TECHNICAL UNIVERSITY OF CRETE
School of Electrical and Computer Engineering

DIPLOMA THESIS

Scaling Text Processing Pipelines using Apache Spark

by

Merieme Katsani

Thesis Committee:
Associate Professor Antonios Deligiannakis (Supervisor)
Professor Minos Garofalakis
Associate Professor Michail G. Lagoudakis

A thesis submitted in fulfillment of the requirements for the degree of Diploma in Electrical and Computer Engineering.

December 2017

Abstract

Big data, which is derived from humans or machines, starting with social media and extending to smartphones or sensors, in forms of texts, images or transactions, is a continuously evolving field. Thus, the ongoing increase of data generated creates a need for knowledge extraction from it, through data analysis. Several areas are engaged in data mining, and in particular the area of machine learning which has been well established over the past years. Various techniques and methods of machine learning are trying to solve big data problems and these two areas consist now an integral part. This particular combination is the main subject of this study, which aims to implement a large-scale text processing architecture. More specifically, this architecture focuses on processing streaming texts derived from Reddit in real-time and the classification thereof as sarcastic or non-sarcastic through a machine learning model. The architecture uses the latest technologies in the field of information processing through distributed platforms such as Apache Kafka and Spark as well as state-of-the-art but also simple and powerful ML algorithms, i.e Random Forests, Naive Bayes and Logistic Regression. After comparing the methodology and design of each individual piece forming the final layout, a selection of the most appropriate model is made followed by the implementation of the framework. Success rates exported were quite close to the relevant literature and sometimes higher, depending on each technique examined. Finally, results are indexed in the distributed search engine Elasticsearch and are evaluated through the Kibana plugin.

Περίληψη

Τα μεγάλα δεδομένα (big data), τα οποία προέρχονται από ανθρώπους ή μηχανές, ξεκινούν από τα μέσα κοινωνικής δικτύωσης και επεκτείνονται σε “έξυπνα κινητά” (smartphones) ή αισθητήρες, σε μορφή κειμένου, εικόνας ή συναλλαγών, είναι ένα συνεχώς εξελισσόμενο πεδίο. Όπως προκύπτει, η διαρκής αύξηση δεδομένων που παράγονται δημιουργεί την ανάγκη για εξαγωγή γνώσης από αυτά, μέσω της ανάλυσής τους. Διάφοροι τομείς ασχολούνται με την εξόρυξη γνώσης (data mining), και πιο συγκεκριμένα ο τομέας της μηχανικής μάθησης (machine learning), ο οποίος έχει εδραιωθεί σημαντικά τα τελευταία χρόνια. Ποικίλες τεχνικές και μέθοδοι μηχανικής μάθησης προσπαθούν να επιλύσουν ζητήματα που πραγματεύονται τα μεγάλα δεδομένα και πλέον αυτοί οι δύο τομείς συνιστούν ένα αναπόσπαστο κομμάτι. Αυτός ο συγκεκριμένος συνδυασμός αποτελεί το κύριο αντικείμενο αυτής της μελέτης, η οποία έχει ως στόχο την υλοποίηση μιας αρχιτεκτονικής επεξεργασίας κειμένων σε μεγάλη κλίμακα. Πιο συγκεκριμένα, αυτή η αρχιτεκτονική επικεντρώνεται στην επεξεργασία ροών δεδομένων σε μορφή κειμένου σε πραγματικό χρόνο, οι οποίες προέρχονται από το Reddit και την ταξινόμηση αυτών ως σαρκαστικές ή μη σαρκαστικές μέσω ενός μοντέλου μηχανικής μάθησης. Η αρχιτεκτονική χρησιμοποιεί τις πιο νέες τεχνολογίες στον τομέα επεξεργασίας πληροφορίας με τη χρήση κατανεμημένων συστημάτων όπως είναι οι πλατφόρμες Apache Kafka και Spark αλλά και τελευταίας τεχνολογίας, καθώς και απλούς και ισχυρούς αλγόριθμους μηχανικής μάθησης όπως Random Forests, Naive Bayes και Logistic Regression. Αφού πραγματοποιηθεί σύγκριση της μεθοδολογίας και του σχεδιασμού κάθε επιμέρους κομματιού που απαρτίζει το τελικό σχεδιάγραμμα, γίνεται επιλογή του πιο κατάλληλου μοντέλου και ακολουθεί η υλοποίηση της δομής. Τα ποσοστά επιτυχίας που προέκυψαν ήταν αρκετά κοντά στη σχετική βιβλιογραφία και μερικές φορές υψηλότερα, ανάλογα με την εκάστοτε τεχνική που εξετάζεται. Τελικά, τα αποτελέσματα ευρετηριοποιούνται στην κατανεμημένη μηχανή αναζήτησης Elasticsearch και αξιολογούνται μέσω του Kibana plugin.

Contents

Chapter 1

Introduction

It is becoming more and more widely perceived that we are living in the Information Age. Enormous amounts of data are being continually generated and studied in numerous engineering and science domains. Particularly, due to the huge, unprecedented dissemination of data through social networks, various social phenomena arise over time, which can be explored and observed in detail, and informative insights can be provided about it based on these observations. Therefore, so-called "big data" becomes a hugely important aspect of reality and its management is considered imperative.

Large-scale data collection and the need to obtain useful information from it necessitates the development of new and effective techniques to store and mine it. Artificial Intelligence (AI) and its leading edge Machine Learning (ML) comprise extremely useful data management tools for retrieving valuable information laying under the "umbrella" of big data by applying high developed algorithms and techniques.

This work presents a distributed and scalable architecture for streaming textual data analysis using Natural Language Processing (NLP) strategies which aim at detecting sarcasm through supervised ML types, including state-of-the-art classification models that improve existing accuracy and performance.

1.1. Motivation

Sarcasm sentiment analysis is an issue of significant challenge, which started from the field of human intelligence itself before it even reached the area of NLP [10]. It exhibited a great interest over the past decade, consisting a combination of both fields of big data and AI. Although there has been a relevant progress in the field of sarcasm detection in recent years, most researches focus on studying Twitter [26] texts without exploiting other social media data such as Reddit [24] comments, that can be proven quite reliable and productive for extracting information, as will be subsequently demonstrated.

Additionally, big data dominates almost all areas of modern technology, with streaming data taking the lead and increasing the demand for effective data handling techniques. The perplexity in this case arises from the fact that amounts of streaming data are becoming illimitable over time, as well as the various sources from which they are being generated, thus complicating even more their processing. Over the past years, a variety of distributed technologies and frameworks has been developed to process and store big data, which should be used appropriately depending on the requirements and expectations of the systems to be realized. The need to create such a system in conjunction with the aforementioned reasons constitute the most fundamental motivation of this study.

1.2. Related Work

Sarcasm detection is integrated with the area of sentiment analysis. In spite of the fact that sarcasm detection research is still in its infancy, numerous substantial studies have been

conducted concerning either the detection process itself or the creation of accordant data samples. A time pass on the researches that have been accomplished precedently is presented in the survey of Joshi et al. [7], as referred below.

Remarkable progress has been observed from the initial steps of sarcasm detection starting from speech to text and other data forms exported from various sources including social media, with Twitter being the most prevailing among them. Ptáček et al. [14] performed sarcasm detection through supervised ML methods on Czech and English Twitter using the Maximum Entropy (MaxEnt) and Support Vector Machine (SVM) classifiers. Buschmeier et al. [3] presented a classification approach to irony detection in product reviews, by comparing state-of-the-art classifiers comprehending Naive Bayes and Random Forest.

Considerable research has been carried out in the case where large-scale data streams arise, making processing or storing them through efficient frameworks essential. Bharti et al. [1] proposed a sarcasm detection structure, based on Hadoop. In a more general context, a plethora of studies has been performed concerning the area of opinion mining (OM) and more specifically in the subject of emotional analysis. Khuc et al. [11] described a distributed system for Twitter sentiment analysis, using a MapReduce algorithm. Another method for sentiment analysis of tweets was introduced in [12], which, unlike preceding works, is based on Apache Spark.

1.3. Thesis Outline

Chapter 2 describes every single component exploited to realize the pipeline. The theoretical background and the software tools used are principally analyzed. Chapter 3 provides a comprehensive and detailed synopsis of the approach suggested, focusing on how it was implemented. Experimental stage is illustrated in Chapter 4, including study and results retrieved from it. Finally, the conclusion is given in Chapter 5, along with suggestions for future work.

Chapter 2

Theoretical Background

This chapter introduces several basic ideas so that the reader can familiarize themselves with the following work. More precisely, reference is made to the theoretical background in concepts deemed necessary as well as to specific tools and components used for the implementation.

2.1. Sarcasm

Various versions of the meaning of the term stand, justified by its special nature. Sarcasm consists *"a way of speaking or writing that involves saying the opposite of what you really mean in order to make an unkind joke or to show that you are annoyed"*, according to LDOCE, a definition that coincides with the use of the term these days. It is worth noting that sarcasm is often used to perceive a particular position or to make a point in a more abstract context.

A fair amount of confusion that deserves to be noted has surrounded the issue of the relationship between sarcasm and irony. Two similar definitions of irony are given by the same dictionary as before, and one of them looks almost identical to the one given for sarcasm: *"when you use words that are the opposite of what you really mean, often in order to be amusing"*. Several cases indicate sarcasm as a subcategory -or a special case- of irony, while others, such as the OED, consider the opposite resulting in divided views.

2.2. Big Data

The term probably dates back to 1999 [4], where the problem of quite large data sets that were taxing the capacities of main memory, local or even remote disks, appeared. Despite the references to the mid nineties, the term became widespread as recently as in 2011, study of Gandomi & Haider [5] reveals, as an evolving concept that describes *"large volumes of high velocity, complex and variable data that require advanced techniques and technologies to enable the capture, storage, distribution, management, and analysis of the information*".

A perception around big data that prevails on a large scale is the idea of the three defining dimensions or the Three V's: Volume, Variety, Velocity and as technology evolves, other dimensions are also being mentioned. An interpretation of each of the dimensions follows below, based on the survey mentioned earlier.

Volume refers to the magnitude of accumulated data. Although big data doesn't equate to any specific volume of data, as it constantly changes, volume nowadays is often used to describe petabytes (PB), exabytes (EB), zettabytes (ZB) or even yottabytes (YB), which is one septillion or 10^{24} bytes of data captured over time and in the future, it is estimated that brontobyte (BB) will be the most suitable data measurement unit.

Variety refers to the diversity in a data set. Different types concerning structured, semi-structured and unstructured data are being generated and captured from various sources including social media, Internet of Things (IoT), weblogs, transactions etc. Structured data refers to any data that is represented according to a strictly defined schema, including data contained in relational databases and spreadsheets and represents only 5 to 10% of all existing data.

Semi-structured data is information that doesn't reside in a relational database. It may have some structure but not necessarily the exact same for all data. XML, a language for data encoding on the web, is a typical example of semi-structured data. Unstructured data is information that is not organized according to a specified schema. Examples include photos, videos, audio files and it represents 80 to 90% data that organizations process daily.

Velocity refers to the rate at which data is being produced, stored and analyzed. The massive and continuous flow of data underscores the need to process it quickly with time consisting a determining factor, thus real- or near real-time processing and analysis of data raise an additional challenge. Accordingly, velocity consists the most significant component in the rapid and immediate transfer of data to the recipient.

Apparently, management and storage of huge amounts of data instantaneously is not feasible utilizing existing systems, therefore the development of new technologies and architectures becomes essential. The now commonplace concept of distributed systems consists the solution to this problem. A distributed system is described [15] as a cluster of autonomous and collaborative computing machines, called nodes, referring to either the software or the hardware side, constituting an integrated entity. A few state-of-the-art, open-source distributed frameworks used to realize this work, are presented below.

2.2.1. Apache Kafka

Apache Kafka [17] is an open-source, distributed messaging system, developed by the Apache Software Foundation for building real-time data streaming pipelines, written in Scala and Java. Kafka was originally built by LinkedIn for log data analysis [9] and eventually ended up as an open-source software in early 2011. Jay Kreps, who was working at LinkedIn at the time, named it after the acclaimed novelist Franz Kafka. An overview of some basic Kafka components, depicted in Figure 2.1, will be introduced below.

Kafka is by definition a distributed system, thus it is executed on a cluster of servers consisting of one or multiple computer units, each of which is called a broker. Kafka takes advantage of the Apache ZooKeeper synchronization system, for operational services such as maintaining and coordinating the cluster. It is considered extremely useful as it guarantees high-throughput combined with low-latency which essentially means hundreds of thousands of reads and writes per second and a large volume of messages reaching the order of TBs, while maintaining performance. In what follows below, basic terminology and ideas that need to be analyzed to better understand Kafka are presented.

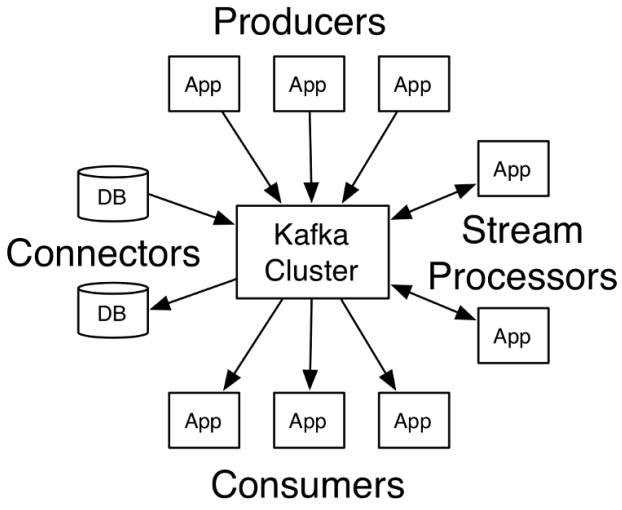


***Figure 2.1**: Kafka's four core APIs*

Kafka constitutes a messaging system operating according to the publish-subscribe model. More precisely, messages sent by Kafka, are organized in categories called topics. Each topic represents a stream of records and each record consists of a key, a value and a timestamp. Processes called producers send messages by publishing data to a topic and processes called consumers read messages by pulling data from a topic they have subscribed to. In other words, producers push data to brokers without waiting for acknowledgment and consumers read data from brokers. A topic is accessible to various consumers that subscribe to the data written to it.

Topics are split into partitions, as shown in Figure 2.2. Each message within a partition acquires a unique, sequential id number called the offset. For each topic, Kafka keeps a minimum of one partition. Parallelism in Kafka is achieved through the distribution of partitions by splitting the data over the servers in the cluster. It is also worth noting that each partition is replicated across a configurable number of servers for fault tolerance.

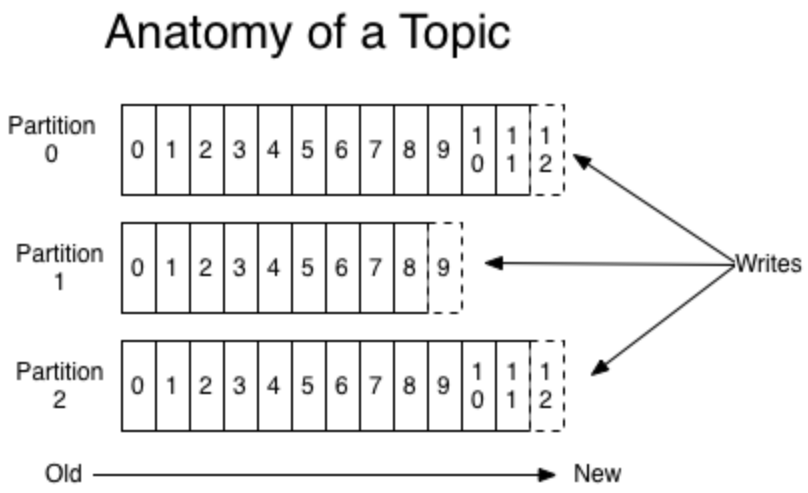


***Figure 2.2**: Anatomy of a Topic*

Among other things, Kafka also offers the ability for real-time stream processing through the Streams API provided, an excellent tool for generating output data streams to topics and consuming input streams from topics.

Moreover, later Kafka releases support interacting with external sources and systems such as databases or key-value stores, through the Connector API, which establishes the connection between these systems and Kafka topics by utilizing reusable producers or consumers.

2.2.2. Apache Spark

Apache Spark [18] is a high-performance, open-source cluster computing system, written in Scala. It was developed in 2009, at the University of California, Berkeley's AMPLab for processing data on a large scale. Spark was donated to the Apache Software Foundation in 2013, which has maintained it since. It came to complement several previous frameworks related to distributed big data processing such as MapReduce and was implemented based on already existing data-parallel processing interfaces such as Dryad [16].

The purpose behind its creation was to cover gaps and difficulties encountered in specific applications that required repeated data access, in particular as regards speed. Assuming iterative ML algorithms as an initial point of reference and the query resolution process as another, the development of Spark, a faster platform which would retain the advantageous properties of existing systems, was considered imperative.

MapReduce model, the heart of the Apache Hadoop framework, notes some performance issues related to the above mentioned categories, due to the continued disk accesses performed to obtain or reload the data needed across multiple parallel operations that cause latency. Spark was built to expand MapReduce model while preserving important properties such as scalability and fault tolerance. Spark runs everywhere, including Hadoop, Mesos, standalone or in the cloud and it can access data in Hadoop Distributed File System (HDFS), Cassandra and any Hadoop data source.

For a better understanding of the architecture behind Spark, it is worth mentioning the basic concepts that compose it. Resilient Distributed Dataset (RDD) consists a primary abstraction which was created to achieve fast computing in a distributed environment. More specifically, RDDs are collections of elements partitioned across cluster nodes that can perform parallel operations and can be rebuilt if a partition is lost because of a node failure. They can be created in four ways: from a file that is inserted in a shared file system, such as the HDFS· by parallelized collections, resulting from dividing the data into pieces and distribute them across multiple nodes· by applying deterministic operations called transformations· by interfering with the persistence of an existing RDD by changing its time duration. They are immutable, that is they don't change once created and they are read-only as they can only be transformed.

RDDs perform operations that are divided into two different categories: transformations and actions. They both contain a set of functions that differ according to how they are executed. More precisely, transactions are those commands that, as their name implies, transform an existing RDD into a new RDD, while actions are the functions that return the final value to the driver program after they have performed all preceding transformations. Behind transformations lies the idea of *lazy evaluation*, a determining factor that enables Spark to run efficiently. According to this idea, the execution of each individual transformation is happening only when an action is called, thus no separate RDD created through transformations, is returned to the driver program.

As a matter of fact, in order to understand how Spark managed to overcome the speed of Hadoop, the concept of in-memory processing, as depicted in Figure 2.3, should be introduced. Spark runs applications up to 100x faster in memory and 10x faster on disk than Hadoop because Spark caches data in Random Access Memory (RAM) instead of slow disk drives. To wit, Spark processes the data by storing the intermediate results in memory, unlike Hadoop that stores them in the disk, given that disk Input/Output (I/O) operations, i.e. moving data from where it is stored to where it needs to be computed, cause the largest time cost in any computing system.

An application in Spark is expressed in the form of an advanced Directed Acyclic Graph (DAG), another optimization factor, which exists to ensure the optimal manner of its performance. DAG is a collection of nodes connected by edges with definite direction and no circular dependency, where nodes represent the RDDs and the edges represent the transformation operation to be performed. Since Spark operating key idea is lazy evaluation, nothing actually happens unless an action operation is applied, as previously noted. When an action operation is executed, which is the last step before the DAG is formed, Spark goes back to the initial steps that were chained together from different RDDs connected to each other and, based on that, it figures out an execution plan or, in other words, it constructs the DAG that will lead to the desired results optimally.

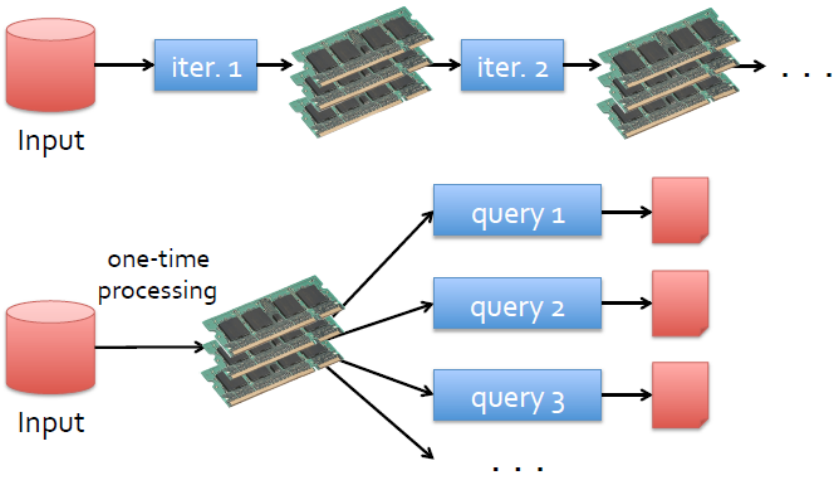


***Figure 2.3**: In-memory data-sharing*

Below is cited a brief introduction regarding the elements of the Spark ecosystem, as illustrated in Figure 2.4. Apache Spark Core API is the base of the execution engine for the development of Spark applications, on top of which is built all other functionality, and provides Java, Scala, Python and R APIs. A description of libraries running on top of Spark follows subsequently.

It is a general-use platform, therefore it offers many potentialities, designed to fill gaps in existing shortcomings. For example, Hadoop offers batch processing rather than real-time processing and Storm offers exactly the opposite, focusing on stream processing or complex event processing. Spark covers the gaps of both systems, offering both batch and real-time stream processing. Spark Streaming is used to process data streams in real-time, which is actually based on micro-batch processing. It readily integrates with a wide variety of popular data sources, including HDFS, Flume, Kafka, Twitter and consists the tool that was mainly utilized in this work.

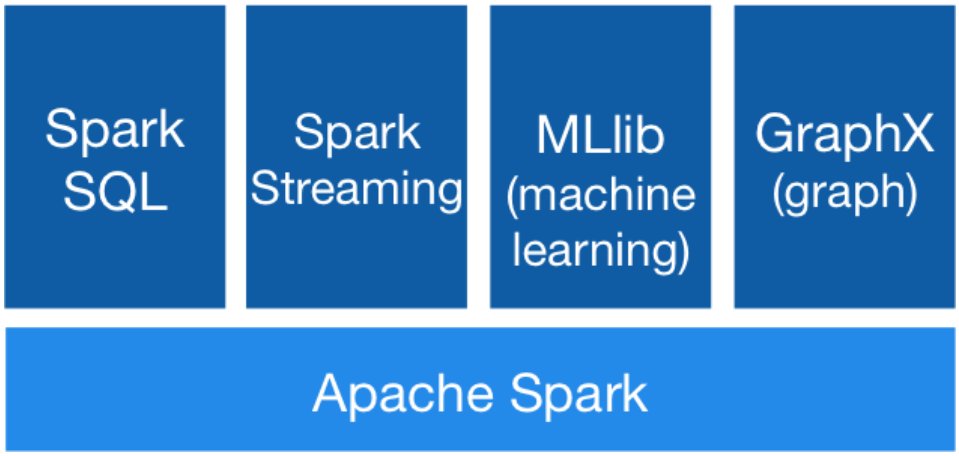


***Figure 2.4**: Apache Spark ecosystem*

As discussed in a previous paragraph, interactive data analysis was a problem that has been attempted to be resolved by the creation of Spark. Users were used to expressing a query, waiting for a response to it, then specifying the query based on that response and continuing until a satisfactory result is reached, i.e. an iterative procedure describing the interactive query analysis for data exploration. One of the newest pieces added to the puzzle of Spark libraries is the Spark SQL module, designed to provide an efficient solution to the previous problem of conventional data warehousing Relational Database Management Systems (RDBMS) users.

The field of ML, which has dominated the modern era, was another significant reason that led to the creation of Spark. In an attempt to improve the speed in ML models which are mainly composed of iterative algorithms and overcome the I/O bottlenecks of MapReduce discussed earlier, a scalable, ML library called MLlib was designed. MLlib guarantees faster computing in a fault-tolerant manner by exploiting in-memory data processing. It contains high quality learning algorithms and utilities and can be used in complete workflows.

Another key element included in the ecosystem of Spark is the GraphX computation engine. It provides an API to improve performance in iterative algorithms included in graph and graph-parallel computation problems over traditional MapReduce programs, as mentioned above. Among other things, GraphX is a growing library of the fastest graph algorithms which simplify graph analytics tasks, while retaining Spark's flexibility, fault tolerance, and ease of use.

2.2.3. Elasticsearch

Elasticsearch [19] is an open-source distributed search-engine built on top of Apache Lucene, firstly released in 2010. A platform developed by the Elastic company, alongside Logstash and Kibana, which are all designed to operate as an integrated package. It is used for storing, searching and analyzing large volumes of data in near real-time and is mainly preferred for applications that require complex queries.

Elasticsearch provides the ability to run queries on various kinds of data really fast as it is based on indexing. More specifically, the idea behind the platform is to store data in the form of documents, which can be expressed in JavaScript Object Notation (JSON) format. Each document is inserted into an index, an abstraction which can be associated with the common databases of

RDBMS, therefore an index consists of a set of documents that denote marked similarity based on a characteristic, e.g. an index for a product catalog.

Since it is a distributed system, it enables the ability of partitioning indexes into smaller pieces called shards in a computing cluster to allow large-data scaling and improve performance through parallel processing. In a realistic system, there is a great chance that a node will fail, which is why the concept of replicas of shards or replicas for short, exist, which are created on a different node than the primary shard to be available for any eventuality. By default, each index in Elasticsearch is allocated five primary shards and one replica, a number assigned at the time an index is created.

Among other things, Elasticsearch provides a compelling RESTful API for interacting with the cluster but also supports many programming languages such as Java, Python etc. Fundamental features provided by the API include managing and controlling statistics and information about the nodes and their progress or performing Create, Read, Update and Delete (CRUD) operations and exceptional tasks and queries against indexes.

The analysis and representation of huge amounts of useful information, which are stored in Elasticsearch indexes, through attractive and understandable dashboards are supported by Kibana, a plug-in for interactive data visualization. It is an open-source, browser-based data exploration engine for searching, analyzing, and presenting data through easy-to-use charts, tables, and maps. It offers services such as real-time analytics to explore massive amounts of data really fast and gain intelligence about them.

2.3. Machine Learning

Many definitions have been given to understand the concept of ML, an integral part of modern computer science, with that of A. Samuels in 1959 as the original but also widely accepted: *"Field of study that gives computers the ability to learn without being explicitly programmed"*. It is an extremely popular and growing field of AI and emerged as a consequence of the big data development as well as the continuous technological evolution that followed the rising of the internet.

The massive flow of data produced in recent years and the need to extract intelligence from it, has urged the development of information exploitation areas such as ML. The idea behind ML is associated with the masterful functionality of the human brain regarding the learning process and how it can be imparted to a computing system. Nevertheless, human brain presents bottlenecks, e.g. the complex task of decrypting it or how time-consuming the human learning process is by nature, problems wherein ML should provide solutions.

Among other fields, ML is closely connected with NLP, as it is a process of learning and understanding concepts arising from natural-language perspective utilizing speech and writing data, tools used for human communication. NLP is an area that deals with the human-computer interaction, focusing on the computer learning process through natural language data residing on the internet, to gain insights from it. ML consists of an area that deals with techniques and methods applied to many NLP related problems, functioning as a complementary element.

There are many algorithms developed for modeling problems which can be categorized depending on input data, as well as the purpose they serve. In general, ML algorithms include input or training data according to which the model is trained so that it can be able to learn patterns and subsequently make predictions about future or testing data. Most well-known categories include supervised, unsupervised, semi-supervised and reinforcement learning.

Supervised is a learning type where the training data has known labels, according to which functions are produced and afterwards used for testing new data e.g. classification and regression problems including algorithms as Naive Bayes, Decision Trees, Support Vector Machines (SVM) etc. Unsupervised methods include unlabeled input data with no expected outcome, e.g. clustering problems including k-means algorithm. Semi-supervised learning is a mixture of the two previous categories, where input data consists of both labeled and unlabeled elements and reinforcement is the learning where the model is called to achieve the most efficient behavior for it, exploiting signals coming from its environment.

2.3.1. Scikit-learn

Scikit-learn [25] is an open-source python library built on top of NumPy, SciPy and matplotlib ecosystems, used in medium-scale ML problems. The project was initially started by Cournapeau in 2007 and made its first public release in 2010 by Pedregosa et al. [13]. It is in its largest part written in python and is widespread for its ease of use, performance and rich documentation. It provides a huge range of state-of-the-art algorithms for supervised and unsupervised learning including the categories of classification, regression, clustering etc. This diploma focuses on the supervised ML algorithms provided through scikit-learn.

Each algorithm includes an estimator, which consists the main component of it and each processing method is implemented around him. Since scikit-learn is a package that deals with supervised and unsupervised learning, what is expected is a model which goal is, given input data called a training set, to produce a function that has been trained accordingly, so that it becomes a "predictor" for future data called a testing set. Data and model parameters are presented as numpy arrays.

Estimators realize the above process through the "fit" method and are able to predict the final result through the "predict" method, when learning is supervised. Furthermore, an estimator can provide methods that measure the success rate of the prediction such as the "score" method, which calculates the accuracy. Some other significant features provided by scikit-learn is the efficient parametres setting strategy called hyperparameters turning with the "GridSearchCV" method where the CV stands for "cross-validation iterator". Parametres are the arguments passed to the constructor of an estimator and can be optimized depending on the requirements of the problem.

Cross-validation method provides the ability to divide training data into train set to train the model and test set to evaluate its performance, which gives a solution to overfitting problems, occuring when the training set is tested on a model already trained by it. Another capability of maximal importance provided by scikit-learn is the combination of all of the above in an element called "Pipeline" and its purpose is to minimize steps and cross-validate them together, while continuing to behave as an estimator.

It should be noted that in ML problems only numeric characters and vectors are used, not texts. A technique for converting text to numeric characters used in this task is called Bag-of-Words or BoW. In its simplest version, this technique measures words in documents and creates a vector which contains numbers that correspond to the frequency with which each word appears in the document. BoW is implemented by the TfidfVectorizer element.

Finally, once the process of creating an estimator has been completed, it is desirable to be able to store the model in a way that is reusable but also for its integration into external systems. Such a way is python's built-in *pickle* module, used for serializing and deserializing an object. During serialization, the structure of the object is converted into a stream of bytes while at deserialize it returns to its original form.

2.3.2. Naive Bayes

Naive Bayes is one of the most popular but rather simple algorithms based on the Bayesian theorem with the "naive" assumption that features are independent of each other. It is a fairly easy and useful method to text classification problems, as it is characterized by high speed even on massive data amounts, a key reason why it was chosen. This classifier relies on the BoW model and involves features probabilities. Bayes' theorem is described by the relationship presented in 2.1, given a feature vector X = {x_1, … , x_n} and a class variable y:

$$P(y \mid X) = P(y \mid x_1, ..., x_n) = \frac{P(x_1, ..., x_n \mid y)\, P(y)}{P(x_1, ..., x_n)} \qquad (2.1)$$

where:

- Conditional probability $P(y \mid X)$ is called the "posterior probability" and describes the probability of an observation belonging to a particular class, given the testing data.
- Conditional probability $P(x_1, ..., x_n \mid y)$ is called the "likelihood" and describes the frequency of an observation appearing in the training set, given the particular class label, and, in this case, $P(x_i \mid y, x_1, ..., x_n) = P(x_i \mid y)$.
- Probability $P(y)$ is called the "prior probability" of the label class and describes the occurrence of a particular label class within the training set.
- Probability $P(X) = P(x_1, ..., x_n)$ is called the "prior probability" and describes the occurrence of a particular observation within the training set.

Therefore, by applying the Maximum A Posteriori (MAP) estimation and dropping the denominator, the relationship 2.1 ends up with a solution of the form:

$$\hat{y} = argmax_y\, P(y) \prod_{i=1}^{n} P(x_i \mid y) \qquad (2.2)$$

2.3.3. Logistic Regression

Logistic regression is the oldest and most widespread approach in binary classification problems. It was developed years ago to solve statistical problems and consists a rather straight-forward method used in various fields other than ML. This method was chosen to be applied to another ML model as it was considered suitable for the particular data set used in this work according to [8]. Furthermore it is a fast method that doesn't require feature scaling.

Assuming y as the label class, $y \in \{0, 1\}$ where 0 is the negative class, i.e. non-sarcastic text and 1 is the positive, i.e. sarcastic text. What is achieved by this method is the prediction of the output value by modeling it with a logistic function, which accepts the training data as arguments.

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}} \qquad (2.3)$$

The standard logistic function is the sigmoid function and that will be the learning algorithm or otherwise the hypothetical function $h_\theta(x)$ used to model the output, as described in 2.3 and depicted in Figure 2.5, where θ corresponds to the parameters of the algorithm.

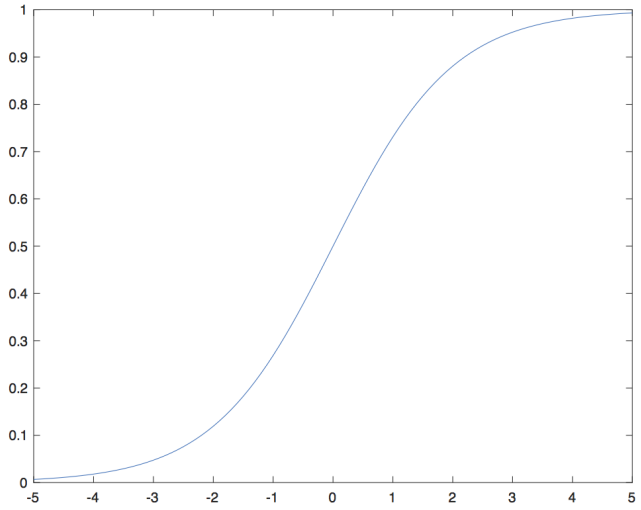


***Figure 2.5**: Sigmoid function*

If the conditional probabilities of each class are assumed as:

$$p(y = 1 \mid x\,; \theta) = h_\theta(x) \qquad (2.4)$$

$$p(y = 0 \mid x\,; \theta) = 1 - h_\theta(x) \qquad (2.5)$$

or

$$p(y \mid x;\theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y} \qquad (2.6)$$

Assuming that m training examples were generated independently, then the likelihood of the parameters would be equal to the relationship described in 2.7. For convenience, the maximization of the log likelihood is then calculated to estimate the regression parameters.

$$L(\theta) = \prod_{i=1}^{m} (h_\theta(x^{(i)}))^{y^{(i)}} (1 - h_\theta(x^{(i)}))^{1-y^{(i)}} \qquad (2.7)$$

2.3.4. Random Forests

Random decision forests or random forests consist a state-of-the-art and powerful ML model for both classification and regression tasks, firstly introduced by Ho [6] in 1995. Plenty new algorithms have been built since then, with the study of Breiman [2] in 2001, consisting one of the most important contributions. Some of its advantages are the speed of execution even on large amounts of data and the avoidance of overfitting the model thus are widely preferred in various applications, consisting a great competitor of other state-of-the-art models such as Support Vector Machines (SVM) etc.

A formal definition [21] given is: *"A random forest is a classifier consisting of a collection of tree-structured classifiers* $\{h(x, \Theta_k), k = 1, ...\}$ *where the* $\{\Theta_k\}$ *are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x."* Bagging is a basic concept behind random forests, which is essentially the combination of many classifiers and, in this case, the combination of many decision trees to provide therethrough a final prediction according to the average majority voting. Parameters $\{\Theta_k\}$ include the structure of the tree by determining which subset of the full dataset is chosen, which variables are split in each node, etc. and are chosen randomly. Therefore, the form of each classifier for each decision tree will be described by the relationship:

$$h_k(x) = h(x \mid \Theta_k) \qquad (2.8)$$

The empirical margin function describes the extent to which the average of the votes of the first correct class elected exceeds the average of the votes for any other class, a metric indicating whether the algorithm is reliable and is given by:

$$\hat{m}(x,y) \equiv P_k(h_k(x) = y) - max_{j \neq y} \hat{P}_k(h_k(x) = j) \qquad (2.9)$$

where $\hat{P}(A) =$ proportion of classifiers $h_k (1 \leq k \leq K)$ for which event A occurs.

Chapter 3

System Design

This chapter analyzes extensively the architecture according to which the subject of this work was realized and each separate stage of it. Initially, a brief introduction to a well-known architecture that was a source of inspiration is presented, followed by the assembly of the elements discussed earlier in an integrated pipeline. It is worth noting that all the processes that follow, as well as the whole architecture, are designed to be applied to large amounts of data, having as reference the 65-GB compressed bz2 testing file, but for reasons of non-access to a cluster of servers, the whole system is applied to smaller files indicatively.

3.1. Lambda Architecture

Lambda Architecture (LA) consists of a scalable and fault-tolerant structure that combines the capabilities of batch and real-time processing, introduced by Nathan Marz [22] in 2011. It was designed to provide answers to complex questions involving large amounts of data efficiently.

The system consists of five different elements, as illustrated in Figure 3.1. Input data is shared between both the batch and the speed layer for processing at the beginning. The batch layer is used for storing the entire dataset in a large-scale file system and precomputing queries, which is a straightforward procedure, so they are already answered when needed. Then, the results derived from it can be easily stored in a database, a component of the batch serving layer.

At the same time, the speed layer is used for processing the input data in parallel by utilizing a real-time data processing system that provides low latency and making it a much simpler procedure as it involves a small percentage of data instead of the entire dataset. Respectively, data coming out of the speed layer can be incremented in another database or a real-time serving layer and the last step is to merge the results obtained from each layer to answer any incoming query.

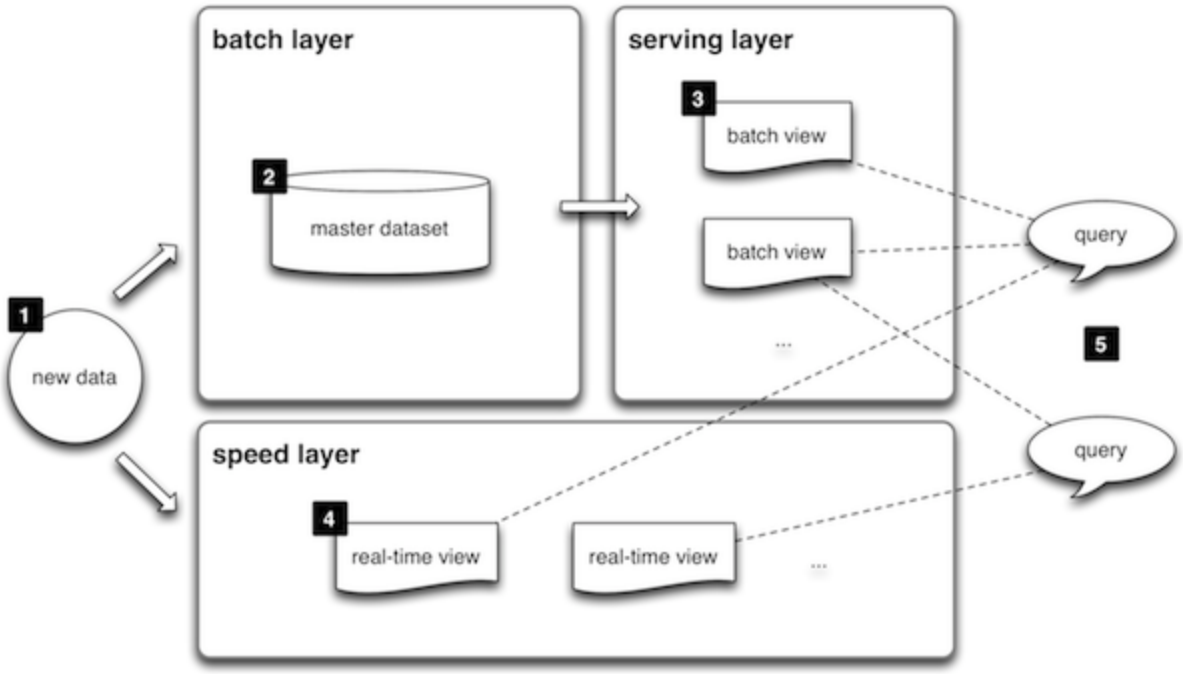


***Figure 3.1**: Lambda Architecture*

LA is a ubiquitous solution to a multitude of approaches, although many concerns have been also expressed due to the complexity it presents. As perceived, it is an approach that combines two different distributed systems which in itself is relatively complex.

According to the original idea, the proposed batch processing framework was Apache Hadoop and the corresponding real-time stream processing was Apache Storm, which both are tools that are not particularly easy to use by themselves, let alone combining these two and synchronizing them to produce a common result.

This hesitation is the main leading to the selection of Apache Spark framework, which is an intermediate solution combining the properties of the two above systems, as previously discussed. Therefore, instead of working on two different and complex platforms, only one could simply be used. Nevertheless, the subject of this thesis is real-time data processing, thus it focuses on the speed layer and its corresponding serving layer.

3.2. System Overview

The main objective of this study is the implementation of a distributed system, which receives and processes large amounts of real-time data streams, aiming at the detection of the sarcasm emotion by leveraging state-of-the-art supervised ML techniques. The difficulty of this problem lies in the enormous amounts of data to be used, in real time, or near-real time, and the implementation of an appropriate ML classifier, which will operate effectively in an integrated framework. The pipeline depicted in Figure 3.2, shows the approach according to which such a system was created.

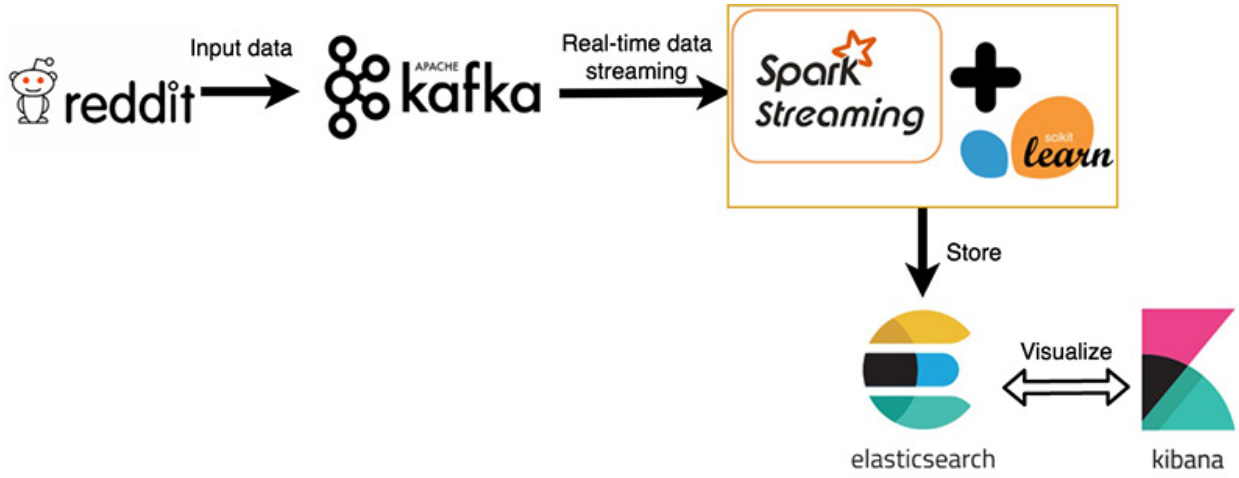


***Figure 3.2**: System architecture*

The diagram presents the general idea according to which the final system was created, consisting all of the tools used as well as the intermediate connection, shown in a more abstract context. As a matter of fact, six different symbols are distinguished, namely Reddit, Apache Kafka, Spark Streaming, Scikit-learn, Elasticsearch and Kibana as well as the interaction between them. Each interaction bears a title relating to the corresponding operation performed. A more detailed schema is shown in Figure 3.3.

As shown, the system is divided into four stages to better organize each component, which are based on different operations performed on data: management, which relates to the data source origin description, the pre-processing procedure, the storage and the representation in a data structure which is preparing the data for the next stage of ingestion, which concerns the connection with the previous stage and the data streaming to the next stage of processing and visualization, i.e. the final data processing phase, the application of the ML models as well as the data visualization on a dashboard. Each of the stages will be described in detail below.

Furthermore, the system is comprised of the following components: a dataset, which concerns the data derived from a data source, an inverted indexer storage structure which is actually a dictionary consisting of keys and values, a random value generator, an Apache Kafka producer of messages which acts as an input data stream source, a Spark Streaming processing system which includes a scikit-learn supervised ML model, an Elasticsearch search engine which is an indexing storage system and a Kibana dashboard for visualization.

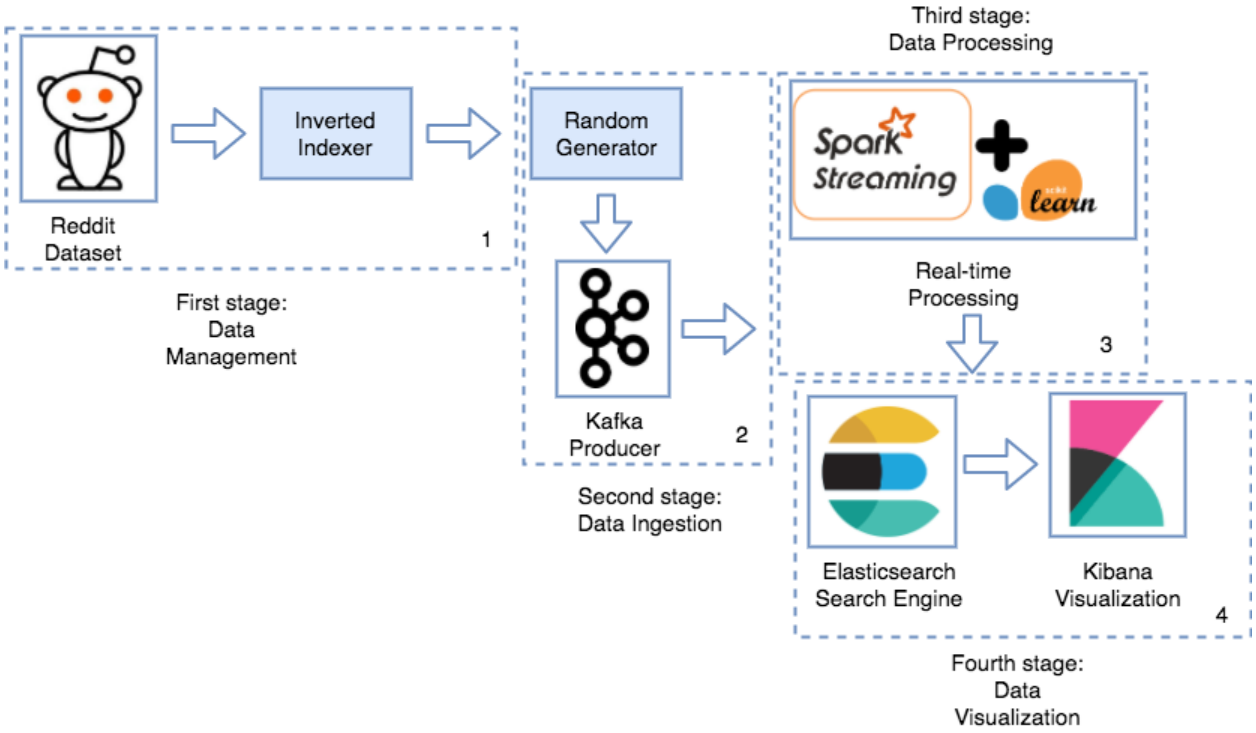


***Figure 3.3**: Detailed system architecture*

3.3. Data Management

The primary process of implementing the system is to review the input data and manipulate it accordingly.

3.3.1. Data Source

The main source of the data is the well-known website Reddit, founded in 2005. Reddit is used for discussions on a variety of issues related to news and areas of interest originating from social media and employs a large proportion of users. It is essentially a site formed by users as it consists of user posts or comments, according to which categories called “subreddits” arise, depending on different topics that comments deal with. Posts can have any form, from texts to photos and videos. Each user has the opportunity to comment on other posts and vote for or against another user. The final score determines which posts will appear higher on the subreddit and eventually on the front page.

Although the original source of the input data is Reddit, as mentioned earlier, the final dataset which was used in this work is the result of the study of Khodak et al. [8]. Self-Annotated Reddit Corpus (SARC) was created for the implementation of tasks related to the NLP area and more specifically to applications focusing on the detection of sarcasm. It mostly focused on the creation of the corpus and not on the development of sarcasm detection systems, although methods such as Bag-of-Words and Bag-of-Bigrams were used for training benchmarks indicatively.

The dataset consists of comments from January 2009-April 2017, which have been processed and filtered appropriately in such a way that they do not contain noise. For each comment there is a label containing the values 0 and 1, depending on whether or not the comment is sarcastic, which is determined by the author himself and other context information, which will be mentioned below.

Reddit was highlighted by the research as a valuable source of data, even more remarkable than others such as Twitter which is used extensively, as it contains higher quality material. Reddit comments have no restrictions on their size, contain more "/s" hashtags, according to which users state that their text contains a sarcastic meaning and are written in a much more comprehensible and clear way than tweets, making SARC more reliable and realistic than other datasets.

The research came up with a 65-GB file containing raw data of various subreddits which is used as a testing set in this work. Furthermore, a 4-MB file containing training balanced data is used as a training set and no other files of data were used. Each entry has the same format in both files, i.e. label, comment, author, subreddit, score, ups, downs, date, created_utc and parent_comment. Finally, some ideas and methods suggested in the research for classifying the balanced data of SARC were considered during implementation.

3.3.2. Data Preprocessing

Regarding the file containing the raw data which was utilized as a testing set, a little preprocessing was made to facilitate subsequent processes. It is worth emphasizing that this is a great quality corpus and the data is presented in distinguishable form. However, since it is a large enough file that goes beyond the capabilities of a single computer machine, it was considered necessary to reduce it by holding 10 million of the 533 million rows included in the original dataset. A rather trivial process using scikit-learn in conjunction with the libraries contained in it. Also, the "label" column, containing values 0 and 1 for non-sarcastic and sarcastic comment respectively, was removed. One last preprocessing procedure that was performed was the removal of unnecessary "\n", "\r", "\t" (newline and tab) separators for data cleansing reasons.

3.3.3. Streaming Simulation

As mentioned above, data is stored in a 3.54-GB csv file which contains 10 million records. Since this thesis deals with the processing of streaming data in real-time its simulation into streams is considered necessary. Although Apache Spark is a general-purpose platform, it was designed for data processing and not for storage, thus Kafka platform was utilized, as a useful integration package between Spark Streaming and Kafka has been created for this purpose.

The first step for simulating data in streams, random access must be ensured, i.e. the ability to access any individual element that exists in a dataset quickly and efficiently, regardless of its size. Apparently, avoiding sequential search is a one-way solution in these cases and utilizing random access results in much better performance and higher speed. To fulfill this requirement a key data structure named "inverted indexer" was created.

An inverted indexer is a data structure that, given a search term, provides the ability to access all documents that contain that term. It consists the most widely used information retrieval

method by well-known search engines for fast full-text searching. It is essentially a dictionary based on records of the key-value form and its design depends on the specific hardware characteristics of the operating computer system.

The implementation of the indexer designed is shown in Figure 3.4, although it is worth noting that the indexer symbol has been zoomed in a bit in that figure for emulation purposes. The difficulty of creating such a structure concerns the size of the dataset file. In this case, the data volume is quite small compared to the original, so the process is relatively simple. However, the algorithm for creating the indexer was designed to work on larger files too, as a function of the system requirements for fast memory access.

Access to data is faster when it's in memory, rather than disk, which is a determining factor to be taken into account when implementing the inverted index structure. As discussed earlier, this architecture is intended for big data and specifically for the original 65-GB file, the size of which exceeds the capabilities of the computer in which the implementation takes place. Therefore, using the python language capabilities, an algorithm emerged which, instead of reading an entire file, reads small blocks of it and creates the corresponding index.

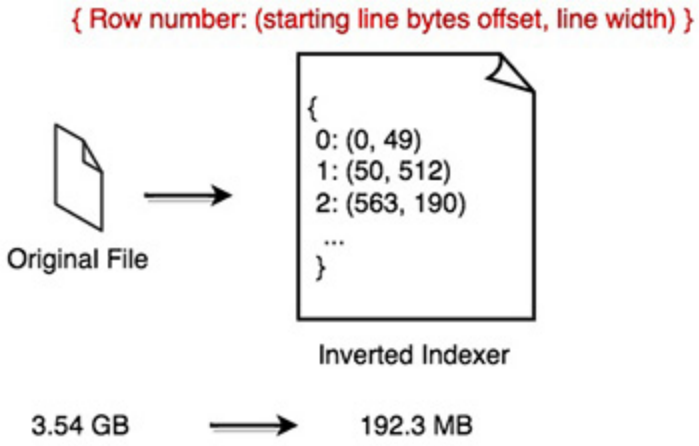


***Figure 3.4**: Inverted indexer*

According to the algorithm, the file is opened, thus made available for reading. Nevertheless, it is not efficient to read the entire file into memory, so only a small block size of 1024 bytes is read at each iteration until the last bytes of the whole file are read. Each iteration occurs for each new line found in the block, for which the bytes offset and width are calculated and stored in the indexer structure. There is no other structure in the implementation than that of the inverted indexer, as the use of fixed-length arrays or lists would be no memory efficient.

Therefore, a much smaller representation of the original file occurred, by using three integers instead of alphanumeric characters and which leads to fast random access as needed. The last step of this phase is the serialization of the indexer using the python pickle module, so that it can be transferred and utilized to the next step.

3.4. Data Ingestion

This stage is the interface between data pre-processing and real-time processing phase. As a matter of fact, the process of randomly generating data from Apache Kafka to Spark Streaming occurs.

3.4.1. Random Line Number Generator

A random number generator was created to implement random data access and generation. The serialized object of the index is initially loaded and then de-serialized. A random number generator with a range of zero to the size of the inverted indexer lines is then created. Random search of each line is achieved through the “seek” function, in which the starting line byte offset or position of the random line is given as an argument and through the “read” function, in which the line width of the same random line is given as an argument. In this way, the retrieval of information contained in each line is realized.

3.4.2. Message Producer

Since the data generation has been implemented, what is left is the phase of the data streams to be sent to the processing stage via the Apache Kafka messaging system. Messages are sent using the Kafka Producer API asynchronously through the “send” method, which means that when called, it adds the message to a buffer thus creating a batch of records.

To activate Kafka server, the ZooKeeper server must initially be called. Then a topic is defined and the number of messages to be sent is set. Kafka is characterized by high performance, i.e. low latency and high throughput even when running on a standalone mode, although its capabilities will be better understood in a computer cluster context. What happens when Kafka produces messages is they are partitioned across Kafka brokers or nodes thus parallelism is achieved. It is worth noting that the parallelism equals the number of partitioning occurring on a topic, which in the case of this study equals one.

Another feature of parallelism that could be used in this work is that of multi-producer calling, i.e. many producers can run in parallel on different nodes of a computer cluster and publish messages to one or more topics to achieve higher throughput, as depicted in Figure 3.5. Streams of messages sent from Kafka are in the key-value format where the value field is assigned with the random line content previously generated while nothing is assigned to the key field as no specific order of messages across partitions of the topic is required.

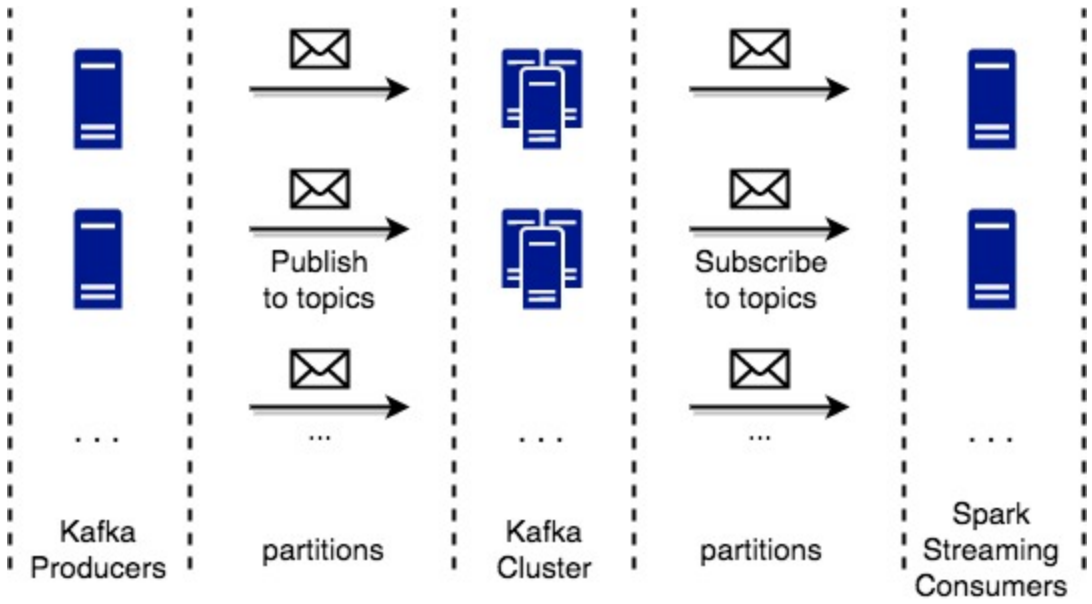


***Figure 3.5**: Parallelism in the current Kafka architecture and the transition to the next stage.*

3.5. Data Processing

At this point, which is the main part of the implementation, the processing of the messages through Apache Spark and the procedure by which the ML models were created are analyzed.

3.5.1. Real-time Stream Processing

The messages were sent by Kafka are now entering Spark Streaming in the form of streams for further processing. At first, connections with SparkContext and StreamingContext are made, which consist the two main entry points for Spark and respectively Spark Streaming functionality. As in fact, Spark Streaming operates at the base of micro-batch processing, or otherwise in near real-time processing, therefore a batch interval of one second is defined. Each second Spark receives a batch of data streams, just in the order they were sent from Kafka producer, as Kafka provides ordered messaging within a partition. It is worth noting that the order in messages is not provided between different partitions of the same topic though.

Streams or records of RDDs are entering Spark in the key-value format, so they need to be processed to get the final form to be applied to the ML classifiers. Initially, the features that are not used need to be separated from the “comment” field, through the “map” operation, which cuts the fields that are separated by “\t” tab separators between them. Then follows the application of the ML model through the “classify_line” function, after the serialized model has loaded

```
#Load the ML pickled model
sklearn_pipeline = pickle.load( open("/path/PIPELINE.pkl", "rb") )

#Predict sarcasm probability for each line
def classify_line(line):
    return sklearn_pipeline.predict_proba([line])[:,1]
```

and the presentation of the final tuples in the form of (comment, author, subreddit, sarcasm_probability) by calling an output operation for each of the RDDs of the stream(similar to actions for RDDs) and for each partition of the RDDs.

```
#Import KafkaUtils and create an input DStream
    kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consumer",
{topic: 1})
    values = kvs.map(lambda x: x[1]) # key-value pairs, keeping the value

    #Apply classification function and store to elasticsearch
    line = values.map(lambda comm: comm.split('\t'))
    sarcTuple = line.map(lambda item: (item[0], item[1], item[2], item[7],
classify_line(item[0])))
    sarcTuple.foreachRDD(lambda rdd: rdd.foreachPartition(store_to_elasticsearch))
```

3.5.2. Classification

Assuming sarcasm detection as a binary classification problem, since there are only two distinct values to be classified, it is considered necessary to model it using ML algorithms. This part concerns the data used as a training set, that is, the 4-MG file in bz2 format, which includes only balanced data. For each algorithm used, the same procedure of data preprocessing was applied.

Initially, the compressed file, which contains more than 50.000 rows, is read and a header with the column names is added and the final training set, which consists of input features represented by the “X” variable and output or “y” target variable is created. “X” variable corresponds to the “comment” column while “y” variable corresponds to the “label” column.

In order to define the ML problem that is encountered, it should be considered that given a “X” incoming dataset, a function or a classifier is trained through a learning algorithm, which will guess whether each new, unlabeled input sentence is tested is sarcastic or not, as shown in Figure 3.6. In fact, what is estimated is the probability of a testing sentence being sarcastic and it should be noted that this definition [23] refers particularly to the supervised ML problems, which this work deals with.

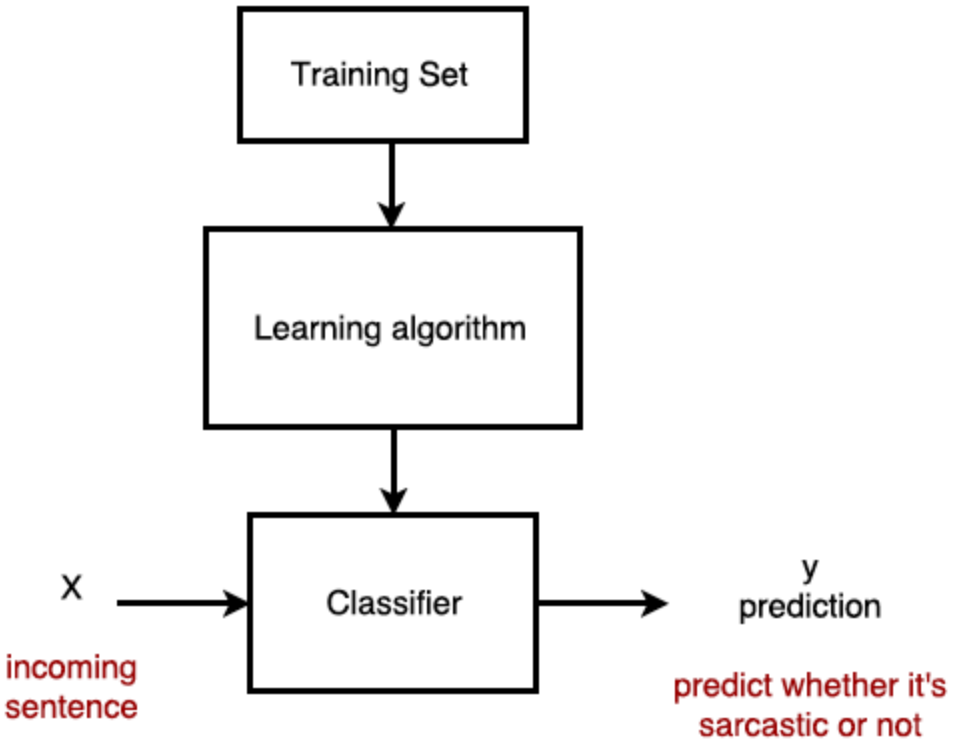


***Figure 3.6**: Supervised ML problem definition*

Subsequently, the data set is then split up into an 80% training set and a 20% testing set and a pipeline of transforms is created. The pipeline consists of a TfidfVectorizer and the classification algorithm implemented.

```
# Split X and y into training and testing sets
from sklearn.model_selection import train_test_split

# Split the dataset the exact same way every single time
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=42)
```

TfidfVectorizer stands for Term Frequency – Inverse Document Frequency (TF-IDF) and is a popular method for measuring word frequencies that appear in a document. “Term Frequency” refers to the times that a particular word appears in a text to the total number of other words and “Inverse Document Frequency” refers to the importance of each word, used to highlight more important words and degrade the unimportant.

However, since BoW does not deal with the order in which the words appear, another model is introduced, which is a generalization of BoW and is called N-gram. BoW is assumed to be identical to the N-gam model for N = 1. N-gram model is a sequence of N words and is used as a parameter of the TfidfVectorizer to clarify meanings or sentiments in pairs of words that as single

words are not clear. Furthermore, the max_features variable considers only the (max) terms that occur more frequently for building the vocabulary.

Finally, the training process follows, through the “fit” function and the final model is loaded into a pickle object to be applied to the Spark Streaming platform. The creation of ML models was realized using the scikit-learn package for python.

```
from sklearn.naive_bayes import BernoulliNB
pipeline = Pipeline([
    ('train_count_vect', TfidfVectorizer( max_features = MAX_FEATURES,
ngram_range=(1,5))),
    ('clf', BernoulliNB()),
])
start = time()
pipeline.fit(X_train, y_train)
```

3.6. Data Visualization

This is the last stage of the architecture and perhaps the simplest. Two basic processes take place at this level: (a) sending the tuples of data calculated to the Elasticsearch system and (b) displaying them on a dashboard through Kibana.

For the first part a function is created, which converts the data into json format, as this is the required from elasticsearch. More specifically, “sarcTuple” includes the following fields: comment, author, subreddit, created_utc and obviously the probability of sarcasm for each comment that was calculated. The data or the documents are organized in indexes of a specific type which is defined when creating the index. Each document has a unique id which is created by encrypting the comment, the author, and the created_utc, all concatenated into a single text. Data is then stored to elasticsearch via the HTTP PUT request method and verified by checking the response of the server.

```
def store_to_elasticsearch(iter):
    headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}
    #Parse each tuple as json
    for record in iter:
        comm, auth, subr, prob = record[0], record[1], record[2], record[4][0]
        doc = {
            "comment" : comm,
            "author" : auth,
            "subreddit" : subr,
            "sarcasm_prob" : prob
        }
        utc = record[3]
```

```
        #Encryption of the doc_id so that it is unique for each document
        doc_id =
hashlib.sha256((comm+""+auth+""+utc).encode('utf-8')).hexdigest()

        #Make HTTP PUT request
        es_record = requests.put(url =
"http://localhost:9200/sarcasm/classified/%s?op_type=create" % doc_id ,
                                                    data = json.dumps(doc),
headers = headers)
        #Check the response of the server
        if es_record.status_code not in [200,201]: #The request has failed
            print("------------- Error: %s -------------" % es_record)
            print("***************** doc_id: %s *****************" % doc_id)

    return es_record
```

For the second part, some representations were created using Kibana, which are shown in the next chapter. A complete representation of the distributed system implemented is shown in Figure 3.7.

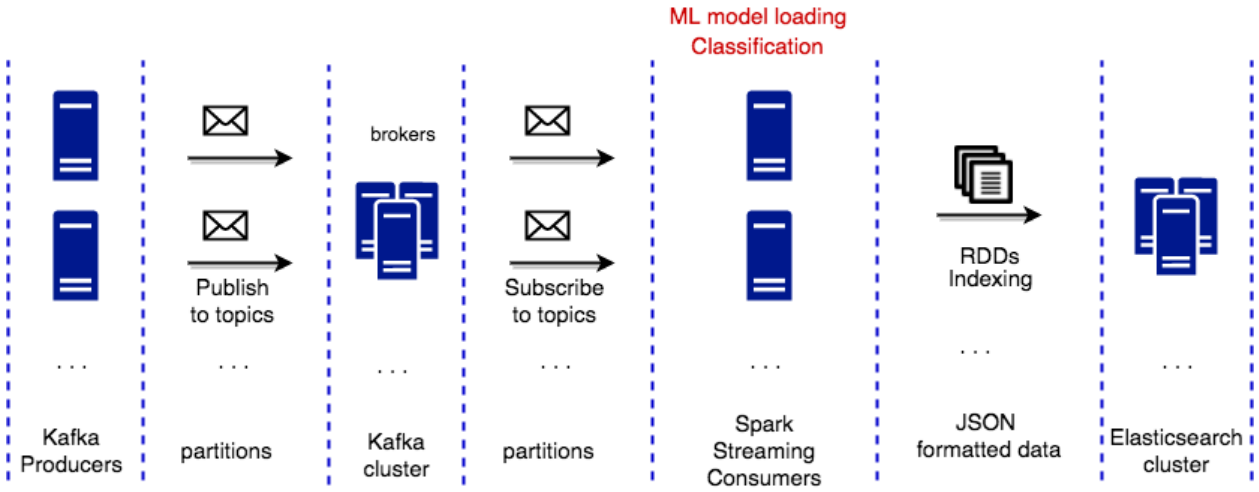


***Figure 3.7**: A comprehensive representation of the distributed architecture*

Chapter 4

Experimental Study

This chapter is a representation of the results obtained from the experimental processes conducted. It is divided into two parts: the first concerns the process of creating the ML model and the corresponding results, while the second concerns the distributed architecture that has been implemented and, in particular, the real-time data processing procedure along with the associated outcomes. As already mentioned in the previous chapter, all experiments were performed on a single computer, as it was unable to exploit a cluster, thus a complete representation of the final results was not feasible.

4.1. Evaluation of Classifiers

This part concerns the creation of the three ML models and results acquired for each one of them. Furthermore, a comparison is made between them and the best one is chosen as the final and most suitable to be added to the architecture. Each algorithm trains about fifty thousand sentences.

One of the metrics used is accuracy, which indicates the values predicted in relation to the actual values. The baseline is an accuracy of 0.5.

```
# predict the response values for the observations in "X"
y_pred = pipeline.predict(X_test)
from sklearn.metrics import accuracy_score
# testing accuracy : train and test the model on different data
print ('Accuracy score:', accuracy_score(y_test, y_pred))
```

Another metric used is f1 scores, which is a more detailed measurement that takes into account false-true negatives and false-true positives.

```
from sklearn.metrics import f1_score
print ('F-1 Score:', f1_score(y_test, y_pred))
```

Also, the execution time of the algorithm and a learning curve are presented for each model.

4.1.1. Naive Bayes Classifier

The Naive Bayes classifier was created for Bernoulli distribution, since it is the most suitable for the specific problem of sarcastic text detection. This is a simple implementation without any special configuration. A partial increment of the alpha smoothing parameter value was performed,

which slightly reduced the accuracy of the model, therefore not utilized. Performance is shown in Table 1.

	Accuracy	F1-score	Time (sec)
Naive Bayes	0.68998	0.67782	11.79

***Table 1**: Naive Bayes classification*

As it seems, the success rate is quite close to the sarcasm detection literature for this classifier and execution time is too small, which is expected as it is a fast enough algorithm. The learning curve is presented in Figure 4.1 to determine if there is room for improvement if the algorithm is trained with more examples.

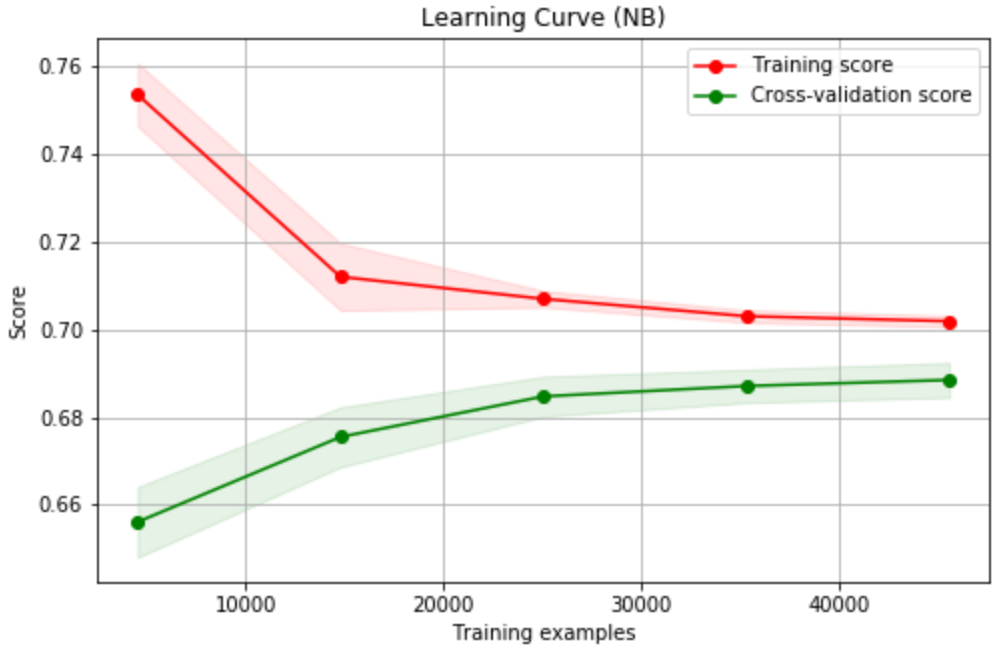


***Figure 4.1**: Learning curve of Naive Bayes*

As distinguished, the training score starts at a high value and lowers as examples are added and the cross-validation score starts at a low value and rises. After some point the two scores converge at a fairly small value as the examples are added to the training set. As can be seen, in this case the addition of more examples does not greatly improve the performance of the algorithm.

4.1.2. Logistic Regression Classifier

The Logistic Regression classifier implements regularized logistic regression using the L1 regularization and no other configuration as none contributed to improving performance. Performance is shown in Table 2.

	Accuracy score	**F1-score**	**Time (sec)**
Logistic Regression	0.69683	0.68608	12.34

***Table 2**: Logistic Regression classification*

As it seems, the success rate is slightly better than previously, although the runtime of the algorithm has increased on a small scale, but the algorithm remained quite fast. The learning curve is presented in Figure 4.2 to determine if there is room for improvement if the algorithm is trained with more examples.

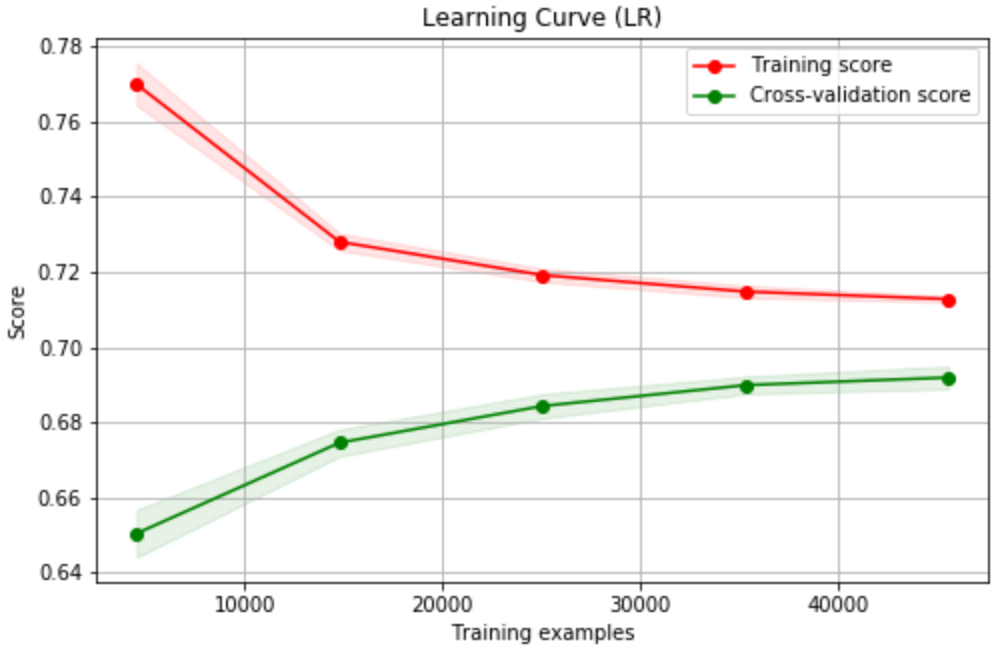


***Figure 4.2**: Learning curve of Logistic Regression*

As distinguished, the learning curve is almost similar to previously, therefore follows that neither in this case the addition of more examples does not greatly improve the performance of the algorithm.

4.1.3. Random Forests Classifier

The implementation of this model is slightly more complicated than previously, as it contains parameterization, which caused better performance. At the beginning, success rates coincided with those of previous classifiers, but after the parameterization they increased. The parameters were set at the optimal values.

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([
    ('train_count_vect', TfidfVectorizer(max_features = MAX_FEATURES,
ngram_range=(1,5))),
    ('clf', RandomForestClassifier(n_estimators=100, criterion='entropy',
max_depth=500, min_samples_split=4, n_jobs = 3, random_state = 42)),
])
start = time()
pipeline.fit(X_train,y_train)
```

It is worth noting that the parameterization can be performed in a distributed environment using the GridSearchCV function, which is provided through the scikit-learn integration package for Apache Spark, by changing a single line of code.

```
# from sklearn.model_selection import GridSearchCV
from spark_sklearn import GridSearchCV # Use spark_sklearn's grid search instead
# Tuning the hyper-parameters indicatively
parameters = {
    'clf__n_estimators': [50,100,200,500],
    'clf__n_max_depth' : [100,200,300,400]
}
grid_search = GridSearchCV(pipeline, param_grid=parameters, cv=3, n_jobs=-1, verbose=1)
grid_search.fit( X_train, y_train)
```

Otherwise it is an extremely time-consuming process, especially in the case of large datasets with multiple, different parameter combinations. Performance is shown in Table 3.

	Accuracy score	**F1-score**	**Time (sec)**
Random Forests	0.71341	0.70559	50.92

***Table 3**: Random Forests classification*

As it seems, the success rate is better than previously, although the runtime of the algorithm has increased considerably, which is expected as there are one hundred different

classifiers combined for a final result, which is ultimately the optimal. The learning curve is presented in Figure 4.3 to determine if there is room for improvement if the algorithm is trained with more examples.

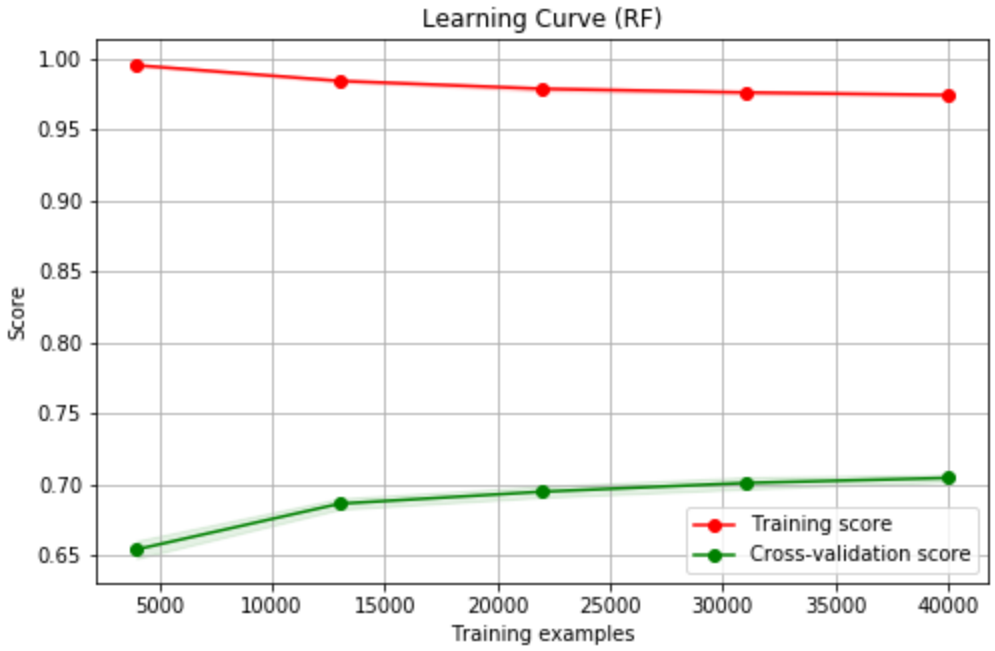


***Figure 4.3**: Learning curve of Random Forests*

As distinguished, in this case the learning curve is quite different than before. The training score starts at a high value and lowers slightly as examples are added and the cross-validation score starts at a low value and rises. The algorithm suffers from high variance, thus more training examples will most likely increase generalization and improve performance.

This is therefore the most appropriate model which is finally loaded into the system, as it has the highest performance and margin for a further increase. Finally, the measurements of the ML model relating the testing data are shown in Table 4.

	Accuracy score	**F1-score**	**Time (sec)**
Random Forests	0.71275	0.70288	29.75

***Table 4**: Random Forests classification (testing data)*

Another information derived for this model is the importance of the features, which are shown in Figure 4.4. Evidently, four features are informative, while the remaining are not. This can be used to reduce insignificant features thus improve the running time of the algorithm, which constitutes the case in which it lags behind others.

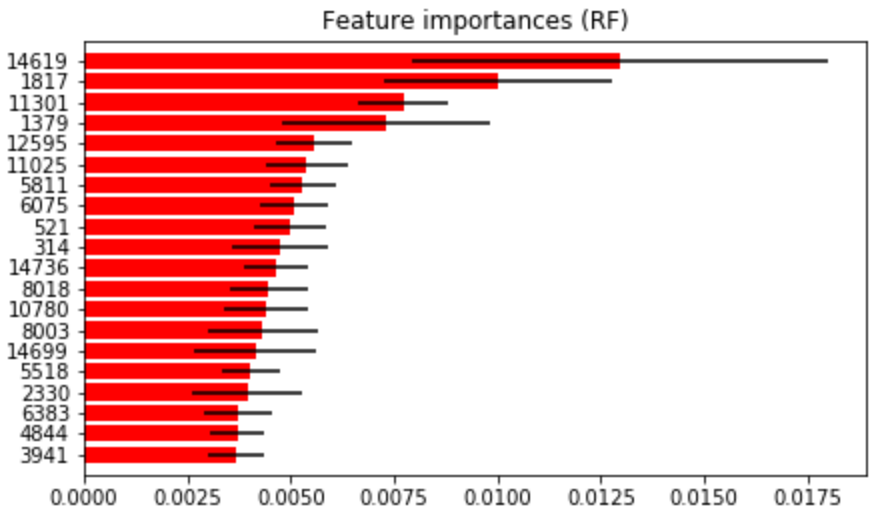


***Figure 4.4**: Feature importances along with their inter-trees variability*

4.2. System Evaluation

The second part of the presentation of the results concerns the system and more specifically the platforms used for its implementation. The web user interfaces related to the system are presented and information is analyzed through them. At this stage, 10K entries were sent and processed out of a total of 10M and the corresponding results are listed below.

4.2.1. Apache Kafka Metrics

Kafdrop [20] was used as a monitoring tool for the processes performed in Kafka. More specifically, the average sending time of 10K records is equal to 240.723 sec, or almost 4 minutes. To wit, Kafka producer sent about *42 messages/sec*. Figure 4.5 presents a cluster overview, which in this case consists of a single broker. It also shows the topics and their partitions.

Kafka Cluster Overview

Zookeeper Hosts: localhost:2181

Brokers

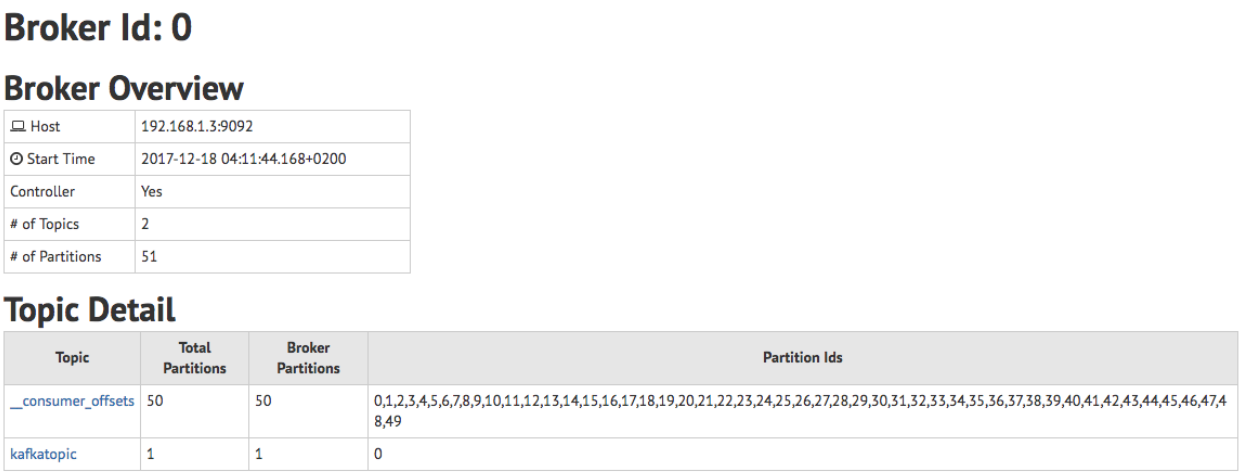
ID	Host	Port	JMX Port	Version	Start Time	Controller?
0	192.168.1.3	9092	-1	1	2017-12-18 05:46:20.307+0200	Yes

Topics

Name	Partitions	% Preferred	# Under Replicated	Custom Config?
__consumer_offsets	50	100%	0	Yes
kafkatopic	1	100%	0	No

***Figure 4.5**: Kafka cluster overview*

Information about each broker is depicted in Figure 4.6.



Broker Id: 0

Broker Overview

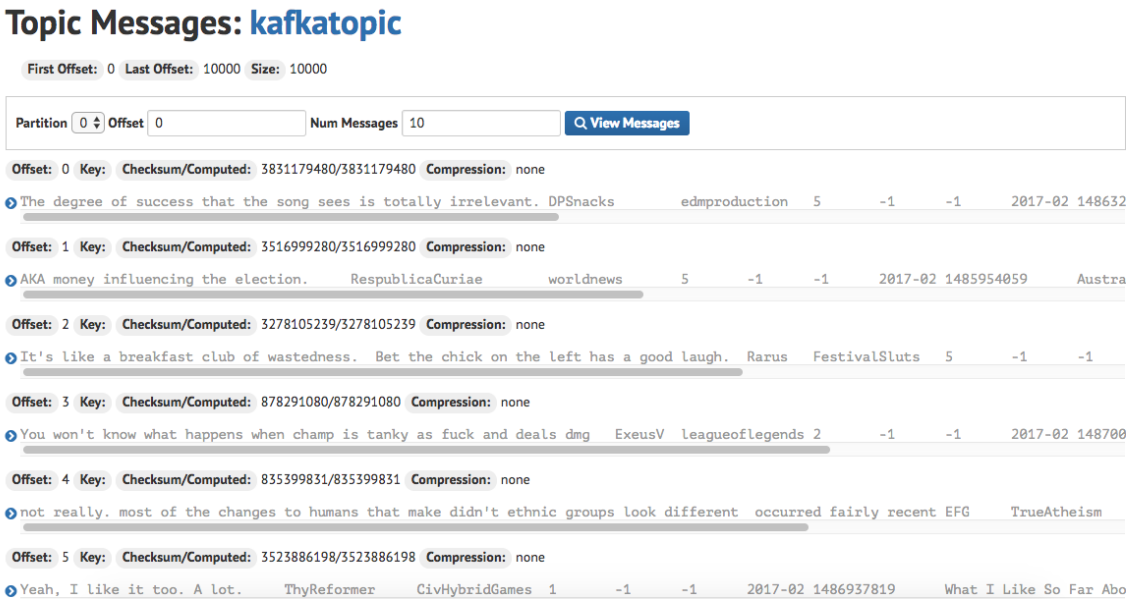
Host	192.168.1.3:9092
Start Time	2017-12-18 04:11:44.168+0200
Controller	Yes
# of Topics	2
# of Partitions	51

Topic Detail

Topic	Total Partitions	Broker Partitions	Partition Ids
__consumer_offsets	50	50	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49
kafkatopic	1	1	0

***Figure 4.6**: Kafka broker overview*

Figure 4.7 shows the messages within the “kafkatopic”.



Topic Messages: kafkatopic

First Offset: 0 Last Offset: 10000 Size: 10000

Partition 0 Offset 0 Num Messages 10 View Messages

Offset: 0 Key: Checksum/Computed: 3831179480/3831179480 Compression: none

The degree of success that the song sees is totally irrelevant. DPSnacks edmproduction 5 -1 -1 2017-02 148632

Offset: 1 Key: Checksum/Computed: 3516999280/3516999280 Compression: none

AKA money influencing the election. RespublicaCuriae worldnews 5 -1 -1 2017-02 1485954059 Austra

Offset: 2 Key: Checksum/Computed: 3278105239/3278105239 Compression: none

It's like a breakfast club of wastedness. Bet the chick on the left has a good laugh. Rarus FestivalSluts 5 -1 -1

Offset: 3 Key: Checksum/Computed: 878291080/878291080 Compression: none

You won't know what happens when champ is tanky as fuck and deals dmg ExeusV leagueoflegends 2 -1 -1 2017-02 148700

Offset: 4 Key: Checksum/Computed: 835399831/835399831 Compression: none

not really. most of the changes to humans that make didn't ethnic groups look different occurred fairly recent EFG TrueAtheism

Offset: 5 Key: Checksum/Computed: 3523886198/3523886198 Compression: none

Yeah, I like it too. A lot. ThyReformer CivHybridGames 1 -1 -1 2017-02 1486937819 What I Like So Far Abo

***Figure 4.7**: Kafka topic messages*

Figure 4.8 shows all the messages consumed so far as well as the Spark Streaming consumer. There is also computed the delay between the consumer and the producer.

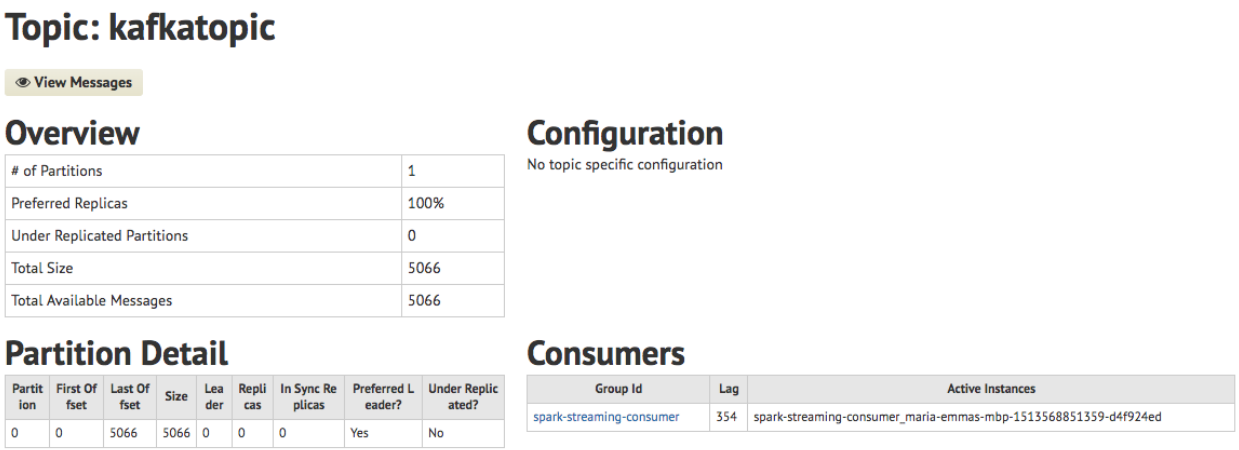


***Figure 4.8**: Topic details about partitions and consumers*

4.2.2. Spark Streaming Metrics

For the presentation of the processes within the Spark Streaming platform, the web UI provided by Spark was used. Figure 4.9 shows Spark jobs running at the moment and already executed jobs that succeeded or failed. Once the executor driver is added the job is ready to start.

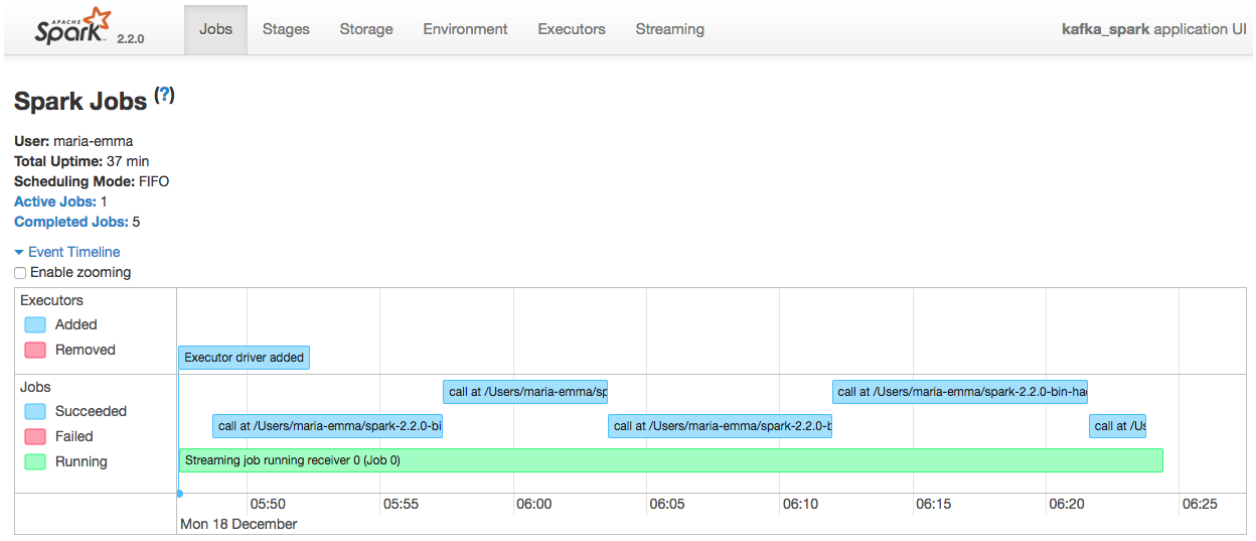


***Figure 4.9**: Spark Jobs*

More detailed statistics on the Spark Steaming part are illustrated in Figure 4.10. As shown, the reading of the messages from the Kafka broker, their processing as well as their writing to Elasticsearch lasted 36 minutes at a 1 second batch interval with an average input rate of 244 records/sec for each batch. The procedure was completed with 24 batches. Scheduling delay is the

time a batch waits in a queue for the processing of previous batches to finish and equals almost 20 minutes.

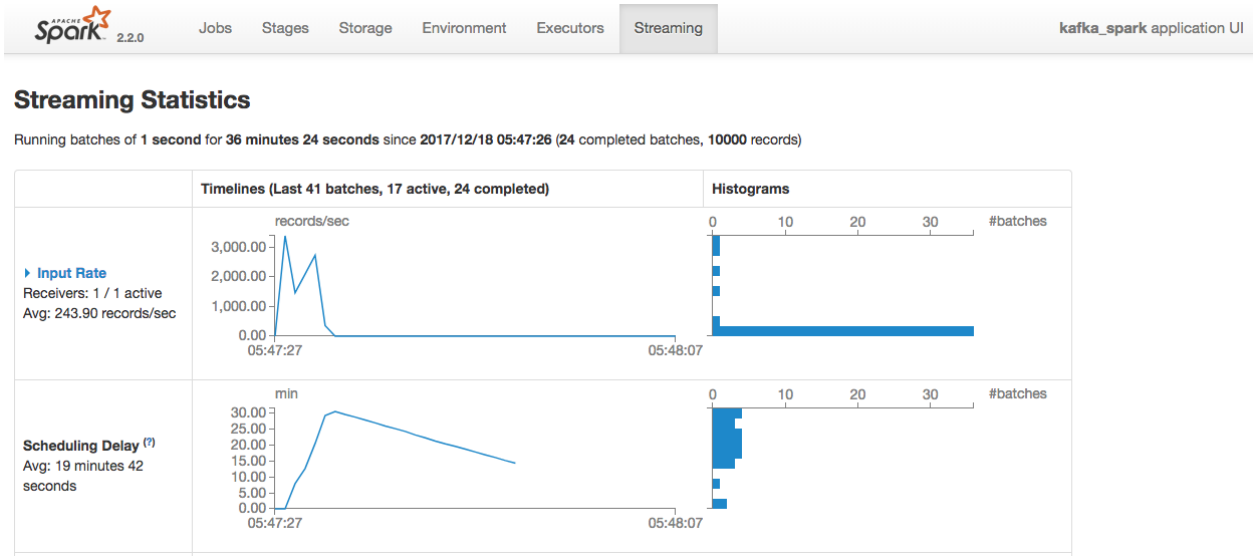


***Figure 4.10**: Spark Streaming statistics concerning the input rate and the scheduling delay*

Figure 4.11 shows the continuation of the above-mentioned statistics, which refers to the processing time and the total delay time. Processing time is the time to process each batch of data and equals almost 1 minute and the total delay time is essentially the sum of the scheduling and the processing time.

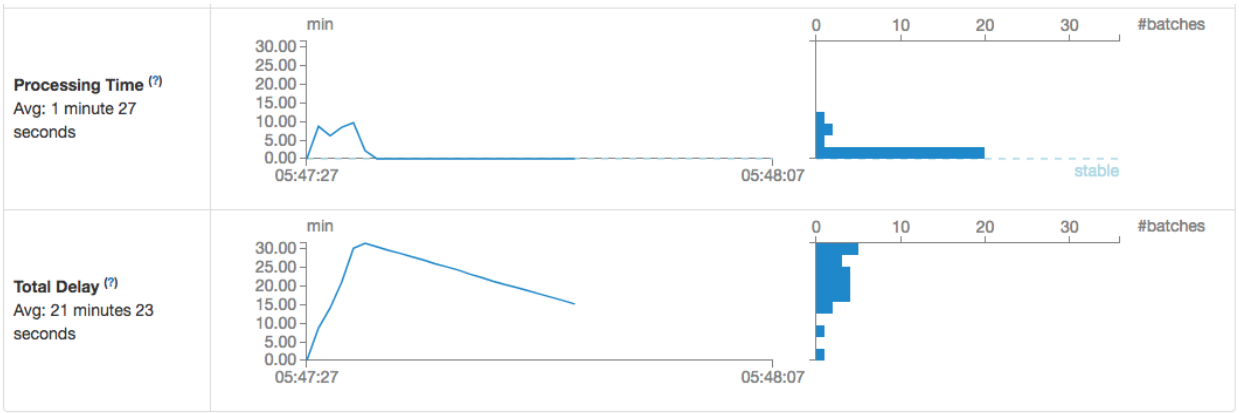


***Figure 4.11**: Spark Streaming statistics concerning the processing time and the total delay*

4.2.3. Kibana Visualization

Kibana tool shows the assignment of data to indexes, as depicted in Figure 4.12.

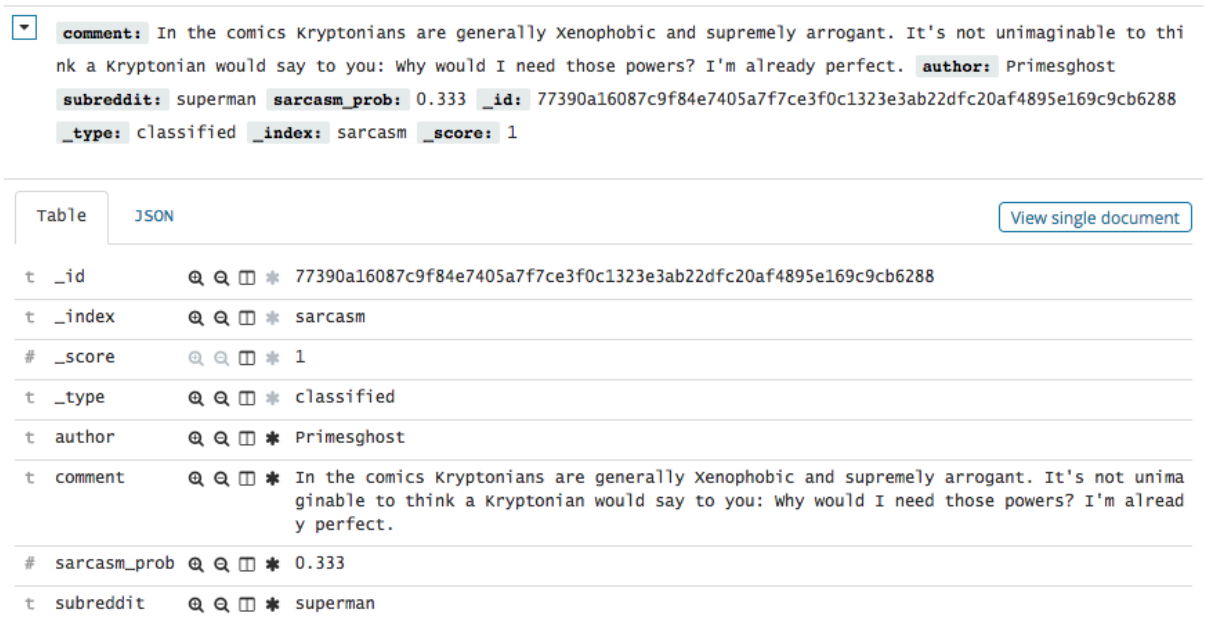


***Figure 4.12**: Documents indexed into Elasticsearch*

Through the features of each document, dashboards may arise based on aggregation metrics. For example, Figure 4.13 shows the total documents contained in the sarcasm index and a chart with the ranges of the sarcasm probabilities.

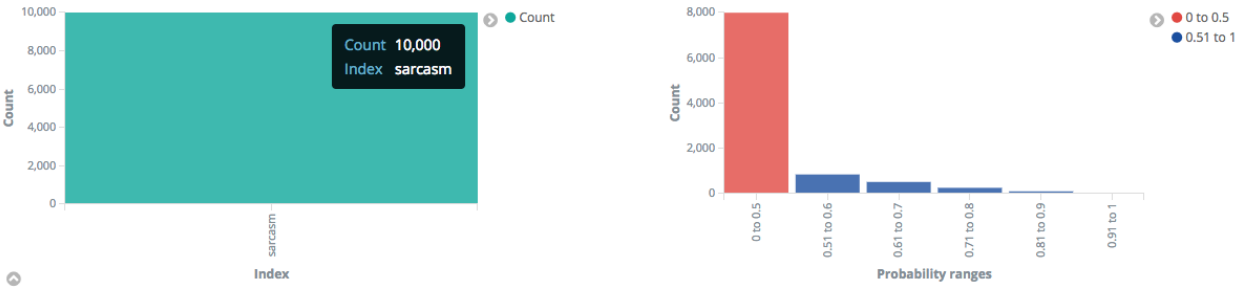


***Figure 4.13**: Number of documents within the sarcasm index and the range in which the probabilities of sarcasm vary*

For example, in the second diagram, the chances of sarcasm vary. Most of the comments (7991) have a probability of less than 0.5 being sarcastic, while the remaining 849 comments have a probability between 0.51 and 0.6, 519 have a probability between 0.61 and 0.7, 259 have a probability between 0.71 and 0.8, 103 have a probability between 0.81 and 0.9, while no comment has a probability between 0.91 and 1.

Chapter 5

Conclusion

In conclusion, an approach combining the well-known fields of big data and ML, is presented. This is a system which has the ability to process texts on a large scale and identify sarcasm in them. Its creation is entirely based on new technologies and their respective combinations and concerns the field of sarcasm detection which remains quite unexplored. It is also an architecture that includes completely independent elements that can be replaced by others, starting from each distributed platform and ending with the ML model. It is worth mentioning that it is based on easily modifiable code for future extensions.

This study used a different source of data than usual and emphasized the processing of data streams in real-time, a challenging task of the modern era. It proved why Reddit can be a reliable source of information compared to Twitter, which prevails in sarcasm detection systems. It also showed ways to exploit large files through the popular inverted indexer data structure, which is widely used by search engines.

Furthermore, the concept of messaging systems was introduced, by utilizing the Apache Kafka platform, which aims at fast and fault-tolerant information transmission to other systems. The study has demonstrated the operations of Kafka as well as the facilities it offers. It also suggested the easy-to-use and effective integration of Kafka with the Spark Streaming framework.

In addition, an introduction to the Spark system and its mode of operation was realized, as well as a brief overview of concepts behind its creation. It has also been compared to other existing systems and has proven to be the most appropriate for this architecture but also a particularly useful and promising tool for future big data issues. Among others, an exploitation of the Spark Streaming extension was carried out, to process data streams in real-time.

An introduction to the scikit-learn library was also accomplished, which was intended to demonstrate the usefulness of this library and the huge range of algorithms it offers. It has been shown that the models of this library are reusable in various contexts through pickled serialized objects. Common and state-of-the-art algorithms were utilized to create ML models for the specific task of sarcasm detection, easily and efficiently, which are reusable and can operate independently of the particular architecture, with minor changes.

A different approach to the concept of data storage in conventional databases was introduced through Elasticsearch, a distributed search engine serving for data analysis and representations using the Kibana plugin.

This study demonstrated that all of the tools utilized can operate in an integrated parallel and distributed environment of a computer cluster noting high performance. Finally, there were performed measurements and experiments that showed the operation and the suitability of each of the tools used.

5.1. Future Work

There are several proposed ideas for future expansion of this work as it deals with elements and approaches that are constantly evolving. One of them is the extension of the ML model. There could be added many other methods that will improve its performance, e.g. feature engineering or feature selection techniques, as already mentioned. Both are categories with a huge range of options and tests, which could well lead to better results. Moreover, as demonstrated in the experiments the random forests algorithm can be further improved by adding more training examples, which will be suitable for the case.

As far as the part of the architecture implemented is concerned, an initial improvement would be its application into a computer cluster and conduct the experiments in this context, in order to highlight the efficiency of the system in a distributed environment. Adding a cluster can make several changes that would lead to a possible improvement, such as saving the initial big data file into HDFS and process it without using the inverted indexer or the random generator. Finally, the whole implementation of the ML model could be transferred to the Spark MLlib library, instead of using scikit-learn.

Bibliography

[1] Bharti, Santosh & Vachha, Bakhtyar & Pradhan, Ramkrushna & Babu, Korra & Jena, Sanjay. (2016). Sarcastic Sentiment Detection in Tweets Streamed in Real time: A Big Data Approach. *Digital Communications and Networks 2.3*, 108-121, 2016.

[2] Leo Breiman. 1996. Bagging predictors. *Machine Learning*. 24, 2 (August 1996), 123-140. DOI=http://dx.doi.org/10.1023/A:1018054314350

[3] Konstantin Buschmeier, Philipp Cimiano, and Roman Klinger. An impact analysis of features in a classification approach to irony detection in product reviews. *In Proceedings of the 5th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis*, pages 42–49, June 2014.

[4] Michael B. Cox and David Ellsworth. Application-controlled demand paging for Out-of-Core visualization. *In Proceedings of Visualization '97*, pages 235–244, October 1997.

[5] Gandomi, Amir & Haider, Murtaza. (2015). Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*. 35. 137-144. 10.1016/j.ijinfomgt.2014.10.007.

[6] Ho, Tin. (1995). Random decision forests. *Document Analysis and Recognition, International Conference* on. 1. 278 - 282 vol.1. 10.1109/ICDAR.1995.598994.

[7] Joshi, Aditya & Bhattacharyya, Pushpak & Carman, Mark. (2016). Automatic Sarcasm Detection: A Survey. *arXiv preprint arXiv:1602.03426*

[8] Mikhail Khodak, Nikunj Saunshi, and Kiran Vodrahalli. 2017. A Large Self-Annotated Corpus for Sarcasm. *arXiv preprint arXiv:1704.05579*, 2017.

[9] Kreps, Jay, Neha Narkhede and Jun Rao. "Kafka: a Distributed Messaging System for Log Processing." (2011).

[10] Maynard, Diana & Greenwood, M.A.. (2014). Who cares about sarcastic tweets? Investigating the impact of sarcasm on sentiment analysis. *Proceedings of LREC*. 4238-4243.

[11] Ngoc Khuc, Vinh & Shivade, Chaitanya & Ramnath, Rajiv & Ramanathan, Jay. (2012). Towards building large-scale distributed systems for twitter sentiment analysis.

[12] Nodarakis, Nikolaos & Sioutas, Spyros & Tsakalidis, Athanasios & Tzimas, Giannis. (2016). Large Scale Sentiment Analysis on Twitter with Spark.

[13] Pedregosa et al.Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[14] Ptáček, Tomás et al. “Sarcasm Detection on Czech and English Twitter.” COLING (2014).

[15] Maarten Steen and Andrew S. Tanenbaum. 2016. A brief introduction to distributed systems. Computing 98, 10 (October 2016) 967-1009.

[16] Zaharia, Matei & Chowdhury, Mosharaf & J. Franklin, Michael & Shenker, Scott & Stoica, Ion. (2010). Spark: Cluster Computing with Working Sets. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 10-10.

Web pages

[17] Apache Kafka. Retrieved from https://kafka.apache.org [Dec.2017]

[18] Apache Spark. Retrieved from https://spark.apache.org [Dec.2017]

[19] Elasticsearch. Retrieved from https://www.elastic.co [Dec.2017]

[20] Kafdrop UI and Monitoring Tool. Retrieved from https://github.com/HomeAdvisor/Kafdrop [Dec.2017]

[21] Kohn, Mark. Mathematics of Random Forests. Boston University. Department of Mathematics & Statistics. Retrieved from http://math.bu.edu/people/mkon/MA751/L19RandomForestMath.pdf [Dec.2017]

[22] Lambda Architecture. Retrieved from http://lambda-architecture.net [Dec.2017]

[23] Ng, Andrew. CS229 Lecture notes. Stanford University. School of Engineering. Retrieved from http://cs229.stanford.edu/syllabus.html [Dec.2017]

[24] Reddit. https://www.reddit.com [Dec.2017]

[25] Scikit-learn http://scikit-learn.org/stable [Dec.2017]

[26] Twitter. https://twitter.com [Dec.2017]