Survival Analysis with Apache Spark and Apache SystemML on Stack Exchange

Mateo Álvarez Calvo

September 10, 2017

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction & main goals

Many studies have been done over the Stack Exchange community [12], one of the biggest Question-Answer (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, Stack Exchange has large amount of data of each interaction. Stack Exchange 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 must 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 (ML) 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, Kaplan-Meier estimates (KM), for example, and then apply some semi-parametric model, in this case

Cox Proportional Hazards model (Cox PH). 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 Machine Learning 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 standalone 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 Stack Exchange 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 Stack Exchange 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, the data schema can be found on the *meta Stack Exchange* page, [**meta-Stack Exchange-data-structure**]:

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 and answers	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 1.1: List of files from the compressed Scifi folder

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.

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 Down-vote and 2 for Upvote
CreationDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time of votation

Table 1.2: 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	Id of the post of which the tag was extracted
WikiPostId	Integer	Id of the wiki of which the tag was extracted

Table 1.3: 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 1.4: 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 1.5: 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 1.6: Badges table

1.2.7 Posts.xml

This file contains all posts posted along with the accepted answers and other info related. The posts can be either questions or answers, both contained on the same file. The file has the following structure:

Feature	Data type	Description
Id	Integer	Unique post identifier
PostTypeId	Integer	Type of post codified as integer
AcceptedAnswerId	Integer	Id of the Answer chosen as the correct one
ParentId	Integer	Parent post id
CreationDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time of post creation in extended format
DeletionDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time of post deletion in extended format
Score	Integer	Post's score, calculated from the users' votes
ViewCount	Integer	Count of visualizations
Body	Text	The post itself, in utf8 format
OwnerUserId	Integer	Id of the user who posted the post
OwnerDisplayName	String	Name of the user who posted the post
LastEditorUserId	Integer	last editor's id
LastEditDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Last edition date
LastActivityDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Last interaction with the post time
Title	Text	Title of the post
Tags	Text	Tags added to the post
AnswerCount	Integer	Number of answers to the post
CommentCount	Integer	Number of comments to the post posted
FavoriteCount	Integer	Number of times the post has been added to favorite
ClosedDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time the post has been closed
CommunityOwnedDate	Timestamp YYYY-MM-DDTHH:MM:SS.dScSmS	Time when the post was transferred to community

Table 1.7: 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 1.8: 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 1.9: 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 Stack Exchange site.
- Verify SystemML integration with Spark for further research and scalability.
- Use SystemML survival analysis algorithms to analyze Stack Exchange's data and obtain conclusions on the main variables affecting the time taken by the community to answer each question.

Chapter 2

Technologies

The downloaded data from Stack Exchange 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

[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 [**Spark˙documentation**]

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 must 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 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 must be in continuous communication with the executor programs, it must 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.

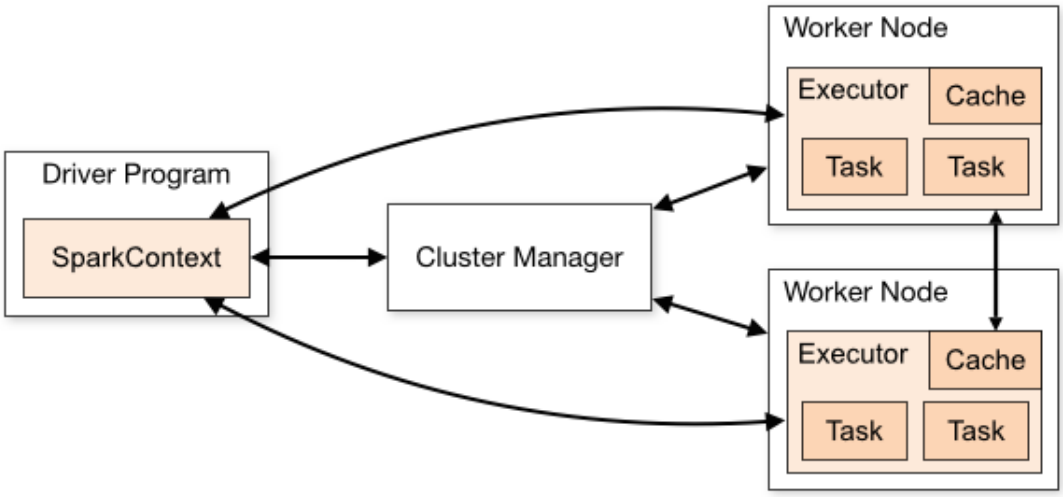


Figure 2.1: Spark Cluster Mode Architecture, from [**Spark˙documentation**]

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.

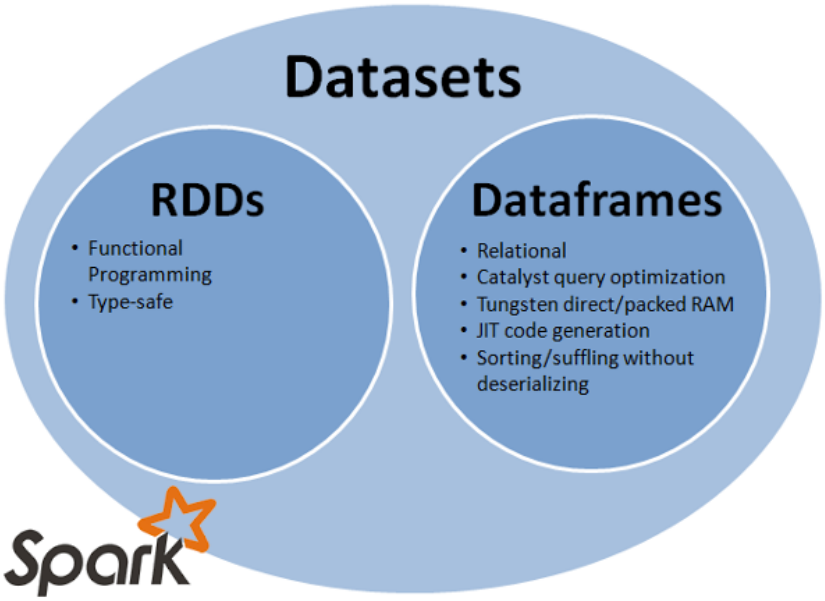


Figure 2.2: Differences between RDDs, dataframes and datasets, from [3]

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.

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.

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.

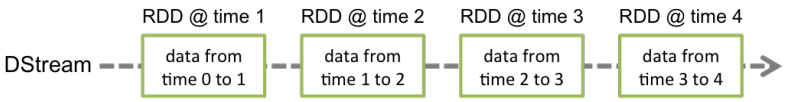


Figure 2.3: DStream structure, from [**Spark˙documentation**]

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 performed over them. This api is addressed by the other, as the main basic structure are the RDDs and the other are buit from it.

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.

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.

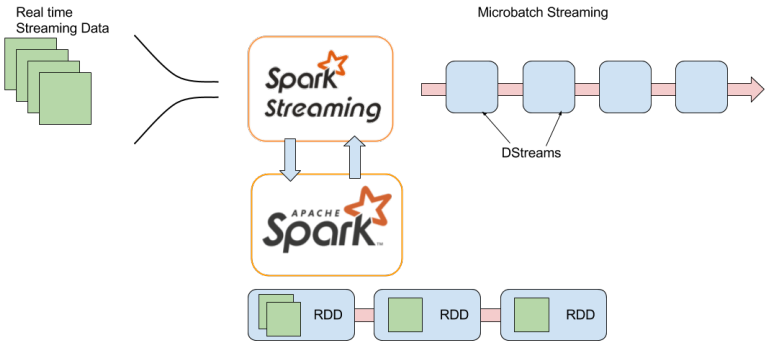


Figure 2.4: Spark Streaming workflow

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.

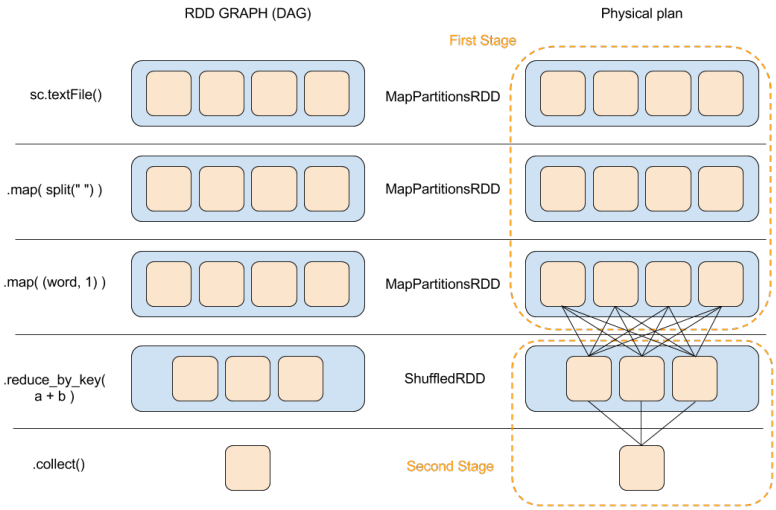


Figure 2.5: Example of translation of DAG to physical plan, for a word count application

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.

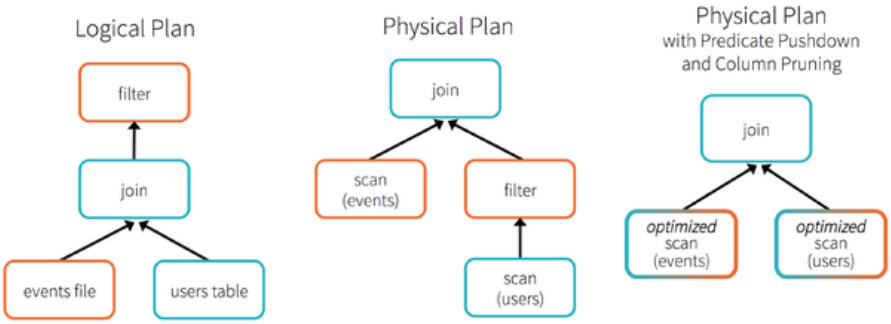


Figure 2.6: Example of an optimized physical plan from a DAG, using Spark SQL, from [8]

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 must 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.

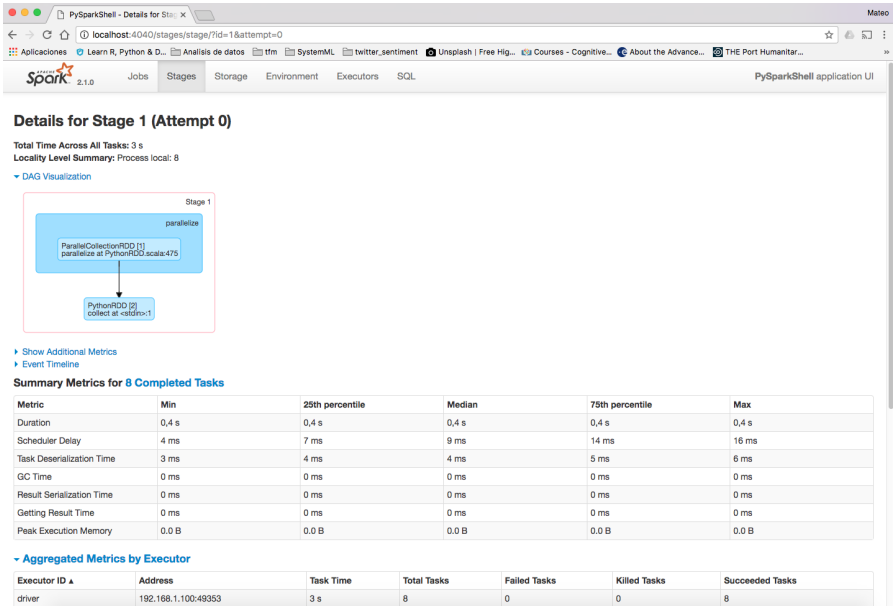


Figure 2.7: Example of Spark UI

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*.

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 2.8: Apache SystemML workflow

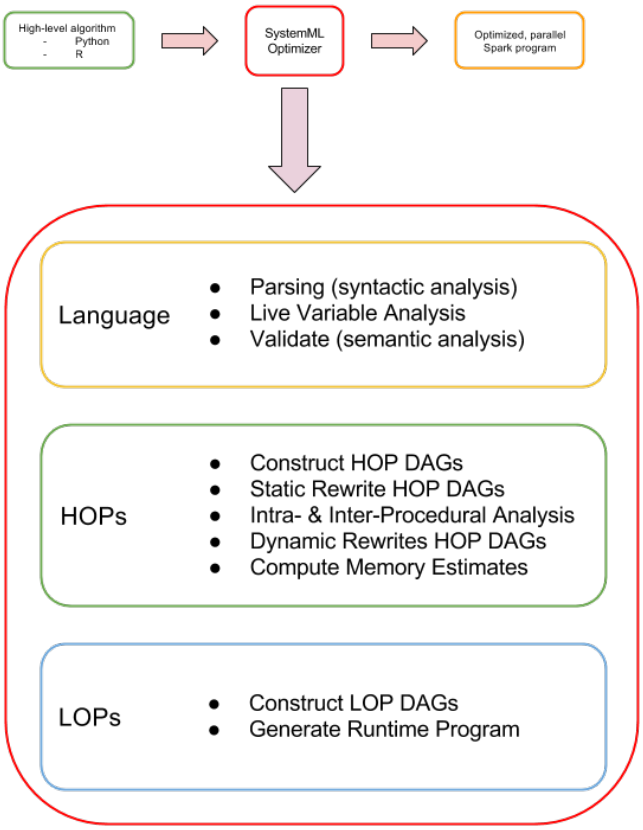


Figure 2.9: SystemML Architecture

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 analyzed, 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

Apart from its general purpose big data analysis capabilities, SystemML has a quite estense algorithm catalog, algorithms that are implemented over the optimizers and thus distributable whenever it is more efficient. Among the catalog SystemML provides, there is an interesting section, specially for this study, the Survival Analysis algorithms, this catalog includes a non-parametric model, the *Kaplan-Meier Estimates model*, and one of the most famous semi-parametric model, the *Cox Proportional Hazards model*. As it'll be explained later, the non-parametric algorithms are useful for exploratory analysis over the data, to have a first approach to the problem, while semi-parametric, and parametric, algorithms are used for further research over the data, giving much more information on the effect of the variables over the survival time.

The main advantage of the semi-parametric models over the parametric models is that the former need less information of the problem itself, not requiring to provide a hazard shape beforehand, while the latter require some assumptions on the behavior of the hazard function on the problem. As it will be shown, this is an important advantage, as

when dealing with an unexplored field, the hazard function is not usually known, having the semi-parametric models the advantage of being more flexible.

Apache Spark's MlLib also has survival analysis algorithm, in particular, the *Survival Regression with Accelerated Failure Time (AFT)* algorithm, a parametric algorithm easier to parallelize than the Cox Proportional Hazards model. This implementation of the AFT is similar to the one implemented on R, but in a distributed way, and uses a *Weibull* distribution function as the hazard function shape. This is one of the main reasons SystemML was chosen for this study, as, unlike Spark, it has a non-parametric algorithm for exploratory analysis of the problem and a flexible semi-parametric algorithm, which is ideal for this study, as the shape of the hazard function is unknown.

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. It 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, and it does not have appreciable effect over the performance of the execution of code.

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.

Figure 2.10: Jupyter Notebook environment example

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, or it can be configured to be the launched when the Spark Shell is executed.

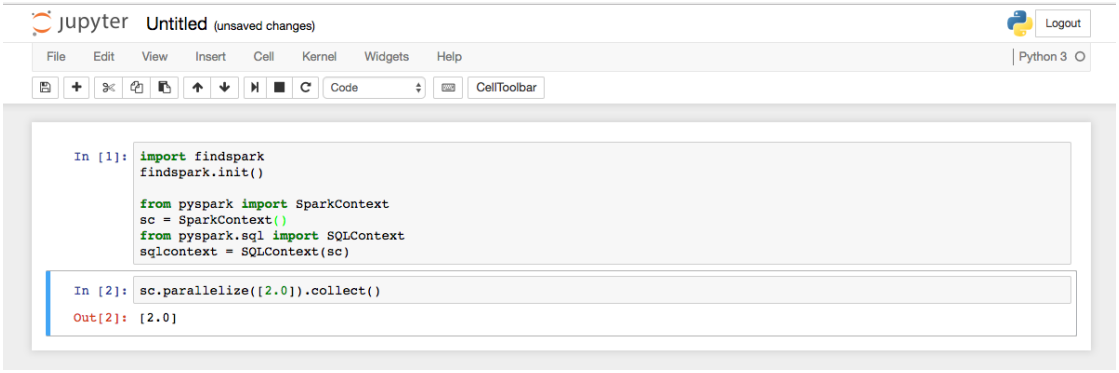


Figure 2.11: Example of a python3 notebook with Spark integrated, running in standalone mode

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	0.14.0-incubating

Table 2.1: 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.html.

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, 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:

Library	Version
appnope[3]	0.1.0
findSpark	1.1.0
ipykernel	4.5.2
ipython	5.3.0
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	$0.14.0 - incubating$
toree[4]	$0.2.0.dev1$
traitlets	4.3.2
fancyimpute	0.4.0

Table 2.2: Installed libraries in Jupyter

The execution of SystemML algorithms can be addressed from the Jupyter Notebook application. For the python kernel, the SystemML library must be installed, instructions for the installation can be found in the Apache SystemML's get started documentation: https://systemml.apache.org/install-systemml.html, while in Scala, the jar file can be included on the startup of the kernel by adding it to the jar files option of Spark or via Spark configuration file.

The alternative to use the Jupyter Notebook environment is using Spark-submit, adding the SystemML.jar file and introducing the algorithm to execute as input parameter, with the desired variables. The configuration of SystemML can be set with the file *SystemML-config.xml*, which is valid for the execution of the standalone mode and the Spark mode using the Spark-submit option with the jar file.

Chapter 3

Infrastructure and resources

The whole study has been executed over a local environment, with a standalone Spark 2.1 installation and both standalone SystemML and SystemML over the standalone Spark infrastructure. This environment does not limit the validity of the present study to be scaled up to a cluster infrastructure, as HDFS has been used to store the data, avoiding local file system restrictions, and the execution over Spark grants the distributed operation of the scripts generated, as the code is the same.

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 single-node installation, on a local network machine.

The data processing has been executed using PySpark over Jupyter Notebook, integrated with the Spark standalone installation, using HDFS for both reading and writing the data. The algorithm execution has also been done over Spark, and using the HDFS infrastructure.

3.2 Configuration

There are 4 main elements to be configure to run this study's code: Jupyter Notebook, Apache Spark, Apache Hadoop and Apache SystemML.

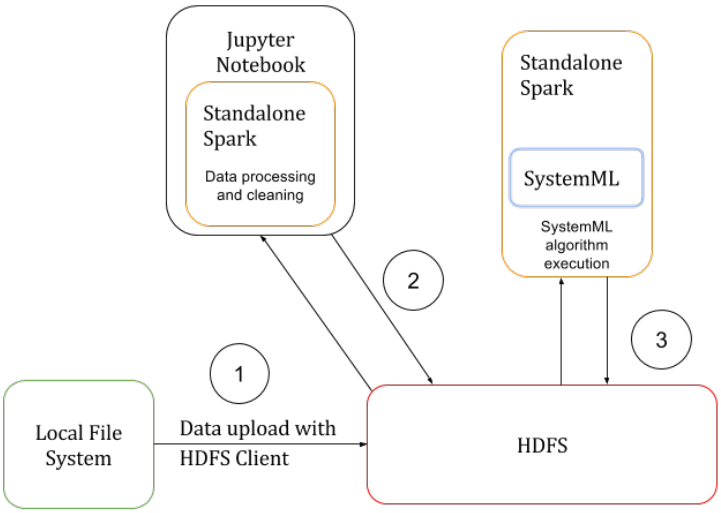


Figure 3.1: Architecture and workflow

3.2.1 Jupyter Notebook

The fastest and easiest way of installing and configuring Jupyter Notebook is using anaconda, which also comes with some important python libraries, such as Numpy or Scipy. The instructions to download and install Jupyter Notebook are on the documentation page: http://jupyter.readthedocs.io/en/latest/install.html.

The list of necessary libraries are show above, 2.2, installing these libraries is simple, just creating a file with the following format: `library==version` and then executing `pip install file-name`.

It is advisable to create a virtual environment to use with this combination of libraries, and not interfering with any other library or version installed. Conda provides this functionality, which is explained on their documentation page: https://conda.io/docs/user-guide/tasks/manage-environments.html.

3.2.2 Apache Spark Configuration

Apache Spark can be downloaded from the official page: https://Spark.apache.org/downloads.html. The downloaded file is a tar.gz file which contains an out of the box runnable Spark, which has been used for this development.

The first step to configure is setting up the environment variables: `$SPARK_HOME` and

`$JAVA_HOME`. The former will indicate python where to find Spark installation and the later will indicate Spark which Java installation use. Spark will also use the environment variables `$HADOOP_HOME` and `$HADOOP_CONF_DIR`.

Spark's configuration can be introduced either on the configuration files located on `$SPARK_HOME/conf/` or directly on the Spark execution, via `--conf` and `--java` options[1].

3.2.3 Apache Hadoop

Hadoop File System has been used for the storage of the files, as in a distributed environment, the file system is not usable. The installation of a Hadoop single node cluster is simple, and explained on the documentation page: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html. Apache Hadoop, comes in a ready to launch tar.gz, which can be downloaded from the official page: http://hadoop.apache.org/releases.html

As Spark and SystemML will need, two environment variables have to be set, `$HADOOP_HOME` and `$HADOOP_CONF_DIR`.

3.2.4 SystemML

SystemML can be downloaded from the official page: https://systemml.apache.org/download.html, and, like Spark and Hadoop, comes in a tar.gz file ready to use. It requires little configuration, apart from the environment variables shown above. The specific configuration can be modified on the `SystemML-config.xml` file.

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 used for the analysis is from the ScyFy community, but, as commented before, the structure is the same for all the Stack Exchange's communities. The data comes in the separated files described on section 1.2. The data comes in *xml* format, with *self-closing* tags, which means that each instance's attributes are stored in the same row

[1] Not all configurations can be indicated this way, for example, the log level needs to be configured using a `log4j.properties` file.

tag as properties, and not in separate subtags of each instance. This particularity is important, as for now, Spark's XML parser does not support this schema, as can be seen on the Spark's repository: *https://github.com/databricks/Spark-xml/pull/149*. Thus the first step is to parse the xml data with Spark and writing it to a *csv* format files. This is done in the notebook: *Convert XML to CSV.ipynb*.

Once the data is in the appropriate format, it is loaded into Spark and processed to get the final values needed for the analysis. All this process is shown in the notebook: *Cleaning data with PySpark.ipynb*, where the data imputation and exploratory analysis is also done. The output of this script are the necessary data files to introduce in the Survival Analysis algorithms in SystemML's desired format. The file generated will be the "fileX" input of the algorithms, and is the result of the combination of the two files *Users.xml* and *Posts.xml*.

Spark SQL api was used to do the processing, as it is necessary to join tables and it is easy to use UserDefinedFunctions to parse data to the appropriate format, specially time series. On the other hand, it is also recommendable to use the Spark SQL api instead of the Core API RDD, as the operations can take advantage of Tungsten and Catalyst optimizers. The result of the process is a dataframe which has still to be cleaned in order to treat the missing values.

FinalTime	CensoringStatus	TagCount	TitleLength	UserReputation	UserAge	PosterReputation	PosterAge
20480.0	1.0	1.0	53.0	846.0	30.0	322.0	40
4096.0	1.0	1.0	85.0	56723.0	54.0	100.0	42
4096.0	1.0	3.0	45.0	21199.0	37.0	218.0	36

Table 3.1: Final table format for SystemML algorithm input, extract from the processed file

The table above shows the final format of data used for the algorithms, the variables are explained below:

- *FinalTime* represents the time passed from the post of the question to the post of the answer or last time, when it is censored.
- *CensoringStatus* whether the instance is censored 0 or not 1
- *TagCount* number of tags in question
- *TitleLength* length of the question's title
- *UserReputation* reputation of the user that answered the question, in case the question is considered answered
- *UserAge* age of the user that answered the question, in case the question is considered answered
- *PosterReputation* reputation of the user that posted the question
- *PosterAge* age of the user that posted the question

3.3.2 Missing values treatment

There are just three variables without missing values: CensoringStatus, TagCount and TitleLength. The other ones have been modified to be completed.

FinalTime

The final time variable is composed with the start and end time of each question. The end of the question is considered whenever an accepted answer is posted, otherwise the data is right censored, and the end time is not known. The proportion of censored instances is about (x%), and the missing values have been imputed using the longest duration of a question, considering it the study's time interval.

UserReputation and UserAge

The information relative to the users that answered the question is only available for the questions that are not censored, as are the ones that were correctly answered. This information is always going to be missing, so in this study these features are not being considered for the survival analysis.

PosterReputation

The majority of data is valid in this section, about (97.8%), and, as it does not follow any specific distribution, random sampled data has been used to impute the missing values.

PosterAge

The proportion of unknown values of the poster's age is about (60%), it does not follow any distribution either. The imputation of this characteristic has been done comparing two different methods, available on the *fancyimput* python library. The objective is to predict the missing values with the other variables available on the dataset for each instance.

The first one is the Multiple Imputation using Chained Equations (MICE). This method uses different models to impute the data, performing analysis over the imputed values to select the best mode for each partition.

The second procedure is using KNN algorithm with different amount of neighbors to fill the missing values. This second algorithm has proven to give more valid results than the former, in this case.

3.3.3 Datasets description for preparing the algorithms

Apart from the data itself, the SystemML algorithms require some specific files, containing the description of the data and variables associated to the model training. It is also required to provide the *.mtd* files for each input file of the algorithm. These files contain metadata that describes the file's format and matrix dimensions, in order to be used by the SystemML optimizer.

3.3.4 Algorithm execution and analysis of results

The last part of the process is to generate and analyze the results. The algorithm execution is shown on each algorithm's section, both have been run on standalone mode and on a standalone Spark environment.

Chapter 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 Input datasets

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.

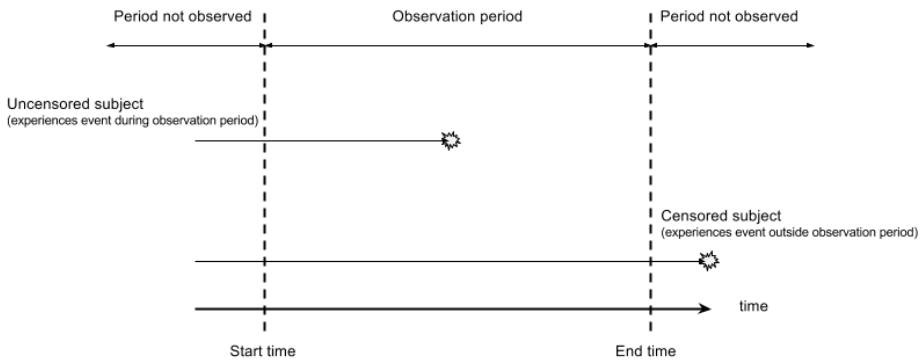


Figure 4.1: Censored and uncensored data

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.

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. It implies that if a subject experiments more than one event, it is registered on the same row, as there ir no information of the instance on any other row.

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 event, where the first event is in the first row of the instance and the subsequent events, in case there are, are represented on the consecutive rows, having each subject as much rows as events the subject experiences.

4.3 Kaplan-Meier Estimator model

4.3.1 The Kaplan-Meier Estimator

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' $\widehat{S}(t)$, 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:

$$\widehat{S}(t_{(j)}) = \widehat{S}(t_{(j-1)}) \times Pr(T > t_{(j)} | T \geq t_{(j)}) \tag{4.1}$$

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

$$\widehat{S}(t_{(j)}) = \prod_{i=1}^{j-1} Pr(T > t_{(i)} | T \geq t_{(i)}) \tag{4.2}$$

The KM model is a non-parametric, maximum likelihood model, which means that it relies on the maximization of the likelihood function of the sample to analyze. This likelihood function takes the following form:

$$L = \prod_{i=1}^{k} h_i^{d_i}(1-h_i)^{n_i-d_i} \tag{4.3}$$

where n_i represents the number of individuals at risk at a certain time, and d_i represents the number of events occurred at this particular time. Maximizing this function, the hazard function can be obtained in terms of n_i and d_i for each time:

$$\widehat{h_i} = \frac{d_i}{n_i},\ for\ i = 1, 2, ..., k \tag{4.4}$$

and the survival function for each time has the following form:

$$\widehat{S}(t_i) = \prod_{i|t_i<t_k} (1 - \frac{d_i}{n_i}) \tag{4.5}$$

Assuming asymptotic result, for big series of data in the sample, the variance of the maximum likelihood estimator can be obtained:

$$\widehat{Var} = \frac{d_i(d_i - n_i)}{n_i^3} \tag{4.6}$$

and with this an estimation of the variance of the survival function, resulting in the *Greenwood equation*:

$$\widehat{Var}(\widehat{S}(t_i)) = \widehat{S}(t_i)^2 \sum_{j=1}^{i-1} \frac{d_j}{d_j(n_j - d_j)} \tag{4.7}$$

Considering the asymptotic convergence to normal distribution of the maximum likelihood estimators, confidence intervals for $(1-a)\%$ can be calculated for this estimations:

$$\widehat{S}(t_i) \pm Z_{a/2}\sqrt{\widehat{Var}(\widehat{S}(t_i))} \tag{4.8}$$

One interesting advantage of the Kaplan-Meier Estimates model is the visualization of the survival function in a graph, as this method is intended for exploratory analysis and for a light approach to the problem itself, the possibility of plotting the results in a easy to understand graph is an interesting feature. In comparison with other algorithms

outside survival analysis, the main advantage is the ability to account for the censored data.

The other side of the simplicity of the model is that it only provides probability estimations on the whole group, it does not take account of other explanatory variables. This limitation constraints the analysis to homogeneous groups, where the other explanatory variables are homogeneous along all the sample. To mitigate this limitation when the sample is not homogeneous on the other explanatory variables, it can be separated in more homogeneous groups and apply the method to each group, being able to compare the results of every subsample.

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, and in second place the method of partial likelihood estimation.

The Cox Proportional Hazards model can be used both for time-fixed or time-varying covariates, on the first one the model is also called Proportional Hazards Model, while the second one is often called Extended Cox Model.

4.4.1 Cox Proportional Hazards Model for time-fixed covariates

Time-fixed covariates are the ones that remain constant on the whole study, and are characteristics of the subjects, for example the gender or the place of birth. The expression of the hazard function for each individual (i) at time (t) in the basic time-fixed for covariates model is the following:

$$h_i(t) = h_0(t) \exp(\beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_k x_{ik}) \tag{4.9}$$

where the x_{ik} represent the values of each covariate (k) for each individual of the sample, h_0 is the *baseline hazard function*, which represents the hazard that has a subject which covariates are all 0, being this subject the reference subject of the study. This function is arbitrary and not specified, and is the reason why the model is semi-parametric, as it can take any form, which is one of the interesting features of the model, providing flexibility. The intercept of the function is contained on this baseline hazard function, which can be estimated.

The equation can be written in terms of *log hazard ratio*, leading to a function that does not depend on the time (t), as the covariates of the model are all constant, considering them time-fixed.

$$\log\left(\frac{h_i(t)}{h_0(t)}\right) = \beta_1 x_{i1} + \cdots + \beta_k x_{ik} \tag{4.10}$$

comparing the hazard functions for two individuals, obtaining the hazard ratio:

$$\frac{h_i(t)}{h_j(t)} = \frac{\cancel{h_0(t)}}{\cancel{h_0(t)}} \times \exp(\beta_1(x_{i1} - x_{j1}) + \cdots + \beta_k(x_{ki} - x_{kj})) = \exp(\beta_1(x_{i1} - x_{j1}) + \cdots + \beta_k(x_{ki} - x_{kj})) \tag{4.11}$$

the previous equation (4.11) shows that the hazard ratio between two subjects does not depend on the baseline hazard function. Using a logarithmic scale for the hazard ratio, it can be seen that it is proportional to the covariates and the parameters of the model:

$$\log(\frac{h_i(t)}{h_j(t)}) = \beta_1(x_{i1} - x_{j1}) + \cdots + \beta_k(x_{ik} - x_{jk}) = \bar{\beta} \cdot (\bar{x_i} - \bar{x_j}) \tag{4.12}$$

This result leads to the assumption of proportional hazards, as the hazard ratio only depends on the betas and values of the covariates, which do not depend on time, following the first assumption. This proportional hazards mean that two individuals that have different values of the covariates, will always have proportional hazard one from the other.

This assumption needs to be verified every time this model is intended to be used. To do so, the effect of every variable on the hazard function needs to be constant over time. To check the behavior of the covariates there are several methods:

The first one is to use a plot of the survival function to check that two individuals with different values of the covariates have paralel survival functions.

The other way may be using the Schoenfeld residuals, either on the statistical or the graphical test.

One example where the assumption of proportional hazards is not reached may be the survival of cancer patients treated with chemotherapy alone or subject to surgical intervention along with the chemotherapy. The covariate representing the treatment could be a dichotomic variable, with 0 meaning chemotherapy alone and 1 representing surgery along with chemotherapy.

Although the variable itself will not change over the time, the treatment will be constant, the effect on the hazard can be a function of time, for example, during the surgical intervention, the survival of the patient is endangered by posible surgical problems, then, if it has been successful, the hazard will decrease as the elimination of the tumor improves the survival.

One posible solution for this case can be separating the patients by the treatment received into independent groups and compare the hazard function for each group, to understand the advantages of one treatment over another.

Other solutions can be starting the analysis after the surgery so that the effect of the treatment variable is constant over time, or use Cox's Extended Model to account for the interaction of the surgical groups and time.

4.4.2 Extended Cox Model for time-varying covariates

The Cox model can also be applied to time-varying series, the formulation of it is the same as the time-invariant but with the time dependence. The time-dependent variables can be continuous or discrete.

Given $\bar{x_i}(t) = x_{1i}, \cdots, x_{ki}$, and time intervals $t_1, \cdots, t_k$, the Cox formulation for time dependent variables has the following form:

$$h_i(t) = h_0(t) \exp(\beta_1 x_{1i}(t) + \cdots + \beta_k x_{ki}(t)) \tag{4.13}$$

where some variables can be time-independent, with fixed value over time. This equation can also be expressed in log hazard ratio form:

$$\log \left[\frac{(h_i(t))}{h_0(t)} \right] = \beta_1 x_{1i}(t) + \cdots + \beta_k x_{ki}(t) \tag{4.14}$$

the the condition of proportional hazards may not be fulfilled now, as, unlike time-fixed covariates, the hazard ratio is function of time, in general, and so, it's value can vary over time.

4.4.3 Partial Likelihood

Cox Model's parameters (β_i) are to be estimated. To do so, a likelihood function is defined to fit the model and estimate its parameters, and this likelihood function estimation is done with the Maximum Likelihood Estimation method.

The likelihood function of the Cox Model is also called partial likelihood, this is because it only accounts for the probability of the uncensored subjects. This way, it is composed by the productory of several likelihoods at different failure times.

Given $L \equiv L(\beta_1, ..., \beta_p)$, the partial likelihood function, supposing there are k ceasing time intervals, and there are no draws, there will be $n - k$ censored times. The ordered censoring timestamps will be $t_1, \cdots, t_k$, the group of subjects at risk at each time interval will be noted by $R(t_i)$, for $i = 1, \cdots, k$. The partial likelihood for each time interval wil be $L_i \equiv L_i(\beta_1, \cdots, \beta_p)$ for $i = 1, \cdots, k$

The likelihood function for the total mode has this expression:

$$L = \prod_{i=1}^{k} L_i \tag{4.15}$$

the objective is to maximize this function, or in this case the log of this function, that is to say that:

$$\begin{aligned} \frac{\partial log(L)}{\partial \beta_i} &= 0, \ \forall \ \beta_i \in \{\beta_1, \cdots, \beta_p\} \\ \frac{\partial^2 \log(L)}{\partial \beta_i \partial \beta_j} &< 0, \ \forall \ \beta_i, \beta_j \in \{\beta_1, \cdots, \beta_p\} \end{aligned} \tag{4.16}$$

From the first of the equations, an estimation of the parameters of the model can be obtained, $\bar{\widehat{\beta}} = \{\widehat{\beta_1}, \cdots, \widehat{\beta_p}\}$. The second expression is to ensure the parameters meet a maximum of the function. This expression can also be used to obtain the variance-covariance matrix:

$$I_{ij}(\beta) = -\frac{\partial^2 \log(L)}{\partial \beta_i \partial \beta_j} \tag{4.17}$$

$$\widehat{\Sigma}_{[p \times p]} = I^{-1} \bar{\widehat{\beta}} \tag{4.18}$$

The resulting estimators of the β parameters of the module are asymptotically unbiased, efficient and normal, with mean $(\widehat{\beta_1}, \cdots, \widehat{\beta_p})$ and variance-covariance matrix $\widehat{\Sigma}$, however the estimator is not completely efficient, as it does not reach the Cramer-Rao limit.

The partial likelihood functions on every time interval have the expression:

$$L_i = \frac{h_0(t_i) \exp\left(\sum_{j=1}^{p} \beta_j x_{ij}\right)}{\sum_{l \in R(t)} h_0(t_i) \exp\left(\sum_{j=1}^{p} \beta_j x_{lj}\right)} = \frac{\exp\left(\sum_{j=1}^{p} \beta_j x_{ij}\right)}{\sum_{l \in R(t)} \exp\left(\sum_{j=1}^{p} \beta_j x_{lj}\right)} \tag{4.19}$$

Finally, the partial likelihood function will have the following expression:

$$L = \prod_{i=1}^{k} L_i = \prod_{i=1}^{k} \frac{\exp\left(\sum_{j=1}^{p} \beta_j x_{ij}\right)}{\sum_{l \in R(t)} \exp\left(\sum_{j=1}^{p} \beta_j x_{lj}\right)} \tag{4.20}$$

This total partial likelihood function does not depend on the baseline hazard function, nor the value of the times, just the ordering of them and the censored data. In case one

of the initial assumptions, the inexistence of draws in decease times, this function has to be modified, for example using the *Breslow Method* [7].

4.4.4 Hypothesis contrasts

The estimation of the model's parameters $\bar{\widehat{\beta}}$, approximately normal with mean $\widehat{\beta_1}, \cdots, \widehat{\beta_p}$, and with variance-covariance matrix $\widehat{\Sigma}$, allows the use of statistical tests similar to linear models to contrast hypothesis.

Two hypotheses can be proposed:

The first hypothesis may be H_0: $\beta_j = 0$ versus H_1: $\beta_j \neq 0 \ \forall \ j \in 1, \cdots, p$. To contrast this hypothesis, the Wald statistic can be used:

$$z = \frac{\widehat{\beta}_j}{\sqrt{\widehat{Var}(\widehat{\beta}_j)}} \tag{4.21}$$

so the $(1-\alpha)\%$ interval can be calculated as:

$$\widehat{\beta}_j \pm z_{1-\alpha/2}\sqrt{\widehat{Var}(\widehat{\beta}_j)} \tag{4.22}$$

The second hypothesis to contrast could be the following: H_0: $\beta = \widehat{\beta}$ versus H_1: $\beta \neq \widehat{\beta}$. Three contrasts can be used:

Wald contrast

Assuming normal distribution for the parameters of the model, $\bar{\widehat{\beta}}$, with mean $\widehat{\beta_1}, \cdots, \widehat{\beta_p}$, and variance-covariance matrix $\widehat{\Sigma} = I^{-1}\bar{\widehat{\beta}}$, this statistic has the following form:

$$\chi_W = (\bar{\widehat{\beta}} - \bar{\beta_0})^T I(\bar{\widehat{\beta}})(\bar{\widehat{\beta}} - \bar{\beta_0}) \tag{4.23}$$

which follows a χ^2 distribution with p degrees of freedom.

Likelihood ratio contrast

This contrast uses the value of the partial likelihood function evaluated in $\hat{\beta}$ and β_0

$$\chi_{LR} = 2\left(\log[L(\bar{\widehat{\beta}}) - \log[L(\bar{\beta_0})]]\right) \tag{4.24}$$

which follows a χ^2 distribution with p degrees of freedom.

Score contrast (Log Rank)

This contrast uses the gradient of the log of the partial likelihood evaluated on H_0, taking this form:

$$\chi_S = \left(\frac{\partial L(\beta_0)}{\partial \beta}\right)^T \left(\frac{\partial^2 L(\bar{\beta}_0)}{\partial \bar{\beta} \partial \bar{\beta}^T}\right)^{-1} \frac{\partial L(\bar{\beta}_0)}{\partial \bar{\beta}} \tag{4.25}$$

which is approximately multi-normal distribution with variance-covariance matrix $I(\bar{\bar{\beta}})$, so the statistic follows a χ^2 distribution with p degrees of freedom.

4.4.5 Model fitness and adequacy testing

There are several tests to address model fitness and adequacy, the log-likelihood test, the likelihood ratio test and the Aike Information Criterion are used for the first purpose, while the Cox-Snell residuals is used for the later.

4.4.6 SystemML input & output format

As described on the oficial documentation of Apache SystemML, to run a survival analysis using the KM estimates method or Cox Proportional Hazards model, three files have to be provided, containing three matrixes, the data itself in the appropriate format, along with two other matrix indicating position of specific variables inside the former. These files are specified on each algorithm's description, as the project is open source, the algorithm implementation can be seen on the project's Github page [4].

Apart from the files specified on the oficial documentation and the algorithm implementation, it is also necessary to provide, along with every matrix file, another file with the same name, and the *.mtd* extension, containing the information of the file containing the matrix in JSON format:

```
1  {
2    "data_type": "matrix",
3    "value_type": "double",
4    "rows": 10,
5    "cols": 3,
6    "nnz": 10,
7    "format": "csv",
8    "header": false,
9    "sep": ",",
10   "description": {
11       "author": "SystemML"
```

```
12              }
13          }
14
```

Code 4.1: Example of matrix metadata file in JSON format

Kaplan Meier Estimates

According to the oficial documentation, the files and parameters needed to execute the KM Estimates algorithm are the following:

File or parameter	**Content or value**
X	input matrix with the survival data itself, containing the following variables, separated in columns: timestamps, whether the event occurred (1) or the data is censored (0), columns with categorical features for grouping and/or stratifying
TE	input column matrix indicating the indices of the columns where the timestamps and the event information are located on matrix X
GI	input column matrix indicating the indices of the columns where the categorical values used for grouping are stored on matrix X
SI	input column matrix indicating the indices of the columns where the categorical values used for stratifying are stored on matrix X
F	input column matrix indicating the indices of X which are to be used for KM analysis
alpha	parameter to compute confidence intervals for the survivor function and its median
etype	parameter to specify the error type, *greenwood* or *peto*
ctype	parameter to modify the confidence interval, *plain* remains the lower and upper bound of the confidence interval unmodified, *log* corresponds to the logistic transformation, and *log-log* corresponds to the complementary log-log transformation
ttype	if data from different groups is available, parameter to specify the test to perform for comparing survival data across multiple groups, the options are: *none*, *log-rank* and *wilcoxon*
fmt	parameter to specify the output format to use for the results: *text* or *csv*

Table 4.1: Input files and parameters for Kaplan-Meier Estimates algorithm

File or parameter	Content or value
O	output matrix containing the results of the Kaplan-Meier analysis, with 7 columns that represent the status on each time interval: • timestamp • number of subjects at risk • number of events • KM estimate of the survival function • Standard error of survival function • Lower confidence interval for survival function • Upper confidence interval for survival function
M	output matrix whose dimension depends on the number of groups (g) and strata (s) in the data (k denotes the number of factors used for grouping, i.e., ncol(GI) and l denotes the number of factors used for stratifying, i.e., ncol(SI)): • unique combination of values in the k factors used for grouping • unique combination of values in the l factors used for stratifying • total number of records • total number of events • median • lower $100 * (1 - alpha)\%$ confidence interval of the median of survival function • upper $100 * (1 - alpha)\%$ confidence interval of the median of survival function
T	if survival data from multiple groups is available, and *ttype* is *log-rank* or *wilcoxon*, output matrix containing the result of the stratified test for comparing multiple groups: • number of groups • degree of freedom for Chi-squared distributed test statistic • test statistic • p-value
T_GROUPS_OE	if exists T, statistics for groups formed on analysis, with as many rows as groups, and the following column values: • number of events • observed value • expected value • $(O - E)^2/E$ • $(O - E)^2/V$

Table 4.2: Output files and parameters for Kaplan-Meier Estimates algorithm

```
1  0,0,1.0
2  0,0,1.0
3  0,0,1.0
4  0,0,1.0
5  0,0,1.0
6  0,0,1.0
7  0,0,1.0
8  0,0,1.0
9  0,0,1.0
10 0,0,1.0
11
```

Code 4.2: Input X matrix for KM Estimates model

```
1  {
2    "data_type": "matrix",
3    "value_type": "double",
4    "rows": 10,
5    "cols": 3,
6    "nnz": 10,
7    "format": "csv",
8    "header": false,
9    "sep": ",",
10   "description": {
11       "author": "SystemML"
12   }
13 }
14
```

Code 4.3: Input X metadata matrix for KM Estimates model

This sample files can be generated via script located in the github repository, on the scripts directory. It is recommendable to do so, to see the appropriate input format the files have for the algorithm, an example of the invocation of this sample generation script is shown below[1]:

```
1  #!/bin/bash
2  <SPARK_HOME>/bin/Spark-submit <systemml-path>/SystemML.
   jar\
3  -f <systemml-scripts-path>/datagen/
   genRandData4SurvAnalysis.dml\
4  -nvargs type=kaplan-meier\
5  n=<number-samples>\
6  p=<probability-of-not-censoring>\
7  O=<O-file-path>\
8  TE=<TE-file-path>\
9  fmt=<format>
10 g=<groups>
11 s=<strata>
12 f=<number-of-levels-for-cathegorical>
13
```

Code 4.4: Sample invocation of the data generation script

[1]More information about the parameters and configuration to run the data generation can be found on the script itself, available on the SystemML github repository.

Cox Proportional Hazards Model

The Cox Proportional Hazards model is, as seen above, quite different from the KM estimates, and needs the following files and parameters:

File or parameter	**Content or value**
X	input matrix with the survival data itself, containing the following variables separated in columns: timestamps, whether the event occurred (1) or the data is censored (0), columns of features for the model
TE	input matrix indicating the indexes of the columns where the timestamps and the event information are located on matrix X
F	input column matrix indicating the indices of X which are to be used for fitting the Cox model
R	If factors (categorical variables) are available in the input matrix X, matrix containing the start and end indices of the factors in X • R[,1]: start indices • R[,2]: end indices Alternatively, user can specify the indices of the baseline level of each factor which needs to be removed from X. If R is not provided by default all variables are considered to be continuous
alpha	parameter to compute confidence intervals, $100 * (1 - alpha)\%$ for the betas of the model
tol	tolerance, ϵ
moi	maximum number of outer iterations, using Newton method
mii	maximum number of inner iterations, using conjugate gradient method, if this value is 0, there will be no maximum
fmt	format of the output matrixes, it can be csv or txt

Table 4.3: Input files and parameters for Cox Proportional Hazards algorithm

File or parameter	Content or value
M	output matrix with dimensions $D \times 7$, where D is the number of covariates of the model, and the columns represent: • betas (β_i) • $exp(\beta_i)$ • standard error of β_i • Z • p-value • lower $100 * (1 - alpha)\%$ confidence interval of β_i • upper $100 * (1 - alpha)\%$ confidence interval of β_i
S	output log file containing a summary of statistics of the model: • number of observations • number of events • log-likelihood • AIC • Rsquare (Cox & Snell) • max posible Rsquare
T	output log file containing a summary of statistics of the model, with the following format in lines: • Likelihood ratio test statistic, degree of freedom, P-value • Wald test statistic, degree of freedom, P-value • Score (log-rank) test statistic, degree of freedom, P-value
RT	output column matrix containing the order-preserving recorded timestamps from X
XO	output matrix, which is the same of X but with timestamps
COV	output matrix containing the variance-covariance matrix of betas
MF	output column matrix containing the column indices of X with the baseline factors removed (if available)

Table 4.4: Output files and parameters for Cox Proportional Hazards algorithm

4.5 Algorithm invocation

The algorithm scripts have a sample invocation of them to be executed in a Hadoop environment, the execution over Spark is similar:

```
1   #!/bin/bash
2   <SPARK_HOME>/bin/Spark-submit <path-to-systemml>/SystemML.
    jar\
3   -f <path-to-systemml-algorithms>/scripts/algorithms/KM.dml\
4   -nvargs X=<file-X-location>\
5           TE=<file-TE-location>\
6           GI=<file-GI-location>\
7           SI=<file-SI-location>\
8           O=<file-O-location>\
9           M=<file-M-location>\
10          T=<file-T-location>\
11          alpha=<alpha>\
12          etype=<error-type>\
13          ctype=<confidence-interval-tunning>\
14          ttype=<test-type>\
15          fmt=<output-format>
16
```

Code 4.5: Example of the invocation of the Kaplan-Meier Estimates model with Apache SystemML over an Apache Spark framework

```
1   #!/bin/bash
2   <SPARK_HOME>/bin/Spark-submit <path-to-systemml>/SystemML.
    jar\
3   -f <path-to-systemml-algorithms>/scripts/algorithms/Cox.dml
    \
4   -nvargs X=<file-X-location>\
5   TE=<file-TE-location>\
6   F=<file-F-location>\
7   R=<file-R-location>\
8   M=<file-M-location>\
9   S=<file-S-location>\
10  T=<file-T-location>\
11  COV=<file-COV-location>\
12  RT=<file-RT-location>\
13  XO=<file-XO-location>\
14  MF=<file-MF-location>\
15  alpha=<alpha>\
16  tol=<tolerance-epsilon>\
```

```
17      moi=<max-outer-iterations>\
18      mii=<max-inner-iterations>\
19      fmt=<output-format>
20
```

Code 4.6: Example of the invocation of the Cox Proportional Hazard model with Apache SystemML over an Apache Spark framework

Note that the input and output matrixes are files, and the algorithms have to have the path to them as input arguments, they also need the parameters set on the invocation of the algorithm, otherwise, default values will be used. It is also important to note that, as commented before, along with the file containing the matrix, another file containing the metadata associated to the matrix have to be provided, as can be seen on the code examples of 4.2 and 4.3.

As the execution of the algorithm is a jar file given to Spark-submit, all the Spark's configuration can be set, thus the execution on a distributed environment would be almost the same, but with the specific Spark's configuration to run in distributed mode with the desired resources and security configuration.

The execution can also be done in the Jupyter Notebook, as the algorithm's script is open source, just copying the code into a cell and calling the function with the desired parameters, or invoking the algorithm's code via scriptUrl method, which will connect to the specified url and dowload the code.

4.6 Results

4.6.1 Exploratory analysis

After cleaning and imputing data, and before introducing the data into the algorithms, an exploratory analysis has been performed over the data. This part has been done using *pandas* python library along with *matplotlib*, so it is not entirely scalable, as it has not been executed on a distributed mode, but the exploratory analysis over a bigger dataset can be done over a sample of it.

The first step is to view the data, plotting their density distribution and obtaining basic statistics.

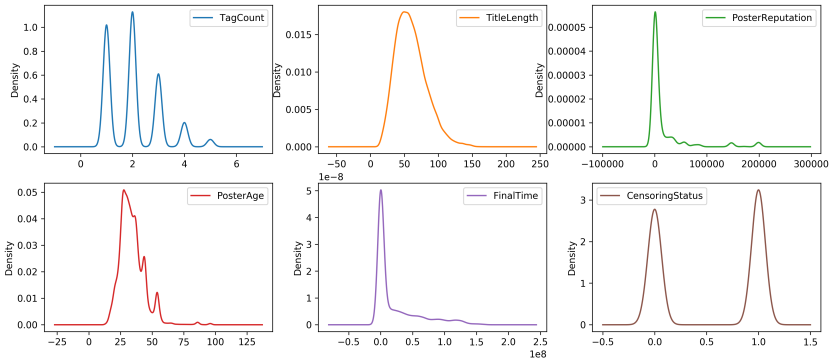


Figure 4.2: Variable density distributions

	TagCount	**TitleLength**	**PosterReputation**	**PosterAge**	**FinalTime**	**CensoringStatus**
count	24009.000000	24009.000000	24009.000000	24009.000000	$2.400900e+04$	24009.000000
mean	2.058895	59.923320	13837.861927	34.053919	$2.195890e+07$	0.538715
std	0.992615	22.637822	34439.894285	9.755022	$3.553241e+07$	0.498509
min	1.000000	15.000000	1.000000	13.999999	$4.096000e+03$	0.000000
25%	1.000000	43.000000	109.000000	27.182650	$8.192000e+03$	0.000000
50%	2.000000	57.000000	803.000000	32.640093	$2.088960e+05$	1.000000
75%	3.000000	73.000000	8117.000000	38.360211	$3.276390e+07$	1.000000
max	5.000000	168.000000	198942.000000	96.000000	$1.624433e+08$	1.000000

Table 4.5: Basic statistics

	TagCount	TitleLength	PosterReputation	PosterAge	FinalTime
TagCount	1.000000	0.060511	−0.015367	0.007575	−0.018741
TitleLength	0.060511	1.000000	−0.003752	0.058853	0.071873
PosterReputation	−0.015367	−0.003752	1.000000	0.210137	0.002616
PosterAge	0.007575	0.058853	0.210137	1.000000	0.051550
FinalTime	−0.018741	0.071873	0.002616	0.051550	1.000000

Table 4.6: Correlation matrix

Analyzing the results above, there are two variables that have important outliers, *User-Reputation* and *FinalTime*, both containing a major part of small values and little amounts of big values. This causes important variations on the mean and on the correlation matrix, as both are sensitive to outliers. To mitigate this effect, a base 10 logarithmic transformation has been applied to both variables.

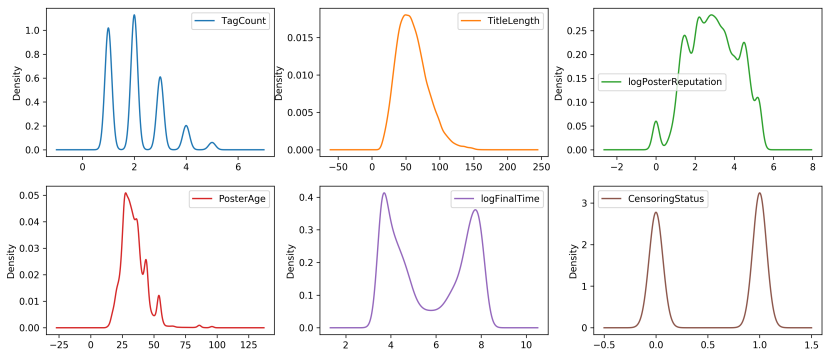


Figure 4.3: Transformed variable density distributions

With the change of scale, the variables PosterReputation and FinalTime are now better distributed 4.7, and do not have outliers. Regarding the correlation matrix 4.8, with this scale, a slight influence of logPosterReputation can be accounted on the PosterAge, in a direct way, increasing with the poster's reputation and with the logFinalTime, with an inverse relation, which means that the time to answer will decrease when increasing the poster's reputation.

	TagCount	TitleLength	logPosterReputation	PosterAge	logFinalTime	CensoringStatus
count	24009.000000	24009.000000	24009.000000	24009.000000	24009.000000	24009.000000
mean	2.058895	59.923320	2.923551	34.053919	5.701602	0.538715
std	0.992615	22.637822	1.245802	9.755022	1.725381	0.498509
min	1.000000	15.000000	0.000000	13.999999	3.612360	0.000000
25%	1.000000	43.000000	2.037426	27.182650	3.913390	0.000000
50%	2.000000	57.000000	2.904716	32.640093	5.319930	1.000000
75%	3.000000	73.000000	3.909395	38.360211	7.515396	1.000000
max	5.000000	168.000000	5.298727	96.000000	8.210702	1.000000

Table 4.7: Basic statistics

	TagCount	TitleLength	logPosterReputation	PosterAge	logFinalTime
TagCount	1.000000	0.060511	0.007440	0.007575	−0.006862
TitleLength	0.060511	1.000000	−0.108555	0.058853	0.093518
PosterReputation	0.007440	−0.108555	1.000000	0.170244	−0.229210
PosterAge	0.007575	0.058853	0.170244	1.000000	0.015670
FinalTime	−0.006862	0.093518	−0.229210	0.015670	1.000000

Table 4.8: Correlation matrix

4.6.2 Algorithm execution and output

Using the Spark-submit execution with the SystemMl.jar file, the algorithms have been executed.

Kaplan Meier

The Kaplan Meier algorithm has been executed grouping the tag-count variable, so that a result with groups comparison can be done, this comparison between groups is necessary for the correct execution of the algorithm, otherwise, the algorithms will not output results. As explained before, the algorithm outputs the complete time series with survival data, which can be plotted using, for example python's *matplotlib*.

The graphic above shows no difference in survival between groups, so the TagCount does not introduce a significative difference in survival times. It is important to note that this example was executed using just one grouping variable, TagCount, which has 5 levels, and took between 3 and 4 seconds to execute, but this algorithm needs to use categorical variables and not continuous to perform these grouping and stratifying, and they have to be moderate in size to work properly.

Cox Proportional Hazards

Cox Proportional Hazards model has been executed with all the features, included the log-transformed PosterReputation variable. The output of the algorithm are the adjusted β parameters of the model with the confidence interval, the information to build survival

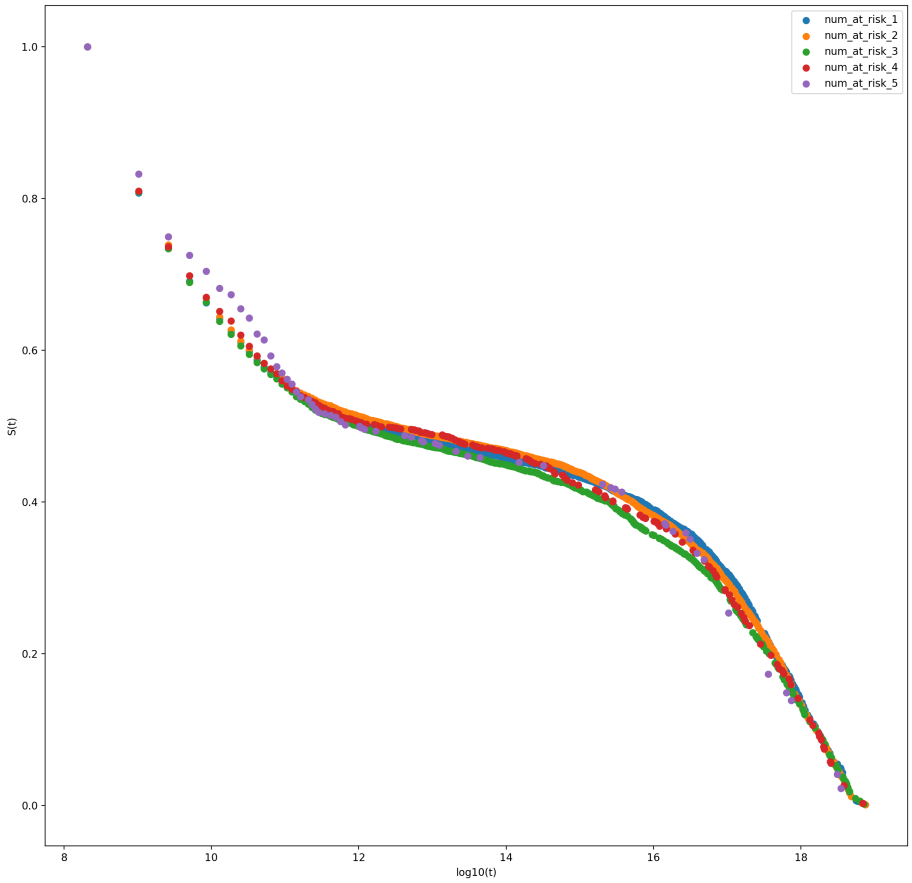


Figure 4.4: Kaplan-Meier survival curves for different amount of tags

curves and the tests to account the adequacy of the model and the verification of the hypothesis of proportional hazards.

The table 4.10 shows the results of the model training, the colors indicate the effect of the feature on the survival time, where the values grater than 1 mean that the feature increases the risk and the ones lower than 1 indicate decrease of the risk.

The model validation tests are shown below:

The results shown on table 4.10 confirms the validity of the model.

	TagCount	TitleLength	logPosterReputation	PosterAge
β	0.022197	-0.002871	0.307105	-0.004196
$\exp(\beta)$	1.022446	0.997132	1.359484	0.995813
stdErr	0.008800	$4.016536E-4$	0.007247	$9.504440E-4$
Z	2.566363	-7.148435	42.37272	-4.414734
p-value	0.011660	$8.777423E-13$	–	$1.011343E-5$
lower_CI $100(1-\alpha)\%$	0.004948	-0.003658	0.292900	-0.006059
upper_CI $100(1+\alpha)\%$	1.973361	1.956691	2.259821	1.954818

Table 4.9: Cox PH Model parameters

log-likelihood	-124565.0716
AIC	249138.1433
Cox & Snell	0.0808
Wald Test	1966.6316
Likelihood ratio test	2022.9469
Log rank test	2011.5584

Table 4.10: Cox PH Model parameters

4.6.3 Algorithms scalability and stability

The algorithms have been executed both in standalone mode and using a Standalone Spark installation. This experiment is intended to prove the scalability of the problem, as it can be parallelized through a Spark cluster. The idea of comparing with the standalone mode is to provide a baseline to start with. The other important feature to prove is the models stability, having the same input parameters, obtain similar execution times and results.

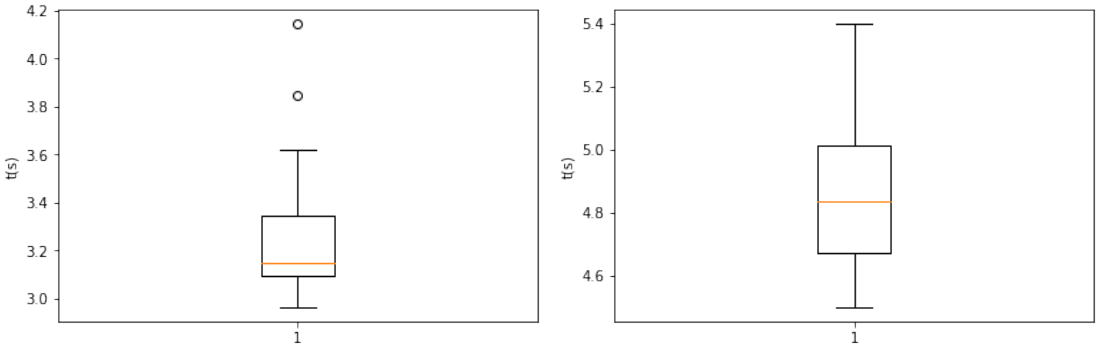
Stability

To have enough data for comparison, 30 executions of the algorithm have been done. Two stability parameters have been tested: execution time and consistency of the results.

The test was executed on 24000 samples with no missing values. For Kaplan-Meier,

one feature was chosen to make the groupings, the data below corresponds to the same parameters on execution. For the Cox PH model, all the 4 variables were introduced as covariates.

The times obtained on the execution with these parameters are resumed on the following graphs:



(a) Cox model

(b) KM model

Figure 4.5: Execution on Spark, time results.

To compare the Kaplan-Meier's results, the survival curves have been printed:

As can be seen on the image 4.6, all the executions provided the exact same results, although there were time variations.

To compare the Cox PH's results, the betas of the models have been compared, all te executions have returned the exact same values for each parameter, comfirming the consistency of executions of the algorithm.

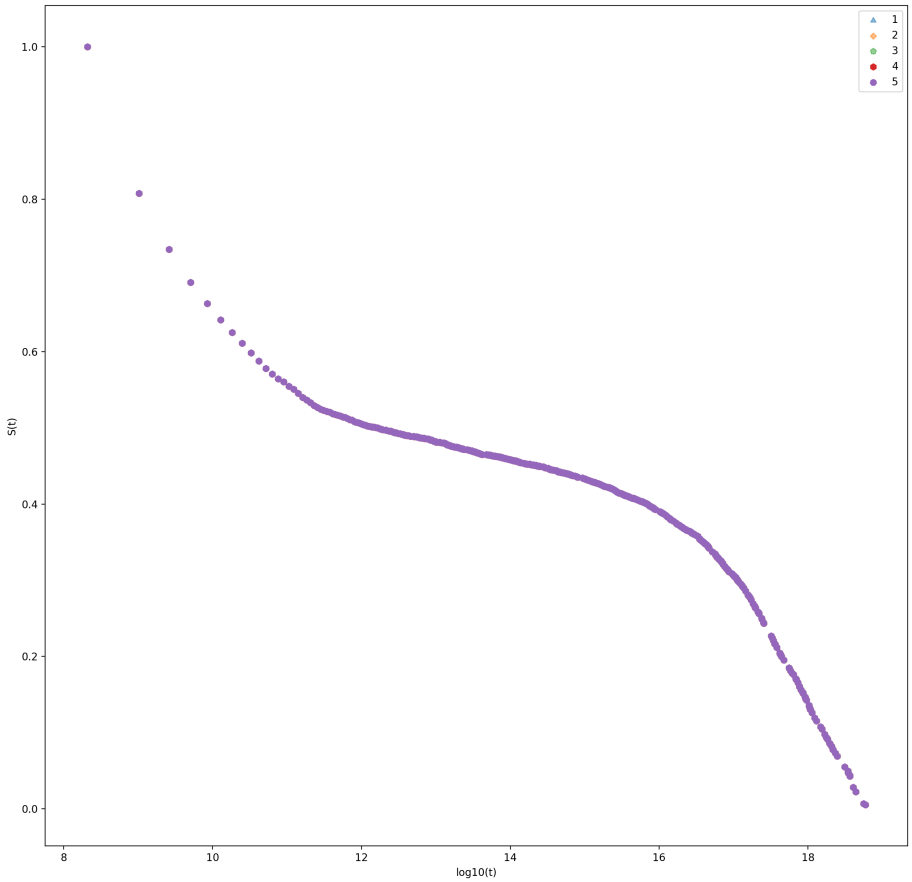
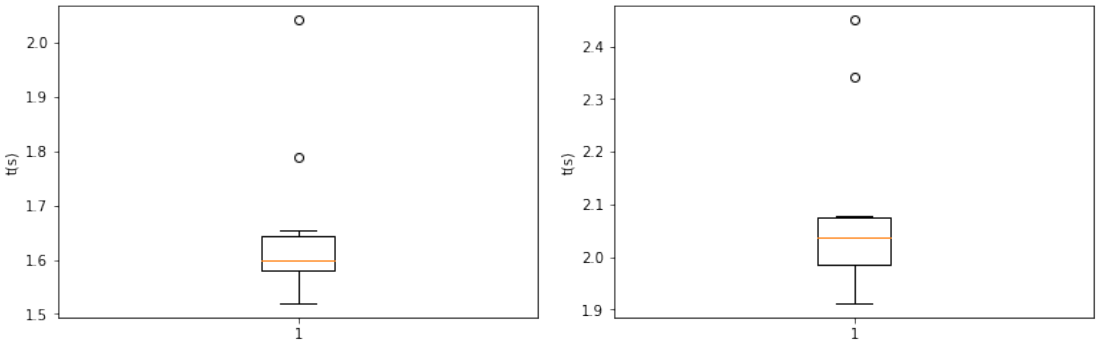


Figure 4.6: Kaplan-Meier results of 5 iterations of the 30 executed

Scalability

Regarding the scalability, two different executions have been done for both KM and Cox PH models, the first one with a big dataset, with 24009 instances, which returned the times shown on the previous image 4.7, the second one using a smaller dataset with 3000 instances. The results are shown below:



(a) Cox model

(b) KM model

Figure 4.7: Execution on Spark, time results.

Chapter 5

Conclusions

The three main objectives were automatize with Spark the data preprocessing to be able to scale up to a big Stack Exchange community, check the integration of Apache SystemML with Apache Spark and use surviving analysis techniques to explore Stack Exchange's data, with the final goal to predict the expected survival time for a question. All these objectives were met, with special focus on the infrastructure, the use and understanding of Spark and SystemML to be used in a scaled-up problem.

5.1 Most important results and lessons learnt

Most important results can be separated in three main parts, the scripts to preprocess the data and perform an exploratory analysis using Spark and Jupyter Notebook for granting scalability, the integration with Spark and learning of SystemML, and the application of Survival Analysis techniques to the Stack Exchange data.

5.1.1 Preprocessing the data

Scripts have been generated to convert the xml formatted data to csv, and adapt the tables information to the algorithms input. Moreover, a basic exploratory analysis have been performed, everything prepared to scale-up except for the visualization part, which runs in python libraries. These notebooks, along with the configuration process explained, allows anyone to perform and test the same analysis.

5.1.2 SystemML

The verification of the integration of SystemML on version *0.14.0-incubating* with Spark on version *2.1.X*, grants the scalability of the problem for bigger communities and further

analysis. Although SystemML is still in development, and some bugs have been found, regarding Kaplan-Meier estimates model, when scaling up the problem, the overall behavour have been tested and proved to work. A small benchmark testing the consistency and initial scalability of the combination of Spark with SystemML has been performed, leading to successfull results in consistency on both algorithms and scalability on Cox PH model.

5.1.3 Survival Analysis techniques

The application of Survival Analysis techniques to Stack Exchange is the first step to validate the utility of these algorithms for solving optimization problems on the behavior of these networks.

Bibliography

[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] N. Breslow. "Covariance Analysis of Survival Data under the Proportional Hazards Model". In: *International Statistical Institute (ISI)* (1974). DOI: 10.2307/1402659. URL: http://www.jstor.org/stable/1402659.

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

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

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

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

[12] B. Namoim et al L. Mamykina. "Design lessons from the fastest q&a site in the west". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (), pp. 2857–2866.

[13] *Material docente de la Unidad de Bioestadística Clínica.* URL: http://www.hrc.es/bioest.

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

[15] *StackExchange data schemas*. URL: https://meta.stackexchange.com/questions/2677/database-schema-documentation-for-the-public-data-dump-and-sede.

[16] 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.