Survival Analysis with Apache Spark and Apache SystemML on Stackexchange

Mateo Álvarez Calvo

August 13, 2017

Contents

List of Figures

List of Tables

1 Introduction & main goals

Many studies have been done over the Stackexchange community [Mamykina, Manoim et al 2011], one of the biggest Q&A sites in the world. The present is yet another study over the data of the famous site, but in this case, the study has two particularities, the use of Apache Spark with the library of Apache SystemML for the processing in a parallel environment, and the use of Survival Analysis to analyze the impact of the variables in the time an answer is accepted for each question, the "survival of each question" in the community.

1.1 Main technologies

As one of the biggest Q&A communities, Stackexchange has large amount of data of each interaction. Stackexchange is separated in several communities, regarding different topics. These communities can be small, as *DevOps* and *InternetOfThings* or really big, as *AskUbuntu* and *Stackoverflow*, the main developers community. This particularity makes necessary the use of technologies prepared to process large amounts of data, in the later case.

The purpose of the present study is to analyze a medium size community, so a distributed processing technology has to be used. For this purpose, Apache Spark, the latest distributed open-source processing technology, has been chosen to parallelize the operations on the data.

Spark ML is the machine learning library of spark, which contains the ML algorithms. Although it includes some Survival Analysis algorithms, all of them are parametric models, which require to specify a hazard function shape in order to be used. This rests flexibility to the models, and for this study the hazard function is not known, so it is wise to start exploring the data with a non-parametric model, KM estimates, for example, and then apply some semi-parametric model, in this case Cox Proportional Hazards model. The absence of non or semi parametric models in Spark ML gives an excuse to use SystemML, a machine learning library developed by IBM and recently adopted as an Apache Foundation project, which has non, semi and parametric algorithms for survival analysis, and is compatible with distributed processing frameworks as Spark or Hadoop.

Regarding the development environment, Jupyter Notebook provides a simple and flexible interface for this analysis, and can also be integrated with Spark, allowing the complete development in just one environment.

1.1.1 Survival Analysis

Survival Analysis is a group of ML models used to predict the time passed until the occurrence of an event. Is a method widely used, specially in the medical and pharmaceutical environments, where the prediction of time until an event is frequently used.

In this case, the objective of using these techniques is to understand the behavior of the variables with the time and to predict when will a posted question be answered. This idea has multiple uses, such as optimizations of the questions themselves, hour of the day, tags added, reputation of the user... or using for example StackOverflow as technical support for problems with software instead of paying the provider's technical support service.

1.1.2 Apache Spark

Apache Spark is a distributed processing technology developed in Scala by Databricks that represents the next step of Apache Hadoop. It includes the best parts of it, such as the Hadoop File System, but under a complete new paradigm that allows operations different from the famous map-reduce, using RAM as storage for results rather than writing to disk, lazy and optimized execution of tasks, and special focus on Machine Learning and SQL-like language, to mention some of the main features.

The use of a distributed processing framework is not strictly necessary in this case, as the community to be analyzed is not that big, but is a good starting point to check the use of SystemML and Spark to make later analysis on a bigger network.

1.1.3 Apache SystemML

Recently included in the Apache Foundation Incubating program, Apache SystemML is a machine learning framework that works in different modes, on both distributed frameworks, Spark or Hadoop, or stand alone mode, written in Java.

This framework provides a language to implement distributed, optimized algorithms ready for big data in a high-level language syntax, for Python and R. Additionally, the framework provides a set of commonly used algorithms already implemented with the sintaxis.

System ML can be executed in a variety of distributed and non distributed modes, with it's standalone mode, and the integration with Hadoop, and Spark via SystemML context, which allows the interaction through Scala, Python and R.

1.2 The Stackexchange data

Stack Exchange is a network of Q&A websites created in 2009 after the great success of *Stack Overflow* in 2008, a Q&A community website for computer programming.

Every question and answer, an all the contents of the communities are licensed under a *Creative Commons Attribution-ShareAlike 3.0 Unported*, so the knowledge is free to be shared with others.

Each community covers a different topic, from physics to software, and is structured in a reputation award format, each user's question and answer can be voted positively or negatively. This feature allows the self administration of the communities, which makes posible the existence of the network, as it is so big that an administrator or moderator could not manage. Whenever moderation is needed, for example when there is an argument, there is a specific place on each community to solve these problems, the Meta section, where the users post settle the disputes to be solved by administrators of the site. The reputation system works as gamification, giving users privileges and functionalities when they earn experience points.

All these communities generate large amounts of data that Stackexchange facilitates every once in a while for data scientists and people in general to download and analyze. The data is available in a torrent file and each package has about 35 - 40 GB of compressed information.

This compressed file has data from different communities for a certain period of time. In this case, the analysis is done over the Scifi community, which is a median size community for science-fiction Q&A.

1.2.1 Scifi community

Scifi is a community in Stack Exchange that focuses on science fiction and fantasy. This community was selected because it has a medium size which is perfect to test the mentioned technologies in a reasonable period of time. Selecting just the data from Scifi community from the big compressed file, it weights around 110 MB in a 7z compressed format. This allows the computation on a local machine for experimentation and then scale the problem to a bigger community such as *Ask Ubuntu* or *Stack Overflow* when the process is refined and it can be launched remotely in a cluster. The data is divided in 8 files, and has the same structure for every community:

Further details about the relational database structure is explained below, the objective is to show the variables obtained from the dataset so that the later variable selection is understood.

File	Description	Size
Votes.xml	Voting results for each question and answer	84,1 MB
Tags.xml	Relational table for tags on each question	169 KB
Users.xml	Users on the net	16,7 MB
PostLinks.xml	links to posts	1,5 MB
Posts.xml	List of all questions	137,3 MB
PostHistory.xml	All interactions of each post	268,6 MB
Comments.xml	List of all comments of each post	66,3 MB
Badges.xml	All users' badges	16,1 MB

Table 1: List of files from the compressed Scifi folder

1.2.2 Votes.xml

This file contains information about votes of the users to each question. The file has the following structure:

Feature	Data type	Description
Id	Integer	Unique vote identifier
PostId	Integer	Foreign key that indicates the post that was voted
VoteTypeId	Integer	Type of vote, 1 for Downvote and 2 for Upvote
CreationDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time of votation

Table 2: Votes table

1.2.3 Tags.xml

This file contains all tags and the posts that contains them. The file has the following structure:

Feature	Data type	Description
Id	Integer	Unique tag identifier
TagName	Text	Name of the tag
Count	Integer	Number of times used
ExcerptPostId	Integer	
WikiPostId	Integer	

Table 3: Tags table

1.2.4 Users.xml

This file contains information about users. The file has the following structure:

Feature	Data type	Description
Id	Integer	Unique user identifier
Reputation	Integer	Reputation level of the user
CreationDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	User creation date
DisplayName	Text	Alias to display on question
LastAccessDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Last login date
WebsiteUrl	Text	Site where the user signed up to
Location	Text	Location of the user
AboutMe	Text	Information user provided
Views	Integer	User views count
UpVotes	Integer	User up votes count
DownVotes	Integer	User down votes count
AccountIf	Integer	Unique user identifier

Table 4: Users table

1.2.5 PostLinks.xml

This file contains information about relation between posts. The file has the following structure:

Feature	Data type	Description
Id	Integer	Unique post links identifier
CreationDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Post links creation date
PostId	Integer	Post unique identifier
RelatedPostId	Integer	Unique identifier of the post related to the PostId
LinkTypeId	Integer	Type of relation between posts

Table 5: Post links table

1.2.6 Badges.xml

This file contains information about the badges the user has obtained.

Feature	Data type	Description
Id	Integer	Unique Badge identifier
UserId	Integer	Unique identifier of the user who obtained the badge
Name	Text	Name of the badge
Date	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time the user obtained the badge in extended format
Class	Integer	Type of the badge
TagBased	Boolean	Whether the badge is based on a tag or not

Table 6: Badges table

1.2.7 Posts.xml

This file contains all questions posted along with the accepted answers and other info related to the time and user who posted the question. The file has the following structure:

Feature	Data type	Description
Id	Integer	Unique question identifier
PostTypeId	Integer	Type of post codified as integer
CreationDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time of question creation in extended format
Score	Integer	Question's score, calculated from the users' votes
ViewCount	Integer	Count of all visualizations of the question
Body	Text	The question itself, in utf8 format
OwnerUserId	Integer	Id of the user who posted the question
LastEditorUserId	Integer	
LastEditDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Last edition date
LastActivityDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Last interaction with the question time
Title	Text	Title of the question
Tags	Text	Tags added to the question
AnswerCount	Integer	Number of answers to the question
CommentCount	Integer	Number of comments to the question posted
FavoriteCount	Integer	Number of times the question has been added to favorite by another user
ClosedDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time the question has been closed
CommunityOwnedDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time

Table 7: Posts table

1.2.8 PostHistory.xml

This file contains information about the interactions with each post. The file has the following structure:

Feature	Data type	Description
Id	Integer	Unique interaction identifier
PostHistoryTypeId	Integer	Type of interaction with the post ()
PostId	Integer	Unique identifier of the post this interaction is related to
RevisionGUID	Text	
CreationDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time of question creation in extended format
UserId	Integer	Unique identifier of the user that created the interaction with the post
Text	Text	Text the user introduced on the interaction

Table 8: Post history table

1.2.9 Comments.xml

This file contains the comments posted for every question created. The file has the following structure:

Feature	Data type	Description
Id	Integer	Unique comment identifier
PostId	Integer	Unique identifier of the post this comment is related to
Score	Integer	Total score of the comment
Text	Text	Comment text
CreationDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time of comment creation in extended format
UserId	Integer	Unique identifier of the user who posted the comment

Table 9: Comments table

1.3 Main Objectives

The main objective of this study is to test the scalability and integration of the proposed technologies, Spark, SystemML and Jupyter Notebook in the usecase of Stack Exchange communities, so further data analysis can be performed. This main goal is divided in three major objectives:

- Use Spark to make the data cleaning and create a script for further research on the Stackexchange site.
- Verify SystemML integration with Spark for further research and scalability.
- Use SystemML survival analysis algorithms to analyze Stackexchange's data and obtain conclusions on the main variables affecting the time taken by the community to answer each question.

2 Technologies

The downloaded data from Stackexchange for the analysis weights about 40 GB, which is enough amount to consider distributed processing. Going down to the distributed processing frameworks, Apache Spark was chosen.

2.1 Apache Spark

Apache Spark is a fast and general-purpose cluster computing framework, widely used for data processing. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL, a SQL-like language prepared for both SQL and NoSQL databases, MLlib and SparkML for machine learning and pipelines, GraphX for graph processing, and Spark Streaming.

This distributed data processing framework was initially developed at the University of California, Berkeley's AMPLab, and donated to the Apache Software Foundation in February 2014, the first release was on May 30th 2014.

Apache Hadoop presented some limitations that Apache Spark tried to solve:

- It is difficult to write most of the algorithms in a MapReduce form.
- It is very slow to write each iteration to disk, which, for example makes difficult to use Hadoop to process streaming.
- Apache Hadoop's support for iterative jobs and Machine Learning restricts it's use for this task.
- Apache Hadoop's SQL tools doesn't work well on complex queries, sometimes it doesn't work at all and other times it is quite slow as it writes on HDD each iteration.
- Streaming functionality is not supported

Some solutions Apache Spark provides to these problems are:

- Lazy computation, Spark only executes a set of tasks when a result is required. This gives the opportunity to optimize jobs before executing them, even at physical level the queries to the data are optimized.
- In-memory data caching, Spark scans HDD only once to read the input data and then uses RAM as much as it can, which is faster than scanning disk on every step.
- Specific Machine Learning libraries, Spark ML and Spark MLlib, including numerous algorithms prepared to run in a distributed mode.

- Spark SQL provides structured (SQL-like) query language for structured and not structured data in SQL or Dataframe API. One of the advantages of this library is the unified access to datastores with the same language, it even provides SQL functionality with streaming data. Other interesting advantages are at an optimization level, using Dataframe API, Spark can optimize operations and queries to the database, using the *Catalyst* optimizer.
- Spark Streaming library allows users to process streaming data using microbatches

2.1.1 Apache Spark Structure

Spark has a master-slave architecture and supports various resource managers: standalone, Mesos and YARN. The resource manager will only be in charge of identify the resources. Independently of the resource manager chosen, the Apache Spark architecture doesn't change.

The deployment of Spark has two variants, client and cluster mode. On the client mode, the Driver is launched on the machine the process has been invoked, and it can be inside or outside the cluster. On cluster mode, the cluster manager is assigned to control the Driver process, and the process itself will be launched inside the cluster. In case Mesos is the cluster manager, it will require an additional service.

Focusing on the Spark cluster mode[1], the architecture is as follows [2]:

All the Spark applications run on the cluster nodes as independent sets of processes, all coordinated by the main program, called *driver program*, that coordinates all the others through the *sparkContext*.

The driver program deploys executor programs on the worker nodes of the cluster via a resource manager, already installed and running on the cluster, which provides the resources needed for the execution of the jobs. The driver first converts the user program into tasks and after that it schedules the tasks on the executors.

The executor programs are in charge of running individual tasks in a given Spark job, all sent by the driver program through the sparkContext. They are launched at the beginning of a Spark application and typically run for the entire lifetime of an application. Once they have run the task they send the results to the driver. They also provide in-memory storage for RDDs that are cached by user programs through Block Manager.

Spark driver program has to be in continuous contact with the executors, as it has to coordinate the workflow and the specific tasks for each executor and monitors the status of them. Moreover, when a result is required by the user, and it is not saved to

[1]The architecture for client mode is the same, but running all the programs and processes in the same machine.

[2]As explained on the Official Spark documentation [2]

a datastore, the result will go back to the driver program. The driver can run in an external machine of the cluster, but, as it has to be in continuous communication with the executor programs, it has to be running all the time the executors are calculating and it can not be disconnected to the cluster.

More than one application can run in the same cluster, as long as there are enough resources. Each application gets its own executor processes and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs), all of them running in different java processes, in general in different machines. However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system.

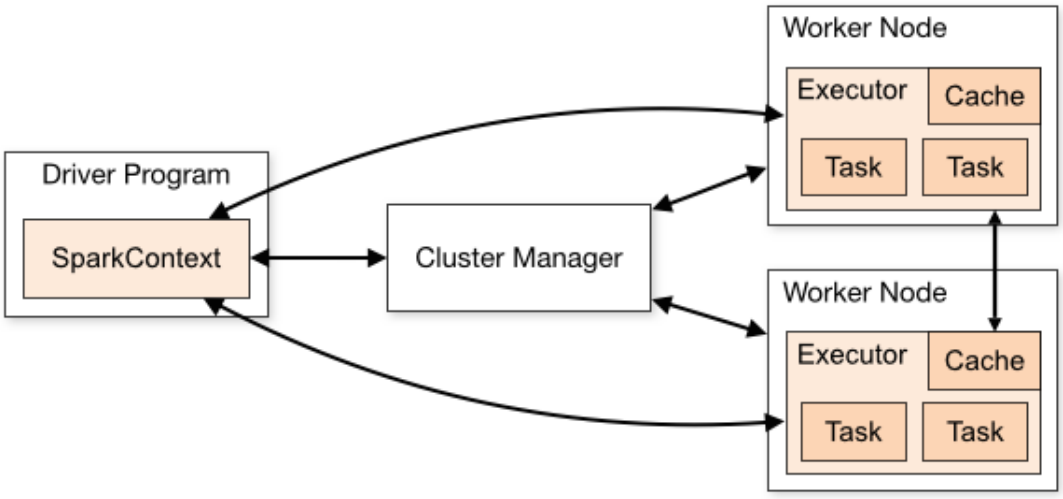

Figure 1: Spark Cluster Mode Architecture, from [2]

Apache Spark is formed by the Spark Core API, available in R, SQL, Python, Scala and Java languages, and built up on it four main libraries: Spark SQL + DataFrames, Spark Streaming, Spark MLlib, Spark ML and GraphX, that complements functionality for Spark, specially on the parts Hadoop failed, Machine Learning, SQL and Streaming. There are other libraries but these are the essential.

These libraries can be imported independently and combined to be used at the same time, for example, Spark SQL can be used in a Streaming environment with Spark Streaming library and this way SQL queries can be launched in

2.1.2 Apache Spark Data Structure

RDDs

Apache Spark started with just one data structure, the RDDs. The RDD responds to Resilient Distributed Dataset, and have the following properties:

- Resilient: an RDD can be computed again in case of failure
- Distributed: the RDD can be partitioned and distributed over nodes, to parallelize the works
- Immutable: an RDD can not be modified, instead, a transformation is applied and other RDD is generated
- Lazy: RDDs represent a result of a series of operations and transformations over data, but it does not trigger any operation
- Statically typed: the values in the RDD are typed

Dataframes

Dataframes are distributed collections structured in named columns, the idea is similar to the R dataframes. Dataframes are part of the Spark SQL API and are built up from RDDs, it is a higher level of data structure, which means that can take advantage of Catalyst to optimize the queries.

Datasets

Datasets are similar to dataframes, but also taking the static typing of the RDDs, combining the best of the two data structures.

This static typing allows datasets to use Tungsten, Spark's optimized memory engine. Tungsten has direct access to off heap memory, to provide even better optimization, and uses data types to minimize the encoding and decoding of the data.

Graphframes

Graphframes are structures used for graph storage and operations. Graphframes are composed by two dataframes, the first one contains all the vertices and the second one contains all the edges. This is useful as dataframes can be optimized by Catalyst. GraphFrames are not part of the Apache Spark core API, as there is still development to do.

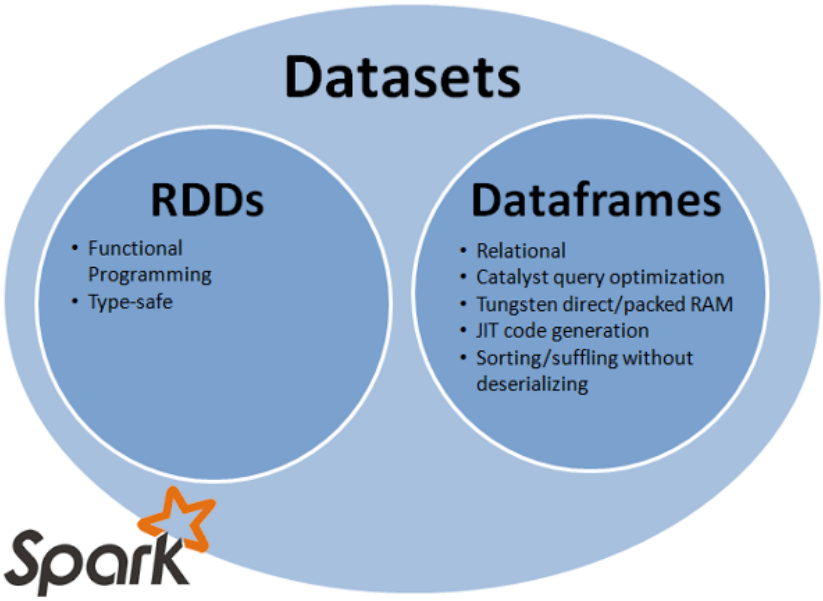

Figure 2: Differences between RDDs, dataframes and datasets, from [3]

Discretized Streams

Discretized Streams, or DStreams, are the basic data structure that Spark Streaming module uses for processing. This DStreams are essentially RDDs in time periods, and represent the data available on each time window. As Spark Streaming works with microbatches, small (or not so small) time intervals of data processing. For a time period, a DStream is basically an RDD, on which operations can be performed giving as a result another temporary RDD or DStream.

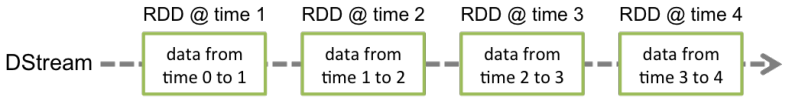

Figure 3: DStream structure, from [2]

Catalyst and Tungsten

As mentioned before, Spark has some specific tools to optimize the workflow when using dataframes and datasets, these optimizers focus both on I/O, writing and reading data, with Catalyst, and on the execution engine itself.

Catalyst is the query optimizer, included in the Spark SQL API. As the execution is not done until an action is called, for example a calculation of a result, Spark can analyze the code and choose the order of operations, which means that, for example, when querying to a database to apply filters, those filters can sometimes, if the database is relational, be applied natively in the database, moreover the sequence of subsequent filters is analyzed in order to reduce the I/O on the database, which can lead to a significant reduction of time and resources, as usually the data available is abundant, but the data used for a process is significantly less. Catalyst works in four phases:

- The *analysis phase* returns a logical plan where all the metadata from the data involved in the operation is known
- The *logical optimization phase* consists on the optimization of the operations performed over the data using ruled-based optimizations
- The *physical planning phase* uses cost-based models to select the best execution plan from the ones available after the logical optimization phase
- The *code generation phase* is the final phase, where Spark compiles parts of the query code to Java bytecode, this speeds up the process as there is no need to use the Scala compiler at runtime to generate bytecode.

Tungsten is the execution engine's optimizer has been developed due to the improvement in the I/O operations thanks to Catalyst, and is focused on the performance of the CPU and the memory, it is focused on three major fields:

- Memory Management and Binary processing: java uses objects, which have a large memory overhead, increasing the size of space occupied to store more simple variables. Apart from Java objects, JVM uses Java Garbage Collector, which manages object creation and destruction according to the life cycle of each object. This is a complex task, as the life cycle can not always be estimated precisely, which causes overhead on the memory, keeping short life cycle objects when they are not necessary and viceversa. Spark understands the data flow through the stages of computation, so it is posible to have a better optimizer than the JVM, for that purpose, Spark introduces an explicit memory manager that converts most operations to use binary data rather than using Java objects and Garbage Collector.
- Cache-aware computation: when computing large amounts of data on an in-memory processing framework, not all data is able to fit in the machine's memory, so information is written to disk. This and fetching data from the main memory are time consuming operations, so large fractions of CPU time are spent on gathering data to process. The solution Spark provides to avoid spending so much time waiting for data to travel from disk or main memory is to design "cache-friendly algorithms" that uses L1/L2/L3 CPU cache as they are orders of magnitude faster than main memory.

- Code generation: Spark dynamically generates bytecode to evaluate expressions, such as SQL expressions rather than using an intermediate interpreter. Using some specific data structures such as dataframes, built from RDDs is an advantage, as data types are already known and specific code can be generate to treat the serialization as there are more information available.

2.1.3 Spark Main APIs

Spark Core API

The core API represents the basic structure of Spark, it can be addressed from any of the supported languages, scala, python, R and java. This API has the main functionality of Spark, which includes a set of operations, that includes Map-Reduce, but is not limited to them, as Hadoop is. Regarding the data management, This API contains the RDDs, explained above and the basic operations.[][][]

Spark SQL + DataFrames

The SQL layer over data is known as Spark SQL. It allows users to use a SQL-like language in Spark programs, to query both structured and not structured databases (SQL & NoSQL), such as Postgres or MongoDB.

The Spark SQL library also provides a main functionality in Spark that is gaining more importance over the time, the Dataframes.

A Dataframe is a data structure introduced in the R programming language that has extended to the Data Science world as one of the most easy to use data structures. Dataframes in Spark are the same concept that in a non-distributed processing framework, but the implementation, as it is for a distributed environment is different, it is built from RDDs with a specific structure.

Spark MLlib + Spark ML

Spark has two main Machine Learning libraries, the first one, Spark MLlib, which is the basic library, that includes the main algorithms and is addressed with RDDs, the second one, Spark ML, which uses Spark MLlib but through DataFrames, and includes further functionality, such as Pipelines, a set of operations performed over data, that allows the user to build sequences of actions over Data Frames.

2.1.4 Spark Streaming

Spark Streaming is the real-time, streaming processing library of Spark. Unlike other streaming processing frameworks such as Apache Flink, Spark Streaming works with microbatches, remaining almost the same structure as Spark itself. The reason is to have the workers processing in time periods, this way the fault tolerance is more robust, as for each microbatch all the operations will be parallelized, and in case any worker falls down, the others complete the work. This way the latency is sacrificed in favor of robustness of fault tolerance.

Spark Streaming runs on top of Spark, so that the benefits of the distribution, scalability and fault-tolerance are inherited from it, but it also has the advantage of using a similar sintaxis and the interoperability with other Spark libraries such as Spark-SQL, giving the possibility to process information with dataframes, with all the advantages they provide.

Spark Streaming has support for numerous data stream sources, including Apache Kafka, ZeroMQ, TCP Socket or Flume, among others, this gives flexibility to the framework to connect to different applications.

The workflow in Spark Streaming is the following: data enters at unspecified rate, it can be a specific amount every second or not a constant quantity, Spark Streaming will be in charge of distributing the data along the workers. Data is transformed into DStreams and distributed at a low level into RDDs, using the Spark Core functionalities, processed, and then returned to the microbatch. At the end of the microbatch, the application will return the processed data in DStreams. The schema below is a diagram of the explained workflow.

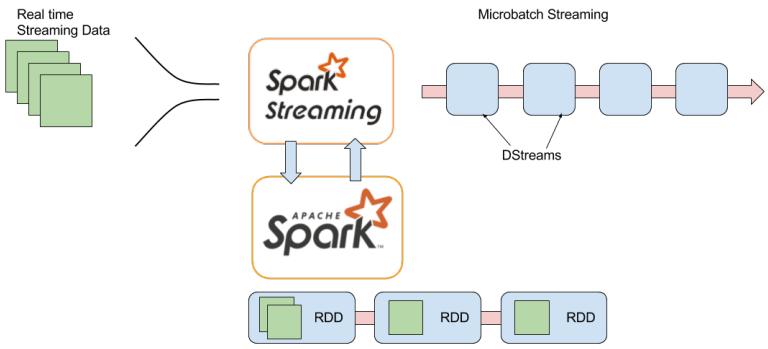

Figure 4: Spark Streaming workflow

Spark GraphX

GraphX is Spark's graph library, which extends Spark's core API with special functionality for graphs. As commented before, Spark GraphX uses a different data structure, *Graph*, a structure that provides functionality to introduce both vertices and edges for the graph. As all other libraries, the GraphX functionality can be mixed with all the rest of the framework, giving this library high flexibility.

Unlike other Spark main libraries, GraphX does not provide access with other language than Scala, it can not be addressed with R nor Python, but it provides an increasing amount of distributed algorithms for graph analysis.

2.1.5 Spark workflow

There are three main phases on the execution of a program in Spark: definition of a DAG from the user's code, translation of the DAG to an execution physical plan, and scheduling and executing the plan in the cluster.

Definition of a DAG

The program defined by the user is a result of a series of operations (transformations and actions) done to a series of RDDs, the first RDD may be the ingest of the data, and the last one, the result the user wants to calculate. This series of operations over RDDs form a graph, an ordered sequence that leads to the final result, therefore the graph is directed, from the first RDD and operation to the last one, and is acyclic, it has a beginning and an end. This DAG is the logical representation of the execution of operations over RDDs and their partitions.

On a lower level, the RDDs are the ones that contain the logical plan under the DAG, each RDD has one or more pointers to one or more parents, along with metadata about the relationship they maintain. These pointers allow an RDD to be obtained through it's ancestors, for example to be recalculated in case of a node failure.

Creation of physical plan

When executing, the DAG is translated to a physical plan which merges multiple operations into tasks. This execution is called whenever an action is called, this execution takes the DAG, looks to the latest RDD and goes backwards to the first ancestor to construct the operations needed. The output of this process is a *job*, composed by a number of *stages*, whose number depend on the operations performed over the RDDs, and *tasks* in every stage.

Formally, a task is a unit of execution that runs on a single machine, tasks group to form stages, which represent the operations performed over a partitioned data in a parallelized way, namely a stage is a group of tasks that will perform the same operation over a partitioned data.

The number of stages created depend on the number of repartitions done to the data to obtain the final RDD, every time a shuffle operation is done to repartition the data over the machines, a new stage is created, for this reason, there can be less stages than group of parallelized tasks.

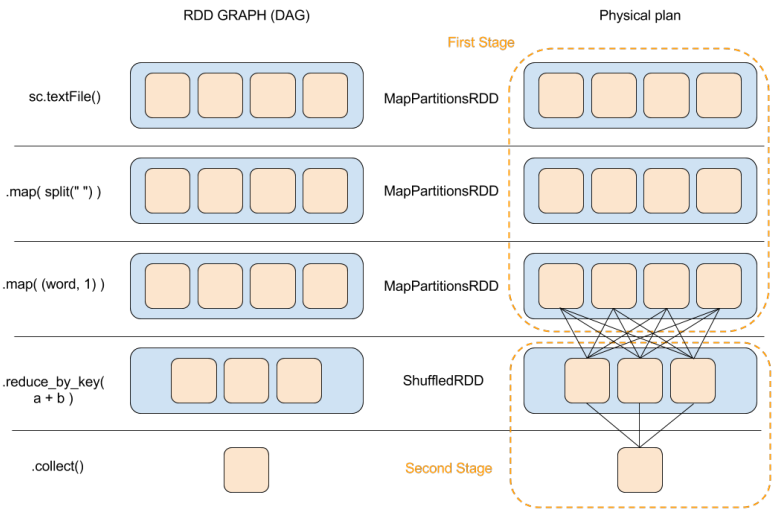

Figure 5: Example of translation of DAG to physical plan, for a word count application

It is interesting to note that, when using Spark SQL, the physical plan will be an optimized DAG, as shown before, using Tungsten and Catalyst.

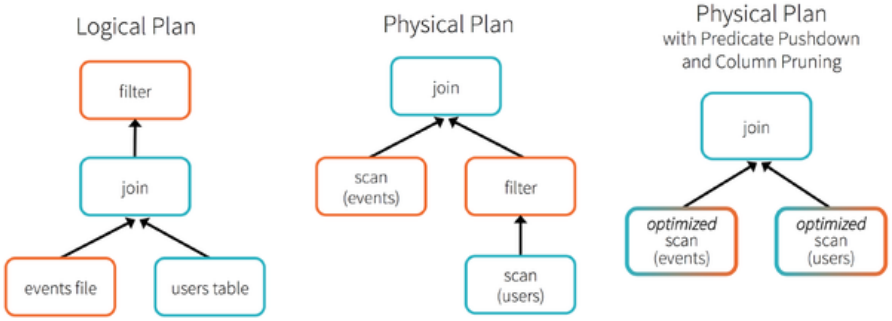

Figure 6: Example of an optimized physical plan from a DAG, using Spark SQL, from [7]

Scheduling and executing the physical plan on the cluster

Finally, the stages are executed in order, launching the tasks over the available nodes to compute the resulting RDD. As the execution runs in-memory, whenever a task fails, the entire sequence of operations of the stage has to be computed for the particular lost partition, but not all the tasks for all partitions. For this reason it is common to *cache* the RDDs after a series of operations, avoiding this way to recalculate all the previous steps.

2.1.6 Spark UI

As seen in the sections above, there is a lot of information of the running process of a Spark application. To help the users monitor the application, configuration and the status of the cluster, Spark has a web user interface, in which all the information, the DAG, the physical plans, the storage with the cached data, the environment variables, the executors available and the SQL options.

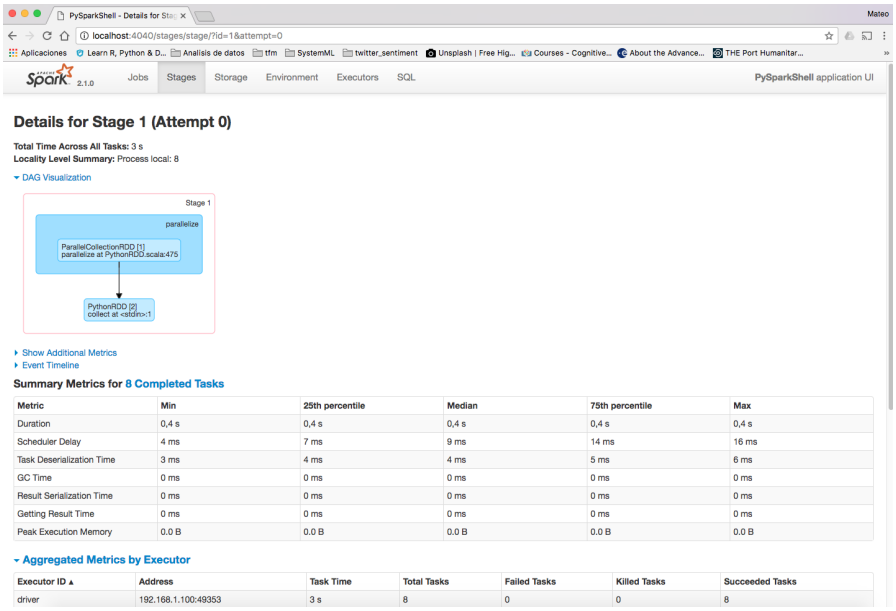

Figure 7: Example of Spark UI

2.2 Apache SystemML

Developed by IBM, SystemML started in 2007 as multiple projects involving machine learning with Hadoop, which in 2009 resulted in a single team dedicated to scalable machine learning research. Through 2009 and 2010, the team observed how clients developed machine learning algorithms, the workflow was (and still is) the following:

First, a data scientist, working in a single PC, and developing in R or Python, used small amount of data to create a ML algorithm, this part of the process works fine, as the algorithms use small data, and the researchers can iterate fast to refine the algorithms. When the algorithm is ready, the data scientist gives it to a systems programmer who implements the algorithm in an optimized way with low level APIs, usually with a distributed processing framework, such as Spark or Hadoop. When the implementation is ready, the distributed algorithm is tested with big data and then the results return to the data scientist to verify the correct implementation of the distributed algorithm.

The later part of the process leads to two major problems, the first one is the time spent for each iteration, which can be large, depending on the complexity of the process itself, and the second one is that during the reimplementation of the algorithm in the distributed framework, mistakes can be made, which leads to different results in big and small data, and to the depuration of the distributed code, a time consuming process.

The objective of Apache SystemML is to attack these two problems, providing the data scientist with a interface in a Python or R like language that optimizes the code to run in a distributed environment, this way, the exact same code is executed in small and big data and the data scientist can verify the behavior of the algorithm on both contexts. The optimization of the code is done by translating this high-level language code written in R or Python to a scalable executable that can run on Spark (or Hadoop), with SystemML compiler and runtime.

The project went open source on 2015 and entered Apache incubation in November 2015, with the first open-source binary release announced on January 2016. The latest release is from April 19^{th} 2017, with the version *v0.14.0-incubating-rc4*.

Figure 8: Apache SystemML workflow

2.2.1 SystemML Architecture

Apache SystemML's architecture involves optimizers that converts code from the high level programming languages to specific code for Spark or Hadoop, for using a distributed infrastructure, or optimized single-node code, when the resources of one machine are enough for the process. All of this is done through three different stages, each one composed by different steps:

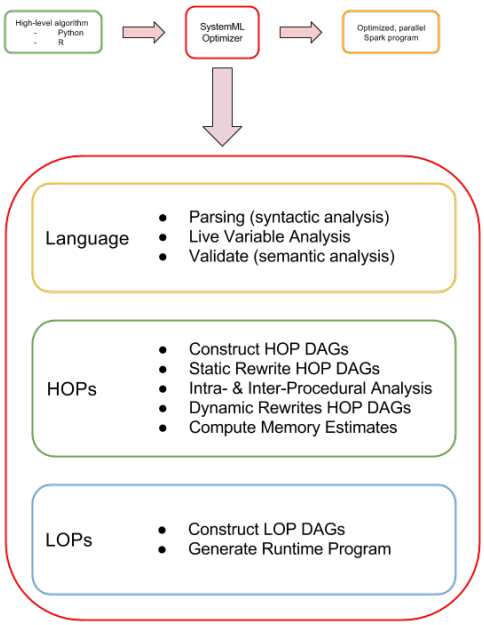

Figure 9: SystemML Architecture

Language level

The first part is the *language processor*, this piece is in charge of interpret the code written on R and Python and optimize the execution, converting complex and resource consuming operations to more optimized code. This piece is divided in three steps:

The first step is to convert each operation into a hierarchical representation of statement blocks and statements, which means that the code written by the user is processed and understood by the optimizer, in order to be able to identify the resource and time consuming operations. This step analyzes the code not only lexically and syntactically, but also the order of operations and other basic aspects. The output of this step is the input code translated into the specific DML grammar.

Once the operations are translated into the DML grammar, variables involved are ana-

lyzed, regarding, for example the input and output variables, their size in memory, and the data flow.

The last step of the language level is to verify the correctness of the code obtained, checking dimensions of variables, necessary variables and parameters for operations among other things. This process is done over the whole program and validates expressions and code blocks, but also simulates the code execution in order to evaluate the resources necessary, analyzing the conditionals and loops, given that the optimizer already knows the size of the variables, as it has been checked on the first step.

Hight-level Operators (HOPs)

This phase of the optimization consists on structuring the operations in order to be able to rewrite the code in a more optimized way, for example changing the order of operations when it implies less resource consumption. To do so, from each basic block of statements, a high-level operators DAG is created, where the nodes represent operations and their outputs, and edges represent data dependencies between operations.

As a result of this process, a single operator tree is built with all the statements and operators of the code. This tree represents a data flow graph is then used to build HOP DAG rewrites. These rewrites are size-independent transformations, for example format conversions or algebraic simplifications.

After this rewrite phase, a simulation of the size of the variables and operations is performed, this way, the program is able to estimate if the execution of all the code is posible or adequated in one machine, otherwise it will be distributed over the machines available.

The fourth step of this phase is to apply dynamic HOP DAG rewrites, this rewrites comprise simplifications done over the operations when the size is enough and cost-based rewrites, for example, the order of operations in matrix algebra, or the transposition of some matrixes over others depending on the size of them.

Low-level Operators (LOPs)

The last part of the optimizer focuses on low-level operations, generating again DAGs, representing operations in the nodes and data dependencies on the edges. These operations are optimized at a physical level, taking into account the specific backend and resources available in which the program will run, focusing specially on MR, instructions run in a Map-Reduce paradigm and CP (Control Program), operations executed in-memory.

At the end of this phase, the main program is compiled and executables are generated for the specific backend used, wether it is in a distributed or a single node execution.

Runtime-Level

Apart from the work done by the optimizer in the stages previous to the execution of the code, it will also be available during the execution of the program. This way, the optimizer can analyze the real time execution and re-compile parts of the code. This is specially important when the execution is in a distributed environment and the optimizer have not had access to the data sizes, which will be available during runtime, giving the opportunity to make live changes and re-compilations of the code.

The use of the optimizer does add an overhead to the execution of the program, typically around $200ms$ per script on the language-level, about $10ms$ per DAG on the HOP-level and $< 1ms$ on the recompilation, including LOP-level.

2.2.2 SystemML Algorithms

Algorithms catalog for systemml, implementation of distributed cox model and kaplan meier

2.3 Reproducible research with Python, Scala and Jupyter

Regarding the selection of the development environment, it was important to use a standardized one so that the analysis could be reproduced by anyone. Jupyter notebook is one of the most commonly used, specially in the education and investigation institutions. Jupyter Notebook is easy to use and configure, and flexible, as different kernels (code interpreters) can be configured, even for the same language with different set of libraries.It can be easily integrated with Big Data technologies, such as databases or Spark itself.

Jupyter Notebook is an open-source web application that contains code interpreters and other functionalities that allow users to develop code and write and share documents. It works with a dedicated file type, the .ipynb, notebooks on which the user can write both code and text in a cell distribution, cells that can be executed independently, having, as a result, an interactive shell for the selected code interpreter. It has interactive interpreters for many languages, including python, ruby, scala, R, etc, as well as markdown interpreters.

Another important advantage of Jupyter Notebook is its simple integration with Spark, it can be launched directly from python via *findspark* library, launching an embebed interactive Spark Shell, to execute the Spark code directly from the cells of the notebook.

To configure the same environment as the one used in this study, some steps have to be followed:

2.3.1 Environment setup

The versions of the software used for this study are listed below:

Technology	Version
Jupyter Notebook	4.3.1
Python	3.5
Scala	2.11
Toree kernel	0.2.0.dev1
Spark	2.1
SystemML	0.12.0

Table 10: Technologies and versions used

The first step is to setup Jupyter Notebook, either running it in a Docker container or installing it directly on the machine. The docker image can be obtained entering the following command in a shell: *docker pull jupyter/notebook*. Regarding the other option, installing it, the instructions can be found in the following link: *http://jupyter.readthedocs.io/en/latest/install.*

Once Jupyter Notebook is running, the kernels have to be configured. In this study, Scala and Python were used for the data processing, so both kernels were configured. The python kernel is usually configured, as it comes with the IPython kernel installation, if not, follow the guide[][][][][][]. To install the scala kernel, several options can be considered, as there are several implementations of the scala kernel. The chosen one was Scala Toree.

Apart from the kernels, some dependencies must be installed to do the data processing in python, those dependencies are:

- appnope==0.1.0
- bokeh==0.12.4
- botocore==1.5.43
- findspark==1.1.0
- ipykernel==4.5.2
- ipython==5.3.0
- ipython-genutils==0.1.0
- matplotlib==2.0.0
- notebook==4.3.1
- numpy==1.12.0
- pandas==0.19.2

- py4j==0.10.4
- pyparsing==2.2.0
- python-dateutil==2.6.0
- scipy==0.18.1
- systemml==0.14.0
- toree==0.2.0.dev1
- traitlets==4.3.2

For the python kernel, the SystemML library has to be installed, instructions for the installation can be found in the Apache SystemML's get started documentation: https://systemml.apache.org/systemml.html

3 Infrastructure and resources

As commented before, selecting an environment to be used is the first step, and it is a essential part.

3.1 Architecture scheme

The whole study has been executed in a standalone model, with a MacBookPro Retina 2015. The data was stored in an HDFS

3.2 Configuration

3.3 Workflow

The first step is to clean the data and make the feature selection. One the data is clean, it is used as the input for the models training, both Kaplan-Meier Estimator model and Cox Proportional Hazards Model.

3.3.1 Data cleaning

The data provided by StackOverflow comes in a reasonably clean xml set of files, which were previously described, on section 1.2. From these files, the Scyfy Community directory has been chosen for the analysis. This is divided on the files described above.

As the input format for the survival analysis methods are quite specific, there is some formatting to do joining and combining the tables from the files:

4 Results

4.1 Survival Analysis

Survival analysis is a set of different techniques and algorithms used to estimate the time passed until the occurrence of an event. All of these methods are based on the conditional probability of an event occurring in a certain period of time, usually called *Hazard Rate*. This basic idea can be applied to numerous environments, such as time of failure of a component in the industry, time of occurrence of an event in economics, and others, but the main field of application of these methodologies is the medical, where the time until some event is usually the main point of interest.

The survival analysis techniques provide with important features other regression methods does not. The main difference between other regression models and survival analysis is the importance of the time in which the events take place, which adds information not only on whether the event has occurred or not, but also the moment it happened. This crucial feature makes possible to take into account the data censoring, that will be explained below, on section 4.2.1, and the comparison of survival between different groups, also analyzing the relationship between the covariates and the survival time.

As in every statistic methodology, there are dependent and independent variables, usually called covariates. In this case, the dependent variable is, as mentioned before, the hazard rate, the conditional probability of an event occurring in a certain period of time giving survival up to this point, but the analysis is not only restricted to assess the effect of the covariates in the time to an event, but also the impact of these variables on the hazard rate.

Among the collection of methods included under Survival Analysis, three groups have to be taken into account: non-, semi- and parametric models. This classification responds to the assumption about the shape of the *hazard function*.

The non-parametric models are simple and fast models used for the initial estimate of the hazard rate, usually applied for initial analysis on groups thanks to their simplicity, although they do not accept the inclusion of covariates and only provide the hazard rate as function of the time via probabilistic estimation on the training subjects.

The semi-parametric methods are more flexible and complex, but have an important advantage over the parametric models, which is that they do no require a specified baseline hazard function before application.

Finally there are parametric models, which make an assumption over the form the hazard rate takes, making them less flexible, as the function form is already defined. Among all the survival analysis methods two have special interest:

The first one is the Kaplan-Meier model, which is a simple, Non-parametric model used for simple and fast analysis, and the representation of the survivor curve. This method

in particular, along with the *Life Table* are classic methods useful for a fast but imprecise analysis of the data. Kaplan-Meier's use is more extended although it is more suitable for small samples.

On the other hand, Cox Proportional-hazards regression is a more complex model, introduced in 1972 by Sir David Cox, in the paper *Regression Models and Life Tables*, which is nowadays one of the 100 most important papers in all of science, as it introduces key innovations in the field.

4.2 Used data

The data used for the analysis has already been presented on section 1.2, so the point of view of this section is to present the input data format necessary for the *Survival Analysis* methods and how has the study's data transformed for it.

4.2.1 Data censoring and truncation

Data censoring refers to a situation where some data of the instance is known but some event times are not known, for example when the event occurs after the end of the observation period. Data truncation refers to the lack of information of some variables outside of the time period considered in the study.

There are different situations of data censoring, right, left and interval censoring. *Right censoring* refers to the situation when the event doesn't occur during the observation period and occurs time after the end of the observation time interval. On the other hand, *left censoring* occurs when the subjects of the study have already experienced the event when the study starts, but it is not clear exactly when. *Interval censoring* refers to the case when the event has occurred during the interval but the information about the time it happened is not available.

Truncation is when there is a period of time when there is no information, inside the observation period. There are also three different types of truncations, right and left truncation, and interval truncation, being the most usual left or interval truncation, and the rarest right truncation.

4.2.2 Data structure

As commented before, there is a specific format for the data input in the survival analysis models, this format is briefly explained below.

There are three main structures of data input: single-episode or subject-based, multi-episode based and person or subject-period files, also called discrete-time data files.

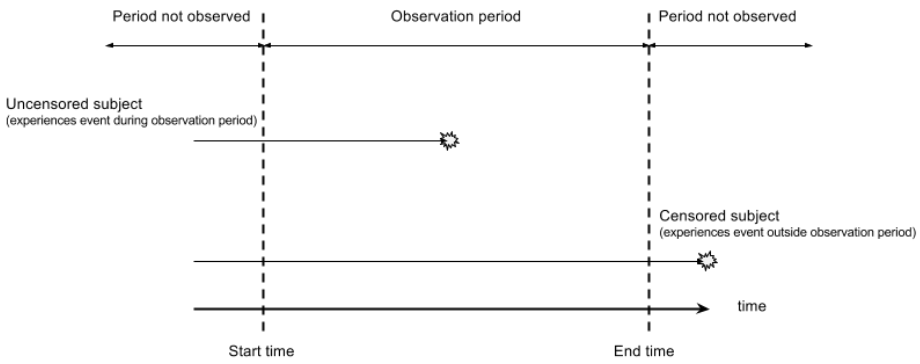

Figure 10: Censored and uncensored data

Single-episode data

In single-episode structured files, each row corresponds to a different subject, and the columns represent the variables and the events occurred.

Multi-episode data

Multi-episode file represents a subjects that experiment the event more than one time, and the data from each subject is separated in different rows for each subject, depending on the number of events the subject experiments.

4.2.3 SystemML input format

¿¿¿¿¿¿¿¿¿¿ The input format for the SystemML library is quite different, it has some particularities such

4.3 Kaplan-Meier Estimator model

Most survival analysis studies start with a non-parametric method as they are simple and fast, and provide with an intuitive graphical form of understanding the data. The two main non-parametric methods, as mentioned before are the Life Tables and the Kaplan-Meier Estimator model. The first is more adequated for large datasets and when the time is not measured with precision. On the other hand, the KM Estimates method is great for when the time is precisely measured, and is widely used, more than the life tables.

The main idea of the KM method is to estimate the survival function at a time 't' $\hat{S}(t)$, which is the probability of a subject surviving until this time 't'. It can be calculated by obtaining the conditional probability of not experiencing the event on time 't', not having experienced it in previous failure times:

$$\hat{S}(t_{(j)}) = \hat{S}(t_{(j-1)}) \times Pr(T > t_{(j)} | T \geq t_{(j)}) \tag{1}$$

The previous equation can also be written as follows, in terms of conditional probability for a specific time:

$$\hat{S}(t_{(j)}) = \prod_{i=1}^{j-1} Pr(T > t_{(i)} | T \geq t_{(i)}) \tag{2}$$

Some advantages of this method are that it takes account for the right censored data, giving more precise and realistic survival times, it is also quite simple and easy to calculate, and gives useful graphs for interpreting the general behavior of the population

4.4 Cox Proportional Hazards Model

Cox Proportional Hazards Model is one of the most relevant regarding time series in general and survival analysis in particular. The model was introduced in 1972 by Sir David Cox in the paper *Regression models and life tables*, and introduced two key features, in first place, the proportional hazards model, in second place the method of partial likelihood estimation.

The

5 Conclusions

5.1 Most important results

5.2 Lessons learnt

References

[1] *Apache Spark data structures [Beyond the lines blog]*. URL: http://www.beyondthelines.net/computing/apache-spark-data-structures/.

[2] *Apache Spark Official Documentation*. URL: https://spark.apache.org/docs/2.1.0/.

[3] *Apache Spark : RDD vs DataFrame vs Dataset [Chandan Prakash's Blog]*. URL: https://http://why-not-learn-something.blogspot.com.es/.

[4] *Apache SystemML Github Repository*. URL: https://github.com/apache/systemml.

[5] *Apache SystemML Official Documentation*. URL: https://systemml.apache.org/.

[6] M. Boehm and D. R. Burdick et al. "SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2014). URL: http://researcher.ibm.com/researcher/files/us-ytian/p52.pdf.

[7] *Databricks: Spark SQL ddata analysis of Liancheng*. URL: http://prog3.com/article/2015-06-18/2824958.

[8] Mohammed Guller. *Big Data Analytics with Spark*. Springer Science+Business Media New York, 2015. ISBN: 978-1-4842-0964-6.

[9] *Inside Apache SystemML [Spark Summit talk]*. URL: https://www.youtube.com/watch?v=n3JJP6UbH6Q.

[10] *Introducing GraphFrames*. URL: https://databricks.com/blog/2016/03/03/introducing-graphframes.html.

[11] Melinda Mills. *Introducing Survival And Event History Analysis*. SAGE Publications, Ltd, 2011. ISBN: 978-84860-102-4.

[12] Reynold Xin and Josh Rosen. "Project Tungsten: Bringing Apache Spark Closer to Bare Metal". In: *Databricks Blog* (Apr. 2015). DOI: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.