UNIVERSITÀ DEGLI STUDI DI PALERMO

SCUOLA POLITECNICA

Corso di Laurea in Ingegneria Informatica
Dipartimento di Ingegneria Chimica, Gestionale, Informatica, Meccanica

INTELLIGENT SYSTEM FOR AUTO-TUNING OF BIG DATA ANALYTICS DEPLOYMENT PROPERTIES

TESI DI LAUREA DI
DAVIDE PROFETA

RELATORE E TUTOR UNIVERSITARIO
PROF. SALVATORE GAGLIO
TUTOR AZIENDALE
ROSOLINO FINAZZO

ANNO ACCADEMICO 2017 - 2018

MAGISTRALE

Summary

Chapter 1
Introduction

In the field of machine learning applied to big data, in this thesis work has been implemented an intelligent system for auto-tuning of big data analytics deployment properties, and which will be integrated on a cloud platform with a model-based BDA-as-a-service (MBDAaaS) approach. A readapted solution proposed by Guolu Wang et al. was first implemented [14]. The original method consists of optimizing a single metric on the Spark platform (an engine for big data processing), readapting it to the use case and aimed to the optimization of multiple metrics. Once the solution proposed for the optimization of a single metric was implemented and tested, the method was extended and tested for the optimization of multiple metrics. A general deployment optimizer method has therefore been proposed, applicable not only to the Spark framework, but also to other frameworks such as Cassandra and Kafka. These frameworks share the same problem: the tuning of deployment properties to optimize application performance. The proposed deployment optimizer then aims to solve these problems automatically, transparently, and with faster times than the manual tuning acted by domain experts and data engineers.
In the context of the TOREADOR project, it can be used for the automatic optimization of various metrics, such as the CPU and RAM utilization, for the execution of big data analytics services that are registered and executed on the platform. Furthermore, the proposed component is designed to optimize different categories of applications and to be easily extensible thanks to the precautions taken during development. The proposed component does not simply provide the already trained models that return a sub-optimal configuration, but defines in a dynamic way, based on the framework used (for example Spark), the conceptual area of application (including ingestion, preparation, analytics…), and the size of the processed input, a series of stages that start from the generation of the labeled dataset up to the search for a sub-optimal configuration, that will be injected at deployment time on the launch configuration of the Spring Cloud Data Flow service orchestrator (SCDF). Finally, the operations that are carried out within the TOREADOR platform will be illustrated on a sequence diagram.

The following chapter deals with the problems that led to the need to introduce an automatic component like the one proposed in the work of this thesis. The main

frameworks related to performance tuning problem are introduced, as well as the architecture oriented to micro-services, today a bridge for the development of modern and large cloud applications that need to scale and evolve quickly.

Chapter 3 provides an overview of the state of the art in the auto-tuning of configuration parameters aimed to performance optimization, focusing in particular to the article and the reference method proposed by Wang et al.

Chapter 4 describes the general method proposed, re-imagining the reference method and extending it to allow the optimization of multiple metrics, and on different target frameworks.

The two following chapters describe the development processes that led, first, to the implementation of a deployment optimizer for the optimization of a single metric, and subsequently to the optimization of multiple metrics. The technologies and models of machines used for the implementation are also described.

Chapter 7 introduces TOREADOR, a project co-founded by EU, including Engineering - Ingegneria Informatica Spa as development partner, and how the deployment optimizer can be integrated into the TOREADOR platform.

In the final chapter personal remarks are made on the basis of the thesis work, and considerations on possible future works.

Chapter 2
Problem Statement

Today Big Data is used by companies to support decision-making processes. The boom in these applications is mainly due to the spread of cloud computing, which has allowed companies to have available unlimited computing power and storage in a scalable way at low cost with an on demand and pay-per-use model. One of the most important aspects of the use of Big Data Analytics that generally deal with processing, storing, and transferring large volumes of data is linked to performance problem. In fact, frameworks such as Spark, Cassandra, and Kafka, have a considerable number of configuration parameters that affect the performance of the application. In a cloud-based infrastructure oriented to micro-services, the client may also want to optimize the services accessed through some service level agreements (SLA), through which service metrics are defined. These can specify, for example, a minimization of the execution time, trying to exploit fully the resources of the infrastructure, or the minimization of costs, and therefore of the resources used in the cloud, or a trade-off between these two. The high number of configuration parameters and difficulties for humans to understand the interactions between several parameters, leads to having a combinatorial problem: the goal, that is to find the optimal configuration, is NP-hard. Furthermore, these settings can have continuous values and have an irregular effect on performance. The human expert, as a data engineer, even if he could find, through many trials and errors, the optimal configuration for a given application, does not make it easier to search for a subsequent application. The best configuration for one application may not be the best for another. This thesis work, starting from a work by Guolu Wang et al. [14], proposes and shows an intelligent system for auto-tuning of Big Data analytics deployment properties based on machine learning, and using a micro services-oriented approach.

In the following paragraphs is provided a brief description of the micro-services oriented architecture, a more in-depth introduction on the framework Spark on which we focused for the test-bed implementation, and on those other frameworks on which it can be applied the deployment optimizer proposed.

2.1 Micro-services architecture

In the beginning the applications were developed and distributed as a single entity: monolithic architecture. This type of architecture lends itself well to small applications or in any case not subject to change. It changes when there is developing complex and rapidly evolving applications. In these situations, monolithic applications can easily become very complex, which makes it difficult to move quickly during development, testing and implementation. Furthermore, the only way to scale a monolithic application is to replicate the entire application with the resulting increase in costs and resources.

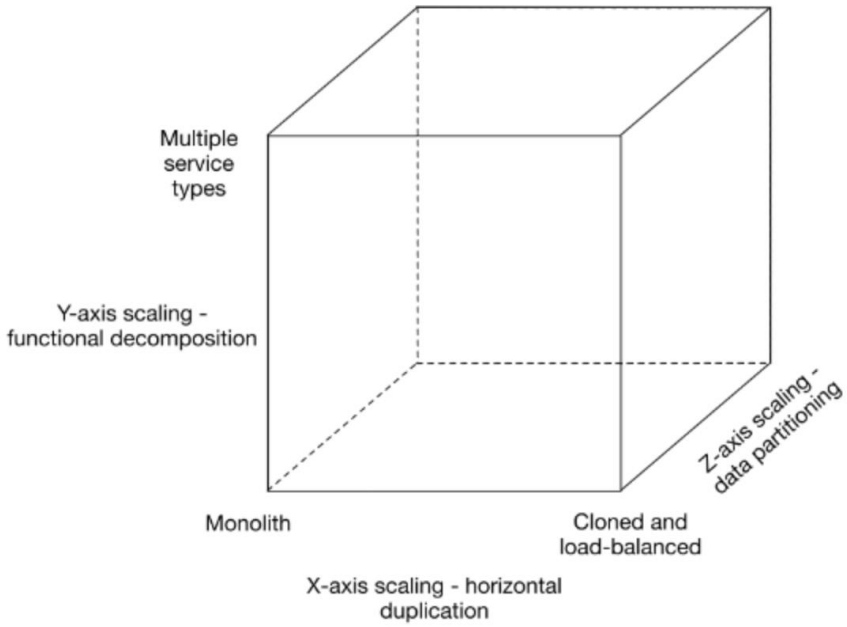

Figure 1. Scalability cube. Source [1]

The most common approach to scaling an application is to replicate by executing multiple identical copies of the application and is known as scaling on the X axis. This is a way to improve the capacity and availability of an application. Similarly, along the Z axis each server makes an identical copy of the code with the difference that each server is responsible for only a subset of the data. These approaches do not solve problems related to the growth of the size and complexity of the application. To solve these problems, it's needed to scale along the Y axis, or axis of functional decomposition, which is the approach used in micro-services. If scaling along the Z axis means dividing things that are similar, scaling along the Y axis means instead dividing things that are different. At the application level, it means dividing a monolithic application into a set of services. Each service implements a series of related features.

The separation of components certainly produces a more effective environment for the build and maintaining highly scalable applications. Services are developed and distributed

independently and are easier to maintain, correct and update, leading to more agile capabilities to respond to today's environmental changes. This architecture, however, leads to disadvantages and problems: you have more distribution flows that must be kept correct and consistent throughout the application life cycle; services need an effective way to communicate without slowing down the whole application, and guarantee high availability using for each service a distributed system. [1]

2.2 Apache Spark

Apache Spark [2] is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Python to R, and supports a rich set of high-level tools that include Spark SQL to process SQL and structured data, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. For computations of huge amount of data, Spark can distribute the computation between computer cluster, and make a data analysis problem that is too big to run on a single machine, with *divide and impera* method, split it between multiple machines. Spark Streaming [5] is an extension of Spark that allows you to process data flows in real time in a flexible, high speed and tolerance way. Batch execution has long been applied to unbounded datasets, despite problems with status management and out-of-order data. Today, with the increasing spread of the *Internet of Things* and the inclusion of social networks in marketing strategies there is having to process many unbounded datasets: from physical sensors that continuously provide measurements, users who interact with web applications, and data from markets financial. The alignment between the type of dataset (bounded or unbounded) and the type of execution model (batch or streaming) offers many advantages in terms of performance.

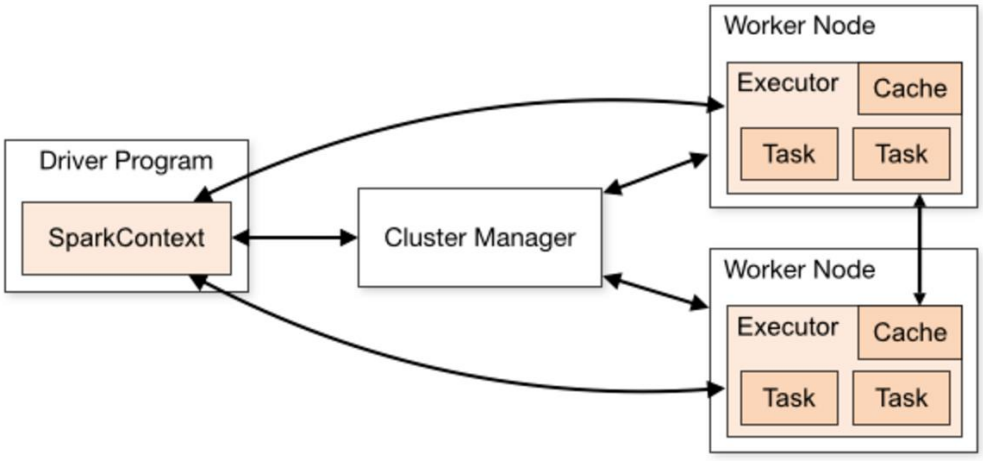

Figure 2. Spark architecture. Source [3]

Spark applications run as independent sets of processes on a cluster, coordinated by the *SparkContext* object in our main program, called *driver program*. To run Spark on a

cluster, *SparkContext* can connect to different types of cluster managers, which allocate resources through applications. Once connected, Spark acquires executors on the nodes in the cluster, which are processes that perform computations and store data for our application. Next, it's sent the code of the application (defined by the JAR or Python file passed on *SparkContext*) to the executors. Finally, *SparkContext* sends the tasks (a unit of work that will be sent to an executor) to the executors to run. This distribution of the load on different nodes allows us to have both a horizontal scalability and to be fault-tolerant, i.e. if one of the executor stops working, it can be recovered without stopping the entire work. [3]

2.2.1 The resilient distributed dataset (RDD)

The RDD is the main object of Spark, which represents an immutable, partitioned collection of elements that can be operated in parallel. Developers have to set RDD objects, and load them with large datasets, then invoke various methods on the RDD object to distribute the processing of that data. Although RDDs are distributed and resilient, they can be spread across an entire cluster that may or may not be launched locally. They can also handle the failure of specific execution nodes in our cluster automatically, redistributing the work as needed when this happens. RDDs are used to transform a dataset, from one set of data, to another, or to perform actions on the dataset itself. [4]

2.2.2 Configuring Apache Spark for performance and resource management

To improve the performance of our distributed Spark applications you could consider to: reduce the processing time of each batch of data using cluster resources; set the right batch size so that batches of data can be processed quickly as they are received. If the data you receive becomes a bottleneck in your system you can parallel it, creating more discretized stream, or DStream, inputs and configuring them to receive different partitions of the source data stream, i.e. adding new receivers, increasing throughput. Internally, a DStream is represented as a sequence of RDDs. Any operation applied on a DStream results in operations on the underlying RDDs. Later these DStreams can be merged together to create a single DStream, in order to obtain a kind of parallelism in the received data.

Cluster resources may be insufficient if the number of parallel activities used is not high enough. Data serialization overheads can be reduced by adjusting serialization formats.

These are some of the criteria with which you can improve the performance of a Spark application. In fact, Spark contains more than 180 configuration parameters that can be manipulated accurately based on application and optimization criteria, discouraging at the

same time the use of manual tuning of parameters through trials and errors, and focusing the research towards automatic solutions. Nevertheless, not all these properties can be considered as tuning properties. Properties related, for example, to Spark UI or security we can consider them and call them as static properties.
Other static properties can be properties, which affect the performance of the application, but which are fixed to static values, taking account of design/architectural choices.

It is therefore essential, as well as necessary, to find first those properties that actually affect the resources used by the application, i.e. to find tuning properties. [6] [7]

2.3 Apache Cassandra

Apache Cassandra is a distributed and open source DBMS, developed to manage large amounts of data located in different servers, also providing a service oriented to availability, without any single break point, or rather, without the loss of a node in the cluster makes the system unusable as it can be in some master/slave architectures. This is due to the fact that each of the nodes belonging to a cluster has the same role as the others: peer-to-peer distributed system.

The nodes in the cluster are arranged in a ring. To understand on which node is a data or where to write it, a partitioner is used, which will map the data, using a hash function on the primary key, for example in a string of 128 bits. Each node will manage a portion of these ranges, and for each scheme, which in Cassandra is called keyspace, you can decide how many times the data must be replicated. [8]

2.3.1 Configuring Apache Cassandra for performance and resource management

The main components in Cassandra to be configured are the partitioner, which determines which node will receive the first replica of a data and how to distribute the replicas on the other nodes in the cluster. The replication factor, that is, the total number of replicas between the cluster. The replication placement strategy, which determines which nodes to place replicas on. But beyond these, there are properties for the cluster, including caching parameters for tables, timeout settings, number of mem-tables that can be flushed concurrently, and many others. [9] [10]

2.4 Apache Kafka

Apache Kafka [12] is an open source instant messaging system, which allows the management of a large number of operations in real time from thousands of clients, both

reading and writing. It arises from the need to have a technology able to process a large amount of data in real time with high throughput and reliability.

Using a publish-subscribe messaging system, the platform:

1. Allows us to publish and subscribe to record streams
2. It allows us to memorize record streams in a fault-tolerant manner
3. It allows us to process record streams as they arise

A messaging system is responsible for transferring data from one application to another, so that applications can focus on data, but do not worry about how to share it.

Kafka can be used in many cases of use: for data monitoring operations; to collect logs from multiple services and make them available in a standard format to more consumers; for stream processing, used by popular framework like Storm and Spark Streaming.

Kafka, unlike most systems, is balanced for both latency and throughput. The performance of Kafka can be optimized improving both producers, brokers and consumers.

2.4.1 Configuring Apache Cassandra for performance and resource management

Two parameters are particularly important for latency and throughput: batch size and lead time. Batch size controls how many bytes of data to collect before sending messages to the Kafka broker. Lead time sets the maximum time to buffer data in a synchronous mode; this parameter improves throughput, but at the same time increase latency.

For Kafka brokers one important tuning is about the correct balance between leaders. In fact, each topic, a category on which the records are published, is divided into several partitions, and for each partition there is a leader. Another important aspect is the choice of the number of leader replicas in order to preserve data.

Consumers side, you need enough partitions to handle all the consumers needed to keep up with the producers. A property that affect throughput is related to checkpoint. You should set checkpoints having a margin of safety with much less impact on throughput. [13]

Chapter 3

State of the Art for Configuration Tuning Problems

In the literature various solutions have been found for the configuration tuning problem. By discarding the idea of having to deal with the problem manually through trials and errors as was mentioned about *Chapter 2*, there are two automatic approaches. The first of these is a so-called "white box" approach, where a thorough knowledge of the internal aspects of the system is required. H. Herodotou et al. [15], presented a self-tuning system for Big Data Analytics, where they proposed a cost-based (analytical) performance modelling method to tuning the parameters of Hadoop platform. The optimization in their "Starfish" system is achieved by a tuning on several levels: workloads, workflows, and jobs (procedural and declarative), as well as across various decision points provisioning, optimization, scheduling, and layout, handling the significant interactions.

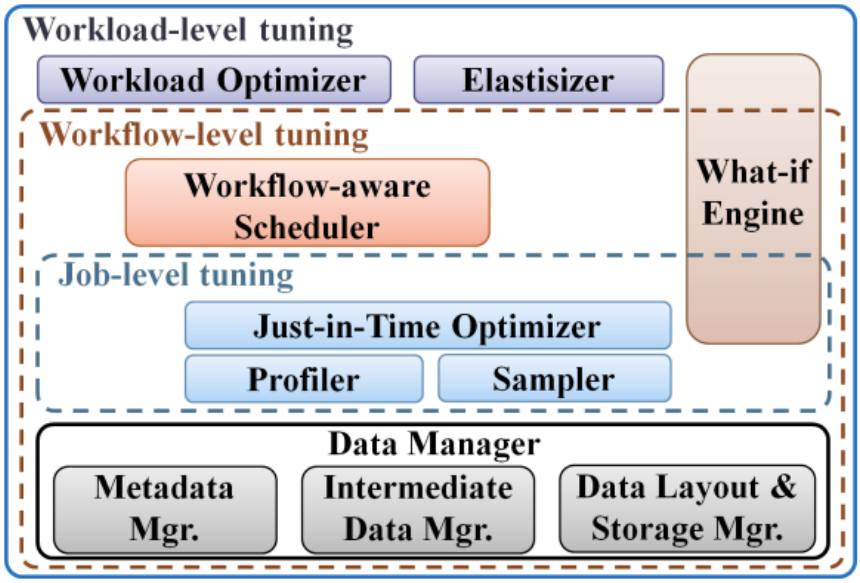

Figure 3. Components in the Starfish architecture. Source [15]

This method cannot be applied to other framework like Spark. Firstly, because of the internal differences between Hadoop and other frameworks. Secondly, being a cost-based model, each framework should be modelled differently taking into account all of its software and hardware stack.

The other approach is based on machine learning to generate "black box" models. These models are easier to build because they do not need detailed information on the internal

aspects of the system, but they are based on direct observations. About auto-tuning of deployment properties it's found a work of G. Wang et al. for the Spark framework, a unified analytics engine for big data processing. In their work "*A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning*" [14], motivate the use of a supervised trained model, with data harvested on a particular platform cluster and workload. Furthermore, these models should be flexible and robust enough to be reusable for multiple types of services and different clusters. Generally, it is expected that the most frequent application changes are input data. For this reason, input data are also part of their model proposed.

This thesis work starts from the implementation and integration of the method proposed by G. Wang et al., and then is extended with new features, in order to propose a general approach for the auto-tuning problem of Big Data Analytics deployment properties.

The following paragraphs briefly illustrate some works found in the literature based on machine learning for the problem addressed, with particular attention to the work taken as a reference.

3.1 Reference work

The method proposed by G. Wang et al. [14] aims to find an optimal configuration in order to minimize the execution time of an application running on the Spark platform, with a performance model represented by the following equation.

$$Perf = F(p, d, r, c)$$

Where *Perf* denotes the performance of the Spark platform, *p* denotes the user's application, *d* denotes the input data, *r* denotes the platform's resources, and *c* denotes the configuration parameters of Spark platform, *F* is a function of *Perf* about *p*, *d*, *r*, and *c*.

Since accurately predicting the execution time is not necessary and difficult, they are concerned with improving the performance under a given set of parameter values compared to the default values. That is, they fixed *p*, *d* and *r* by tuning on the parameter *c*, i.e. on the configurations.

3.1.1 Optimization method for Spark

The optimization method is composed of binary classification and multiple classification. The binary classifier is first generated. It takes as input the values of a subset of Spark's properties, and the size of the input, and as output it returns 1 if the execution time with those parameters is less than to the default configuration, 0 otherwise.

The Spark tuning properties they chosen is shown in the following figure.

Parameter	Description	Default	Rules of Thumb
spark.driver.cores	Number of cores to use for the driver process	1	1--8
spark.driver.memory	Amount of memory to use for the driver process	1g	1g—4g
spark.executor.cores	Number of cores to use for the executor process	1	10-40
spark.executor.memory	Amount of memory to use per executor process	1g	2g--8g
spark.reducer.maxSizeInFlight	Maximum size of map outputs to fetch simultaneously from each reduce task	48m	24m--96m
spark.shuffle.manager	Implementation to use for shuffling data	sort	sort/hash
spark.shuffle.compress	Whether to compress map output files	true	true/false
spark.shuffle.spill.compress	Whether to compress data spilled during shuffles	true	true/false
spark.broadcast.compress	Whether to compress broadcast variables before sending them	true	true/false
spark.io.compression.codec	The codec used to compress internal data	Snappy	lz4/lzf/snappy
spark.broadcast.blockSize	Size of each piece of a block for TorrentBroadcastFactory	0.2	0.1-0.3
spark.default.parallelism	Default number of partitions in RDDs returned by transformations like join, reduceByKey and parallelize when not set by user.	0.6	0.4-0.8
spark.speculation	Whether to perform speculative execution of tasks.	4m	2-8m
InputDataSize	The size of the input dataset	NULL	NULL

Figure 4. Parameters used. Source [14]

These parameters are chosen by domain experts. For each one a brief description of the parameter, the default value, and the range of values that is explored to make tuning has been reported. From this scheme you can consider two things: the metric, in this case the execution time, depends not on all the properties of Spark (which are more than 180), but on a significant subset; secondly, that the ranges of some properties also depend on the type of cluster used. In fact, Spark works in a distributed environment, that is the load on a specific number of nodes that have a specific number of cores and available RAM.

The inputs that exceed the binary classifier, then with output 1, are passed as input to a multiple classifier, generated on the basis of the reduction of the execution time in percentage terms. To give an example, if on the basis of dataset collected, there is a decrease of the execution time of 5, 10, 15, 20 and 25% compared to the default parameters, then multiple classifier will have 5 labels.

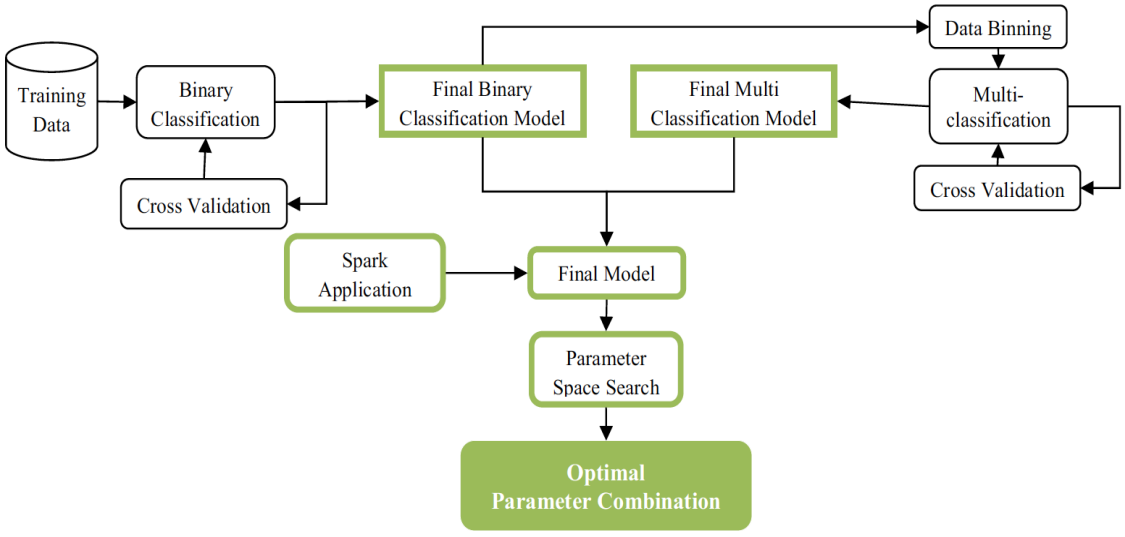

Figure 5. Optimization method for Spark. Source [14]

Once these models have been obtained, you can use them to predict if a configuration is better than the default one, i.e. if it leads to a reduction in execution time; furthermore, if that configuration improves compared to the default one, moving to the multiple classifier, you will be able to predict how much it improves. However, it's not possible explore the entire parameter space, since this is generally very huge. An efficient search algorithm is then used: the recursive random search algorithm (RRS).

3.1.2 Recursive random search

The basic idea of RRS is to maintain the initial efficiency of random sampling by "restarting" it before its efficiency becomes low. However, unlike the other methods, such as hill-climbing, random sampling cannot be restarted by simply selecting a new starting point. Instead you accomplish the "restart" of random sampling by changing its sample space. Basically, the algorithm consists of two phases: the exploration phase where, among *n* samples, the best one is taken; in the second phase, the exploitation phase has run: a random search is performed on a window centred on the point found; if it is found a better point within *p* times that point become the new centre point of the window, otherwise after *p* times the exploitation window is shrunk. [16]

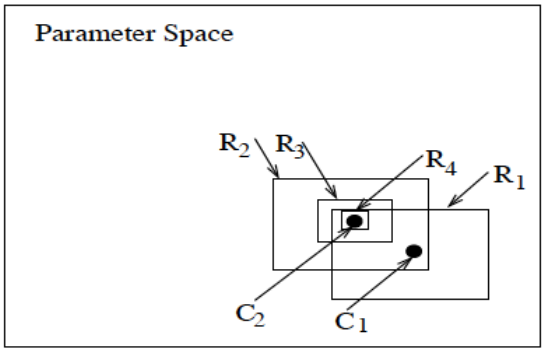

Figure 6. Shrink and re-align process. Source [16]

This phase is repeated until it is satisfied a stop criterion decided by us, while *n* and *p* are parameters of the algorithm. In this case the search should stop when the optimal configuration is found, i.e. the one with the highest reduction predicted by multiple classifier.

3.1.3 Data collection

To build the required models is necessary a training dataset. To generate this dataset, they selected four types of workloads such as *Sort*, *Wordcount*, *Grep* and *NavieBayes*. These types of workload are simple and easy to understand, but most of all they cover different application types including I/O intensive, CPU intensive, memory and iterative intensive.

So, after generating a list of random configurations on the specified ranges, data from real executions on the Spark cluster for each workload is collected.

3.1.4 Training and testing

They explored four common machine learning models applied into parameter tuning: decision tree, logistic regression, support vector machine and artificial neural network. With experimental results, they showed decision tree model (C5.0) is the best model considering the accuracy and computational efficiency. Instead, to improve the ability of model generalization fitting, the method of cross validation is used. They randomly split dataset into a 70% training set to train the models, and a 30% test set to evaluate their accuracies, lastly taking the average performance metrics obtained performing five times training testing with different random split.

The method of G. Wang et al. firstly is implemented and integrated for our deployment optimizer. Subsequently, based on initial considerations and the choice of a "black box" approach, the method is extended by considering several metrics at the same time. Furthermore, based on the same considerations, it would be possible to extend the method also considering other frameworks than Spark, such as Kafka and Cassandra, according to other types of workloads and input parameters.

3.2 Other works

Another optimization problem is related to the DBMS configuration tuning. As with Big Data Analytics, the tuning complexity due to the large number of DBMS knobs, and the non-reusable configurations for more workloads, has led to the exploration of automated solutions. D. Van Aken, in their work "*Automatic Database Management System Tuning Through Large-scale Machine Learning*" [17], presented an automatic tool, OtterTune, that leverages the knowledge gained from one application to assist in the tuning of others. They utilize a combination of supervised and unsupervised machine learning methods to (1) select the most impactful configuration "knobs", (2) map unseen database workloads to previous workloads, and (3) recommend knob settings. OtterTune also works with any DBMS, and maintains a repository of data collected from previous tuning sessions. This data is used to build the models to recommend optimal settings.

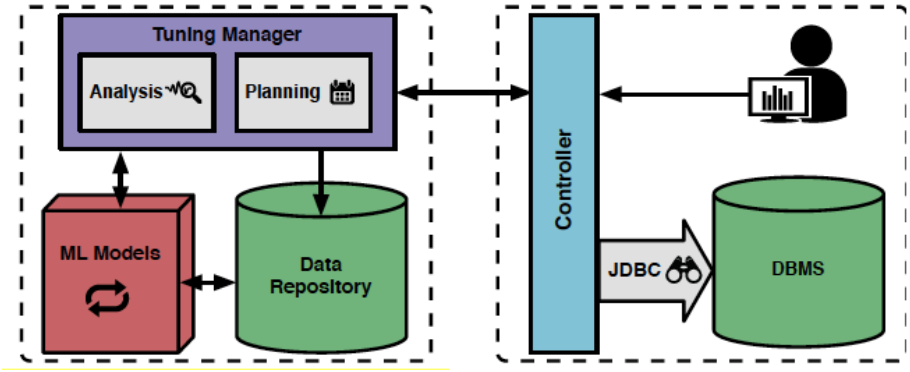

Figure 7. OtterTune architecture. Source [17]

OptCon, a machine learning-based predictive framework by S. Sidhanta et al. [18], is used for automatic consistency setting for quorum-based distributed data stores. OptCon it's implemented and tested on Cassandra. It tunes the datastore according to any dynamic variations in the workload and any given SLA using decision tree model. The configuration predicted is weak enough to satisfy the latency threshold, while being strong enough to satisfy the staleness threshold, expressed by SLAs.

Chapter 4
Deployment Optimizer process

The deployment optimizer proposed is based on machine learning and takes inspiration from the work of G. Wang et al. [14] for the tuning configuration parameters of Spark, and thought to be used it within a platform with a model-based BDA-as-a-service (MBDAaaS) approach.

The platform, when fully operational, should allow the client to access a catalogue of available services. From this catalogue the client, through REST interface, selects one or more services defining a workflow, where for each service they are associated a set of tuning properties to be optimized on the basis of the metrics performances. Since each service of the workflow, even if launched on the same framework (Spark), could have a different set of tuning properties and metrics, properties must be merged workflow-wise and run the optimizer on this aggregation. Examples of metrics are execution time, CPU utilization, RAM utilization, etc. The deployment optimizer proposed uses black-box models and a fast search algorithm to find a sub-optimal configuration. Model's inputs are the tuning property, while the outputs express if the configuration of input parameters leads to better performances on the chosen metrics, and how much it improves. Unlike the method we were inspired by, the size of the input that the user wants to process is not used as an input to the model, but is a parameter that is however passed to the optimizer for the *workload executions* stage. The model training, being supervised, requires a labelled dataset. The idea is to have a few models for many different services. That is, each service is associated with a workload on which the dataset must be built to training the models, and then used to search a sub-optimal configuration. This avoids having a workload for each service, and therefore a specialized model for a single service, reusing the same model for more similar services. In the test-bed we focused on working on the Spark framework, but the same method can be extended to other frameworks such as Kafka and Cassandra. Instead, as services to be optimized, we focused on the big data analytics services of which the Toreador platform is supplied, developing an ad hoc workload for this category of services and model training. Going deep into the process, the optimizer consists of four stages schematized in the following figure.

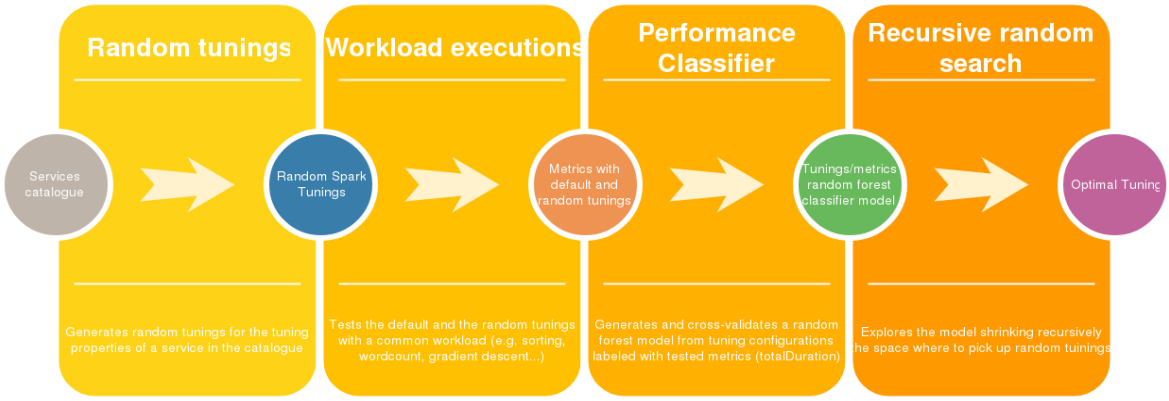

Figure 8. Deployment optimizer process

4.1 Random tunings stage

The first stage, *random tunings*, takes as input a set of tuning properties resulting from the choice of the metrics that the user wants to optimize, and as output returns a set *N* of random configurations, where *N* will be the record number of the dataset.

The deployment properties to use as tuning properties can be chosen by domain experts and/or relying on official framework documentation. Each metric could have a different, higher or lower number of properties than another. For the simultaneous optimization of multiple metrics, the union of all properties should be taken. Instead, the range of values to be randomly sampled for each property, and then tuned, is also chosen based on platform resources, for example based on the number of nodes, available ram, number of cores. This can be considered as a middle way, between using only human knowledge and manually tuning properties through trials and errors, and using machine learning models that make prediction considering all deployment properties for the framework in question. Remember that Spark owns more than 180 properties, most of which do not affect system resources. Being able to identify a significant set of properties on which tuning allows not only to have more compact models, but also a smaller parameter space where to search a sub-optimal configuration on the last stage.

All information on tuning properties to be used by metric, and the associated workload by service category are taken from the service catalogue. In order to generate random configurations on this stage, for each tuning property information on the ranges of the permissible values is also taken. If it admits continuous or discrete values, or categorical, the unit of measure, and all the information to generate a random number and be able to lead back to a value admitted by the property.

4.2 Workload executions stage

The randomly generated configurations are passed as input for workload executions stage. They are executed *N* runs of the assigned workload with the configurations generated in the previous stage, plus a run with the default configuration. This allows us to obtain, at the end of the workload processing, a dataset for the training of a supervised model. The dataset inputs are the values of randomly generated tuning properties. The workload is processed with a data workload chosen based on the size of the input that the user wants to process, allowing us to remove the parameter regarding the size of the user input from the model. This is because the data, and subsequently the models, are generated and trained dynamically, and based on the client's interaction and requirements.

In the work of G. Wang et al. [14] the workloads are set to four: *WordCount*, *Grep*, *Sort* and *NavieBayes*, taken from the BigDataBench benchmark suite [19]. BigDataBench is an open-source big data and AI benchmark suite developed by Institute of Computing Technology, Chinese Academy of Sciences. The current version BigDataBench 4.0 provides 13 representative real-world data sets and 47 benchmarks. These workloads represent the most of applications, because at low level, especially in the first three workloads, operations executed are sums, searches and comparisons. In our use case examined, we want to auto-tune the deployment of big data analytics services. Instead of using *NaiveBayes* as a workload target, i.e. a Bayesian classifier, it is implemented and used a workload that perform low-level operations that can generalize most of the machine learning algorithms used. In general, a workload should contain elementary operations and should be simple to understand, but at the same time be representative for a category of applications. To generate data workloads, it’s possible to use the string and numeric dataset generators provided by the same BigDataBench, allowing you to specify the size of the dataset required.

Harvesting a dataset labelled by launching workload services takes a lot of time compared to the other stages of the deployment optimizer process. So that makes it to not choose a number of launches and dimension of the workload dataset too high. At the same time, it motivates us to choose not just real applications but simple applications as workloads, in order to execute as many runs of workloads as possible, and then get a good number with records to train the model that sufficiently covers the entire parameter space.

4.3 Performance classifier stage

Once the workload executions are completed and the labelled dataset is generated, it is passed and processed by another sub-component to train and save the binary and multiple classification models. The performance classifier for the optimization of a single metric follows the work taken as a reference. The binary classifier is trained with the entire dataset collected in the previous stage, labelling with 1 the configurations that lead to a better metric performance than the default configuration, labelling with 0 otherwise. This classifier filters out the best configurations from the worst ones compared to not doing any tuning of the configuration parameters, i.e. compared to the default configuration. Configurations labelled with 1 are subsequently used to train a multiple classification model. This time the configurations are labelled based on the increase in performance in terms of percentages compared to the metric with the default configuration. The following figure shows an example of the training process with only one metric to optimize.

DATASET

TP1	**TP2**	**…**	**METRIC**	**METRIC WITH DEFAULT CONF.**
X11	X12	…	12	10
X21	X22	…	7	10
X31	X32	…	8	10
…	…	…	…	

TRAINING DATASET BINARY CLASSIFIER

TP1	**TP2**	**…**	**Y**
X11	X12	…	0
X21	X22	…	1 →
X31	X32	…	1 →
…	…	…	…

TRAINING DATASET MULTIPLE CLASSIFIER

TP1	**TP2**	**…**	**Y**
X21	X22	…	30
X31	X32	…	20
…	…	…	…

Figure 9. Example training process with single metric

An extension to the original method was to add the possibility of optimizing multiple metrics at the same time. In the case of multiple metrics, instead of training a single binary classifier, *N* binary classifiers are trained, one per metric, or training a single multi-task model, obtaining a binary filter for each metric. To filter the configurations that will be used for the training of the multiple classifier, it's used an aggregation function to bring us back to having a single binary filter. The selected aggregation of binary classifiers outputs, produces the maximum number of occurrences between 0 and 1. In case of a tie, 0 is returned. Configurations labelled with 1, as previously, are passed to the training of *N* multiple classifiers, or a single multiple classifier multi-task. But additionally, if in the configurations that pass to the multiple classifier there are metrics that have a worse

performance than the default configuration they are labeled with 0. The following figure shows an example of the training process for multi-metric optimization.

DATASET

TP1	TP2	…	METRIC 1	METRIC 1 WITH DEFAULT CONF.	METRIC 2	METRIC 2 WITH DEFAULT CONF.
X11	X12	…	12	10	10	20
X21	X22	…	7	10	5	20
X31	X32	…	8	10	30	20
…	…	…	…	…	…	…

TRAINING DATASET BINARY CLASSIFIERS

TP1	TP2	…	Y1
X11	X12	…	0
X21	X22	…	1 →
X31	X32	…	1
TP1	**TP2**	**…**	**Y2**
X11	X12	…	1
X21	X22	…	1 →
X31	X32	…	0

TRAINING DATASET MULTIPLE CLASSIFIERS

TP1	TP2	…	Y1
X21	X22	…	30
…	…	…	…
TP1	**TP2**	**…**	**Y2**
X21	X22	…	75
…	…	…	…

Figure 10. Example training process with multiple metric

4.4 Recursive random search stage

The output of the deployment optimizer is a sub-optimal configuration for the real application. This configuration is found by performing a search on the parameter space, using the RRS algorithm introduced in the previous chapter, based on the outputs predicted by the models trained with a specific assigned workload.

In this stage, it used the random tuning component to generate new random configurations. For a single metric optimization, the random configurations generated are given as input to the binary classifier. If the output produced is 1, it means that with the chosen configuration there is a better metric performance than the default configuration, and it can be passed as input to the multiple classifier, which instead predicts how much it improves in percentage terms. The RRS operates by carrying out a first phase of exploration, where *N* random configurations take the best one, that is, with the output of the highest multiple classifier. Subsequently, the exploitation phase occurs: in the neighbourhood centred at this point on the parameter space, it searches for new configurations. If it's found a better configuration, it's reapplied the exploitation centring on the new point, otherwise, after some steps, it's restricted the search window to the current point. The RRS should end

when a configuration with the highest possible classifier output possible is found. But to avoid never ending the search, due to the high number of parameters, a maximum number of calls to the classifier was entered. When this limit is reached, the best configuration found up to that moment is returned.

The search for a sub-optimal configuration for several metrics operates in exactly the same way, but at the prediction stages a process of aggregation of the outputs has been added, with the same criteria used for the model training phase. Configurations that are ingested by multiple classifiers or multi-task multi-classifier are those configurations that improve at least half plus one of the metrics compared to the default configuration. While the outputs of the multiple classifier are aggregated, so as to always return a scalar while keeping the RRS algorithm unchanged.

Chapter 5
Single Metric Optimization

This chapter describes the implementation and design choices to integrate the G. Wang et al. [14] method for the specific use case. At the beginning, working on the Spark framework, we focused on the optimization of the execution time as the only metric.
The deployment optimizer has been implemented as Maven project, and conceived to be a micro-service within of a platform with a model-based BDA-as-a-service (MBDAaaS) approach.
The chapter starts from the test-bed used for the workload deployment and a description of the data model through which each sub-component realized communicates. Follow the steps and the choices made for the selection of tuning properties. Then the process of dataset harvesting from the execution of a workload for model training. The technologies used for training and saving the machine learning models. How the algorithm search on parameter space is implemented, how the tests are performed and the results obtained.

5.1 Test-bed for the workload deployment

To run workloads, and then evaluate the strength of the method, a Spark cluster with four nodes is deployed. Each node, including three slave nodes and one master, has the same hardware and software stack as described in the following table.

Node	Role	CPU Type	Cores	Memory	Disk
1	NameNode, DataNode, Master	Intel Xeon E312xx@2.3GHZ	4	16 GB	250 GB
2	DataNode, Worker	Intel Xeon E312xx@2.3GHZ	4	16 GB	250 GB
3	DataNode, Worker	Intel Xeon E312xx@2.3GHZ	4	16 GB	250 GB
4	DataNode, Worker	Intel Xeon E312xx@2.3GHZ	4	16 GB	250 GB

OS	Spark	Scala	Hadoop
CentOS 7	2.1.1	2.11	2.7.2

Figure 11. Test-bed configurations

As already described we want to optimize a model of the performances that we assume depends on the following factors: the workload associated with the user application, the size of the input dataset, the resources of the platform, and the configuration parameters of

Spark platform. Fixed the first three factor, tuning on the configuration parameters of Spark is performed. The performances could therefore vary considerably depending on the resources available. This translates into having models, trained by a dataset obtained from the execution of a workload on a Spark cluster, that are specific not only for the type of workload but also for the platform used.

5.2 Data model

For the whole process pipe, illustrated in the previous chapter, the same data model is used, in order to contain both the metrics to be optimized and the tuning properties. The data model is described by the following class diagram.

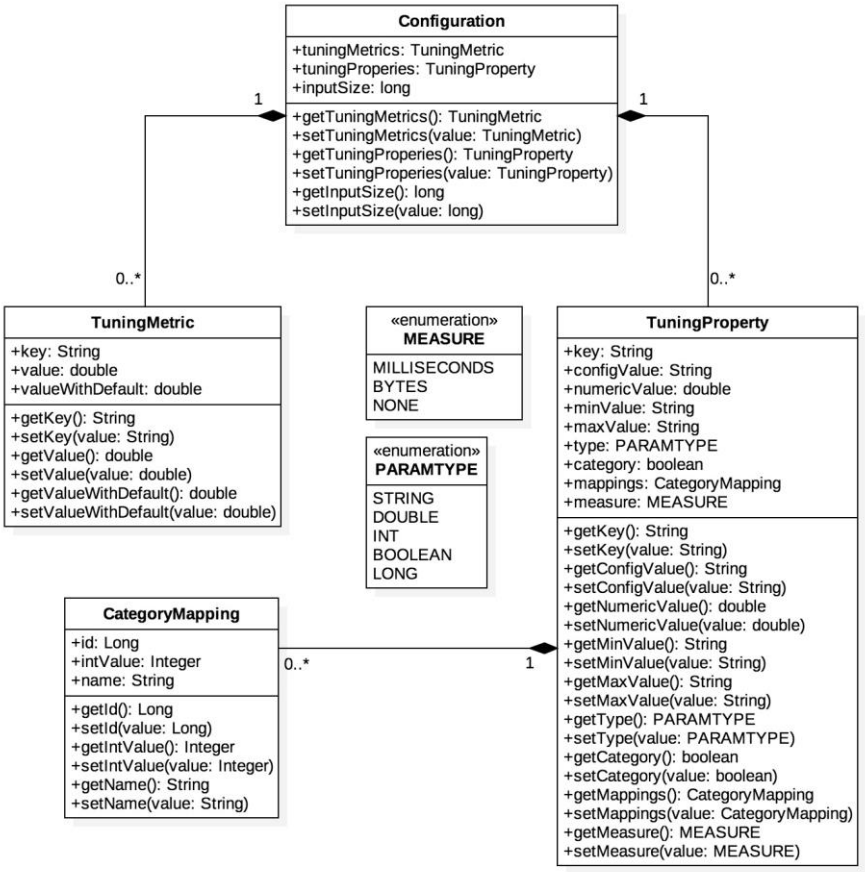

Figure 12. Class diagram: deployment optimizer data model

The *Configuration* class consists of a list of one or more tuning metric, a list of one or more tuning properties and an input size that represents the size of the data to be processed. The *TuningMetric* class contains as attributes the name of the metric to be optimized, the value of the metric recorded with the configuration to which it is linked, and its value with the default configuration. The *TuningProperty* class contains, in addition to the name of the property, information related to the type, e.g. if the property takes Boolean, integer or categorical values, and information on the range of allowable values to be explored for that

given property. If the tuning property is categorical, one o or more *CategoryMapping* are associated to the class *TuningProperty* with a composition relationship.

5.3 Spark tuning properties selection

For the selection of Spark tuning properties, we started from those proposed in the work taken as reference. Initially, the values of the ranges to be explored to the cluster used were simply adapted for our test-bed. Later, after a study of the line guides and documentation for Spark tuning, they were added to these two new tuning properties: *spark.task.cpus* and *spark.serializer*. The first represents the number of cores to allocate for each task, which by default is set to 1. Varying this property between 1 and 4 it's possible to control the parallelism of the tasks in the cluster. The second represents the type of serializer used, choosing between the Java serializer and Kryo. By default, Spark serializes objects using Java's *ObjectOutputStream* framework, while the other, Kryo serializer, is especially recommended for network-intensive applications.

After some test and results, the following tuning properties have been converted into static properties, i.e. the amount of memory and cores to be allocated to the driver and executors are set.

Static Properties	Fixed Value	Description
spark.driver.cores	2	Number of cores to use for the driver process
spark.driver.memory	4g	Amount of memory to use for the driver process
spark.executor.cores	4	The number of cores to use on each executor
spark.executor.memory	11g	Amount of memory to use per executor process

Figure 13. Static properties

This choice is made by considering different aspects:

- By default, the memory allocated per executor is 1 GB. This low value has been noted to worsen heavily on the output metric with default configuration; this value is set at 11 GB
- The Spark property *spark.task.cpus* can conflict with the *spark.executor.cores* property. The number of cores allocated for each task must be smaller, or more equal, the number of cores allocated to the driver and executors. This condition can be violated during the dataset collection process. In fact, the random sampler component randomly generates the values in the set range of each tuning properties

independently. Furthermore, even these properties, default to 1, heavily affected the default metric

The following table shows the parameters used in performance model chosen to optimize the execution time metric.

Parameters	Default	Tuning Range	Description
spark.reducer. maxSizeInFlight	48m	24 - 96m	Maximum size of map outputs to fetch simultaneously from each reduce task
spark.shuffle. compress	true	true/false	Whether to compress map output files
spark.shuffle.spill. compress	true	true/false	Whether to compress data spilled during shuffles
spark.broadcast. compress	true	true/false	Whether to compress broadcast variables before sending them
spark.io.compression. codec	lz4	lz4/lzf/snappy	The codec used to compress internal data such as RDD partitions, event log, broadcast variables and shuffle outputs
spark.broadcast .blockSize	4m	2 - 8m	Size of each piece of a block for *TorrentBroadcastFactory*
spark.default. parallelism	12	1 - 12	Default number of partitions in RDDs returned by transformations like join, reduceByKey, and parallelize when not set by user
spark.speculation	false	true/false	If set to true, performs speculative execution of tasks
spark.task.cpus	1	1 - 4	Number of cores to allocate for each task
spark.serializer	JavaSerializer	JavaSerializer/ KryoSerializer	Class to use for serializing objects that will be sent over the network or need to be cached in serialized form

Figure 14. Spark parameters used in performance model

5.4 Random sampler implementation for tuning properties

In the first phase of the optimization process, the random sampler sub-component generate a certain number, passed as input, of configuration parameters with random values within a specified interval. These will constitute the input for the next phase and represent the dataset, not yet labelled, for the training of the machine learning models. The range of values on which to sample the tuning properties chosen, and the type of data, for example numerical or categorical, is specified in the data model previously shown. A strategy pattern was used, in order to keep only one interface and to manage the sampling of types of parameter at runtime. The following diagram shows the random sampler classes diagram.

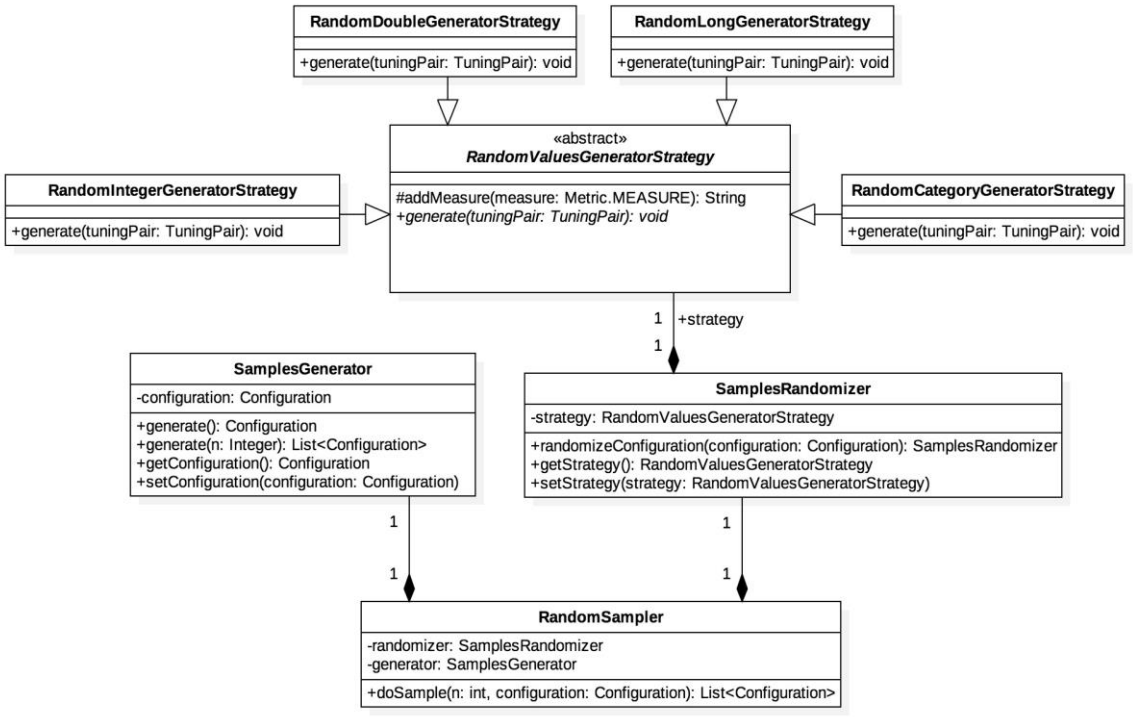

Figure 15. Class diagram: random sampler

As shown above, starting from a data model expressed by the *Configuration* class, and an integer *N*, a list of *Configuration* is generated with random tuning properties.

5.5 Workload dataset harvesting

For the generation of labelled datasets, four different workloads have been implemented using the Spark version 2.1.1. The first three workloads implemented are the same as those employed by G. Wang et al. [14]: *WordCount*, *Grep* and *Sort*, which can be used for general applications. The fourth workload is taken as target for big data analytics applications. The choice of a simple and at the same time representative workload for most used machine learning algorithms, has fallen into a workload that makes low-level derivatives on a numerical input dataset. A function of Spark's MLlib library is used to perform one of the simplest method to solve optimization problems: the gradient descent. The method of descent along the gradient is a technique that allows you to determine the maximum and minimum points of a function of several variables. This method is applied for example in the training of feedforward neural networks, and in gradient boosting. For our use case and tests, the gradient descent as target workload is set.

The fifth workload is instead implemented only for the testing and validation phase of the models generated by the executions of gradient descent workload.

The execution of the target workload is run a number of times equal to the number of configurations passed as input, which were generated by the random sampler. Other key

inputs parameters to set are the name of workload to run, the access configurations to the Spark cluster, the HDFS path where to take the data workload, and the HDFS paths where to store the workload metric logs and the training dataset, saved as a JSON file. Another parameter to set is the number of runs to be performed with the default configuration. This is because metrics' performance is affected by bias during workload execution. To capture this variability, especially in the default configuration, multiple run are executed with the default configuration and the average of that metric for each run is returned.

To collect the metrics, the REST API offered by Spark is used in this first phase of implementation for the optimization of a single metric. Spark allows access, via the web UI on the default 4040 port, to information about applications including: a list of scheduling stages and tasks, environmental information, information about the running executors, and other information from which it is possible to extract the metrics to be optimized. All this information can be accessed during application execution, while at the end of the application it is stored and accessible in the Spark's history server, by default in port 18080. From the Spark API, an application is referenced by its application ID, *[app-id]*. Accessing at *http://<server-url>:18080/api/v1/applications/[app-id]/executors* it is displayed a JSON array, where each JSON reports information for a given executor [20] [21]. The following figure shows the web UI of the spark's history server, and an example of a JSON array reachable through endpoints, showing the metrics reported for each executor.

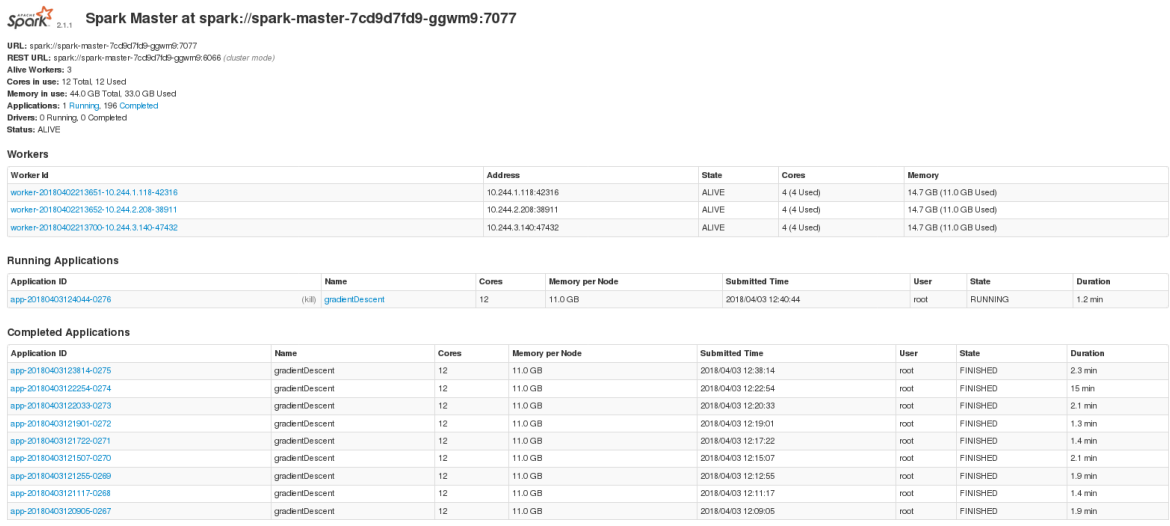

Figure 16. Spark history server

```
[ ⊟
  { ⊟
    "id":"2",
    "hostPort":"10.244.3.34:49765",
    "isActive":true,
    "rddBlocks":0,
    "memoryUsed":0,
    "diskUsed":0,
    "totalCores":4,
    "maxTasks":4,
    "activeTasks":0,
    "failedTasks":0,
    "completedTasks":160,
    "totalTasks":160,
    "totalDuration":146297,
    "totalGCTime":9782,
    "totalInputBytes":15073851632,
    "totalShuffleRead":366644,
    "totalShuffleWrite":574775,
    "maxMemory":6110576640,
    "executorLogs":{ ⊟
      "stdout":"http://10.244.3.34:8081/logPage/?appId=app-20180511212818-2196&executorId=2&logType=stdout",
      "stderr":"http://10.244.3.34:8081/logPage/?appId=app-20180511212818-2196&executorId=2&logType=stderr"
    }
  },
  { ⊞ },
  { ⊞ },
  { ⊞ }
]
```


Figure 17. Metrics reported by Spark Web UI at the executor endpoint

A JSON array is shown at the selected endpoint, with a JSON per executor. Each JSON provides a set of information and metrics.

For the dataset harvesting of a single metric, the execution time, the following steps are performed:

- Target workload is run *M* time with the default configuration.
- Then is run a number of time based on the number of random configurations generated on the last stage. During these run the *app-id* with the default configurations and with the random properties are stored in two separate lists.
- Once all the run are completed, accessing to the spark's history server to the endpoint executors of each application identified by an ID, and among all the executors the max value to key *totalDuration* has taken, identifying with that the execution time of the application.
- All this metrics harvested are added to the deployment optimizer's data model.
- Finally, the data model, now complete with all the input information represented by the values of the tuning properties, and output, represented by the values of the chosen tuning metric, is saved on a HDFS path as a JSON file and will represent the labelled dataset for the training models.

A sample of the JSON file representing the labeled dataset is shown below.

```
[
    {
        "inputSize": 1063834011,
        "tuningMetrics": [
            {
                "key": "totalDuration",
                "value": 128798,
                "valueWithDefault": 151977
            }
        ],
        "tuningPairs": [
            {
                "key": "spark.reducer.maxSizeInFlight",
                "configValue": "95155084b",
                "numericValue": 95155084,
                "minValue": "25165824",
                "maxValue": "100663296",
                "type": "INT",
                "category": false,
                "mappings": null,
                "measure": "BYTES"
            },
            {
                "key": "spark.shuffle.compress",
                "configValue": "true",
                "numericValue": 1,
                "minValue": "0",
                "maxValue": "1",
                "type": "BOOLEAN",
                "category": false,
                "mappings": null,
                "measure": "NONE"
            },
            {
                "key": "spark.shuffle.spill.compress",
                "configValue": "true",
                "numericValue": 1,
                "minValue": "0",
                "maxValue": "1",
                "type": "BOOLEAN",
                "category": false,
                "mappings": null,
                "measure": "NONE"
            }, …
```


Figure 18. A sample of workload output

The JSON file consists of a JSON array of configurations. In the sample of the file shown above, a part of the JSON relative to the first configuration is shown.

The inputs used later by the machine learning model are located within the JSON array *tuningPairs*, where each JSON inside it represents a tuning property. Specifically, the input value is represented by the value corresponding to the *numericValue* key. The value of the *configValue* key, on the other hand, represents the same value, but in the formatting required by the Spark framework. The outputs related to the metrics chosen and measured for a particular configuration are located within the JSON array *tuningMetrics*. In this case, we have inside only one JSON with the metric representing the total duration of the workload, where it is specified, in addition to the name of the metric, the value for that

particular configuration and the value with the default configuration (expressed in milliseconds).

It's used abstraction process to implement the workloads. As shown in the following class diagram, each workload is implemented as an extension of the abstract *Workload* class. Since there are many common methods and fields, this approach easily allow the addition of new workloads to the project starting from a pattern defined in the abstract class.

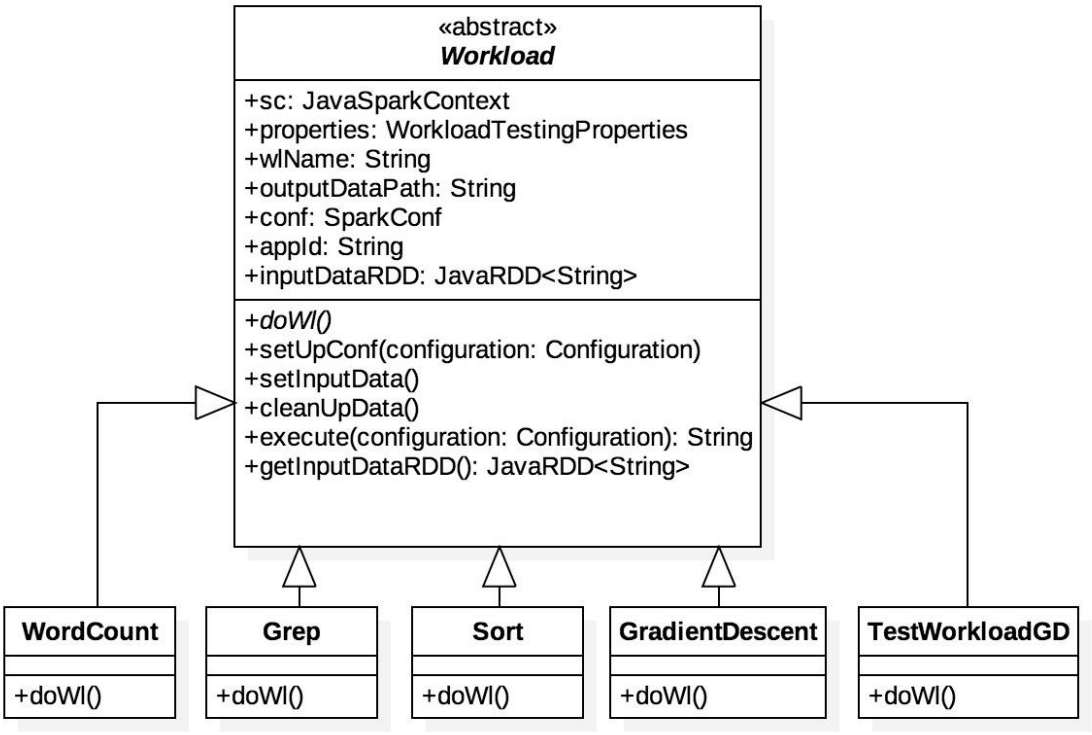

Figure 19. Class diagram: workload

5.6 Technologies and machine learning models used

For the choice of the model it's considered that in the work of G. Wang et al. [14], the best performance in terms of model accuracy is achieved using the decision tree model. For this reason, and based on the models supported by the machine learning technology used, the Random Forest model is used for the optimization of a single metric. Being a supervised model, this model needs a labelled dataset with inputs and outputs for its training. The model's performances were then improved through its validation with datasets harvested from two different distributions, i.e. from two different workload executions. TensorFlow, an open source software library for machine learning models, has been used.

5.6.1 TensorFlow

TensorFlow [22] is an open source library developed by Google for applications based on machine learning, also working on a large scale in a distributed way. It is used in various Google products like GMail, in speech recognition and others. It provides multi-level APIs, from high-level APIs for Python, C / C ++, Java, but also low-level APIs like *TensorFlow Core* to have complete programming control. TensorFlow works using graph structures, or rather the computation is defined on a direct acyclic graph (DAG). The nodes of the graph represent the mathematical operations, while the arcs, called Tensor, represent the flow of data from one node to another. Each node takes zero or more Tensor, which consists of a multi-dimensional array, and produces a Tensor as output. The graph is defined in a high-level language, is compiled, and optimized, and can be executed on one or more CPUs, GPUs. Furthermore, Google has developed a specific integrated circuit for machine learning applications called TPU (Tensor Processing Unit).

A TensorFlow application, in general, consists of two phase:

- Creation of the graph: where we define what each node does
- Execution of the graph: where the complete flow is performed

TensorFlow provides a utility called TensorBoard that can show an image of the computational graph that we have described through some supported language.

5.6.2 Random Forest model

Random Forest [23] is an ensemble learning method for classification, regression and other tasks, that operate by constructing multiple decision trees at training time and outputting the class that is the mode of the classes, for classification tasks, or mean prediction, for regression tasks, of the individual trees, to get a more accurate and stable prediction.

Random Forest can prevent over-fitting creating random subsets of the features and building smaller tress using these subsets. Then, the sub-trees are combined. A lot of number of trees cannot always prevent over-fitting and it also makes the computation slower as the number of trees increase. It's considered very easy to use algorithm, because it's default hyperparameters often produce a good prediction result and are straightforward to understand. The hyperparameters used for the validation of the model on TensorFlow were: the number of steps for training and the number of trees to be generated, leaving others unchanged to their default values.

5.7 Random Forest implementation for deployment optimizer

The chosen model is implemented on Python using the TensorFlow library. As input the script takes the HDFS path to the configuration data model in JSON format, saved at the end of the workload execution stage. Once imported, this JSON have to be processed in order to construct two labelled datasets for binary and multiple classifier training, using the random forest model.

Each value of tuning properties related to a configuration represents an input row of the binary classifier dataset. In the first pre-processing phase, it's built this dataset labeling with 1 if the value of the metric for that given configuration is better than the default one, otherwise 0. If it returns 1 that same row is also added to the dataset for the training of the multiple classifier, but now labelling it with the percentage increase of the performance that was had. These percentage ratios between the metric obtained with a given configuration and the default one is then rounded up to the nearest integer, so as not to have isolated labels with just one example for training.

The second phase of pre-processing consists in transforming the labels of the multiple classifier so as to be mapped in the interval [0, N-1], where *N* are the number of labels, in order to be properly trained by the Random Forest model. The labels are then mapped in descending order, representing with 0 the label with which the higher performance increase is obtained, up to *N-1* representing the minor increase obtained, based on the training dataset.

Once prepared the datasets, the models are trained and lastly are saved using the *SavedModelBuilder* class. This class [24] provide functionality to build a *SavedModel* protocol buffer, allowing to save meta graphs as part of a single language-natural, but at the same time sharing variables and assets. TensorFlow provides APIs for use in Java programs, but the training of machine learning models is only possible on Python language. For this reason, the task of training models is delegated to a Python script, saving them as proto-buffer files and then executing them within a Java application.

The following figure shows the computation graph of the binary classifier training shown by TensorBoard visualization tool.

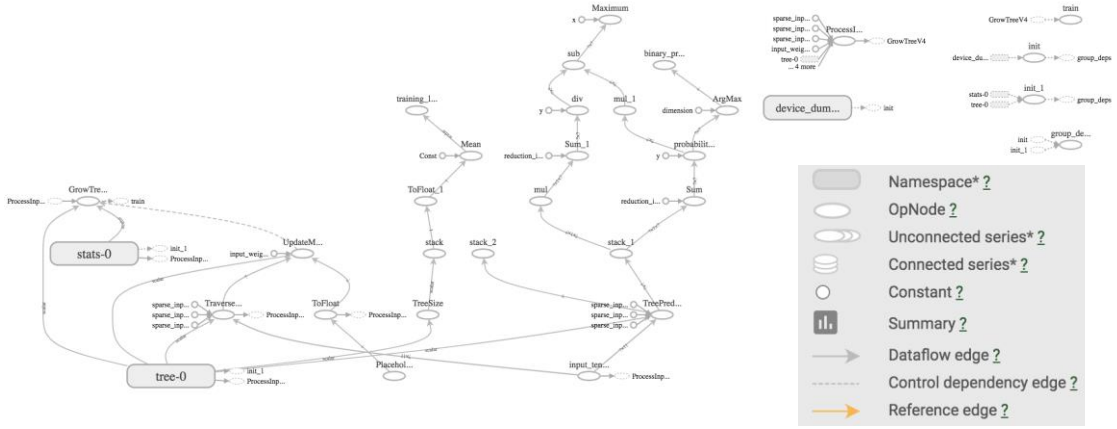

Figure 20. TensorBoard: Binary Classifier Training

5.8 Recursive random search implementation

In the last phase of the process the RRS search algorithm is used, the models saved by the Python script are loaded, and the random sampler sub-component is used again to generate new random configurations. The implementation is described by the following class diagram:

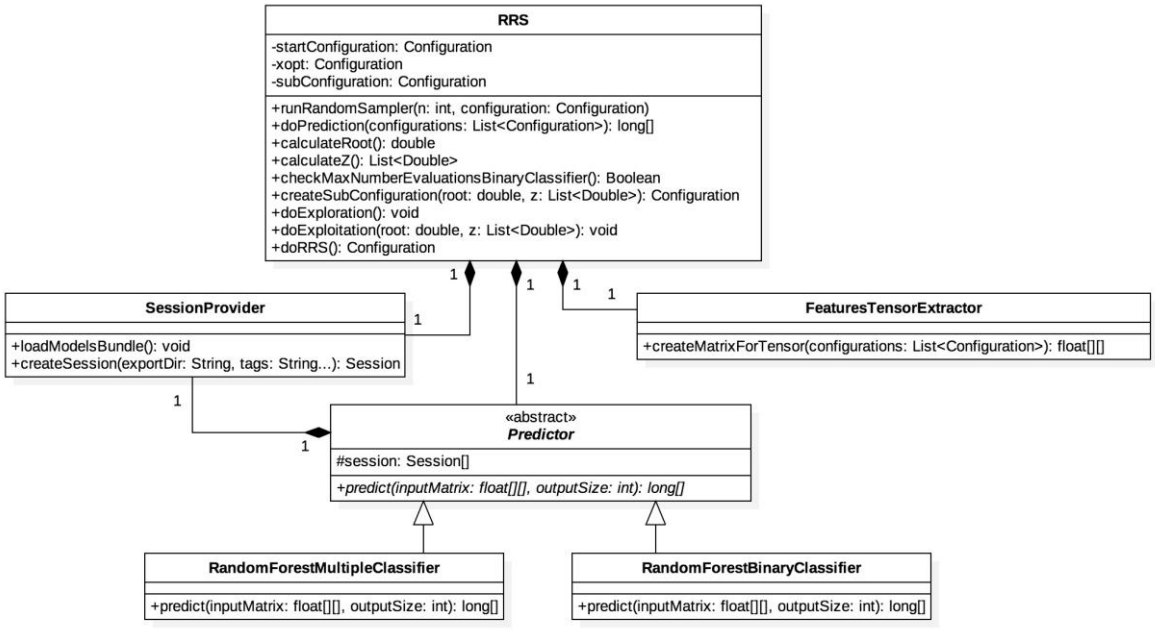

Figure 21. Class diagram: workload

The main class, RRS, executes the efficient search algorithm described in chapter 3. The inputs to be passed are: the paths relative to the models saved by the Python script in proto-buffer format; and the initial *Configuration* containing all the information of the tuning

properties in order to be processed by the random sampler. A sub-optimal configuration found by the algorithm is returned as output.

Two stop criteria were chosen for the recursive random search algorithm. The algorithm, in the first place, should terminate once the best possible output for a configuration is returned by the multiple classifier. The optimal output is labeled with '0', representing the best possible increase, in percentage terms, compared to the default configuration. If you are in a very large parameter space, this search, even due to local minima, may never end, or take a long time. For this reason, as a second stop criterion, a limit on the predictor calls equal to 500 has been set. Once this limit has been reached, the best configuration found up to that time is returned.

In order to import the random forest models, and then use them to make predictions, we had to manually import some missing libraries, not listed in the TensorFlow Java API. It's used abstraction process to implement the predictors for *RandomForestBinaryClassifer* and *RandomForestMultipleClassifer*, starting from the abstract *Predictor* class. This pattern allows you to easily add new predictor types, and modify or extend existing ones. The predictions are made on *Configuration* or *Configuration* lists generated by the random sampler sub-component.

During the exploration phase of the RRS is taken the best *Configuration* object from a list of *Configuration* according to the outputs returned by the classifiers.

During the next exploitation phase, random configurations are generated in a parameter window space that gradually shrinks and re-aligns. The shrinking and alignment of the window operationally results in getting a new *Configuration*, starting from the initial one, and modifying the values of *minValue* and *maxValue* fields accordingly. These fields in fact identify the range of values from which to extract a random value for a parameter.

5.9 Tests and results

The random forest models used for the generation of the binary and multiple classifier model, to optimize a single metric, are validated in order to find the hyper-parameters that would lead to a better overall performance.

The workload and the metric chosen are the gradient descent workload for big data analytics applications and the execution time. To find these hyper-parameters, a process of error analysis is performed on different types of sets: train, validation and test set.

So it's harvested datasets from two different distributions. The first is the dataset harvested through the executions of the gradient descent workload. The second through the execution of a workload test for the gradient descent. This workload does not perform basic

operations like the first workload, but does the training of three machine learning models using Spark's MLlib library: k-means, Naive Bayes and decision tree.

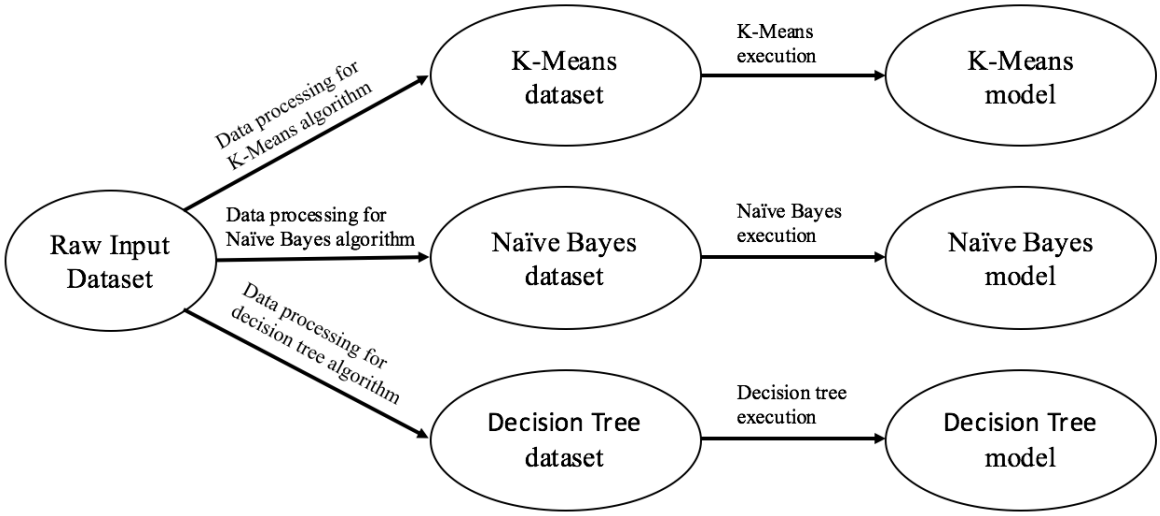

Figure 22. Test Workload Workflow for Gradient Descent

The workload takes as input a generic dataset with categorical and numeric values; also specifying as inputs: the numeric and categorical columns, and the output columns to the workload datasets. Building a validation/test set on the metrics performances of both supervised and unsupervised ML algorithms, we end up having a distribution which slightly differs from the training one. This is due to the variety of algorithms and computations not involved previously on the building of the training set that was based on Gradient Descent only. As a matter of fact, cross-out validation step is crucial to refine the model hyperparameters.

Having to train two supervised and one unsupervised models, starting from a generic dataset, the respective model datasets are generated in an appropriate manner.

These transformation operations on the dataset have been defined on the *DatasetTransformer* class.

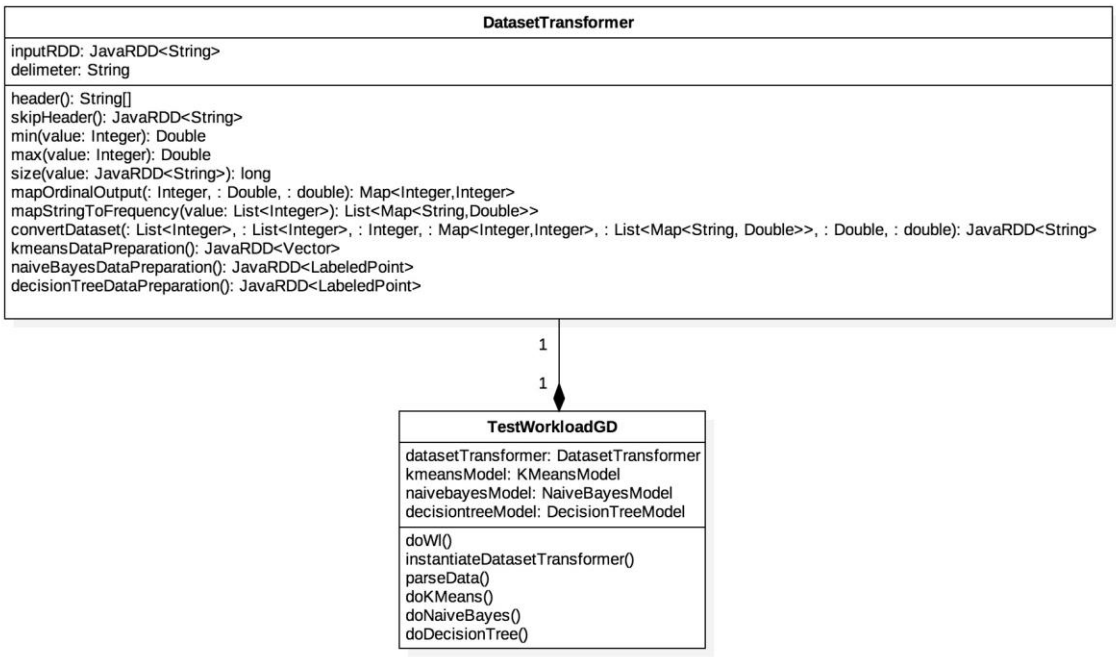

Figure 23. Class diagram: TestWorkloadGD

Once the two distributions have been obtained, the first one has been divided into training and training-dev set. The second, obtained by the execution of the test workload for gradient descent, has been divided into dev and test set.

5.9.1 Error analysis process

The model is trained with the training set and the gap between errors, or equivalent to accuracy, on different sets are analyzed as schematized in the following figure, in order to solve problems related to model training such as under-fitting and over-fitting.

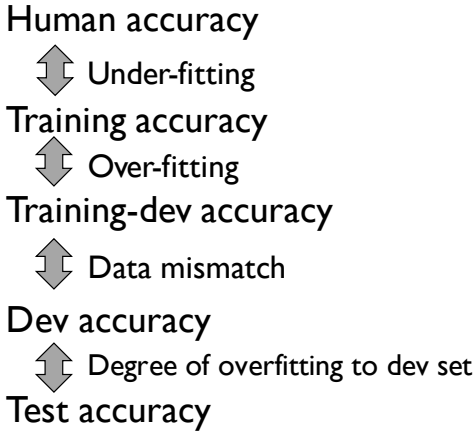

Figure 24. Error analysis process

In the first phase, human error is compared to the error on the training set. If the gap between these two is too high you have an under fitting problem. This arises when the model created is too simple: in the case of a decision tree for example when there are few

nodes. It is therefore necessary to have, in general, a larger model that is better suited to the data. Not having data about human error, this phase was not taken into account, but we tried to keep a training error low.

In the second phase is compared the error on the training with the error on the training-dev. A high gap means that you have an over-fitting problem. In this case, the model is very well suited to the training set, leading to training errors many lows, but fails to generalize. To avoid this problem, you could adopt variance reduction techniques or collect more training data.

In the third phase is compared the error on the training-dev with the error on the dev from a second distribution. A high gap in this phase means that there is a data mismatch. This gap can be mitigated by analyzing the differences between training and dev set, and synthesizing artificial training data.

In the last phase is compared the error gap between dev and test set, and if too high it means there is overfitting on the dev set. This is adjustable by increasing the dev set.

5.9.2 Results on test-bed

In the test-bed a dataset of 1000 row is generated for the first distribution processed with a data workload of 1GB, while 100 for the second one processed with a data workload of 1.2GB. In both workload executions, a number of run has been set with the default configuration to 3, in order to have on the default configuration a value of the chosen metric, the execution time, less affected by the bias.

The datasets collected from the two distributions were then split into:

- Training: 900
- Training-dev: 100
- Dev: 80
- Test: 20

The hyper-parameters explored for the Random Forest model are:

- Number of steps → [1, 10, 25, 50, 100]
- Number of trees → [1, 10, 50, 100, 250]

The split process its carried out randomly 5 times, thus obtaining 5 training, training-dev, dev and test set. At this point, a validation process is performed by iterating on the combination of the hyper-parameters listed above, and imposing a gap maximum error equal to 10%. For a given combination of hyper-parameters, the training process and prediction for the error analysis is run on the 5 sets split randomly; finally making an average on the reported accuracy.

The accuracy on the binary classifier is calculated as follows:

$$Accuracy_{BC} = \frac{\sum True\ positive + \sum True\ negative}{\sum Total\ population}$$

In the multiple classifier, during the prediction phase the random forest model returns a label between 0 and N-1, where with 0 the maximum achievable performance increase is identified, while with N-1 the minimum one. This label is then remapped to its original value, and compared to the true value. Since the model may have been trained with labels that are not present in the non-training dataset, a threshold value of 10% has been set. For example, if the multiple classifier predicts an X label with a given configuration, which corresponds to a performance increase of 15%, while the true value is 20%, it's considered as belonging to the chosen label because their difference in absolute value it is less than 10%.

At the end of the validation, the combination of hyper-parameters that led to better performance accuracy on test set, and at the same time passing the error analysis, for the binary and multiple classifier respectively is reported below:

Random Forest	Training accuracy	Training-dev accuracy	Dev accuracy	Test accuracy	Number of trees	Number of steps
Binary Classifier	0,949	0,892	0,827	0,832	10	50
Multiple Classifier	0,82	0,81	0,796	0,775	10	1

Figure 25. Random Forest: one single metric optimization – Validation results

With the combination of best hyper-parameters found, a final accuracy on the test of 83% for the binary classifier was obtained, while 78% for the multiple classifier.

The sub-optimal configuration found by the deployment optimizer for the optimization of the total duration metric, compared with the default one, is shown in the following table.

Tuning Properties	Sub-Optimal Value	Default Value
spark.reducer.maxSizeInFlight	35m	48m
spark.shuffle.compress	true	true
spark.shuffle.spill.compress	true	true
spark.broadcast.compress	true	true
spark.io.compression.codec	snappy	lz4
spark.broadcast.blockSize	7m	4m
spark.default.parallelism	8	4
spark.speculation	false	false
spark.task.cpus	3	1
spark.serializer	JavaSerializer	JavaSerializer

Figure 26. Sub-optimal configuration

The most remarkable tuning properties that differ from the corresponding default values are:

- *spark.broadcast.blockSize*: this property sets the size of each piece of a block for the broadcasted data to the executors. The optimal value found is almost double that of the default one, and that means to have a smaller exchange of data between the executors but still keeping a high level of parallelism during broadcast.
- *spark.default.parallelism*: the default number of partitions in RDDs returned by transformation like *join*, *reduceByKey*, and *parallelize* are set to the number of cores on all executor nodes (4 in this case). The optimal value found is 8, doubling the number of partitions and the leveraging the parallelism of Spark.
- *spark.task.cpus*: by default for each task is allocated only 1 core. Increasing the number of cores to allocate for each task, it reduces the number of tasks that are performed in parallel. The optimal value found is 3, and having 4 cores per executor means performing a single task per execturor. It may seem disadvantageous, but having more tasks in parallel also means having more overhead from GC and I/O.

These properties together highlight the fact that Spark is mostly a data parallelism engine and its parallelism is achieved mostly through representing the data as RDDs. Thus optimization of the execution time of big data analytics applications is achieved through the tuning of the resource allocation and parallelism, fully exploiting the potential of Spark.

Chapter 6
Multiple Metrics Optimization

This chapter describes the implementation choices for extending the deployment optimizer proposed to the optimization of multiple metrics. For the optimization of multiple metrics the sub-components that need an extension are those concerning the following phases:

- The metrics collection stage through the executions workload, now using a metric collection tool rather then Spark web UI.
- In the training stage we have now to take into account multiple metrics, working as in the proposed method: by training a model per metric, or by using a single multi-task model.
- In the search for the sub-optimal configuration, the RRS sub-component have to be modified accordingly, considering predictions for each metric.

The only unaltered component is the random sampler, since it operates independently from the number of metrics to be optimized. In fact, even if the metrics take into account different tuning pairs with respect to other metrics, the merging process carried out upstream makes it possible to optimize all the metrics simultaneously, and considering a single set of tuning properties.

6.1 Workload dataset harvesting using SparkOscope component

For the collection of multiple metrics to be optimized, a component called SparkOscope, was used. This component extends the Spark web UI and allows to detect and save custom metrics on HDFS which are more than those available from the Spark web UI. Among these are, for example, the metrics *cigar.cpu* and *sigar.ram*, that respectively return the percentage of CPU and RAM usage.

As reported on the GitHub project documentation [27], "in order for the executor metrics to be stored in HDFS and therefore be retrieved by the UI, you need to have the following in the *metrics.properties* file" (of Spark framework):

```
executor.sink.hdfs.class=org.apache.spark.metrics.sink.HDFSSink
executor.sink.hdfs.pollPeriod = 20
executor.sink.hdfs.dir = hdfs://localhost:9000/custom-metrics
executor.sink.hdfs.unit = seconds
```

Figure 27. Configuration HDFS metrics sink. Source [27]

To understand the structure of the logs saved by the component, a test has been performed locally with the Docker image provided on the project, and is set as follows. Inside the custom-metrics folder there are the log folders named with the app ID of the applications run by Spark. Inside the log folder there are several log files, each generated by an executor, and which we will call executor log. During the execution of the workload, every Spark executor will append every 10 seconds, each in its executor log, in JSON format, the measurements of the metrics. In fact, SparkOscope is mainly designed to show plots of the custom metrics available, in real time, on the web Spark UI, and using MQTT rather then HDFS (that is slower).

Therefore not having a single metric value, as there are timed detections, it's implemented a method of aggregation of the metrics harvested. The method performs for each executor log an average on the present detections and finally, for each metric, the largest among the executor logs is taken.

A metric collection tool is therefore necessary and essential for the deployment optimizer process, not only in terms of the quantity and types of metrics available, but also on the quality of the detections.

6.2 ANN multi-task and multiple random forest model

Once the dataset labeled with the metrics collected with SparkOscope is obtained, it must be processed by the Python script that deals with the training of the performance models. For the choice of machine learning models, this time is chosen for two ways:

- The random forest model is reused, but this time generating *N* models for the binary classifier and *N* for the multiple classifier, where *N* is the number of metrics.
- A neural network multi-task approach is used to train only 2 models.

6.2.1 Neural network multi-task approach

In the field of machine learning, an artificial neural network (ANN) is a mathematical model of calculation based on biological neural networks. This model consists of a group of information interconnections consisting of artificial neurons and processes that use a

calculation connection approach. In most cases an artificial neural network is an adaptive system that changes its structure based on external or internal information flowing through the network during the learning phase.

In practical terms, neural networks are non-linear statistical data structures organized as modeling tools. They can be used to simulate complex relationships between inputs and outputs that other analytical functions can not represent. There are multiple types of neural network, each of which come with their own specific use cases and levels of complexity. The most basic type is the feedforward neural network, in which information travels in only one direction from input to output.

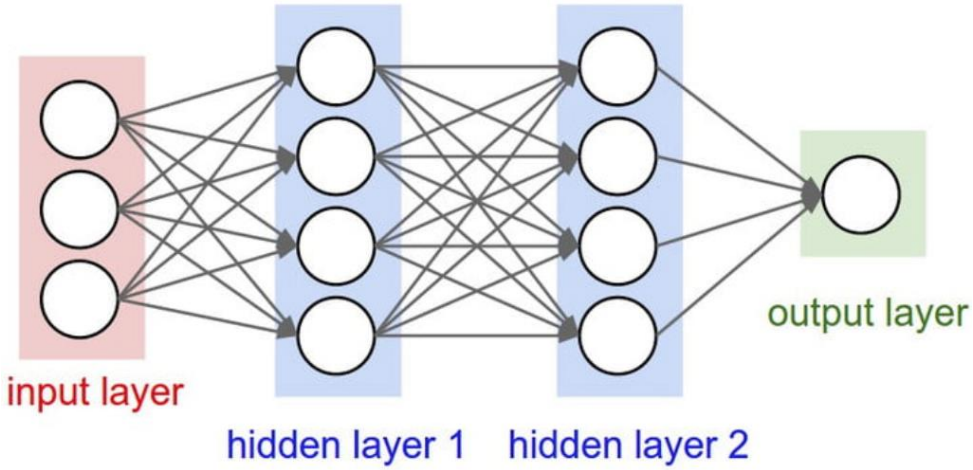

Figure 28. Configuration HDFS metrics sink. Source [28]

As shown in the figure above, an artificial neural network receives external signals on a layer of input nodes (processing units), each of which is connected to numerous internal nodes, organized in several levels. Each node processes the received signals and transmits the result to successive nodes.

We generally train a single model or an ensemble of models to perform our desired task. But, by sharing representations between related tasks, we can enable our model to generalize better on our original task. This approach is called Multi-Task Learning (MTL). The multi-task learning as being inspired by human learning. For learning new tasks, we often apply the knowledge we have acquired by learning related tasks. [28]

Multi-task learning has been used in many fields application of machine learning, from natural language processing to computer vision.

With TensorFlow it's possible to create multi-task neural networks by following two approaches [11].

The first approach, called alternate training, the input batches for training are alternately used to minimize a given task. All tasks share a part of the network, continually transferring some of the information from each task to the other.

This approach is typically used when having different training datasets for each task. Nevertheless, alternate training can become biased towards a specific task, especially if the dimensions of the datasets are disproportionate towards the training of one of the tasks.

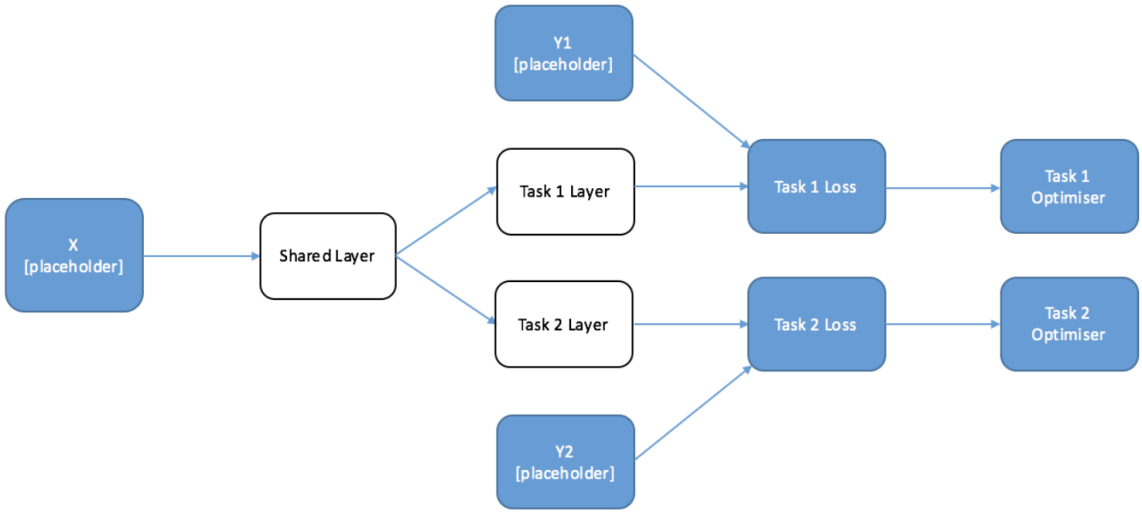

Figure 29. Multi-task learning – Alternate training. Source [11]

Taking the example of alternate training in the figure above, if a training data batch is used to train the parameters of task 1, it will not touch the parameters of task 2 layer. Only the layer shared by the tasks is modified.

In our exam case, we have a dataset with multiple labels. So, what we would like to do is train the tasks at the same time preserving the independence of the task-specific functions. This can be achieved easily by optimizing the join of loss functions of the individual tasks: joint training approach.

The following figure shows and example of this approach.

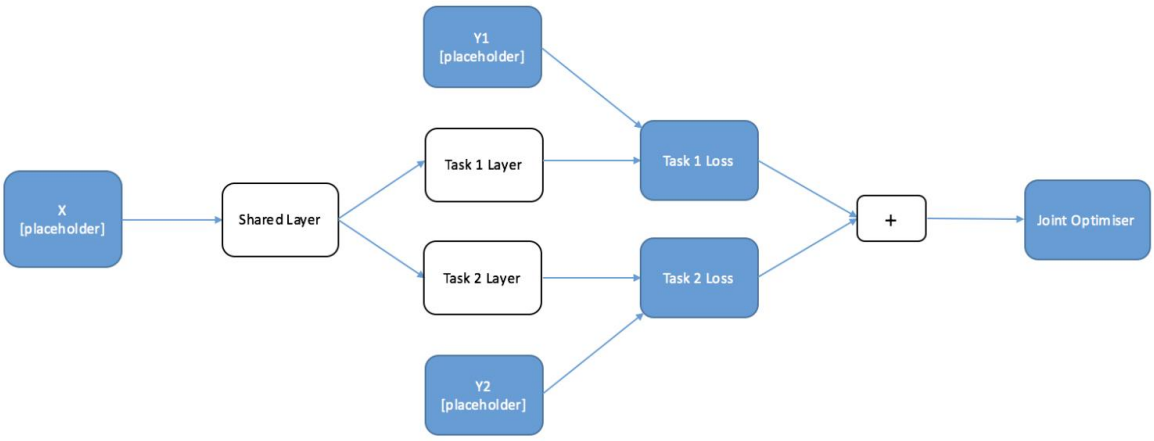

Figure 30. Multi-task learning – Joint training. Source [11]

6.2.2 Models implementation

The multi-task neural network was used following the joint training approach. The Python script, using the TensorFlow library, dynamically generates a multi-task neural network according to the number of metrics in the labeled dataset. Similarly, in the case of the Random Forest model, a total of *2N* models are trained and saved.

For implementation, abstract class as design pattern was chosen as shown in the following class diagram.

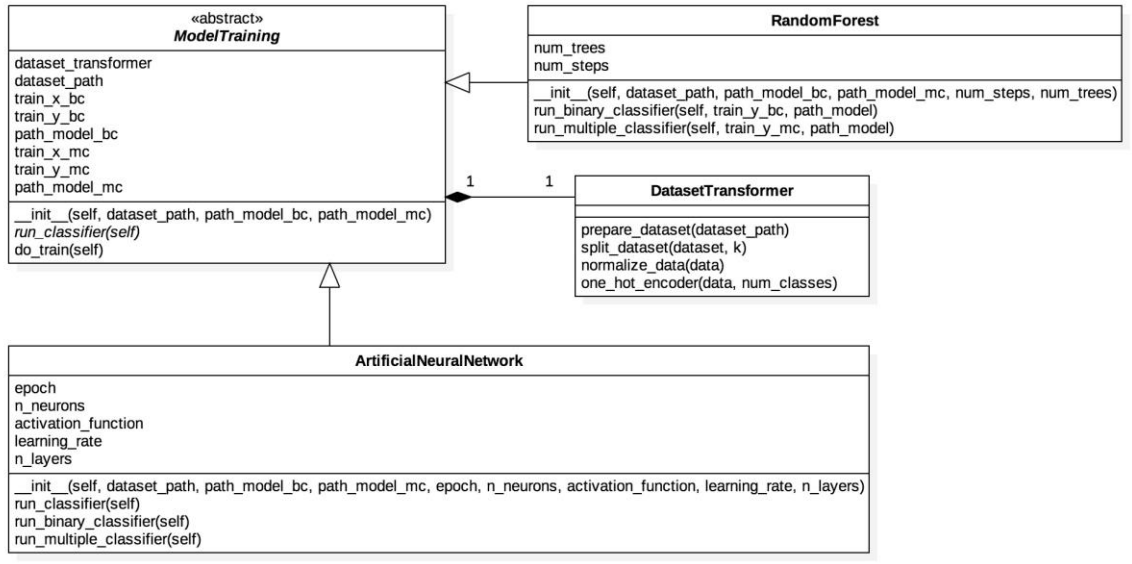

Figure 31. Class diagram: machine learning models

From the *ModelTraining* abstract class derive the specialized classes related to the training of different models of machine learning, such as: *RandomForest* and *ArtificialNeuralNetwork* class. The *RandomForest* class contains among the fields the hyper-parameters of the model: the maximum depth of the tree and the number of steps to be performed; in a similar way the *ArtificialNeuralNetwork* class contains hyper-parameters such as the learning rate and the number of epochs.

Instead, *DatasetTransformer* is a support class used to prepare the dataset for binary and multiple classifiers.

The pre-processing of the neural network requires a standardization of data, while one-hot encoding have to be applied to the labels.

To filter the configurations that will be used for the training of the multiple classifier, it's used an aggregation function to bring us back to having a single binary filter. The selected aggregation of binary classifiers outputs, produces the maximum number of occurrences between 0 and 1. In case of a tie, 0 is returned. Configurations labelled with 1, as in the

case of a single metric optimization, are passed to the training of *N* multiple classifiers, or a single multiple classifier multi-task. But additionally, if in the configurations that pass to the multiple classifier there are metrics that have a worse performance than the default configuration they are labelled with 0.

The concept of a binary filter was thus maintained, also applied when we wanted to optimize multiple metrics at the same time.

6.3 Recursive random search implementation applied to multiple metric optimization

The search in space of tuning properties now consists in finding a configuration that is sub-optimal at the same time for all the metrics. That is, the ideal goal would be to find a configuration that has as output from the binary classifier all 1, while from the multiple one, for how it was implemented, all labels to 0 (remembering that with 0 is represented the greatest possible improvement of performance compared to default configuration).

Leaving the execution of the search algorithm unchanged, the modification was made on the predictor method. An aggregation method has been added for the predictor of the binary classifier. The method returns the maximum number of occurrences between 0 and 1, returning 0 in case of tie. This aggregation result is calculated by the following formula:

$$BinaryAggregation = \begin{cases} 1, & \sum_{i=0}^{n-1} BC_{Prediction}\,(i) > 0 \\ 0, & otherwise \end{cases}$$

, where *n* represents the number of metrics used and $BC_{Prediction}(i)$ represents the prediction performed by the binary classifier for the *i-th* metric.

If a configuration has output 1 from the binary aggregation output, it will be submitted to the predictor of the multiple classifier. For the same reason, an aggregation method has been added downstream of this predictor.

This time the aggregation consists of the arithmetic mean of the outputs of the multiple classifier reported, and calculated by the following formula:

$$MultipleAggregation = \frac{\sum_{i=0}^{n-1} MC_{Prediction}\,(i)}{n}$$

In this way, even if more metrics have sub-optimal configurations distant in the parameter space, the algorithm is getting closer and closer to a configuration that on average improves all the metrics.

In order for the algorithm to converge for the metrics chosen to be optimized, they should all be perfectly orthogonal. Each metric should depend on a different sub-set of tuning

properties set. Even in the case of non-convergence, the search process after 500 iterations returns the best configuration found up to that moment, operating as just described.

6.4 Tests and results

The random forest models and artificial neural network with multi-task approach used for the generation of the binary and multiple classifier model, to optimize multiple metric, are validated operating as in the case of the optimization of a single metric, i.e. through the so-called error analysis, in order to find the hyper-parameters that would lead to a better overall performance.

The metrics chosen to be optimized are captured by the SparkOscope tool, and are: *sigar.cpu* and *sigar.ram*, which respectively express the CPU and RAM usage of the application. This time, the tests were performed locally from the Docker image provided by the developers themselves, in which there is already everything you need to use Spark with the SparkOscope tool, in order to save a series of custom-metrics on HDFS.

In this local test-bed a dataset of 500 row is generated for the first distribution processed with a data workload of 100MB, while 100 for the second one processed with a data workload of 100MB. In both workload executions, a number of run has been set with the default configuration to 3.

The datasets collected from the two distributions (from the gradient descent workload and test workload execution) were then split into:

- Training: 450
- Training-dev: 50
- Dev: 80
- Test: 20

The hyper-parameters explored for the Random Forest model are:

- Number of steps → [1, 10, 25, 50, 100]
- Number of trees → [1, 10, 50, 100, 250]

The hyper-parameters explored for the Artificial Neural Network model are:

- Learning rate → [0.3, 0.5, 0.7, 0.9]
- Number of neurons → [10, 25, 50, 75, 100]
- Number of hidden layers → [2, 5, 10, 25, 50]
- Activation function → [sigmoid, relu]
- Number of epochs → [1, 10, 50, 100, 250, 500]

In the process of calculating accuracy, it must be taken into account of having more then one metric and so a number of different accuracy for each metric. The process of

calculating the aggregate final accuracy for the two classifiers is described by the following figure, assuming this calculation for two metrics.

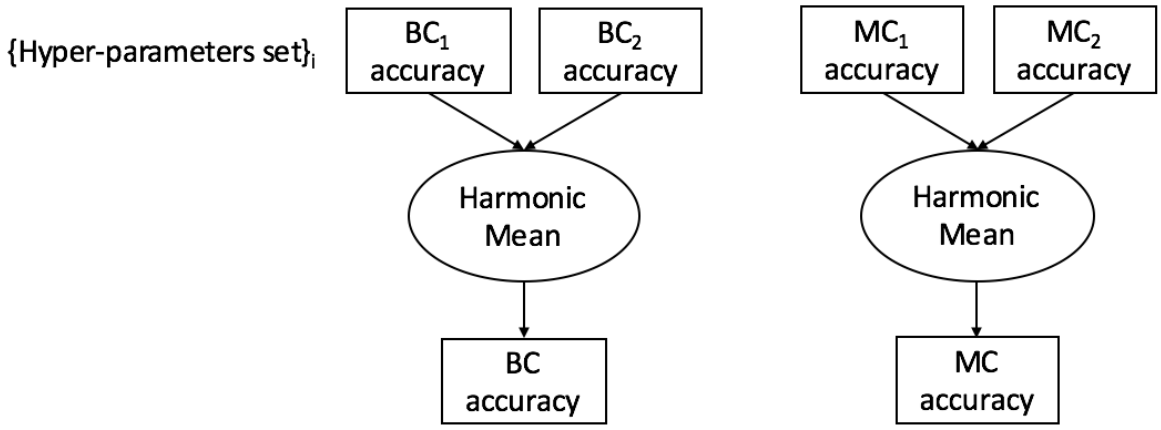

Figure 32. Accuracy aggregation process for multi-metric optimization

Given the trained models with a set of hyper-parameters specified, we calculate the accuracy for each metric and these are aggregated through the harmonic mean, that is the result will be given by the reciprocal of the arithmetic mean of the reciprocal of the individual accuracy. This process is repeated both for the binary classifier and for the multiple classifier, obtaining a single accuracy value. In turn, the individual accuracy are given as the average of 5 individual accuracy that come from 5 different models trained with a different portion of training set.

At the end of the validation, the combination of hyper-parameters that led to better performance accuracy on test set, and at the same time passing the error analysis, for the binary and multiple classifier respectively is reported below:

Random Forest	Training accuracy	Training-dev accuracy	Dev accuracy	Test accuracy	Number of trees	Number of steps
Binary Classifier	0,72	0,63	0,584	0,547	1	50
Multiple Classifier	0.693	0.614	0.627	0.667	25	10

Figure 33. Random Forest: multiple metric optimization – Validation results

The validation of the ANN model did not lead to good results, maintaining with a high degree of over-fitting of the model therefore with a low accuracy in the test set, or in a reverse situation of under-fitting. The error analysis was therefore not exceeded due to the high error deviations between the sets.

The following tables show the problems related to under-fitting and over-fitting of the model, respectively.

ANN Multi-Task	Training accuracy	Training-dev accuracy	Dev accuracy	Test accuracy	Learning rate	Neurons per hidden layer	Hidden layers	Activation function	Epochs
Binary Classifier	0,418	0.405	0.626	0.635	0.1	10	5	sigmoid	50
Multiple Classifier	0.483	0.295	0.612	0.596	0.3	10	5	sigmoid	250

Figure 34. ANN multi-task: multiple metric optimization – Validation results with under-fitting problem

In fact, in the first configuration, using a few training steps and a few hidden layers, the best result was obtained on the test test, to the detriment of the training accuracy, having therefore a problem upstream of under-fitting, assuming that human error is lower.

ANN Multi-Task	Training accuracy	Training-dev accuracy	Dev accuracy	Test accuracy	Learning rate	Neurons per hidden layer	Hidden layers	Activation function	Epochs
Binary Classifier	0.642	0.45	0.303	0.265	0.3	10	10	relu	500
Multiple Classifier	0.641	0.51	0.326	0.234	0.1	25	25	sigmoid	500

Figure 35. ANN multi-task: multiple metric optimization – Validation results with over-fitting problem

In the second configuration, instead, using a bigger network and with a higher number of epochs, there is a problem of over-fitting due to the high gap between the training set and the training-dev set.

The sub-optimal configuration found by the deployment optimizer for the optimization of the CPU and RAM usage, compared with the default one, is shown in the following table.

Tuning Properties	Sub-Optimal Value	Default Value
spark.reducer.maxSizeInFlight	39m	48m
spark.shuffle.compress	false	true
spark.shuffle.spill.compress	true	true
spark.broadcast.compress	true	true
spark.io.compression.codec	lz4	lz4
spark.broadcast.blockSize	8m	4m
spark.default.parallelism	10	4
spark.speculation	true	false
spark.task.cpus	2	1
spark.serializer	KryoSerializer	JavaSerializer

Figure 34. Sub-optimal configuration for multi-metric optimization

The values found are similar to the configuration found for the optimization of execution time. Both in fact show a finer tuning of the parallelism properties on the data and on which Spark relies mainly, thus separating from the default defaults.

Chapter 7
TOREADOR Platform

The thesis work produced, aims to be integrated into the TOREADOR platform, as a deployment optimizer component. In particular, for big data processing on the frameworks like Spark, supported by the platform.

TOREADOR [29] is a project, co-funded by European Union's Horizon 2020 research and innovation programme, aimed at overcoming some major hurdles that until now have prevented many European companies from reaping the full benefits of Big Data Analytics (BDA). Companies and organisations in Europe have become aware of the potential competitive advantage they could get by timely and accurate Big Data analytics, but lack the IT expertise and budget to fully exploit BDA. To overcome this hurdle, TOREADOR takes a model-based BDA-as-a-service (MBDAaaS) approach, providing models of the entire Big Data analysis process and of its artefacts.

TOREADOR open, suitable-for-standardisation models will support substantial automation and commoditization of Big Data analytics, while enabling it to be easily tailored to domain-specific customer requirements. Besides models for representing BDA, TOREADOR will deliver an architectural framework and a set of components for model-driven set-up and management of Big Data analytics processes.

Once TOREADOR MBDAaaS will become widespread, price competition on Big Data services will ensue, driving costs of Big Data analytics well within reach of EU organizations (including SMEs) that do not have either in-house Big Data expertise or budget for expensive data consultancy.

TOREADOR framework addresses automatically all major problems of on-demand data preparation, including handling Big Data opacity, diversity, security, and privacy compliance, and will support abstract modelling of the BDA life cycle from distributed data acquisition/storage to the design and parallel deployment of analytics and presentation of results.

In the TOREADOR project, Engineering Ingegneria Informatica SpA is responsible for the integration and deployment of the TOREADOR Platform and for the implementation of data representation, storage and retrieval services.

7.1 TOREADOR architecture

The TOREADOR Platform architecture aims at turning the conceptual specification for DBA-as-a-service into platform and integration specification for the actual services supporting the modelling, integration and deployment of specific BDA applications in several domains. The TOREADOR architecture defines and specifies the ‘consumption’ of big data analytics as service by integrating and exploiting the toolsets and services for big data value management and analysis resulting from the project.

As the following figure illustrates, the communication of the directives concerning the service management is granted by a TOREADOR Platform API which exposes the following REST resources:

- Service registration
- Workflow definition
- Workflow execution

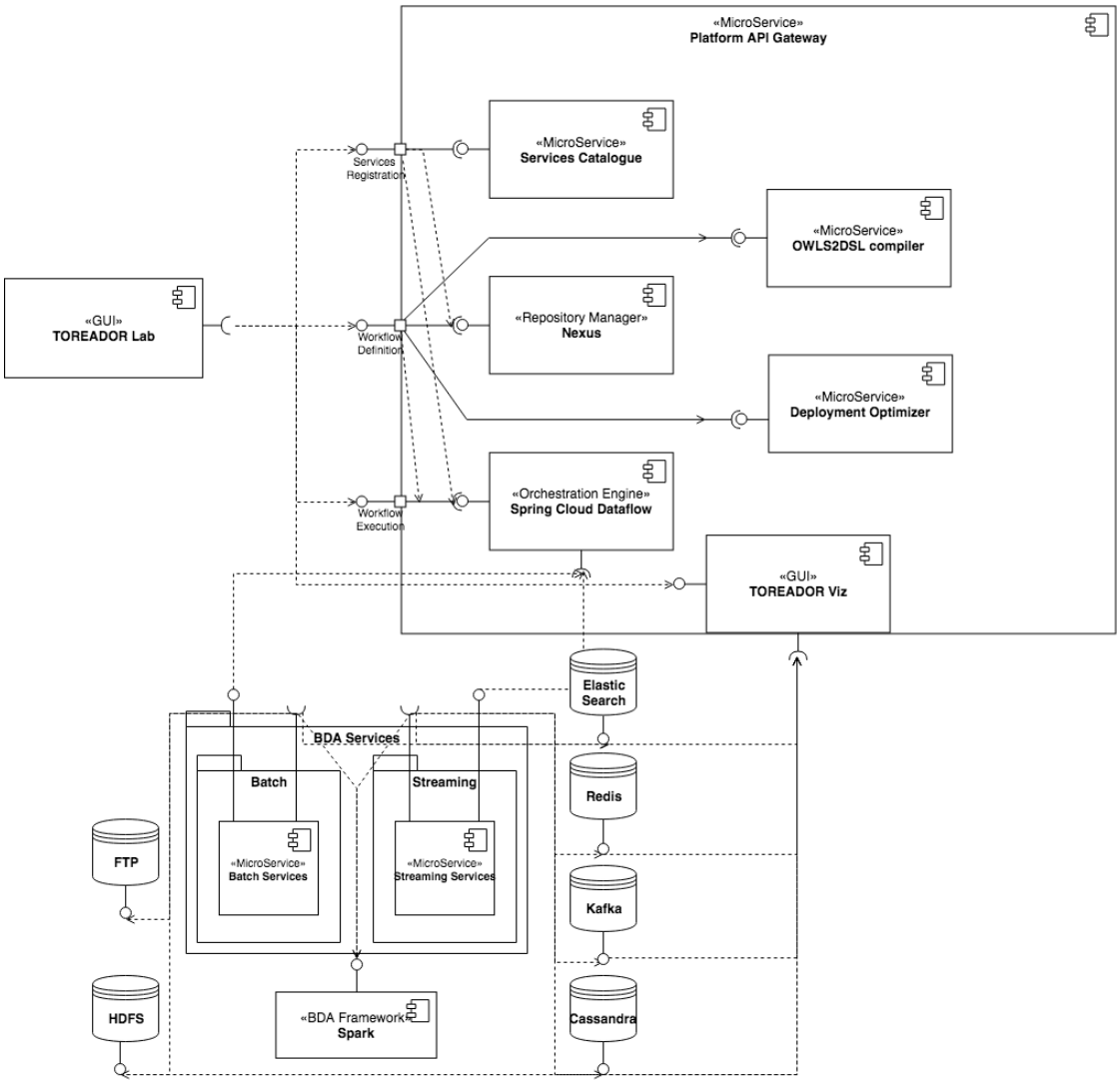

Figure 35. TOREADOR component diagram

The exposure of a REST API enables the Machine-to-Machine (M2M) interaction with the model definition interfaces, which, according with the TOREADOR methodology, supply a set of user-level and abstract definitions of the expected behaviour of the *BDaaS* services, running over the underlying execution environment.

7.2 Deployment optimizer component on TOREADOR Platform

The implemented component will be started by the API platform after the workflow definition phase. Once the deployment optimizer process is completed, workflow execution can be started with the sub optimal configuration found. This scenario is described by the following sequence diagram.

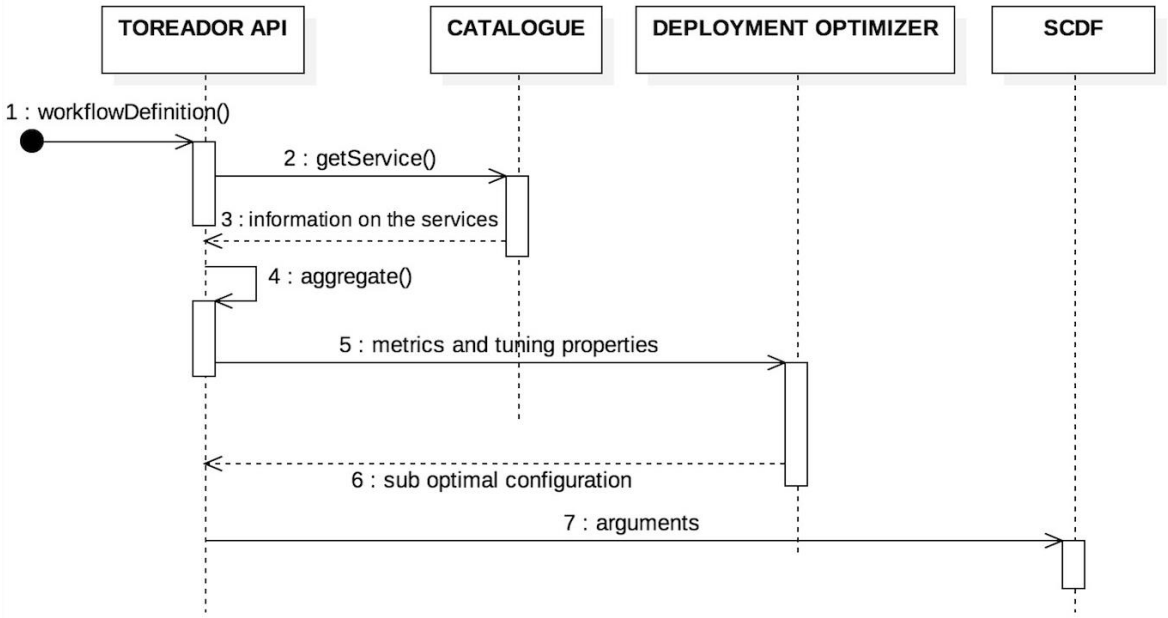

Figure 36. Sequence diagram: deployment optimizer on TOREADOR workflow

Because of the considerable processing time of the optimizer, all calls are asynchronously. The definition of the workflow specifies a series of platform services that will be executed by the client. For each service, present in the services catalogue, are associated a series of metrics and tuning properties. Once the information about the services to be optimized is collected, an aggregation of the tuning properties is made. The aggregation will allow to group the tuning properties of the service pipe to be optimized per framework and associated workload. Thus, for all the services in the workflow definition associated with the same framework and workload, merge of the tuning properties and metrics will be done, running an instance of the deployment optimizer with the resulting aggregation for each framework-workload pair.

Finally, the configurations will be passed as arguments to the Spring Cloud Data Flow (SCDF) [26] service provider. Spring Cloud Data Flow is a cloud native programming and operating model for compassable data microservices on modern runtimes. With Spring Cloud Data Flow, developers can create and orchestrate data pipelines for common use cases such as data ingest, real-time analytics, and data sinking.

Chapter 8
Exploitation

ENGINEERING has the mission of developing new solutions and promoting research outcomes to business units. In particular, the TOREADOR platform and its deployment optimizer are of interest of the Industry business unit, which could integrate them in the offerings or in new solutions to convey them to several existing (and potential) customers.
In the manufacturing sector, considered the main business domain where project results can be exploited, ENGINEERING has 200+ customers in Italy and around the world, requesting new ICT solutions supporting them to increase yields and reduce costs.
In Defence & Security context the TOREADOR approach will enable advanced Big Data analysis and smart visualization taking into account data protection and securely data sharing for forensic prosecution and preventive investigations.
Therefore, the results of this thesis could be integrated in the solutions developed and deployed by ENG in several real world scenarios, exploiting the enabling technology in other domains in which huge amount of data are expected to be produced and analysed, ranging from utilities/DSO in smart grids, to water distribution, to real time control of device in transport (railways) infrastructures, and so forth and so on.

Chapter 9
Conclusions and Future Works

In this thesis work, a general method for the auto-tuning of the configuration parameters for several big data frameworks such as Spark, Cassandra and Kafka was described, in order to optimize multiple metrics based on machine learning models. The proposed method stems from the extension of the work of G. Wang et al. [14] for the optimization of a single metric on the Spark framework. The method was then implemented on the Maven environment, focusing on the Spark framework for the optimization of metrics related to big data analytics applications. The main extension proposed, compared to the original method, was to be able to optimize multiple metrics simultaneously through a system of aggregation and the use of multi-task learning (MTL).

Finally, a possible integration into the TOREADOR platform was proposed and described through a sequence diagram. TOREADOR is a project co-founded by EU, including Engineering - Ingegneria Informatica Spa as development partner. TOREADOR takes a model-based BDA-as-a-service (MBDAaaS) approach, providing models of the entire data processing process and of its artefacts. The platform does high usage of the Spark framework, and in particular of the MLlib library providing all the tools for the processing machine learning algorithms.

In general, the generation of a sub-optimal configuration is performed through a series of stages. The initial stages aim to generate a dataset labelled with the metrics detected through the execution of a specific workload with random configurations, but specified within a range of values, and with the default configuration. To filter the configurations that will be used for the training of the multiple classifier, we used an aggregation function to bring us back to having a single binary filter. The selected aggregation of binary classifiers outputs, produces the maximum number of occurrences between 0 and 1. In case of a tie, 0 is returned. Configurations labelled with 1 are passed to the training of N multiple classifiers, or a single multiple classifier multi-task. For the multiple classifier the configurations are labelled based on the increase in performance in terms of percentages compared to the metric with the default configuration.

The deployment optimizer was then tested and validated to be integrated into the TOREADOR platform. Thus a specific workload was implemented that generalized the most used machine learning algorithms. In particular, the gradient descent workload makes

low-level derivatives on a numerical input dataset. The models used for the testing phase were the Random Forest for the optimization of a single metric, while for the optimization of multiple metrics the artificial neural network model with a multi-task approach were also taken into account, using the TensorFlow software library. The validation and testing phase was performed through the so-called error analysis, in order to avoid the main problems related to model training such as over-fitting and under-fitting. For validation, datasets from two different distributions were used. The first distribution is obtained from the execution of the gradient descent workload. The second is obtained from the execution of a workload that does not perform basic operations, but does the training of three machine learning models using Spark's ML library: k-means, Naive Bayes and decision tree.

By generating two independent models, it was possible to perform the validation separately, in order to find the best hyper-parameters for each model. The best results for the optimization of the "execution time" metric on the test dataset have been obtained by setting as hyper-parameters of the Random Forest model a number of trees equal to 10 for both classifiers, while a single training step for the binary classifier and 50 steps for the multiple classifier. The accuracy obtained is 83% for the binary classifier, while 77% for the multiple classifier. Regarding the results obtained for the optimization of two metrics, the consumption of CPU and RAM, the use of Random Forest models has led to better results of the single multi-task neural network. This may be due mainly to factors related to the type of input data that the Random Forest model can easily manage and generalize thanks to its binary tree structure; or it can also be linked to an insufficient number of training data for the neural network. In terms of aggregate accuracy on the test, however, the situation has worsened, with a 55% accuracy for the binary classifier, while 67% for the multiple classifier. The choice to continue to explore the multi-task and neural network models should therefore not be excluded, because there is still much scope for improving through the use of optimization techniques and other models.

The sub-optimal configuration found for both metrics highlight the fact that Spark is mostly a data parallelism engine and its parallelism is achieved mostly through representing the data as RDDs. Thus the optimization of metrics for big data analytics applications is achieved through the tuning of the resource allocation and parallelism, fully exploiting the potential of Spark. Consequently, the proposed component can be used automatically and transparently by the TOREADOR platform with respect to the end user, taking full advantage of the available cluster resources, and therefore the Spark engine, in order to optimize metrics such as execution time, and/or other consumption-related

metrics. All of these are benefits for the end user, especially in a BDA-as-a-service model, where there is often a pay-per-use business model based on resource and time consumption.

Among the possible future works for the deployment optimizer implemented there is its extension for the auto-tuning of deployment properties for other frameworks such as Cassandra and Kafka. These frameworks present the same problem as the Spark framework on the performance optimization through a considerable amount of properties that can be tuned. To achieve this goal, it would be enough to extend only the workload execution stage, defining and adding workloads that operate on these target frameworks. This is in fact the only stage of the deployment optimizer that operates differently depending on the type of the application and framework on which it is launched, returning a dataset labelled with metrics to be optimized collected with some metrics detection tool.
For the detection of metrics on Spark, the SparkOscope tool was used, developed by IBM researchers, as extension of the Spark Web UI. A proposed alternative would be to use directly from this tool the metrics detected on MQTT (a publish-subscribe lightweight messaging protocol), in order to have an improved event capturing compared to writing metrics on HDFS and therefore having more accurate metrics.
Another future work could concern the definition of priorities on the metrics to be optimized, through a static or dynamic choice given by the user, for example on the basis of a service level agreement (SLA). In this way, during the search phase of the sub-optimal configuration, by giving different weights to the outputs of the multiple classifier, the optimization is shifted accordingly, giving more importance to certain metrics than others.
Finally, the exploration of other algorithms for the multi-task learning (MTL) offered by software libraries such as Scikit-learn, in order to find a model of machine learning more suited to the optimization of the required metrics.

Bibliography

[1] Microservices architecture
[2] Apache Spark
[3] Apache Spark - Cluster Mode Overview
[4] Apache Spark - RDD
[5] Apache Spark - Spark Streaming Programming Guide
[6] Apache Spark - Tuning Spark
[7] Apache Spark - Spark Configuration
[8] Codingjam - Introduzione a Cassandra
[9] Apache Cassandra – Architecture in brief
[10] Cassandra Configuration File
[11] Multi-task learning with TensorFlow
[12] Apache Kafka
[13] Cloudera - Apache Kafka for Performance and Resource Management
[14] Guolu Wang, Jungang Xu1, Ben He, "A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning", Sydney, NSW, Australia, High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on, 2016.
[15] H. Herodotou, H. Lim, et al., "Starfish: a self-tuning system for big data analytics", Proc. of the 5th Biennial Conference on Innovative Data Systems Research (CIDR 2011), Asilomar, USA, January 9-12, 2011: 261-272.
[16] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for large-scale network parameter configuration", Proc. of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS 2003), San Diego, USA, June 9-14, 2003: 196-205.
[17] Dana Van Aken, Andrew Pavlo , Geoffrey J. Gordon , Bohan Zhang, "Automatic Database Management System Tuning Through Large-scale Machine Learning", Proceedings of the 2017 ACM International Conference on Management of Data, May 14-19, 2017, Chicago, Illinois, USA.
[18] Sidhanta, Subhajit and Golab, Wojciech and Mukhopadhyay, Supratik and Basu, Saikat, "OptCon: An Adaptable SLA-Aware Consistency Tuning Framework for Quorum-based Stores", 2016.
[19] W. Gao, Y. Zhu, Z. Jia, et al.,"Bigdatabench: a big data benchmark suite from web search engines," Proc. of the the 40th International Symposium on Computer Architecture (ISCA 2013), Tel-Aviv,June 23-27, 2013:1307-1320.
[20] Apache Spark - Monitoring and Instrumentation
[21] Mastering Apache Spark - Spark History Server
[22] TensorFlow
[23] Wikipedia - Random Forest model
[24] Towards Data Science - The Random Forest Algorithm
[25] TensorFlow - SavedModelBuilder
[26] Spring Cloud Data Flow

[27] GitHub - SparkOscope
[28] Digital Trends - Artificial Neural Network
[29] TOREADOR Project