropped from the set and new data points come in to replace them. Thus, the data set is constantly "moving" to account for new data as it becomes available. This method of calculation ensures that only the current information is averaged.

The weighted average formula is used to calculate the average value of a set of numbers according to their relevance. The relevance of each number is called its weight. The weights can be between 0 and 1 inclusive.

$$\text{Weighted Avg} = w_1x_1 + w_2x_2 + \ldots$$

In the above equation $x_i$ denotes the incoming stream $i$-th component and $w_i$ denotes the relevant weight.

The exponential average is a weighted average, where the weighting decreases and increases exponentially.

### 3.7 Frequent Directions

Frequent directions is a matrix sketching algorithm. One of its main features is that it is deterministic. The algorithm is built on the similarity between matrix sketching and frequency estimation problem and provides specific error bounds for it.

Let B be axb sketch matrix of A which is an input matrix. For every input row from A, the matrix B is updated. The algorithm keeps the invariant that last added row of sketch always all-zero valued. These all-zero valued rows are replaced by the rows that are transferred from A to B. During the execution, sketch is rotated using its singular value decomposition (SVD). As a result of this operation, row of the sketch matrix B are orthogonal and remain in descending magnitude order. More details of the algorithm can be found in original paper [28].

Listing 13 shows main class of FrequentDirections. We extend from StreamTransformer interface and set feature count, sketch size and aggregation hint. These methods are essential to provide user-specified parameters to the algorithm. The main function, on the other hand, is shown in Listing 14. Basically, we keep sketches in stateful flatmaps. So, out sketching problem is translated to Flink’s DataStream API. If user enables aggregations, we use folding technique to aggregate relative values inside the sketch.

```scala
class FrequentDirections extends StreamTransformer[FrequentDirections] {

    import FrequentDirections._

    def setFeaturesNumber(count: Int): FrequentDirections = {
        parameters.add(FeaturesNumber, count)
        this
    }

    def setSketchSize(size: Int): FrequentDirections = {
```
parameters.add(SketchSize, size)
    this
}

def enableAggregation(enabled: Boolean): FrequentDirections = {
    parameters.add(AggregateSketches, enabled)
    this
}

Listing 13. Frequent Directions class

```scala
implicit def transformFrequentDirections[T <: Vector : TypeInformation : ClassTag] = {
    new TransformDataStreamOperation[FrequentDirections, T, T] {
        override def transformDataStream(
            instance: FrequentDirections,
            transformParameters: ParameterMap,
            input: DataStream[T]) :
            DataStream[T] = {
            val resultingParameters = instance.parameters ++ transformParameters
            val statefulStream = FlinkSolmaUtils.ensureKeyedStream[T](
                input, resultingParameters.get(PartitioningOperation))
            ell = resultingParameters(SketchSize)
            d = resultingParameters(FeaturesNumber)
            assert(ell < d * 2, "the sketch size should be smaller" +
                " than twice the number of features")
            val sketchesStream = statefulStream.flatMapWithState(
                (in, state: Option[Sketch]) => {
                    val (elem, _) = in
                    val out = new ListBuffer[BreezeVector[Double]]()
                    val sketch = updateSketch(elem.asBreeze, state, out)
                    (out, Some(sketch))
                })
            if (resultingParameters(AggregateSketches)) {
                sketchesStream.fold(None.asInstanceOf[Option[Sketch]])((acc, item) => {
                    Some(updateSketch(item, acc))
                }).flatMap((acc, out) => {
                    val Sketch(zeroRows, matrix) = acc.get
                    val toOutputType = (x: BreezeVector[Double]) =>
                        x.copy.fromBreeze.asInstanceOf[T]
                    if (zeroRows.isEmpty) {
                        (0 until matrix.rows) foreach (i => out collect
toOutputType(matrix(i, ::).t))
                    }
                })
            } else {
                sketchesStream.map(x => {
                    x.fromBreeze.asInstanceOf[T]
                })
            }
        }
    }
}
```

Listing 14. Main transformation for Frequent Directions.
4 Conclusions

Scalable online machine learning is one of the fundamental pieces of PROTEUS. In this work package, we develop SOLMA library, which is used not only in the scope of PROTEUS, but also for other scalable machine learning problems. The library is open source and open for contributions. In this deliverable, we provide a general abstraction for SOLMA library. The abstraction is useful to structure algorithm implementation. We benefit from state-of-the-art designs, such as FlinkML and python scikit. One main difference of SOLMA abstraction and existing works is that the former is designed to online machine learning scenarios, while for latter ones are designed for mainly offline algorithms. Moreover, we provide a general overview of algorithms implemented on top of SOLMA. The full version of our library is open source and ready to use.
References


[5] Apache Kafka project


