Horizon 2020 Program (2014-2020)

Big data PPP

Research addressing main technology challenges of the data economy

Industrial-Driven Big Data as a Self-Service Solution

Industrial-Driven Big Data as a Self-Service Solution

D3.1: Batch Processing Analytics module implementation[†]

Abstract: This deliverable reports on the design and approach of the I-BiDaaS batch-processing platform. This report describes the state of the work done in work package 3 until month 12th. We have two different types of advances. The first one is related to the Advance Machine Learning (ML) submodule of the batch processing module of the I-BiDaaS platform. The second one are advances on the technological components of the batch processing module that will support the execution of the applications. This type of advances has been mainly driven by the implementation of the Minimum Viable Product of I-BiDaaS that is based on a use case scenario provided by CAIXA.

Contractual Date of Delivery	31/12/2018
Actual Date of Delivery	31/12/2018
Deliverable Security Class	Public
Editor	*Yolanda Becerra (BSC)*
Contributors	BSC, IBM, UNSPMF, FORTH
Quality Assurance	*Christos Tzagkarakis (FORTH), Kostas Lampropoulos (FORTH) Dusan Jakovetic (UNSPMF)*

[†] *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780787.*

The *I-BiDaaS* Consortium

Foundation for Research and Technology – Hellas (FORTH)	Coordinator	Greece
Barcelona Supercomputing Center (BSC)	Principal Contractor	Spain
IBM Israel – Science and Technology LTD (IBM)	Principal Contractor	Israel
Centro Ricerche FIAT (FCA/CRF)	Principal Contractor	Italy
Software AG (SAG)	Principal Contractor	Germany
Caixabank S.A. (CAIXA)	Principal Contractor	Spain
University of Manchester (UNIMAN)	Principal Contractor	United Kingdom
Ecole Nationale des Ponts et Chaussees (ENPC)	Principal Contractor	France
ATOS Spain S.A. (ATOS)	Principal Contractor	Spain
Aegis IT Research LTD (AEGIS)	Principal Contractor	United Kingdom
Information Technology for Market Leadership (ITML)	Principal Contractor	Greece
University of Novi Sad Faculty of Sciences (UNSPMF)	Principal Contractor	Serbia
Telefonica Investigation y Desarrollo S.A. (TID)	Principal Contractor	Spain

Document Revisions & Quality Assurance

Internal Reviewers

1. *Christos Tzagkarakis, Kostas Lampropoulos (FORTH)*
2. *Dusan Jakovetic (UNSPMF)*

Revisions

Version	Date	By	Overview
0.0.1	16/10/2018	Cesare Cugnasco	Initial version of the index
0.0.2	8/11/2018	Cesare Cugnasco	Review of the index
0.0.3	19/11/2018	Yolanda Becerra	Added contributions from BSC
0.0.4	19/11/2018	Dusan Jakovetic, Stevo Rackovic	Added contributions from UNSPMF
0.0.5	21/11/2018	Omer-Yehuda Boehm	Added contributions from IBM
0.1.0	23/11/2018	Yolanda Becerra	First draft to internal reviewers
0.1.1.	17/12/2018	Giorgos Vasiliadis	Added contributions from FORTH
0.2.0	21/12/2018	Yolanda Becerra	Second draft to internal reviewers
0.2.1	29/12/2018	Internal Reviewers	Approval of the second draft
0.2.2	29/12/2018	Yolanda Becerra	Final version released

Index of Contents

Index of Figures

Index of Tables

1 Executive summary

This report describes the advances in the work package 3 of the I-BiDaaS project: Batch processing innovative technologies for rapidly increasing historical data. The work described in this report summarises the tasks developed from month 6th, the starting month of the work package, until month 12th.

This work package focuses on the software stack required for the batch module of the I-BiDaaS platform. We can divide the work developed until now into two parts: advances in the machine learning algorithms that the I-BiDaaS platform will provide to users, and advances in the technological components that, until this moment, are driven by the requirements of the Minimum Viable Product (MVP) of the project. The MVP is based on a real use case provided by one of the project partners (CAIXA). To support this prototype, we have implemented a data analytics application, and we have extended the software components of the platform to meet the requirements of this application.

2 Introduction

2.1 Overview and contributions to the project goals

The goal of work package 3, Batch processing innovative technologies for rapidly increasing historical data, is to develop the I-BiDaaS batch processing module. This module is a software component that is responsible for machine learning and batch analytics within the I-BiDaaS platform. In this report, we describe the work done by the tasks of this work package during the six initial months (from the 6th month to the 12th month of the project). This work package is organized into four tasks:

- Task 3.1: Advancing data analytics via structured (non) convex optimization
- Task 3.2: Innovative Distributed solvers library implementation with COMPSs programming framework
- Task 3.3: Data management: Hecuba
- Task 3.4: Hecuba interaction with TDF

Following, we identify which part of the reported work by this deliverable has been performed by each task and how this work contributes to the final goals of the project:

- Advances in the Machine Learning algorithms that the I-BiDaaS platform will provide to users. This work is part of both task 3.1 (the theoretical advances) and task 3.2 (the implementation of the algorithms using PyCOMPSs [5]). This work contributes to the final goals of the projects by starting the building of the pool of data analytics algorithms that the I-BiDaaS platform will provide to users.
- Advances in the Hecuba tool [6]. This work has been developed as part of task 3.3. This work contributes to the final goals of the project by starting the implementation of the data management software of the batch module of the I-BIDaaS platform.
- Advances in the integration of Test Data Fabrication (TDF) tool [19][20] and Hecuba, as part of the work developed by task 3.4. The contribution to the final goals of the project is the definition of how to integrate the tool that generates synthetic but realistic data with the data management component of the I-BiDaaS platform.

2.2 Structure of the report

This report is organized as follows. First, to give context to the work package advances, we start the report with a description of the architecture of the I-BiDaaS platform (see section 3) as well as of the main components of the software stack (see section 4).

Following, we describe the work done until now. We can divide this work into two threads: advances in the machine learning algorithms that the I-BiDaaS platform will provide to users, and advances in the technological components that, until this moment, are driven by the requirements of the Minimum Viable Product (MVP) of the project. The MVP is based on a real use case provided by one of the project partners (CAIXA). To support this prototype, we have implemented a data analytics application, and we have extended the software components of the platform to meet the requirements of this application.

Section 5 describes the advances of the first thread (machine learning algorithms), section 6 describes the algorithm and the implementation of the data analytics application of the MVP and section 7 describes the extensions implemented on the software stack to support the application.

3 The role of the batch module in I-BiDaaS architecture

The I-BiDaaS batch processing module is a software component that is responsible for machine learning and batch analytics within the I-BiDaaS platform. It contains two sub-modules: 1) COMPSs (COMP Superscalar) programming model [4], and 2) Advanced Machine Learning sub-module (provided by UNSPMF). The former is a task-based programming model that allows for a simplified development of implementations of algorithms that are to be run over a distributed infrastructure (cluster, gird, or cloud). The latter is an actual pool of machine learning and analytics algorithms implemented within the COMPSs programming model and Python, incorporating external codes like Message Passing Interface (MPI), when needed. The utilization of COMPSs allows the users to program their applications through a sequential paradigm with an annotation of code pieces that can be parallelized, while the actual parallelization is done automatically by the COMPSs runtime module. This allows for a short development time with respect to standard solutions. The algorithmic pool will be evolving as the platform lives, and the implementations will be available in the I-BiDaaS knowledge repository for re-use. The pool will also consider integration with, and import from, existing libraries of algorithms like Apache Mahout [1] and Tensorflow [2].

The batch processing module will be exposed to the I-BiDaaS user interface so that various modes of operation will be available to end users. More precisely, the users will be able to utilize ready-to-use algorithms from the pool that are activated via one click from the user interface. In addition, while the algorithms' parameters will have built-in default values, they will also provide a user interface for tuning parameters without the need for re-coding. Furthermore, code templates will be available so that new-but-similar-to-existing applications are not developed from scratch. Finally, interactive querying to facilitate exploration and experimentation will be enabled by the Qbeast (see Section 4.3).

The position of the batch processing module with respect to the overall I-BiDaaS platform is described next (**Figure 1**). The module interacts with the Universal Messaging (UM), Advanced Visualizations, COMPSs runtime, and Hecuba DataBase (DB) system, including Hecuba database and Qbeast indexing system. Another important interaction of the module is with the Test Data Fabrication platform that is accomplished indirectly through the Universal Messaging. In more detail, the interactions are as follows. The batch processing module gets data from the Hecuba DB system, the main database system of the I-BiDaaS platform. The COMPSs runtime module enables that the tasks annotated as parallelizable within the COMPSs programming model are actually executed in parallel. Interactive querying to facilitate exploration and experimentation will be enabled by Qbeast. Through the Universal Messaging, the module delivers to the data fabrication platform certain analytics results and meta-data that are used for semi-automatic and automatic data fabrication.

One can see that, in a broader sense, due to tight interactions, the batch processing module encapsulates in its software stack Test Data Fabrication (TDF), COMPSs, Hecuba, Cassandra, and Qbeast. The software stack is detailed in section 4.

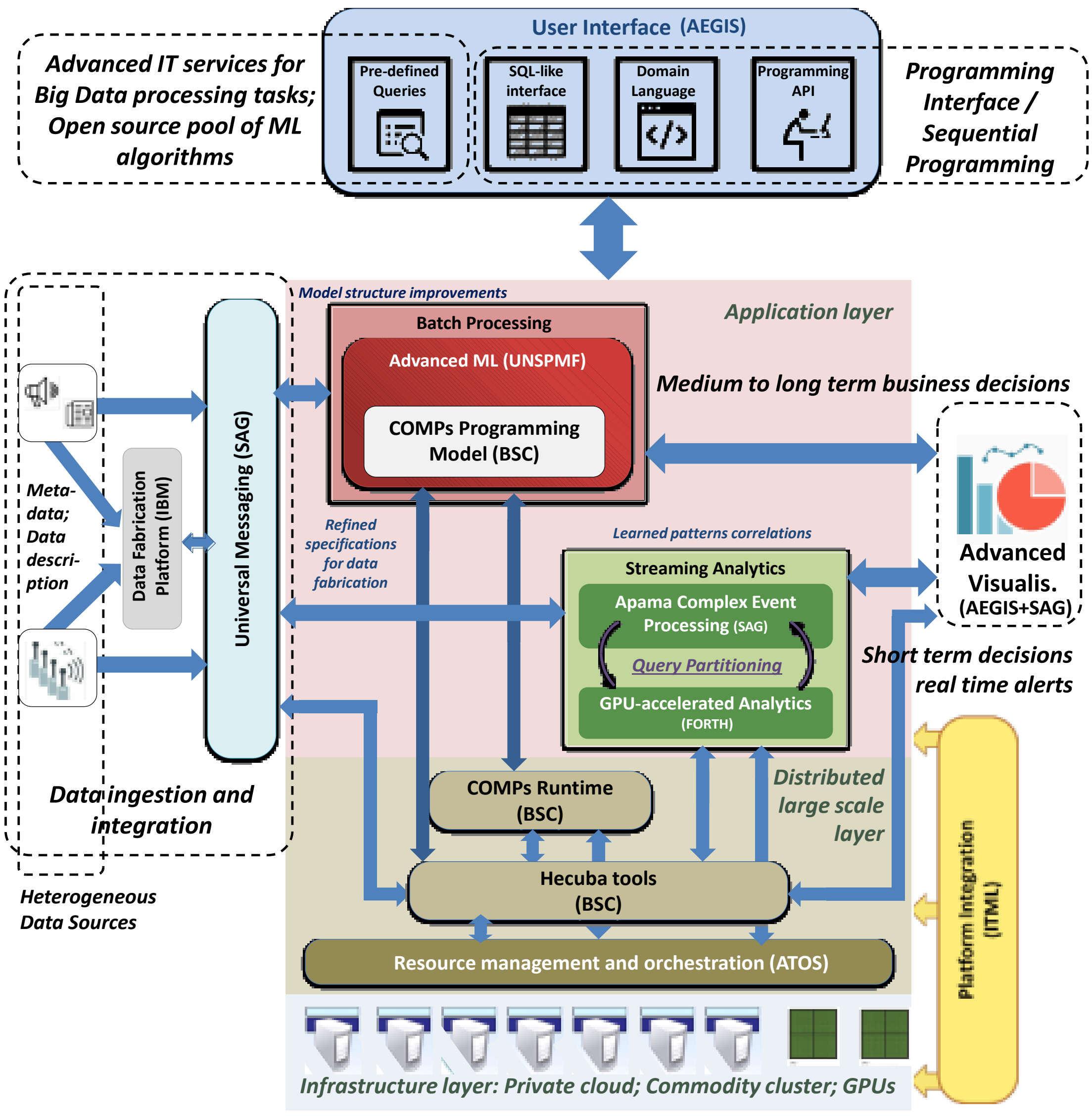
User Interface (AEGIS)
Pre-defined Queries
SQL-like interface
Domain Language
Programming API
Advanced IT services for Big Data processing tasks; Open source pool of ML algorithms
Programming Interface / Sequential Programming
Model structure improvements
Application layer
Batch Processing
Advanced ML (UNSPMF)
COMPs Programming Model (BSC)
Medium to long term business decisions
Advanced Visualis. (AEGIS+SAG)
Meta-data; Data descri-ption
Data Fabrication Platform (IBM)
Universal Messaging (SAG)
Refined specifications for data fabrication
Learned patterns correlations
Streaming Analytics
Apama Complex Event Processing (SAG)
Query Partitioning
GPU-accelerated Analytics (FORTH)
Short term decisions real time alerts
Data ingestion and integration
COMPs Runtime (BSC)
Distributed large scale layer
Hecuba tools (BSC)
Heterogeneous Data Sources
Resource management and orchestration (ATOS)
Platform Integration (ITML)
Infrastructure layer: Private cloud; Commodity cluster; GPUs

Figure 1: I-BiDaaS platform

4 The software architecture

In this section, we give an overview of the main software stack of the batch module in the I-BiDaaS platform. Namely, these are the following: Test Data Fabrication (TDF), the platform to generate realistic synthetic data provided by IBM; COMPSs the programming model and the runtime to control the execution of the application, provided by BSC; Hecuba and Qbeast, provided by BSC, that implements the interface to access persistent data and Cassandra, which is the database used as backing storage.

4.1 Test Data Fabrication (TDF)

TDF is a web-based central platform for generating high-quality, realistic data for testing, development, and training. The platform provides a consistent and organizational wide methodology for creating test data for data bases and files. A user can specify rules and meta-data, based on which the platform fabricates realistic synthetic data and places the data in a database or a file. The data and the meta-data logic are extracted automatically and are augmented by application logic and testing logic modelled by the user. The modelling is performed via rules (constraints) that the platform provides. Once the user requests the generation of a certain amount of data into a set of test databases or test files, the platform core constructs a constraint satisfaction problem (CSP) and runs the IBM integrated solver to find a solution (which is the generated data) such that the modelled rules as well as the internal data consistency requirements are all satisfied. The platform is capable of generating data from scratch, inflating existing databases or files, moving existing data, and transforming data from previously existing resources, such as old test databases, old test files and also production data. In essence, the platform provides a comprehensive and hybrid solution that is capable of creating a mixture of synthetic and real data according to user requirements. The generated data can be written into most common databases, e.g., DB2, Oracle, Ms SQL Server, PostgreSQL and also SQLite, or into files with common formats, e.g., CSV, XML, JSON and positional flat files. In I-BiDaaS, we plan to provide two different options to connect TDF with the database: the first one will be TDF to access Apache Cassandra using the Hecuba interface and, the second one will be using UM as the mechanism to deliver data to Hecuba.

4.2 COMPSs

COMPSs [4], is a task–based programming model designed to facilitate the development of applications for distributed infrastructures, such as Clusters, Grids and Clouds.

Starting from the original sequential application, written using different programming languages (Python, Java, C++), the COMPSs runtime exploits the inherent parallelism of the code execution by detecting and taking care of the data dependencies that may exist between invocation of different parts of the code (tasks). One of the most relevant features of this programming model is the transparency with regard to the computational infrastructure that allows to completely isolate the application from the execution logic.

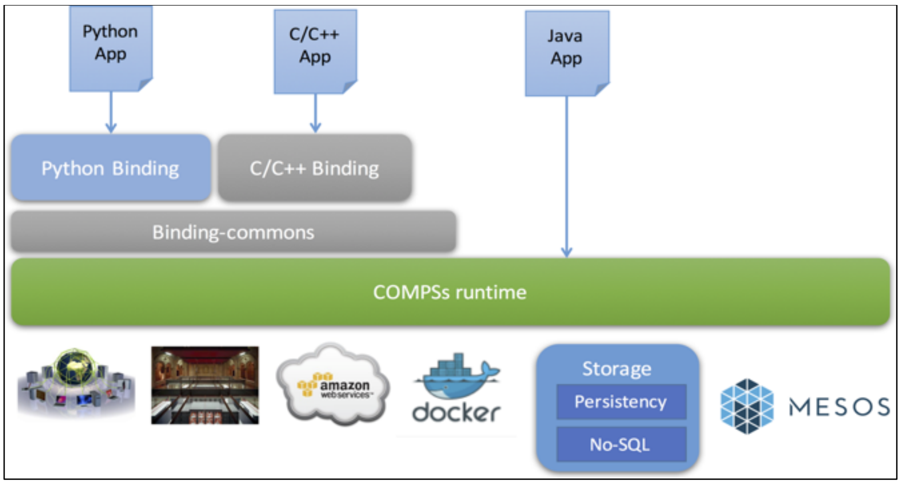

Figure 2: COMPSs architecture

As depicted in Figure 2, COMPSs has native support for Java applications with bindings for C/C++ code and Python scripts. In the model, the developer is mainly responsible for:

- Identifying the functions to be executed as asynchronous parallel tasks;
- Annotating them with a standard Python decorator or Java interface.

Then, the runtime [3] is in charge of exploiting the inherent concurrency of the execution of the functions, automatically detecting the data dependencies between tasks and spawning those tasks to the available resources, which can be nodes in a cluster, cloud or containers. A key aspect of providing an infrastructure-unaware programming model is that programs can be developed once and run on multiple backends, without having to change the implementation.

In COMPSs, the programmer is freed from having to deal with the details of the specific platform, since these details are handled transparently by the runtime. The availability of different connectors, each implementing the specific provider API (e.g. Cloud providers), makes it possible to run computational loads on multiple backend environments without the need for code adaptation. In cloud environments, COMPSs provides scaling and elasticity features that allow the number of utilised resources to be dynamically adapted to the actual execution needs.

Calls to annotated functions are wrapped by a function of the Python binding [5], which forwards the function name and parameters to the Java runtime. With that information, the Java runtime creates a task and adds it to the data dependency graph, immediately returning the control to Python. At this point, the main program can continue executing right after the task invocation, possibly invoking more tasks. Therefore, the Java runtime executes concurrently with the main program of the application, and as the latter issues new task creation requests, the former dynamically builds a task dependency graph. Such graph represents the inherent concurrency of the program and determines what can be run in parallel. When a task is free of dependencies, it is scheduled by the run- time system on one of the available resources, specified in XML configuration files.

The default scheduling policy of the runtime is locality-aware. When scheduling a task, the runtime system computes a score for all the available resources and chooses the one with the highest score. This score is the number of task input parameters that are already present on that resource, and consequently they do not need to be transferred.

4.3 Hecuba and Qbeast

Hecuba [6] is a set of tools and interfaces developed at BSC, that aims to facilitate programmers an efficient and natural interaction with a non-relational database. Using Hecuba, the applications can access data like regular objects stored in memory and Hecuba translates the code at runtime into the proper code, according to the backing storage used in each scenario. Hecuba integrates with the COMPSs programming model [4]. This integration provides the user with automatic and mostly transparent parallelization of the applications. Using both tools together enables data-driven parallelization, as the COMPSs runtime can enhance data locality by scheduling each task on the node that holds the target data.

The current implementation of Hecuba supports Python applications that use data stored in memory, in Apache Cassandra database or in ScyllaDB.

Hecuba supports the interaction between Apache Cassandra and both the advanced machine learning module and the TDF.

Qbeast is a multidimensional indexing system integrated with Hecuba that is provided also as part of the software stack. Qbeast implements a novel indexing algorithm based on D8tree algorithm [18], which is designed to support both analytical and data-thinning queries on multidimensional data while aiming to lower latency and achieving linear scalability. The main idea of the indexing algorithm is to merge multidimensional indexing, data sampling and denormalization techniques to achieve high scalability, availability, and performance.

4.4 Key-value database

I-BiDaaS platform uses Apache Cassandra [7] as the common database to keep all the data. Apache Cassandra is a distributed and highly scalable key-value database. Cassandra implements a non-centralized architecture, based on peer-to-peer communication, in which all nodes of the cluster can receive and serve queries. Data in Cassandra is stored on tables by rows, which are identified by a key chosen by the user. This key can be composed of one or several attributes, and the user has to specify which of them compounds the partition key and which of them the clustering key. A partitioning function uses the partition key to decide how rows distribute among the nodes.

ScyllaDB [8] is a drop-in replacement for Apache Cassandra that targets applications with real-time requirements. The primary goal of ScyllaDB is to reduce the latency of the database operations by increasing the concurrency degree using a thread-per-core event-driven approach thus lowering locks contentions and better exploiting the modern multi-core architectures. The programmer interface and the data model are the same with Apache Cassandra. The peer-to-peer architecture that provides scalability and availability is also inherited from Cassandra.

In I-BiDaaS we plan to perform some experiments replacing Cassandra by ScyllaDB. This change will not affect the implementation of the algorithms nor the architecture of the platform (because Cassandra and ScyllaDB are fully compatible) and will allow us to evaluate which of the two data management solutions fit better with the requirements of the I-BiDaaS use cases.

5 Machine learning advances

This section describes current advances that are carried out with respect to the development of the pool of I-BiDaaS batch algorithms, i.e., the Advanced ML sub-module. The focus in the current work is given on two main largely independent and parallel directions. The first is on advancing the theory that will underlie a range of future COMPSs algorithmic implementations of I-BiDaaS. The second is on the actual COMPSs implementations of a number of standard machine learning algorithms, namely K-means, K-NN, decision trees, random forests (see, e.g., [12], and sparse regression via the alternating directions method of multipliers (ADMM), e.g., [14]. While the second direction does not provide algorithmic novelty, it serves several important purposes. First, it provides an actual set of standard algorithmic implementations that will be available in the I-BiDaaS pool. Second, it will be used as a set of templates from which one can build more complex implementations. Third, it allows for testing, through standard methods, several features of the COMPSs framework regarding the implementation of complex parallel and distributed ML tasks, and applying the conclusions and lessons to the implementation of novel methods in the next project stage. It is also worth noting that another algorithmic implementation, currently available in the I-BiDaaS pool and corresponding to the MVP of the project, is discussed in a separate section (see section 6).

5.1 Theoretical advances: Distributed convex optimization with increasingly sparse communications

From the theoretical perspective, the I-BiDaaS advanced ML module is based upon the framework that maps several optimization paradigms that enable fast Big Data optimization (e.g., asynchronous updates; block-coordinate updates; mini-batching/stochastic gradient; map-reduce-type parallelization) onto the task-based COMPSs programming. This approach will allow for fast and scalable execution over a broad class of the underlying physical infrastructures. The framework is mainly based on structured (non)-convex optimization, with the following underlying rationale. Often, a machine learning task corresponds to minimizing an empirical loss function over the data; the loss is then minimized via an optimization algorithm. This scenario subsumes a broad range of models and methods, including, e.g., training a deep neural network via back propagation (the backpropagation is an instance of a stochastic gradient descent method), classification via support vector machines/logistic regression, sparse regression, sparsity+low rank models, dictionary learning, etc. Our framework considers several computational models, including master-worker, peer-to-peer, and hybrid solutions.

We report here on the recent advances in the peer-to-peer models and novel distributed algorithms for solving generic problems that are represented as large but finite sums of convex losses [9][10][11][23][24][25]. Specifically, we develop and rigorously analyze distributed first and zeroth order stochastic optimization methods with the novel feature of increasingly sparse communications. With the proposed algorithms, each peer/node/agent communicates with its immediate neighbours in a randomized fashion, so that the probability of communicating decreases with time in a carefully crafted way. We demonstrate that the algorithms exhibit strictly faster rates of decay of the mean squared error with respect to the state of the art.

In more detail, with the proposed methods, each node, at each time epoch, decides whether to communicate or not based on a binary decision variable, where different nodes' decisions are independent across time epochs and are independent across nodes. Each node, at each time epoch, makes an update by weight-averaging its solution estimate with the solution estimates of its neighbours in the underlying graph, and by performing a step according to a stochastic gradient approximation of its local loss function. The probabilities of communication across time epochs are then carefully designed, in order to achieve the best possible mean square error decay rate, both with respect to the overall communication cost, and with respect to the number of time epochs. Reference [9] demonstrates and analyzes the proposed methodology on non-linear least squares problems, while references [10][25] and [11][24][25] are concerned with generic loss minimization problems and consider first and zeroth order stochastic methods, respectively.

The next stage of the described research corresponds to actually implementing the developed first and zeroth order optimization methods and their variants in COMPSs and consider adding the implementations to the I-BiDaaS pool.

5.2 COMPSs implementations of a pool of standard methods

5.2.1 Brief review of the PyCOMPSs programming model

We briefly review the PyCOMPSs programming model concepts relevant to implementation of machine learning methods, while a general introduction to COMPSs is given in section 4.2. The PyCOMPSs programming model is based on the principle wherein a user is mainly responsible for identifying the functions to be executed as asynchronous parallel tasks and annotating them with a standard Python decorator. A runtime system is in charge of exploiting the inherent concurrency of the script, automatically detecting and enforcing the data dependencies between tasks and spawning these tasks to the available resources, which can be nodes in a cluster, cloud or grid. Arguments for task decorators and their possible values are presented in Table 1.

Table 1: Arguments for task decorators in PyCOMPSs

Argument	Value
Formal Parameter Name	-INOUT: read- write object -OUT: write – only object -FILE: read – only file -FILE_INOUT: read – write file -FILE_OUT: write – only file
Returns	int, long, float, str, dict, list, tuple, user-defined classes
isModifier	True (default) of False
Priority	True or False (default)

After the tasks have been defined, the framework creates the dependency graph, and the application can be executed in parallel. In order for results to be synchronized, we use the PyCOMPSs' API function, compss_wait_on. This command steams the main control flow until all the values of the invoked tasks are obtained so that the main program can work with the correct version of the data.

With respect to the described model, we implemented an initial pool of machine learning algorithms in PyCOMPSs. In the following, we describe these implementations. First, a brief explanation of each algorithm is provided. Then, we present the methodology of parallelization of each algorithm in PyCOMPSs. Initial testing of the accuracy of the method is provided in section Appendix.

5.2.2 List of algorithms implemented

K-means [12] is a standard algorithm for data clustering. It works iteratively and demands only one parameter denoted here by *K*, which stands for a number of clusters to create. Firstly, K initial mean values are randomly generated, where the arithmetic values belong to the data domain. Then, K clusters are created by assigning every data point to the nearest mean. After cluster association process, the mean of each cluster becomes the new centroid. The previous steps are iteratively performed until convergence. The algorithm converges when the distances between two successive sets of centroids are zero or small enough. In order to reduce the number of distance calculations and speed up the algorithm, we use distance estimations based on triangular inequality [13].

ADMM-LASSO. Alternating direction method of multipliers (ADMM) [14] is a widely used general purpose optimization method that is amenable to distributed implementations. We implement here the variant that solves sparse regression LASSO (Least absolute shrinkage and selection operation) problems [14].

K-NN Classification and Regression. K-nearest neighbours (K-NN) [12] is a standard machine learning algorithm used both for regression and for classification. It takes one mandatory parameter, K, which stands for the number of nearest neighbours to consider when making the decision. In order to accelerate the prediction phase, we perform K-means clustering in the training and use triangle inequality to reduce the number of distance measurements [15].

Decision Tree Classification. This is a tree-like structured model for classification [13] based on the principles of entropy and information gain and which try to create simple sets of rules for classification that maximize information gained at each step.

Random Forest Classification. Random Forest [14] is an ensemble algorithm formed by multiple decision trees. It inherits all the advantages of decision trees, but due to ensembling, it is more robust in making predictions. One needs to set the number of trees to be used – n_estimators and the number of attributes for each tree – n_features.

5.2.3 PyCOMPSs Implementations

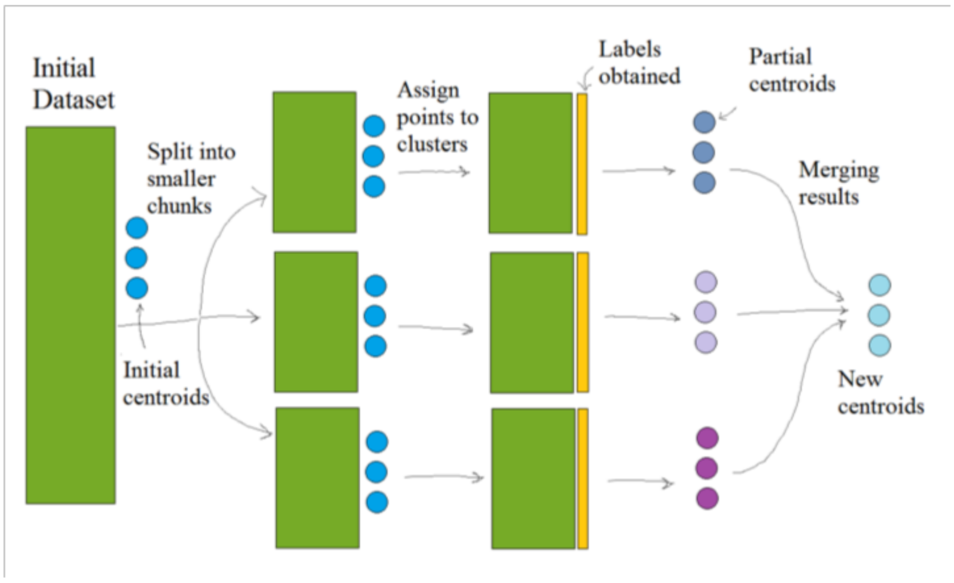

Figure 3: Scheme for the K-means implementation

K-means Clustering (see Figure 3). This algorithm requires parameter K (default=8) and additionally parameters *num* (the number of parallel processes, default=10), *max_iter* – the maximal allowed number of iterations (default=300) and *tol* – the tolerance allowed to terminate training (default=0.0001). The methods provided are .fit(), .predict() and .fit_predict(). Input to the fit method is a set of data points for training the algorithm and it calculates the centroids of the clusters. The input for the predict method is a set of points to predict and it predicts cluster labels for each point. Method .fit_predict() is equivalent as invoking fit and predict on the same dataset.

Parallelization is done in the fitting phase. In every iteration, the dataset is broken into smaller chunks and for each of these, we perform functions in parallel to find closest centroids for points, using triangular inequality to reduce the number of computations. Then for each cluster in each chunk, we note sum and count of the points contributing and then merge all the outputs into one, synchronizing calculations, to obtain new centroids. The process is repeated until convergence. The fast K-means implementation based on the triangular inequality can be developed straightforwardly in the parallel version. For distance estimation, one needs the specified point and all the centroids. As each core has this information, running in parallel can be carried out immediately.

ADMM LASSO (see Figure 4). The input parameters are matrix A and vector b, and the output is vector x. The matrix A is splitted into smaller submatrices $A_1,\ldots,A_n$, and vector b is splitted accordingly to $b_1,\ldots,b_n$. Then, on each (A_i, b_i) pair, we perform the following steps for computing x in parallel (here k corresponds to the current iteration and $S_{\frac{\lambda}{\rho}}$ denotes the element-wise soft thresholding operation with threshold equal to λ/ρ, see [14] for details):

$$x_i^{k+1} = (A_i^T A_i + \rho I)^{-1}\left(A_i^T b_i + \rho\left(z^k - u_i^k\right)\right)$$

$$z^{k+1} = S_{\frac{\lambda}{\rho}}(w^{k+1} + v^k)$$

$$u_i^{k+1} = u_i^k + x_i^{k+1} - z^{k+1}$$

Here, I is the identity matrix, λ, ρ are appropriately set positive parameters, u_i^k and z^k are dual and auxiliary variables, respectively, w^k is the entry-wise arithmetic mean of the x_i^k's, $i = 1,\ldots,n$, and v^k is the entry-wise arithmetic mean of the u_i^k's, $i = 1,\ldots,n$

After each iteration, the results are synchronized and the residuals are updated. The process repeats until convergence. In the first step of the ADMM algorithm, an inverse matrix operation is performed. In order to reduce the computational complexity of this operation, Cholesky factorization is used (Figure 4).

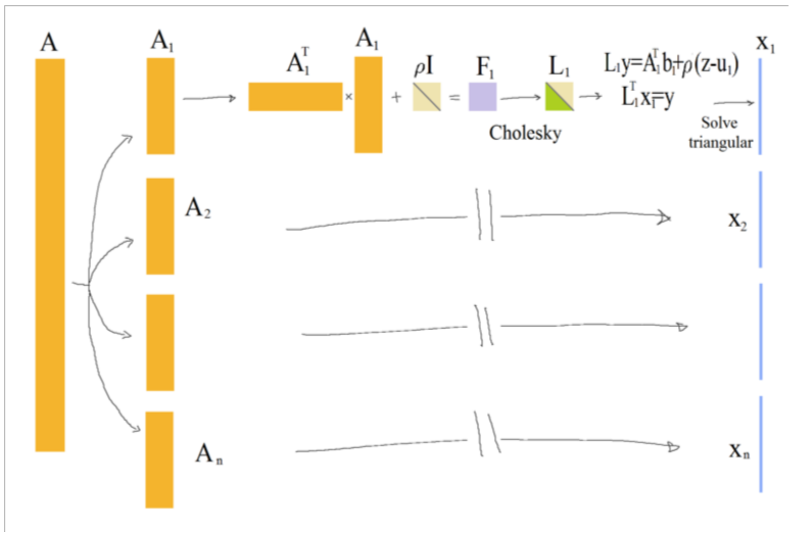

Figure 4: Scheme for ADMM implementation

K-NN Classification and Regression (see Figure 5). The algorithm requires parameter K (default=3) and additionally parameter *num* that stands for the number of parallel processes (default=10). The methods provided are .fit(), .predict() and .fit_predict(). The inputs to the fit method are two arrays, one containing the data points and other with labels for training an algorithm, and it calculates the centroids of the clusters. Input for the predict method is a set of points to predict and the output is an array of labels for each point (or in case of regression it is an array of estimated values). Method .fit_predict() is equivalent as invoking fit and predict on the same dataset.

When fitting this algorithm, we use the K-means algorithm introduced above; that makes the fitting phase slower, but speeds up predictions. In the predicting phase, training data is split into chunks and partial K-NN is performed, using K-means clusters obtained and triangle inequalities to reduce the number of distance computations, in parallel. For each of the chunks, we merge result, synchronizing calculations, and reduce them to the K closest neighbours. Then, in the case of classification, voting is performed to find the majority class, or, in the case of regression, we calculate the new value as a weighted average.

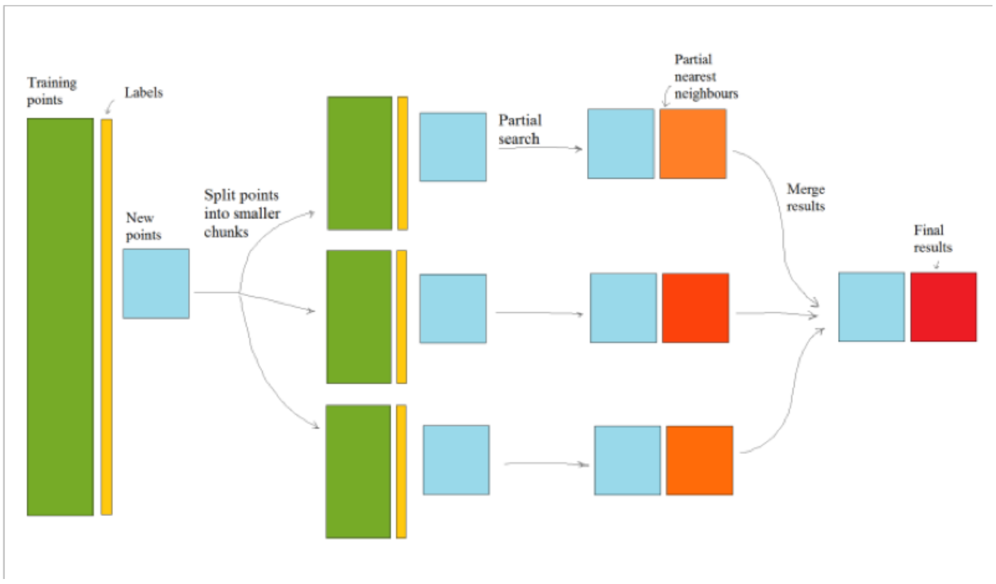

Figure 5: Scheme for K-NN implementation

Decision Tree Classification (see Figure 6). The arguments for this algorithm are *max_depth* as the maximal allowed depth of the trees (default=4), *min_sample* – minimal number of points in a leaf to split it (default = 5) and *impurity_tol*, which is the allowed impurity for terminating branching (default=0). This implementation consists of three methods. Method .fit() takes an array of data points for training the model and returns a tree. Method .predict_proba() takes a set of points to predict and returns the probability of belonging to each available class. Method .predict() takes a set of points to predict and returns the most probable class. Calculations are split in a manner that in one step each attribute is examined in parallel at the same time, and for each value information gain after the split is calculated, also in parallel. Then, all the calculations are synchronized and split is performed according to best value-attribute pair. We continue these steps until convergence.

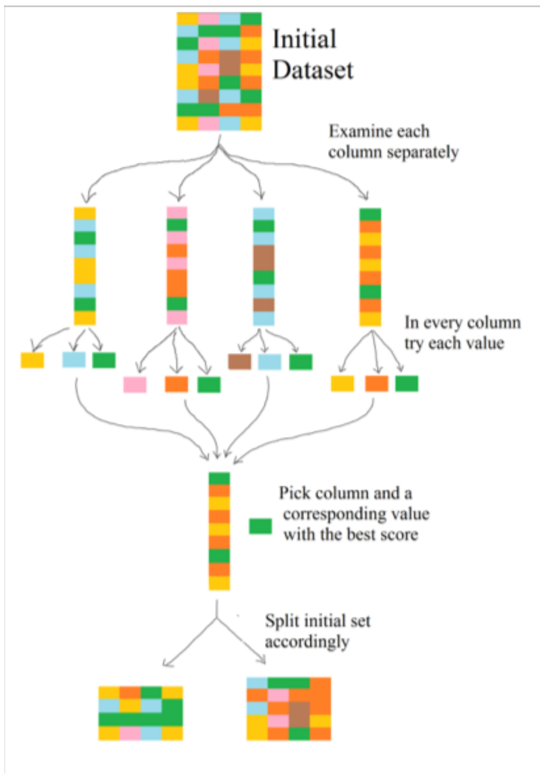
Initial Dataset
Examine each column separately
In every column try each value
Pick column and a corresponding value with the best score
Split initial set accordingly

Figure 6: Scheme for decision tree implementation

Random Forest Classification. The arguments for this algorithm are *n_estimators*, which tells how many trees will build the model (default = 10); *n_features* – the number of attributes for building each tree (default = 4), *max_depth* as the maximal allowed depth of the trees (default=4), *min_sample* – minimal number of points in a leaf to split it (default = 5) and *impurity_tol*, that is allowed impurity for terminating branching (default=0). The Implementation consists of three methods. Method .fit() takes an array of data points for training the model and returns a tree. Method .predict_proba() takes a set of points to predict and returns the probability of belonging to each available class. Method .predict() takes a set of points to predict and returns the most probable class. When fitting the model, each tree is created in parallel as a task, and considering that we use above the described implementation of the decision tree method, it is even further parallelized. When predicting, the results obtained by all the trees are merged together as the final answer.

This deliverable reports some preliminary tests for the developed methods for moderate-sized data sets, sequential implementation and only accuracy-type results (see the Appendix). Subsequent work will include testing of parallel execution on a computer cluster, scalability with respect to the number of machines, execution times, large data sets, etc.

6 The batch application for the MVP

The use case that we are using for the MVP comes from CAIXA requirements. CAIXA stores information about their customers and the operations they perform (bank transfer, check their accounts, etc.) using channels such as mobile apps or online banking. Related to this information, CAIXA has deployed different techniques to find new kind of relationships between customers using the IP address (to a more detailed description of our use cases you can refer to the chapter 3 of the deliverable D1.3 of I-BiDaaS [17]).

The main objective of this use case is to validate and test if synthetic data generated with the same characteristics, provides the same or additional insights, and in particular if the number of relationships detected by CAIXA could be detected using synthetic data.

6.1 Algorithm and main data structures

The goal of the application is to find relations between people, given a set of connections to IP addresses. We want to maximize the detections of close relations between users.

First, the algorithm detects IP addresses that cannot be used to represent individual users, such as IP addresses of public spaces (e.g. libraries, campus, etc.), or IP addresses used by electronic banking tools that operate on behalf of different users. The selection of these IP addresses is based on the number of different users that connect from them: the IP addresses discarded are those for which this number of users exceeds a configurable threshold.

There is a relation between two users when they have connected from two different IP addresses on a month, discarding the relations with public IP addresses. In the end, the application must take groups of a maximum of two users. The main data structures are the following:

- IPConnections: the goal of this data structure is to register which users have connected to a given IP address. It is implemented as a dictionary indexed by IP address and with a set of user identifiers as a value.
- DailyPairs: the goal of this data structure is to keep the information about all the connections of each user in a month. It can be implemented as a nested dictionary. The outer dictionary is indexed by month and the inner dictionary is indexed by the user identifier. The value of the inner dictionary is a set of the user connections, where a connection is a pair composed of the day and the IP address.

The algorithm implements the following steps:

- First, the algorithm iterates over all the data in order to create the data structures, only considering those IP addresses that are not from known aggregation bank tools.
- Secondly, using the IPConnections structure the public IP addresses can be detected. They are the IP addresses with a length of the set of users greater than the public space threshold. The public IP addresses are kept because in the next step it will be used to discard the relations with public IP addresses.
- Finally, in order to find the relations, the algorithm uses the DailyPairs structure. For each month, it intersects the set of connections of each user with that of all the others except the set of connections of the users already computed.

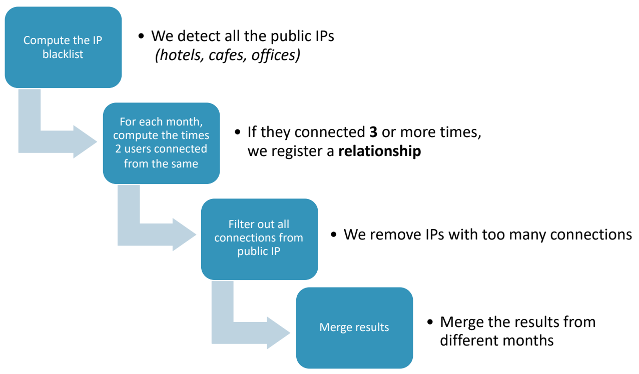
Compute the IP blacklist
• We detect all the public IPs (hotels, cafes, offices)
For each month, compute the times 2 users connected from the same
• If they connected 3 or more times, we register a relationship
Filter out all connections from public IP
• We remove IPs with too many connections
Merge results
• Merge the results from different months

Figure 7: Algorithm to detect relations between users

We have developed three different versions for this application: the sequential version, the parallel version using COMPSs and the parallel version with Cassandra, using COMPSs and Hecuba. All these codes are available in a GitHub repository [16] that will be made publically available. The sequential version is just an implementation of the algorithm described above and depicted in Figure 7. In the following sections, we describe the two parallel versions implemented. For all three versions, the input data, describing all the connections, is assumed to be stored in Apache Cassandra and is accessed through the interface implemented by Hecuba.

6.2 Parallel version using COMPSs

This section describes an initial version of the code using the PyCOMPSs programming model. In this initial version, we have defined several parallel tasks, not only to exploit parallelism but also to benefit from the automatic detection of the dependencies. The main tasks of our algorithm are the following:

- Divide in chunks the input data (randomly), and for each chunk initialize in parallel the data structures with the input data
- Merge the results of the initialization
- Compute the blacklist with the discardable public IP addresses
- Identify the months that appear in the input data
- For each month, compute in parallel the relations between users
- Merge the information about relations detected for each month

Figure 8 shows the code that implements the initialization of the IPConnections and the DailyPair structure together with the corresponding merge phase, and Figure 9 shows the portion of the dependency graph of the COMPSs version of the application for this code fragment.

```
@task(returns=(dict,dict))
def chunk_aggr(partition):
    # A map of maps with format
    #  { 'june': { 'userA': {('127.0.0.1','15th'}]}}
    result = defaultdict(lambda: defaultdict(set))
    # A map of users for ip to ban public ips
    ip_connections = defaultdict(set)
    for ((FK_NUMPERSO, PK_ANYOMESDIA), (IP_TERMINAL, _)) in partition.items():
        if IP_TERMINAL not in KNOWN_AGGR_BANK_TOOLS_IP:
            day_of_the_month = PK_ANYOMESDIA[-2:]
            connection = (day_of_the_month, IP_TERMINAL)
            month = PK_ANYOMESDIA[:-2]
            result[month][FK_NUMPERSO].add(connection)
            ip_connections[IP_TERMINAL].add(FK_NUMPERSO)
    return result, ip_connections

@task(returns=dict)
def get_reduced_IPConnections(*ip_connections):
    return reduce(compute_IPConnections, ip_connections)

def compute_IPConnections(a_ip_connections, b_ip_connections):
    ip_connections = defaultdict(set)
    for ip in set(a_ip_connections.keys() + b_ip_connections.keys()):
        ip_connections[ip] = a_ip_connections[ip].union(b_ip_connections[ip])
    return ip_connections


def main():
(...)
daily_pairs,  ip_connections  =  zip(*map(chunk_aggr),  records.split())  #compute  local
results

compss_barrier()
```

Figure 8: COMPSs version: Initialization of IPConnection and DailyPairs

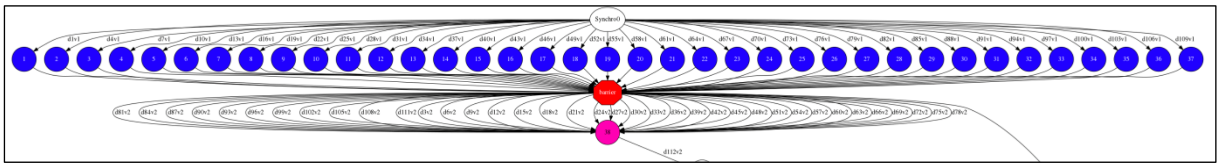

Figure 9: Dependency partial graph

Notice that, as the temporary data is stored in memory, it is necessary to add merge tasks to join the partial results. However, if we use a database to store this temporary data, we can omit the synchronisation and the merge task, maximising the parallelism degree. Figure 10 illustrates three different approaches to dealing with this splitting-parallel computing-merging scheme: the one that we have using with COMPSs programming model, the alternative using Spark/Hadoop engines and, finally, the one that can be obtained using Hecuba as the backend for the data.

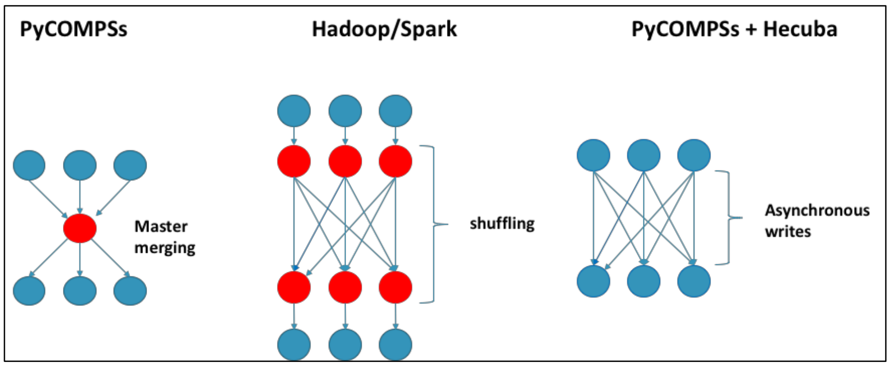

Figure 10: Split-compute-merge Solutions

6.3 Parallel version using COMPSs and Hecuba

The third version of the application is a COMPSs application implemented using Hecuba as the data interface and Cassandra as data backend.

As we have described in section 6.2, using Cassandra to store the data allows us to delegate on the database the management of the global view of the data. This approach frees programmers from implementing an explicit synchronization between those parallel tasks that modify the data structure. This way, removing the synchronization points, we are able to maximize the parallelism degree of the application and thus the utilization of the hardware resources. Notice that the interface for inserting data in Cassandra is asynchronous with the execution of the application, overlapping this way the data storing with the computation.

To implement this application version, we have extended Hecuba to support new data types: sets and dictionaries of sets. These extensions are described in section 7.2.

Hecuba allows users to access the data as regular Python objects stored in memory. The only requirement is to define previously the class that supports each data structure, specifying the data types.

Figure 11 shows the definition of the classes that support the two main data structures of our use case.

```
class IPConnections(StorageDict):
'''
@TypeSpec <<ip_terminal:str>, users:set<int>>
'''
class DailyPairs(StorageDict):
'''
@TypeSpec<<month:str,user:int>,connections:set<day:str,ip:str>>
'''
```

Figure 11: Classes definition for the data structures

Both classes are implemented as dictionaries, that Hecuba converts into the suitable tables on the database. To achieve this transparent translation, it is necessary to make them inherit from the Hecuba class StorageDict. With the decorator @TypeSpec, the programmer can specify the type for both the key and the value of each element of the dictionary. Optionally, it is also possible to specify a name for each of these fields.

The keys of dictionaries of class IPConnections are of type string while the values are sets of integers (i.e. for each IP address we store the identifier of all users that have connected to it).

The key of dictionaries of class DailyPairs is a tuple composed of a string and an integer, while the value is a set of tuples, each of them composed of two strings.

Once these classes are defined, the programmer can create an instance of some object and then use the regular Python operators over those instances. Hecuba offers two options for the object instantiation: volatile objects or persistent objects. All volatile object can become persistent at any point by using the make_persistent method. Instantiation of persistent objects requires as a parameter a string to identify the data in the database.

Figure 12 shows the code of the initialization of the IPConnections and the DailyPairs structures. Note that in this code fragment, the merge of partial results is performed within the initialization function without the requirement of a subsequent reduction function, as was the case of the code shown in Figure 8. This is possible because the object containing the IP connections is global and the synchronization is a responsibility of the database.

```
task()
def chunk_aggr(partition, ip_connections, daily_pairs):
   # A map of maps with format
   #  { 'june': { 'userA': {('127.0.0.1','15th'}]}}
   for ((FK_NUMPERSO, PK_ANYOMESDIA), (IP_TERMINAL, _)) in partition.items():
      if IP_TERMINAL not in KNOWN_AGGR_BANK_TOOLS_IP:
         day_of_the_month = PK_ANYOMESDIA[-2:]
         connection = (day_of_the_month, IP_TERMINAL)
         month = PK_ANYOMESDIA[:-2]
         try:
            daily_pairs[month, FK_NUMPERSO].add(connection)
         except KeyError:
            daily_pairs[month, FK_NUMPERSO] = {connection}
         try:
            ip_connections[IP_TERMINAL].add(FK_NUMPERSO)
         except KeyError:
            ip_connections[IP_TERMINAL] = {FK_NUMPERSO}

def main():
(...)
ip_connections = IPConnections( "test.IPConnections" )
daily_pairs = DailyPairs( "test.DailyPairs" )
map(lambda x: chunk_aggr(x, ip_connections, daily_pairs), records.split())
```

Figure 12: COMPSs+Hecuba version: Initialization code of IPConnections and DailyPairs

7 MVP-driven technological advances

The development of the MVP has driven the technological advances that we report in this deliverable. In addition to the implementation of the application described in Section 5, some extensions have been required both on the TDF platform and Hecuba. In the current stage of the MVP, we have decided that the fabricated data will be inserted on the Apache Cassandra database, using the interface provided by Hecuba. Until this moment, this insertion is done through a conversion tool that reads the data from the files generated by TDF and stores it into Apache Cassandra. Shortly, the TDF tool will be adapted to insert the data directly into the database. In addition, we plan to provide an alternative implementation for the connection between TDF and Apache Cassandra. This alternative is the same as the Streaming analytics module of the I-BiDaaS platform will use to access the data stored in Apache Cassandra, and consists of using the UM component to communicate both with TDF and Hecuba.

In the following subsections, we describe the advances made on both TDF and Hecuba modules.

7.1 TDF advances

The advances made on the TDF tool related to the batch module are mainly the study of the changes that should be done in order to be integrated with the database.

The TDF tool has two types of data fabrication projects, one for databases and another for files. In database projects, the tool imports the database structure from the database meta data automatically, whereas in file projects, the user must define the structure of the data manually. Once the structure is defined, the user is now required to model the data using various types of constraint rules. Both the data structure description and the model are translated into an abstract constraint satisfaction problem (CSP) and sent to the integrated CSP solver. The solver returns a solution, which contains a desired data point that satisfies all the constraints. This solution is translated and written into the desired output resource. An output resource in database project could be one of the various commonly used relational databases e.g. DB2, Oracle, PostgreSQL etc. An output resource in file project could be one of the various commonly used formats e.g. positional flat file, xls, csv, xml or json.

TDF has been extended to support streaming data output using the MQTT (Message Queuing Telemetry Transport) protocol. The solution that is generated by the solver is converted from the TDF internal representation into the json format. This new feature enables the interaction of TDF and the SAG universal messaging bus. The feature supports publishing json objects, one per generated table row, in database projects, and the entire file in file projects, to the specified broker. The new feature supports wide configurations options including user authentication, secure communications and desired quality of service.

TDF will be extended to support Cassandra DB similarly to the way relational databases are supported. TDF will be able to automatically import the structure and to write the solution back to the database.

In order to automate the modelling process which is known to be complex and time consuming, the platform must support generation of new data from rules which were automatically learned and extracted from real/production data. The fabricated data must be similar, to the training data, thus realistic. Data similarity in this context is characterized as new data with similar statistical and distributional properties, similar relationships etc. This is for both column wise and cross-column wise. Such properties may involve numeric type, dates, word sequences, distributions, means, variances, ranges, quantiles, confidence intervals, patterns, distinctness, uniqueness, cross-column correlations and more.

The feature to automatize the data modelling will know how to utilize basic information like the schema-table-column names, data types and existing database internal constraint rules applied on it (e.g. primary keys, check constraints) to syntactically find best match for a known representative term for which a set of categorized, pre-defined rules already exists.

Provided with real-data analysis results, TDF will automatically construct the data modelling rules that will enforce the creation of similar data. A trivial example could be when the data analysis suggests that a column called 'age' has a normal distribution with mean equals to 39 and standard deviation equals to 5, and the modelling logic will create a constraint requiring that exact same distribution. TDF has a generic API for importing data analysis results.

7.2 Hecuba advances

We have extended Hecuba to support the MVP use case. In particular, we have added support to sets and dictionaries of sets. In the following subsections, we describe the changes involved in this extension.

7.2.1 Hecuba support to sets

The StorageSet is a class that contains all the functionalities for sets in Hecuba. It is useful for cases when a StorageObject only contains a set, as the code will get simpler. In the same way as the StorageDict, to give Python sets the possibility to be stored persistently, the Python class has to inherit from the StorageSet class and they have to contain the annotation describing the type of each key and value. The rest of the class can be empty or can contain any definition decided by the programmer. The format of the annotation is a Python comment. The line must start with the keyword @TypeSpec and continue with the types of the set content (see Figure 13).

```
class ClassName(StorageSet):
'''
@TypeSpec <set_type_specification>
'''
```

Figure 13: Specification of a StorageSet definition

For example, Figure 14 shows the code of a definition of a set with a tuple of a string and an integer.

```
class MyClass(StorageSet):
'''
@TypeSpec <str,int>
'''
```

Figure 14: Example of class definition of a StorageSet

In the current implementation, StorageSets can contain Strings, Integers, Floats or Tuples. StorageSet offers programmers the same interface as Python defines for its regular sets and thus, the programmer can use, for example, a method union, intersection, add or remove on StorageSets with the same external behaviour. If the result of a StorageSet method is a new StorageSet, then the new StorageSet will be persistent if the target StorageSet for the method is persistent. If the resulting StorageSet is persistent, Hecuba assigns an automatic identifier in a transparent way. If the StorageSet is defined to be persistent, Hecuba translates the operations to keep the data and to read the data from the database.

In Figure 15, we show a simple example of performing a union between two different StorageSets (A and B). This operation creates a new StorageSet (C) with the contents from both A and B. In this example, the resulting set C will be persistent because A is persistent.

```
A = MyClass("test.setA")
B = MyClass("test.setB")
C = A.union(B)
```

Figure 15: Union between two StorageSets

Additionally, Hecuba implements the split method on the StorageSet, equivalent to the split method of the StorageDict class, that allows to partition the set and then iterate over each partition. This partitioning method allows dividing the work between parallel PyCOMPSs tasks. The split method, creates a pre-defined number of partitions (sets) and returns an iterator on that partitions. At this moment, the number of partitions is a configurable parameter of Hecuba and should be related with both the task granularity and the desired parallelism degree. The criteria to define the content of each partitions is to favour the data locality (all the elements in a partition are stored in the same Cassandra node). Figure 16 shows an example of how to use the split method.

```
for ss in myStorageSet.split():
      # Here we will have access to each partition of the StorageSet
      for value in ss:
             # Here we will have access to each element of the partition
```

Figure 16: Example of partitioning and iteration on sets

Internal implementation

The internal implementation of the StorageSet management distinguishes between two different cases: volatile StorageSet and persistent StorageSet. The volatile StorageSets implements in memory all the operations. The implementation of persistent StorageSets uses the Cassandra interface to read and write the persistent data.

Each persistent StorageSets is kept in a Cassandra table, with just one row per each element of the set. In the case of simple elements (in the current implementation, integers, strings or floats) there is only one column that also acts as primary key. In the case of tuples, there is one column per each element of the tuple and all of them will be part of the primary key: the first element of the tuple is the partition key and the rest are clustering keys. Each access to a persistent StorageSet is converted into an access to the corresponding table in Cassandra.

When a method that returns a new set, such as union, is performed, it returns a new StorageSet, a new Cassandra table is created with the suitable scheme. The name of the new table is decided transparently to the programmer. It consists of the name of the base StorageSet (the one that is executing the creation method) followed by suffix "_method_i", where "method" is the creation method (for example, union) and "i "is an incremental integer (to avoid collisions between table names).

7.2.2 Hecuba support to dictionaries of sets

We have extended Hecuba to support dictionaries of sets, that is, dictionaries which values are of type Hecuba sets. From the point of view of a user, the usage of a dictionary of sets is mostly like the usage of any other dictionary: the same methods apply. We have defined the syntaxes to specify that the values can be of type set and we have extended our classes parser to detect it. Figure 17 shows an example of how to specify a class describing a dictionary of sets. In this example, the key is of type string and the value is a set of integers.

```
class MyClass(StorageDict):
'''
@TypeSpec <<str>,set<int>>
'''
```

Figure 17: Definition of a dictionary of sets

Furthermore, Figure 18 shows some examples of how to instantiate and use an object of this class. As you can see, once the object is instantiated, the utilization is like a regular Python dictionary. Hecuba is responsible for translating these operations on the suitable database operations.

```
d = MyClass("test.dict")
d["key0"] = {}
d["key0"].add(1)
d["key0"].add(2)
d["key1"] = {1,2,3}
d["key1"].remove(1)
```

Figure 18: Examples of usage of a dictionary of sets in Hecuba

One of the goals of Hecuba is to keep the same behavior between Hecuba objects and the equivalent Python objects. In the case of the current implementation of dictionaries of sets there exists a slightly different behavior between Hecuba and Python, but this is something that we will change in the near future. This difference is related to the methods that in Python create new sets like, for example, union or intersection. In the case of Hecuba, when the target set is a value of a dictionary, we modify the target set instead of creating a new set. Notice that this is something irrelevant for the implementation of the use case of the MVP and simplifies the implementation. For this reason, in order to avoid a delay in the testing of the use case we decided to assume this simplification and complete the implementation in the next release of Hecuba.

Internal implementation

We have considered two different options for the internal implementation of the dictionary of sets in Hecuba. The first one was to follow an approach equivalent to the implementation of Storage Objects containing Storage Objects: in the table of the outer Storage Object we store the identifier of the internal Storage Object which is stored on a different Cassandra table. However, as we are dealing with dictionaries that may have many different entries it would be very inefficient to create a different table for each entry. For this reason, we have decided to discard this option and to embed the content of the set in the table of the dictionary.

With this embedding option, we have only one table: the one created with the dictionary. The primary key of the table is the key of the dictionary together with the values of the set associated to that key: the key of the dictionary acts as partition key while the values of the set act as clustering keys.

7.3 Integration with the Streaming Analytics module

In the context of MVP, we also consider the integration of the batch analytics module with the streaming analytics module. In particular, as detailed in D5.2, the batch analytics output therein is passed to the SAG's Apama complex event processing engine (CEP). Besides that, we will consider the integration of the batch analytics module with the FORTH's GPU-accelerated pattern matching, in order to deliver filtering rules that may be applied to the incoming streaming data. By doing that, the historical data and the produced analytics may be used by FORTH's sub-module to filter incoming streaming data. We will research various options for the data format and messaging protocols through which this integration can be implemented in the best possible way; e.g., direct communication, communication through Apama, etc.

8 Conclusions and future steps

This report describes the state of the work done in work package 3 until month 12th. We have two different groups of advances.

The first one is related to the advance ML submodule of the Batch module of the I-BiDaaS platform: we report advances on the theory that will underlie the machine learning applications that the platform will provide and a number of actual COMPSs implementations of some machine learning algorithms, namely K-means, K-NN, decision trees, random forests, and sparse regression via ADMM. These applications will be part of the batch analytics module of the I-BiDaaS platform and will provide users with the necessary tools to perform the target data analytics.

The second one is advances on the technological components of the Batch module that will support the execution of the applications. This group of advances has been mainly driven by the implementation of the MVP of I-BiDaaS that is based on a real use case provided by one of our partners (CAIXA). The advances in the technological components are related both to the TDF tool and the data management tools. First, we have defined the type of communication to be implemented between the TDF tool and the data management tools in the Batch layer of the platform. We plan to implement and to evaluate two different approaches: one will be based on the utilization of the Universal Messaging (UM) bus and the other one will be TDF to use Hecuba to write directly on the database. Related to this, we have extended TDF to implement the utilization of the UM bus to communicate both with the streaming analytics module of the platform and with the data management tools of the batch processing module. Second, we have extended Hecuba to support the data structures required by the application of the use case. The implementation of the use case using PyCOMPSs and Hecuba is also part of the contributions of this work package.

As part of the future work of this work package we include the following tasks:

- Continue working on the theoretical studies of the machine learning algorithms and provide more algorithms to be part of the applications pool in the batch module.
- Adapt the developed machine learning algorithms to use Hecuba to access the database.
- Continue working on the extensions to Hecuba to support the new algorithms. Evaluate and refine the implementations of the modifications already done to Hecuba.
- Adapt Hecuba to use the UM component to provide access to the database to both TDF and the streaming module of the platform.
- Adapt TDF to use Hecuba to access directly the database.
- Design the communication between the database and the visualization component of the I-BiDaaS platform.
- Develop further communication between the batch processing module and the streaming analytics module of the I-BiDaaS Platform.

Appendix: Initial accuracy testing

For each of the algorithms in section 5.2.2 a corresponding benchmark implementation is considered. For ADMM LASSO we use as a benchmark the MPI implementation of the same algorithm [14] For the rest of the algorithms, we used their implementations from the Python's widely adopted library Scikit-Learn [21]. We use the following metrics for accuracy testing. For classification algorithms, the standard accuracy score metric is used. More precisely, in multiclass classification tasks, we compute the accuracy for each class, i.e., the ratio of correctly estimated samples over the total number of samples. The mean value of all class accuracies is finally computed. In binary classification, we calculate the ratio of correctly estimated samples over the total number of samples. For the regression algorithms, the metric used is r^2 score. The silhouette score is used for clustering algorithms. For the LASSO algorithm, we track the behaviour of the primal and dual residuals and objective functions across the iterations. We carried out only accuracy-type testing over a sequential, single-node running environment, and moderate-sized data sets. Future tests include large scale data sets and actual run-times over a computer cluster. The data sets used are commonly used real open data sets. The iris dataset consists of three species of Iris flowers (multiclass classification); each species is represented by 50 samples and 4 attributes, describing the sepal and petal lengths and widths. The data set is available at UCI repository [22] as well as in Scikit-Learn's datasets. The wine dataset consists of three different red wine types represented by 59, 71 and 48 samples, respectively (multiclass classification). Each sample has 13 attributes obtained by chemical analysis, and the data set is available at the UCI repository as well as within the Scikit-Learn's datasets. The wine quality dataset is similar to the wine quality data set, but now it concerns white wine types. The set consists of 4898 instances and 12 attributes and is available at the UCI repository [22] (multiclass classification). The breast cancer dataset is a binary classification dataset with 569 (212 and 357 respectively) samples and 30 attributes, and it is available within the Scikit-Learn's data sets. The student performance data set represents student achievement in secondary education in two Portuguese schools. The set consists of 649 samples and 30 attributes and it is available at the UCI repository. In addition, we use two synthetic data sets, dubbed Dataset 1 and Dataset 2, which are artificial sets created via Scikit-Learn. The first data set is suitable for clustering created by the method *make_blobs*, and the second data set is intended for regression and is created by the method *make_regression*. Dataset 3 is the artificial data set used for testing and is generated as described in Chapter 11 of [14]. The results for different algorithms, metrics, and different datasets are presented in the tables below.

Table 2: K-means performance on Iris Dataset

Iris Dataset	Mean score	Min score	Max score	25 percentile Score	75 percentile score
PyCOMPSs	0.5789	0.5789	0.5789	0.5789	0.5789
Scikit-Learn	0.5816	0.5816	0.5816	0.5816	0.5816

Table 3: K-means performances of Dataset 2

Dataset 2	Mean score	Min score	Max score	25 percentile Score	75 percentile score
PyCOMPSs	0.2669	0.2224	0.3433	0.2633	0.2676
Scikit-Learn	0.3433	0.3433	0.3433	0.3433	0.3433

Table 4: LASSO performance on Dataset 3

Dataset 3	Primal residual	Dual residual	Primal epsilon	Dual epsilon	Objective function
PyCOMPSs	0.029	0.0477	0.0973	0.0785	18.8159
MPI	0.0355	0.0667	0.0985	0.0696	17.1890

Table 5: K-NN classification performance on Iris Dataset

Iris Dataset	Mean acc.	Min acc.	Max acc.	25 percentile acc.	75 percentile acc.
PyCOMPSs	0.961	0.8666	1	0.9333	0.9666
Scikit-Learn	0.961	0.8666	1	0.9333	0.9666

Table 6: K-NN classification performance on Wine Dataset

Wine Dataset	Mean acc.	Min acc.	Max acc.	25 percentile acc.	75 percentile acc.
PyCOMPSs	0.7066	0.5555	0.8888	0.6666	0.75
Scikit-Learn	0.7066	0.5555	0.8888	0.6666	0.75

Table 7: K-NN classification performance on Breast Cancer Dataset

Breast Cancer Dataset	Mean acc.	Min acc.	Max acc.	25 percentile acc.	75 percentile acc.
PyCOMPSs	0.9254	0.8684	0.9649	0.9122	0.9473
Scikit-Learn	0.9254	0.8684	0.9649	0.9122	0.9473

Table 8: K-NN regression performance on Student Performance Dataset

Student Performance Dataset	Mean score	Min score	Max score	25 percentile score	75 percentile score
PyCOMPSs	0.7704	0.312	0.952	0.7163	0.8541
Scikit-Learn	0.7663	0.3649	0.9418	0.7071	0.8501

Table 9: K-NN regression performance on Carbon Nanotubes Dataset

Carbon Nanotubes Dataset	Mean score	Min score	Max score	25 percentile score	75 percentile score
PyCOMPSs	0.9636	0.9497	0.9732	0.9605	0.967
Scikit-Learn	0.9708	0.9597	0.9786	0.968	0.974

Table 10: K-NN regression performances of Dataset 1

Dataset 1	Mean score	Min score	Max score	25 percentile score	75 percentile score
PyCOMPSs	0.7958	0.0911	0.9505	0.7235	0.8847
Scikit-Learn	0.8071	- 0.3589	0.9389	0.7754	0.8829

Table 11: Decision Tree performance on Wine Dataset

Wine Dataset	Mean acc.	Min acc.	Max acc.	25 percentile acc.	75 percentile acc.
PyCOMPSs	0.9102	0.6944	1	0.8888	0.9444
Scikit-Learn	0.918	0. 6944	1	0.8888	0. 9722

Table 12: Decision Tree performance on Iris Dataset

Iris Dataset	Mean acc.	Min acc.	Max acc.	25 percentile acc.	75 percentile acc.
PyCOMPSs	0.9396	0.7666	1	0.9	0.9666
Scikit-Learn	0.945	0.7666	1	0.9333	0.9666

Table 13: Random Forest performance on Iris Dataset

Iris Dataset	Mean acc.	Min acc.	Max acc.	25 percentile acc.	75 percentile acc.
PyCOMPSs	0.9473	0.8333	1	0.9333	0.9666
Scikit-Learn	0.9556	0.8666	1	0.9333	0.9666

Table 14: Random Forest performances of Student Performance Dataset

Student Performance Dataset	Mean acc.	Min acc.	Max acc.	25 percentile acc.	75 percentile acc.
PyCOMPSs	0.2397	0.0886	0.3924	0.1898	0.2911
Scikit-Learn	0.3098	0.1772	0.3924	0.2784	0.3544

Table 15: Random Forest performances of Wine Quality Dataset

Wine Quality Dataset	Mean acc.	Min acc.	Max acc.	25 percentile acc.	75 percentile acc.
PyCOMPSs	0.5205	0.4744	0.5591	0.5109	0.5336
Scikit-Learn	0.5376	0.4918	0.5806	0.5275	0.5471

References

[1] Apache mahout. https://mahout.apache.org/

[2] Tensorflow. https://www.tensorflow.org/

[3] ServiceSs: an interoperable programming framework for the Cloud, Lordan, F., E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, Journal of Grid Computing, March 2014, Volume 12, Issue 1, pp 67–91, DOI: 10.1007/s10723-013-9272-5

[4] COMP Superscalar, an interoperable programming framework, Badia, R. M., J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent, SoftwareX, Volumes 3–4, December 2015, Pages 32–36, DOI: 10.1016/j.softx.2015.10.004

[5] PyCOMPSs: Parallel computational workflows in Python, Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, Jesús Labarta, IJHPCA 31(1): 66-82 (2017), DOI: 10.1177/1094342015594678

[6] Hecuba, https://github.com/bsc-dd/hecuba

[7] Apache Cassandra, http://cassandra.apache.org/

[8] Scylla DB, https://www.scylladb.com/

[9] Communication Efficient Distributed Weighted Non-Linear Least Squares Estimation, A. Kumar Sahu, D. Jakovetic, D. Bajovic, S. Kar, EURASIP Journal on Advances in Signal Processing, 2018, invited paper, Special issue on optimization, learning, and adaptation over networks, https://doi.org/10.1186/s13634-018-0586-0

[10] Convergence rates for distributed stochastic optimization over random networks, A. Kumar Sahu, D. Jakovetic, D. Bajovic, S. Kar, to appear in proc. IEEE International Conference on Decision and Control, CDC 2018, invited paper, Dec 17-19, Miami, FL, USA

[11] Non-asymptotic rates for communication efficient distributed zeroth order strongly convex optimization, A. Kumar Sahu, D. Jakovetic, D. Bajovic, S. Kar, to appear in proc. GlobalSIP 2018, IEEE Global conference on signal and information processing, Nov 26-28, Anaheim, CA, USA

[12] Pattern Recognition and Machine Learning, Springer Science+Business Media, C. M. Bishop, New York, 2006

[13] Using the Triangle Inequality to Accelerate k-Means, Charles Elkan, Department of Computer Science and Engineering, University of California, San Diego, California 2003

[14] Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers, Boyd S., Parikh N., Chu E., Peleato B., Eckstein J., 2011, Foundations and Trends in Machine Learning Vol. 3, No. 1, pp. 1–122

[15] A Fast Exact k-Nearest Neighbours Algorithm for High Dimensional Search Using k-Means Clustering and Triangle Inequality, Xueyi Wang, in Proc Int Conf Neural Netw. 2012

[16] I-BiDaaS MVP repository. https://github.com/cugni/I-BiDaaS-MVP/tree/master/batch

[17] Positioning of I-BiDaaS. Deliverable D1.3. https://www.ibidaas.eu/deliverables

[18] C. Cugnasco, Y. Becerra, J. Torres, E. Ayguade, D8-tree: a de-normalized approach for multidimensional data analysis on key-value databases, in ICDCN '16: Proceedings of the 17th International Conference on Distributed Computing and Networking, 2016

[19] Create high-quality test data while minimizing the risks of using sensitive production data. IBM InfoSphere Optim Test Data Fabrication, IBM, 2017, https://www.ibm.com/il-en/marketplace/infosphere-optim-test-data-fabrication.

[20] Test Data Fabrication. Security and Data Fabrication, IBM Research, 2011, https://www.research.ibm.com/haifa/dept/vst/eqt_tdf.shtml.

[21] Scikit-learn: Machine Learning in Python, Pedregosa et al., Journal of Machine Learning Research 12, pp. 2825-2830, 2011.
[22] UCI repository, https://archive.ics.uci.edu/ml/index.php
[23] Distributed Zeroth Order Optimization Over Random Networks: A Kiefer-Wolfowitz Stochastic Approximation Approach, A. Kumar Sahu, D. Jakovetic, D. Bajovic, S. Kar, to appear in proc. IEEE International Conference on Decision and Control, CDC 2018, invited paper, Dec 17-19, Miami, FL, USA
[24] Communication-Efficient Distributed Strongly Convex Stochastic Optimization: Non-Asymptotic Rates, A. Kumar Sahu, D. Jakovetic, D. Bajovic, S. Kar, submitted to IEEE Transactions on Automatic Control, 2018, https://arxiv.org/abs/1809.02920
[25] Exact spectral-like gradient methods for distributed optimization, Dusan Jakovetic, Natasa Krejic, Natasa Krklec Jerinkic, submitted to Computational Optimization and Applications, 2018, http://people.dmi.uns.ac.rs/~natasa/ExactSpectral.pdf