| | |
|---|---|
| **The Islamic University of Gaza** | **الجامعـــــــــة الإســــلامية بغـزة** |
| **Deanship of Research and Postgraduate** | **عمادة البحث العلمي والدراسات العليا** |
| **Faculty of Information Technology** | **كـليـــــة تكنولـــوجيـا المعلـومـــات** |
| **Master of Information Technology** | **مـــاجستير تكنولـوجيـا المعلومـات** |

# Parallel Text Classification Applied to Large Scale Arabic Text

# تصنيف النصوص العربية ذات النطاق الواسع على التوازي

**by**

**Bushra Omar Alqarout**

**Supervised by**

**Dr. Rebhi S. Baraka**

**Associate Professor of Computer Science**

**A thesis submitted in partial fulfilment of**
**the requirements for the degree of**
**Master of Information Technology**

**October/2017 – Muharram/1439**

## إقــــــــرار

**أنا الموقع أدناه مقدم الرسالة التي تحمل العنوان:**

**Parallel Text Classification Applied to Large Scale Arabic Text**

**تصنيف النصوص العربية ذات النطاق الواسع على التوازي**

أقر بأن ما اشتملت عليه هذه الرسالة إنما هو نتاج جهدي الخاص، باستثناء ما تمت الإشارة إليه حيثما ورد، وإن هذه الرسالة ككل أو أي جزء منها لم يقدم من قبل لنيل درجة أو لقب علمي أو بحثي لدى أي مؤسسة تعليمية أو بحثية أخرى.

## DECLARATION

The work provided in this thesis, unless otherwise referenced, is the researcher's own work, and has not been submitted elsewhere for any other degree or qualification

| | | | |
|---|---|---|---|
| Student's name: | **Bushra O. Alqarout** | **بشرى عمر القـاروط** | اسم الطالب: |
| Signature: | | | التوقيع: |
| Date: | | | التاريخ: |

الرقم: ج س غ/٣٥/ Ref: ....

التاريخ: ٢٠١٧/١٠/١٠ Date: ....

## نتيجة الحكم على أطروحة ماجستير

بناءً على موافقة عمادة البحث العلمي والدراسات العليا بالجامعة الإسلامية بغزة على تشكيل لجنة الحكم على أطروحة الباحثة/ **بشرى عمر علي القاروط** لنيل درجة الماجستير في **كلية *تكنولوجيا المعلومات*** **برنامج تكنولوجيا المعلومات** وموضوعها:

تصنيف النصوص العربية ذات النطاق الواسع على التوازي

**Parallel Text Classification Applied to Large Scale Arabic Text**

وبعد المناقشة التي تمت اليوم الثلاثاء ٢٠ محرم ١٤٣٨هـ، الموافق ٢٠١٧/١٠/١٠م الساعة الحادية عشر صباحاً، اجتمعت لجنة الحكم على الأطروحة والمكونة من:

| | | |
|---|---|---|
| د. ربحي سليمان بركة | مشرفاً و رئيساً | [illegible] |
| د. أشرف يونس مغاري | مناقشاً داخلياً | [illegible] |
| د. سناء وفا الصايغ | مناقشاً خارجياً | [illegible] |

وبعد المداولة أوصت اللجنة بمنح الباحث درجة الماجستير في **كلية *تكنولوجيا المعلومات*/** برنامج **تكنولوجيا المعلومات.**

***واللجنة إذ تمنحه هذه الدرجة فإنها توصيه بتقوى الله ولزوم طاعته وأن يسخر علمه في خدمة دينه ووطنه.***

والله ولي التوفيق ،،،

عميد البحث العلمي والدراسات العليا

[illegible]

أ.د. مازن اسماعيل هنية

عمادة البحث العلمي والدراسات العليا
الجامعة الإسلامية غزة
Deanship of Research and Graduate Studies

## Abstract

Arabic text classification is becoming the focus of research and study for many researchers interested in Arabic text mining field especially with the rapid grow of Arabic content on the web. In this research, Naïve Bayes (NB) and Logistic Regression (LR) are used for Arabic text classification in parallel. When these algorithms are used for classification in a sequential manner, they have high cost and low performance. Naïve Bayes cost a lot of computations and time when it is applied on large scale datasets in size and feature dimensionality. On the other hand, logistic regression has iterative computations which cost heavy time and memory. Also, both algorithms do not give satisfying accuracy and efficiency rates especially with large Arabic dataset taking into account that Arabic language has complex morphology adding complexities to the computing cost. Therefore, in order to overcome the above limitations, these algorithms must be redesigned and implemented in parallel.

In this research, we design and implement parallelized Naïve Bayes and Logistic Regression algorithms for large-scale Arabic text classification. Large-scale Arabic text corpuses are collected and created. This is followed by performing the proper text preprocessing tasks to present the text in appropriate representation for classification in two phases: sequential text preprocessing and term weighting with TF-IDF in parallel. The parallelized NB and LR algorithms are designed based on MapReduce model and executed using Apache Spark in-memory for big data processing. Various experiments are conducted on a standalone machine and on a computer clusters of 2, 4, 8, and 16 nodes. The results of these experiments are collected and analysed.

We found that applying stemming approach reduced dataset documents' sizes and affects the classification accuracy where root stemming gets more accurate results than light (light1) stemming. For fast results, NB is suitable and returns high accuracy rates around 99% for large-scale documents with high dimensionality. LR also gives accurate results except it takes longer time than NB. It gives 93% accuracy for Al-Bokhary corpus compared to NB which gives 89% accuracy for the same corpus.

***Keywords:*** *Text Classification, Apache Spark, Naïve Bayes, Logistic Regression, MapReduce.*

## الملخص

أصبح تصنيف النص العربي محور البحث والدراسة لكثير من الباحثين والمهتمين في مجال التنقيب في النصوص العربية لسرعة نمو المحتوى العربي السريع على شبكة الإنترنت. في هذا البحث نستخدم خوارزميات التصنيف Naïve Bayes و Regression Logistic لتصنيف النص العربي على التوازي. هاتان الخوارزميتان عندما تستخدم للتصنيف بطريقة متتابعة تكلف الكثير من الحسابات والوقت والمساحة خصوصا عندما يتم تطبيقها على مجموعات كبيرة من البيانات على نطاق واسع في الحجم والخصائص. كما أنهما لا توفران دقة وكفاءة كافية لتصنيف البيانات العربية الكبيرة لما تتميز بها اللغة العربية من طبيعة مركبة. هذه القضايا تجعل من الضروري تصميم وتنفيذ هاتان الخوارزميتان على التوازي.

في هذا البحث، قمنا بتطوير وتنفيذ خوارزميات التصنيف Naïve Bayes وLogistic Regression على التوازي لتصنيف النص العربي ذو النطاق الواسع. حيث تم جمع هذا النص الكبير من المكتبة الشاملة و إنشاء نصوص من أحاديث البخاري ، ثم أداء المهام التحضيرية المناسبة لتجهيز النص في الشكل المناسب للتصنيف وتم ذلك على مرحلتين: معالجة النص مسبقا باستخدام برنامج جافا وتمثيله بشكل TF-IDF باستخدام مكتبة Apache Spark التي تطبق منهج MapReduce. تم تصميم خوارزميات التصنيف لتعمل على التوازي حسب نموذج MapReduce وطبقت باستخدام Apache Spark المختص في معالجة البيانات الكبيرة. تم تنفيذ تجارب متعددة على جهاز حاسوب مستقل لتعمل على التوالي وعلى مجموعة من اجهزة الحاسوب تتكون من 2، 4، 8، و16 جهاز لتعمل على التوازي حيث تم جمع وتحليل نتائج هذه التجارب. وجدنا أن تطبيق طرق التجذير (stemming approaches) المختلفة يقلل من حجم الملفات وتأثيرها على دقة التصنيف إذ أن root stemmer أرجع نتائج أكثر دقة من light1 stemmer. خوارزمية Naïve Bayes عملت بأداء عالي للتصنيف السريع حيث وصلت دقة النتائج الى 99% تقريبا. خوارزمية Logistic Regression أرجعت نتائج أكثر دقة حيث وصلت دقتها الى 93% مع نصوص أحاديث البخاري في حين لم تتعد دقة Naïve Bayes 89% مع نفس النصوص ولكن بسرعة أعلى من سرعة Logistic Regression.

***الكلمات المفتاحية:*** *تصنيف النصوص، مكتبة Apache Spark، خوارزمية التصنيف Naïve Bayes، خوارزمية التصنيف Logistic Regression، نموذج البرمجة المتوازية MapReduce .*

بسم الله الرحمن الرحيم

قال تعالى:

﴿ وَقُل رَّبِّ زِدْنِي عِلْمًا ﴾

صدق الله العظيم

[ طـه: 114 ]

## Dedication

I dedicate this work and give special thanks to my dear husband Mohammed J. Abu Mere for being there by my side all the way through the work of this thesis.

I also have huge gratitude and appreciation to my loving parents, Omar and Sahar Alqarout whose words of encouragement and push for persistence ring in my ears. Moreover, I thank my brothers who have always been by my side.

## Acknowledgements

Special thanks to Dr. Rebhi S. Baraka, my supervisor, for his efforts of reflecting, reading, encouraging, and most of all patience throughout the entire research process.

I would like to thank Dr. Alaa M. El-Halees and Dr. Iyad M. Alagha for their feedback and comments.

Also, I would like to acknowledge and thank the IT faculty staff members, Mr. Arafat Abu Jrai and Mr. Mohammed Al-Shorafa for facilitating the lab to conduct my experiments.

## Table of Contents

## List of Tables

## List of Figures

## List of Appendices

## List of Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| FN | False Negative |
| FP | False Positive |
| IDF | Inverse Document Frequency |
| JDK | Java Development Kit |
| LAN | Local Area Network |
| L-BFGS | Limited memory - Broyden Fletcher Goldfarb Shanno |
| LR | Logistic Regression |
| MLlib | Machine Learning library |
| NB | Naïve Bayes |
| RDD | Resilient Distributed Dataset |
| TC | Text Classification |
| TF-IDF | Term Frequency-Inverse Document Frequency |
| TN | True Negative |
| TP | True Positive |

`

# Chapter 1

# Introduction

# Chapter 1

# Introduction

Text classification plays an important role in organizing and ordering various types and sizes of data. Data is available in different forms like text, images, video, sensor produced and social media and in different languages. In this research, we are investigating text classification of Arabic text using parallelized algorithms and observing its behaviour. Through this chapter, we describe the workflow of this research. We start with a background of text classification, Arabic, techniques and tools for performing text classification. Then we state the research problem specify the research objectives, determine the scope and limitations, identify the importance of the research, define the methodology to be followed to achieve the research objectives, and finally we present the structure of the thesis.

## 1.1 Background

Due to continuous growth of Arabic text content on the web, Arabic text mining has become the focus of research and study for many researchers interested in Arabic text mining especially on Arabic text classification (Hmeidi, Al-shalabi, & Al-Ayyoub, 2015).

Automatic Text Classification involves assigning a text document to a set of predefined classes automatically using a machine learning technique. The classification is usually based on specific words or features existing in the text document. Since the classes are pre-defined, it is a supervised machine-learning text classification.

Arabic language is known of its rich vocabulary and complex morphology. Therefore, Arabic text pre-processing and classification take a lot of computational time to produce the classifier model and applying it, especially, when large text is encountered.

There are different algorithms used for text classification such as Naïve Bayes (Thabtah, Eljinini, Zamzeer, & Hadi, 2009), K-Nearest Neighbour (K-NN) (Alhutaish & Omar, 2015), Support Vector Machines (SVM) (Alsaleem, 2011), Decision tree

(Bahassine, Kissi, & Madani, 2014). Most of these algorithms are used for text classification in a sequential manner which need much time on computation and do not result in high accuracy and efficiency especially with large datasets. Therefore, parallel implementation of these algorithms would enhance their accuracy and efficiency using models such as Map-Reduce and tools such as Hadoop (Hadoop, 2014) and Spark (Spark, 2014).

One of the most used classifiers on Arabic text (Mamoun & Ahmed, 2014) is Naïve Bayes (NB). It is a simple probabilistic classifier based on applying Bayes theorem with strong independence assumptions between the features. It works very well on numerical and textual data and requires a small volume of training data for classification but it has difficulty with noise or irrelevant features in training data.

In 2015, Al-Tahrawi find out Arabic text classification can be preferred using Logistic Regression (LR). It is a statistical method used to analyse a dataset with one or more independent features to determine a class. It estimates the probability of each class directly from the training data by minimizing errors.

Contemporary parallel techniques and tools could be used to improve the accuracy and the efficiency of these classification methods. One such tool is Apache Spark which is a fast and general-purpose cluster computing system for large-scale data processing in parallel. 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 for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

In this research, we develop and implement a text classification approach for Arabic language in parallel manner. We conduct the research with two corpuses: a large-scale Arabic text corpus and Arabic text corpus with large number of categories and small number of instances. Each corpus is exposed to the proper text preprocessing tasks to represent the text in appropriate form for the classification. The Naïve Bayes and Logistic Regression are the considered algorithms in this work and designed based on MapReduce architecture. The proposed approach is implemented using Apache Spark framework. We conduct different experiments on a single machine and on multiple computer clusters of 2, 4, 8, 16 nodes respectively for the evaluation of the proposed classifiers efficiency and accuracy.

## 1.2 Problem Statement

Different text classification algorithms like Naïve Bayes and Logistic Regression when applied to Arabic text take extra computational power and memory space on training the data for building the model and applying it leading to less efficiency and accuracy especially when the size of the dataset is large and text documents has high dimensionality taking into consideration the morphological complexity of the Arabic language.

The problem of this research is how to use parallelization with these algorithms to improve their efficiency and to use most convenient text pre-processing methods to enhance results accuracy for classifying large-scale Arabic text.

## 1.3 Objectives

### 1.3.1 Main Objective

To develop and implement a parallelized approach to classify large-scale Arabic text using Naïve Bayes and Logistic Regression algorithms and to measure their accuracy and efficiency compared to the sequential versions of these algorithms.

### 1.3.2 Specific Objectives

The research is conducted through achieving the following objectives:

1. Collect and create in house Arabic corpus for the approach.
2. Consider the proper text preprocessing techniques for each of its tasks: stemming and term weighting schemes.
3. Select and setup the suitable parallel system environment considering factors such as size of the cluster, suitability for realizing text classification algorithms and ability to deal with large-scale data.
4. Redesign and implement each classification algorithm in a suitable way for the selected parallel system.

5. Conduct sufficient experiments and evaluate the performance and classification results. Based on the results, compare the parallelized algorithms to each other and to the sequential versions.

## 1.4 Scope and Limitations

This research considers parallelizing text classification algorithms: Naïve Bayes and Logistic Regression for large-scale Arabic text to improve the level of efficiency and accuracy. The work is conducted with the following scope and limitations:

1. We use free large data corpus of Arabic language and we use in house collected corpus with a lot of numbers of classes with small number of samples.
2. The Arabic corpus contains text based on various Islamic domains such as hadith, feqh, and history.
3. The proposed classification algorithms are Naïve Bayes which is the most commonly used in Arabic text classification and Logistic Regression which is the rarely used in Arabic text classification. They are parallelized and compared with each other and with their sequential versions.
4. Implementation phase does not include text preprocessing. Text preprocessing is accomplished separately prior to conducting the experiments.
5. We conduct the experiments on a single machine and on a multicomputer clusters of 2, 4, 8 and 16 nodes to measure the efficiency and the accuracy of the proposed approach.
6. The cluster is built using Apache Spark standalone cluster.

## 1.5 Importance of the Research

1. Determine suitable factors and methods to take into consideration with Naïve Bayes and Logistic Regression algorithms for classifying Arabic text with best results and performance.
2. Enhance the efficiency of the text classification algorithms when applied to large-scale datasets or small datasets with large number of classes through parallelism.

3. The proposed parallel approach can be used with applications considering text classification as a major task such as text summarization and question answering systems.
4. The proposed parallel approach can play a major role in different domains such as health and business.

## 1.6 Research Methodology

We follow the following methodology to achieve the research objectives:

### 1.6.1 Research and Survey

This includes reviewing the recent literature closely related to the research problem and to the main research objective. The literature review covers three topics including Sequential Text Classification Algorithms, Text Preprocessing Techniques which affect Classification Accuracy and Text Classification with Parallel Computing. The reviewed researches were analyzed and summeraized in Chapter 2. We formulate the specific objectives to overcome the drawbacks and achieve the research main objective, hence solving the research problem.

### 1.6.2 Arabic Corpus Collection and Preparation

We search for appropriate free large-scale Arabic text corpus for classification and we create one on our own that has many classes with small number of samples to satisfy the research needs as described in Section 4.2 and 5.1.

### 1.6.3 Arabic Text Preprocessing

We apply Arabic text pre-processing tasks on the corpus to optimize the text quality and transform it into a suitable form for classification using appropriate techniques and tools as described in Section 4.3. This phase includes:

- **Tokenization**: breaking text into words called tokens using the appropriate Arabic tokenizer.
- **Normalization**: normalizing each token into its canonical form. In Arabic there are few letters which are often misspelled and thus need normalization and that include:
    - The Hamzated forms of Alif (أ , إ, آ) are normalized to bare Alif (ا).

- The Alif-Maqsura (ى) is normalized to a Ya (ي).
- The Ta-Marbuta (ة) *is normalized to a* Ha (ه).
- Remove tatweel. For example: (حركـــات) to (حركات)
- Remove numbers and special characters
- Remove Excessive spaces, tashkeel, and punctuation marks

- **Stop words removal**: remove any token considered as a stop word and does not bear content such as هما ,هي ,هم ,في.

- **Stemming:** use appropriate stemmer to derive the stem or root of each token.
- **Representation:** applying the suitable term weighting scheme to enhance text representation as feature vector.
- **Feature Selection**: apply the appropriate method to select group of features to reduce the training time needed and have better results for the approach construction like setting the number of features.

### 1.6.4 Setup the Parallel Environment

We setup the parallel environment as listed in Section 5.2 for the development and the implementation of the approach using the required models, frameworks and programming languages. MapReduce model is used as the parallel programming model. Apache Spark is a well-known framework that is used to realize MapReduce.

### 1.6.5 Design and Implementation of the Classification Algorithms

We design the parallel model for each considered classification algorithm (Naïve Bayes and Logistic Regression in Section 4.4 and 4.5) by choosing an appropriate MapReduce partitioning and mapping technique with load balancing .

We implement each algorithm using Apache Spark and its MLlib library in client mode that runs on Apache Spark standalone cluster.

### 1.6.6 Experimentation

We perform a set of experiments on the implemented approach using the pre-processed Arabic text corpus and observe the results and performance.

The experiments are described in detail at Section 5.3 where they run on a single node and on clusters of 2, 4, 8, 16 nodes respectively. They include dividing the dataset into two sets: one for training to build the classification model with the proposed approach and the other for testing to evaluate the generated model.

### 1.6.7 Evaluation

We evaluate the proposed approach in each mode according to the performance metrics such as time needed to train the model, speed up and scalability as measures of efficiency and they discussed in Section 5.4.1. We also analyse the results according to the classification measures such as accuracy, precicion, recall, and f-measure as dicussed in Section 5.4.2. After that, we discuss the results, and find out all related factors influencing the performance of the proposed approach.

## 1.7 Thesis Structure

The thesis is organized as follows: Chapter 2 covers the technical and theoretical foundation of the research. Chapter 3 presents the review of related works. Chapter 4 presents and describes the proposed approach for Arabic parallel classification. Chapter 5 discusses the experimental results and evaluation. Finally, Chapter 6 includes the conclusions and future work.

# Chapter 2

# Theoretical and Technical Foundation

# Chapter 2

# Theoretical and Technical Foundation

Arabic language is known as the mother tongue of Arabs and many Muslims around the world. Therefore, Arabic content on the web grows rapidly and hence increases the need for studying different classification algorithms like Naive Bayes and Logistic Regression for classifying Arabic text documents needed in many areas and for various purposes. Through this chapter, we present concepts, algorithms, models, tools and techniques used in this research. Starting with a conceptual overview of Arabic language in Section 2.1, followed by describing text classification and presents used classifiers in this research namely NB and LR in Section 2.2. We use Apache Spark as a parallel programming model. In Section 2.3 described Apache Spark in detail, how it manages memory, and its available libraries and APIs. Finally, evaluating text classification is defined using the mentioned metrics in Section 2.5.

## 2.1 Arabic Language

Arabic language is the language of Quran and the native tongue of more than 200 million people across the world (Versteegh & Versteegh, 2014; Wahba, Taha, & England, 2014). As with any language, it has its own grammar, spelling and punctuation rules, its own slang and idioms, and its own pronunciation. The Arabic alphabet consists of 28 letters, reading from right to left.

Arabic letters do not have a case distinction. Most letters connect with one another using slight modifications to the basic alphabet forms to combine words. Vowels on letters of Arabic word as resembled in Table 2.1, or the position of a word in the sentence could affect the meaning of the word.

For example:

كُتُب: (noun) pronounced kotob, means the plural of book كتاب.

كَتَبَ: (verb) pronounced katab, means write.

**Table (2.1):** Various Arabic Vowels represented on Letter ta'a (ط)

| طٌ | طٍ | طً | طُ | طِ | طَ | طْ | طّ |
|---|---|---|---|---|---|---|---|
| Tanwin Damma | Tanwin Kasra | Tanwin Fatha | Damma | Kasra | Fatha | Sukun | Shadda |

Arabic language preprocessing in text mining research includes various tasks such as normalization and stemming and these tasks considered difficult to maintain. Ayedh, Tan, Alwesabi, and Rajeh (2016) are explained that considering the following justifications:

- The rich nature and complex morphology of Arabic language in which a root word can produce many words with different meanings.
- In Arabic, there are not only suffixes, which added at the end of the root, or prefixes, which added at the beginning of the root, but also infixes that placed between the letters of the root and sometimes it is hard to differentiate them from the root. For example, adding alif (ا) to the root (عمل) to become (عـامل).
- There are Arabic words have various meanings and the proper meaning is identified according to its presence in the paragraph or how the diacritical marks set on the word letters.
- Since Arabic letters do not have capital or small shapes, it make recognizing proper names, acronyms, and abbreviations challenging.
- Lack of available free Arabic datasets applicable for Arabic document classification and large scale Arabic datasets in particular.
- Arabic language is rich with synonyms and broken plural forms that differ from its initial form in singular. For instance, (عَطِيَّة، صَدَقة، هِبَة) are synonyms that mean gift. (قُلُوب) is a broken plural forms means hearts that differ from its singular (قَلْب).
- There is words in Arabic language have many lexical classes (noun, verb, etc). As (قلب) in (في قلب المجريات) means core, (عملية قلب مفتوح) means heart.
- Some words in Arabic language cannot derive its root because it came from another language like program (برنامج) and internet (انترنت).

Therefore, various algorithms are developed to apply one of the preprocessing tasks such as light stemmer and root stemmer algorithms were developed for stemming in which stemming. In addition, each algorithm has memory and time complexity that should be taken into consideration besides the applied environments to preprocess the Arabic language text properly for classification.

## 2.2 Text Classification Algorithms

Text classification (TC) is a subfield of text mining used to assign each document to its related category or more. TC has a number general steps applied by different algorithms as shown in Figure 2.1 and they are explained as follows:

A. **Data Collection/Creation:** Data corpus used for classification could be in home collected or already collected by others.
B. **Text Preprocessing:** process corpus text using known text preprocessing tasks to be in suitable form for classification like stemming, tokenization, normalization, etc.
C. **Feature Weighting and Selection:** text will be represented as a bag-of-words such as position of the word or its meaning would not affect on the classification process and each word will be weighted and selected using different methods like tf-idf (Term Frequency-Inverse Document Frequency), Chi-Square, etc.
D. **Data Splitting:** usually data corpus split into a part for *training* the data using any chosen algorithm to generate the classifier model, and into another part for *testing* the generated model.
    - **D.1 Training:** The first part of data is the input to the classification algorithm to produce the classifier model.
    - **D.2 Testing:** The second part of data is used to test the produced classifier model that predicts the class of each input and then is compared to its real class. At the end, it computes the error percentage and other required performance measures like precision, and recall.

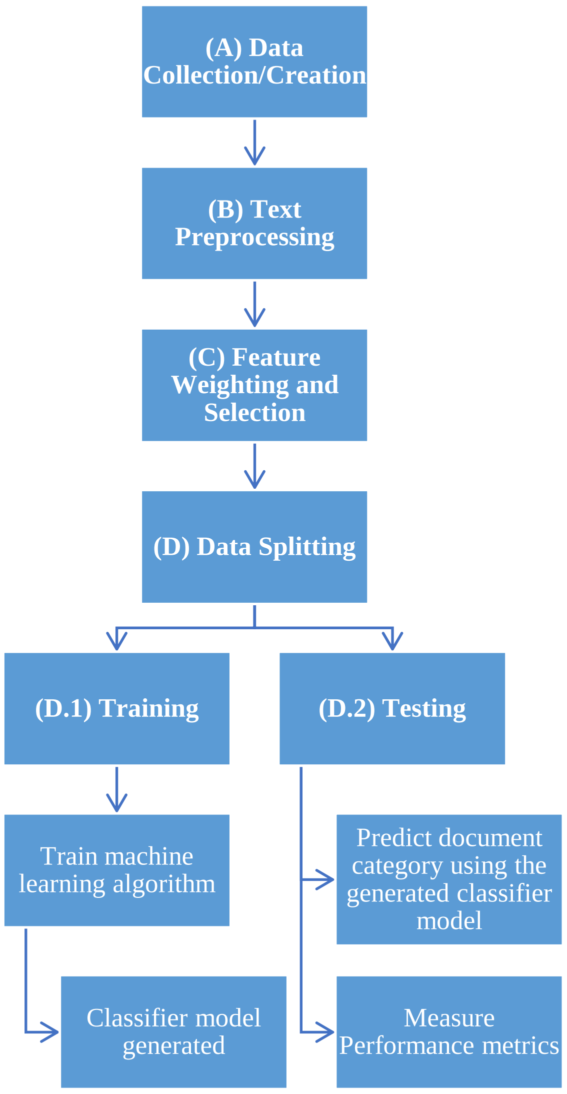
(A) Data Collection/Creation
(B) Text Preprocessing
(C) Feature Weighting and Selection
(D) Data Splitting
(D.1) Training
(D.2) Testing
Train machine learning algorithm
Predict document category using the generated classifier model
Classifier model generated
Measure Performance metrics

**Figure (2.1):** General Text Classification Steps

Now, we talk about the used text classification algorithms in this research namely Naïve Bayes and Logistic Regression.

### 2.2.1 Naïve Bayes

NB is a supervised machine-learning algorithm, which can be faster than other classification algorithms. It is constructed based on Bayes theorem of probability to predict the class of unknown data set and assume the independence between features. For example, a patient diagnosed with flu if he is exposed to the following symptoms (Mäkelä et al., 2000): fever, sore throat, cough, and headache. Even if these features depend on each other or upon the existence of the other features, all of these properties independently, contribute to the probability that the patient have a flu and that is why this algorithm is known as Naive. NB model is easy to build and particularly useful for very large data sets. Along with simplicity, it also known to outperform even highly sophisticated classification methods.

The classification model is established by applying Bayesian rule (Krishnaveni & Sudha, 2016; McCallum & Nigam, 1998) as indicated by equations 2.1 and 2.2.

$$\text{Posterior} = (\text{Likelihood} * \text{Prior}) / \text{Evidence} \qquad \textbf{(2.1)}$$

$$P(c|x) = (P(x|c) * P(c)) / P(x) \qquad \textbf{(2.2)}$$

Where:

- P(c|x): the posterior probability of *class* **c** given *predictor* (**x**, *attributes*).
- P(c): the prior probability of *class*.
- P(x|c): the likelihood which is the probability of *predictor* given *class*.
- P(x): the evidence that is the prior probability of *predictor*.

Given approximations of these parameters calculated from the training documents, classification can perform on test documents by calculating the posterior probability of each class given the evidence of the test document, and selecting the class with the highest probability.

### 2.2.2 Logistic Regression with L-BFGS

Logistic regression is commonly used with binary classification. It is a linear method with the loss function calculated by the logistic loss:

$$L(w;x,y) = \log(1 + \exp( -yw^T x )) \qquad \textbf{(2.3)}$$

In this research, we are targeting multinomial logistic regression that its' model m has K-1 binary logistic regression models regressed against the first class 0 for K possible classes. Given a new data points, K−1 models will be run, and the class with largest probability will be chosen as the predicted class. L-BFGS is an optimizer used with LR.

#### *- Multinomial Logistic Regression*

Multinomial logistic regression generalizes logistic regression to multiclass classification cases. It is used to predict the probabilities of the different possible outcomes of a categorically distributed dependent variable, given a set of independent variables. It takes assumptions for granted that each independent variable has one value for each class and it cannot anticipate the dependent variable of any class accurately. But statistically it does not demand to be self-reliant contrasting NB. However, it becomes challenging to discriminate the impact of several variables if this is not the class.

Polamuri (2017) simplified the Multinomial LR computations cycle into simple few steps as shows in Figure 2.2:

a. **Inputs:** All the features which exist in the dataset considered as the inputs (F) to the multinomial LR. And their values must be numerical. For that reason, the features are converted to numerical if they are not using proper methods.

b. **Linear Model**: The linear equation in the linear regression is also used as the linear model equation.

$$\text{Linear model} = W*F + b \qquad \textbf{(2.4)}$$

where **F** is the set of inputs, and **W** is set of weights.

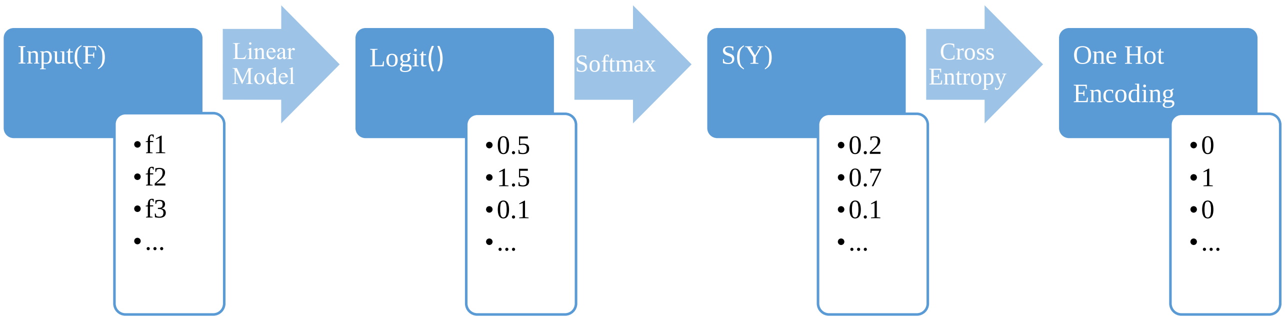

**Figure (2.2):** Multinomial LR Steps (Polamuri, 2017)

Assume F (numerical values) = [f1, f2, f3]. Where W includes the same input number of weights W = [w1, w2, w3]. Then the linear model output will be w1*x1, w2*x2, w3*x3. The weights w1, w2, w3 will be updated in the training phase using parameters optimization also called *loss function* which is an iteration process where the calculated weights for each observation used to calculate the cost function and ends when the loss function value is less or considered insignificant.

c. **Logits:** they are the outputs of the linear model that their scores are changing with the calculated weights.

d. **Softmax Function:** it computes the probabilities for the given score that returns the high probability value for the high scores and fewer probabilities for the remaining scores. We can observe from Figure 2.2 that softmax probabilities are 0.2 and 0.7 for the given logits 0.5 and 1.5. We use the highest probability value for predicting the target class for the given input features. Remembering that the probabilities range of the softmax function are between 0 and 1 and the summation of all its computed probabilities are equal to 1.

e. **Cross Entropy:** The last step in the multinomial LR that determines the similarity distance between the probabilities calculated from the softmax function and the target one-hot-encoding matrix. And the shortest distance will be for the true target class.

f. **One-Hot-Encoding:** This function is used to show the target values or categorical attributes into a binary representation. It is easy to create in which for every input features (f1, f2, f3) the one-hot-encoding matrix is with the values of 0 and the 1 for the target class. The total number of values in the one-hot-encoding matrix and the unique target classes are the same.

Next, we will talk about L-BFGS as an optimization parameter used with the multinomial LR.

***- Limited memory- Broyden Fletcher Goldfarb Shanno (L-BFGS)***

Limited memory BFGS is well known optimization algorithm in machine learning better version of the **B**royden **F**letcher **G**oldfarb **S**hanno (BFGS) algorithm using a limited volume of computer memory. It is also called L- "the algorithm of choice" for fitting log-linear (MaxEnt) models and conditional random fields with $L_2$-regularization.

It computes an approximation to the inverse Hessian matrix to steer its search through variable space in which it stores only a few vectors that represent the approximation implicitly. Due to its resulting linear memory requirement, it is appropriate for optimization problems with a large number of variables.

## 2.3 Map-Reduce Programming Model

MapReduce is a parallel and distributed programming model mainly associated for processing large data sets that could be executed on a large cluster of machines. Map and Reduce are the main procedures of MapReduce paradigm. *map()* processes an input of key/value pair to perform some filtering or sorting and generate a new set of intermediate key/value pairs, and *reduce()* joins all intermediate values associated with the same intermediate key. The MapRecuce system orchestrates the processing by partitioning the input data, scheduling the execution across a cluster machines, for redundancy and fault tolerance, and managing the required inter-machine communication. This tolerates unexperienced developers with parallel and distributed systems to easily consume the resources of a large distributed system.

There is various examples that applies the MapReduce model like *Distributed Grep* in which map() emits a line if it matches a supplied pattern, and reduce() is just copies the supplied intermediate data to the output.

Dean and Ghemawat (2008) present an overview of the MapReduce workflow as shown in Figure 2.3. The following actions occur when a MapReduce program runs:

1. The MapReduce program splits the input files into M pieces then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is the master and the rest are workers. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split, parses it into key/value pairs and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. From time to time, the buffered pairs are written to the local storage, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local storage are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is reported by the master about these locations, it uses remote procedure calls to read the buffered data from the local storages of the map workers. When a reduce worker has read all intermediate data, it

sorts it by the intermediate keys so that all occurrences of the same key are grouped. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program.

After successful completion, the output of the MapReduce execution is available in the output files. Typically, users do not need to combine these output files into one file. They often pass them as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

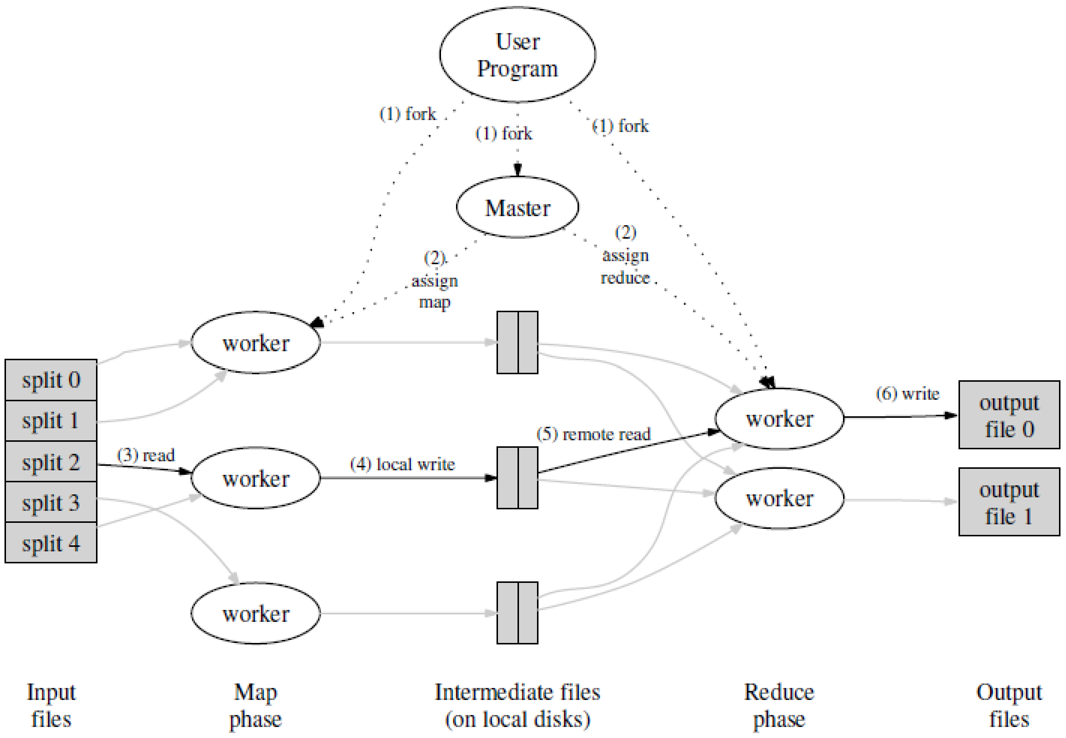

**Figure (2. 3):** MapReduce Execution Workflow (Dean & Ghemawat, 2008)

There are various frameworks and extensions that realize MapReduce model like Apache Hadoop and Couchdb. Now, we are describing in details Apache Spark that is an extension of MapReduce model used for processing big data.

## 2.4 Apache Spark

Apache Spark is an open-source platform for processing large-scale data. Spark offers the ability to access data in a variety of sources, including Hadoop Distributed File System (HDFS), OpenStack Swift, Amazon S3 and Cassandra. Spark is mainly designed to run in-memory, it to handle iterative analysis and more rapid, less expensive data chomping. It provides libraries like a fully-featured machine learning library (MLlib) which used in this work, a graph processing engine (GraphX), data frames and SQL processing, and stream processing.

### 2.4.1 Spark Programming Model

Spark is a parallel programming model runs on clusters in which there is a master node handles the spark driver and send tasks to the executors using a cluster manager. Spark provides two main abstractions for parallel programming and they are RDDs and operations applied on these RDDs.

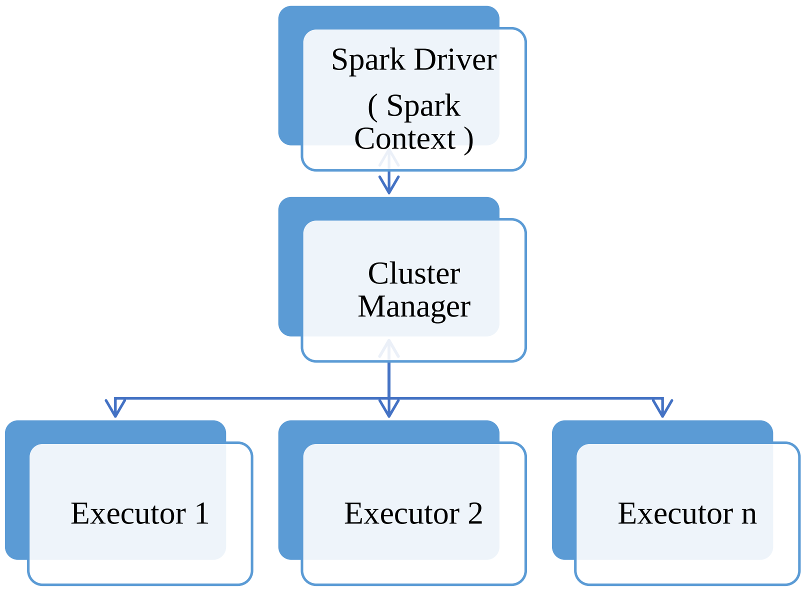

**Figure (2.4):** Spark Programming Model

Zaharia, Chowdhury, Franklin, Shenker, and Stoica (2010) present RDD which stand for Resilient Distributed Dataset as a read only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. The elements of an RDD need not exist in physical storage; instead, RDD contains enough information to compute the RDD starting from data in a reliable storage. This means that RDDs can always be reconstructed if nodes fail. In Spark, each RDD can be constructed using transformations in different ways:

- From a *file* in a shared file system, such as the Hadoop Distributed File ystem (HDFS).
- By *"parallelizing"* a Scala collection in the driver program, which means dividing it into partitions that will be sent to multiple nodes.
- By *transforming* an existing RDD using an operation called *flatMap*, which passes each element through a user-provided function of type $A \Rightarrow List[B]$. Other transformations can be expressed using *flatMap*, including *map* that pass one-to-one function of type $A \Rightarrow B$) and *filter* (pick elements matching a predicate).
- By changing the *persistence* of an existing RDD. By default RDDs are lazy in which show up on demand when they are used in a parallel operation and are discarded from memory after use. However, the persistence of an RDD can change through two actions:
    - *cache* action leaves the dataset lazy, but hints that it should be kept in memory after the first time it is computed, because it will be reused.
    - *save* action evaluates the dataset and writes it to a distributed file system such as HDFS and can be used in future operations on it.

RDD can be operated using various operations called actions like *reduce()* which combines dataset elements using an associative function to produce a result at the driver program. *collect()* sends all elements of the dataset to the driver program. *foreach()* passes each element through a user provided function.

### 2.4.2 Spark Memory Management

The memory model used since Spark version 1.6.0 and up is the unified memory model as shown in Figure 2.5 and it is consist of three main sections as described by (Grishchenko, 2016):

1. **Reserved Memory:** is reserved by the system, which has approximate size of 300MB from RAM, which means it does not participate in Spark memory region size calculations, and its size cannot be changed in any way without Spark recompilation. Note that it is only called *reserved* and not used by Spark in any way, but it sets the limit on what you can allocate for Spark usage. Even if you want to give all the Java Heap for Spark to cache your data, you won't be able to do so as this "reserved" part would remain. Each Spark executor should has at least *1.5 * Reserved Memory = 450MB* heap, it will fail with "please use larger heap size" error message.

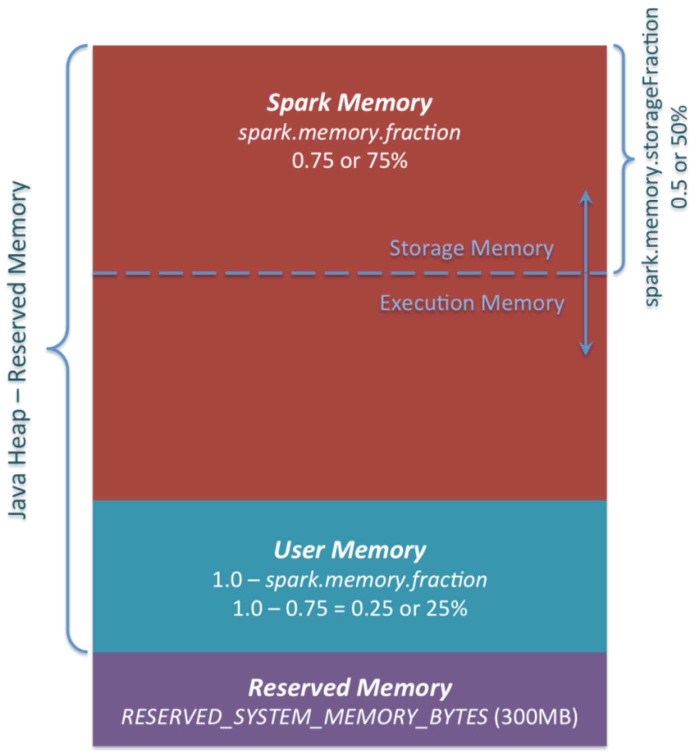

**Figure (2.5):** Apache Spark Unified Memory Model

2. **User Memory:** This is the memory pool that remains after the allocation of Spark Memory, and it is up to the developer to use it in a way he want. It

can store  data structures that would be used in RDD transformations. The size of this memory  can be calculated using the following formula

$$(\textit{Java Heap} - \textit{Reserved Memory}) * (1.0 - \textit{spark.memory.fraction}) \quad \textbf{(2.5)}$$

3. **Spark Memory:** it managed by Apache Spark. Its size calculated as

$$(\textit{Java Heap} - \textit{Reserved Memory}) * \textit{spark.memory.fraction} \quad \textbf{(2.6)}$$

This section is split into 2 areas namely *Storage Memory* and *Execution Memory*, and the boundary between them is set by *spark.memory.storageFraction* and equal 0.5 as default. This boundary is not static, and in case of memory pressure the boundary would be moved, for example one area would grow by borrowing space from another one.

- **Storage Memory:** It is used for both storing Apache Spark cached data and for temporary space serialized data. Also all the broadcast variables are stored there as cached blocks. It does not require that enough memory for unrolled block to be available, in case there is not enough memory to fit the whole unrolled partition it would directly put it to the drive if desired persistence level allows this. As of broadcast, all the broadcast variables are stored in cache with *MEMORY_AND_DISK* persistence level.
- **Execution Memory:** It is used for storing the objects required during the execution of Spark tasks. For example, it is used to store shuffle intermediate buffer on the Map side in memory. It also supports spilling on disk if not enough memory is available, but the blocks from this pool cannot be forcefully evicted by other tasks.

## 2.5 Python Programming Language

Python is an open source high level programming language. Since its rich standard library and dynamic typing and binding, encourages rapid developing of programs and integrating systems more efficiently. In addition, it supports other libraries and extensions available on the web without any charges that gives it ability to be more productive in different fields like web development, game and desktop programming, big data analysis.

Python offers increased productivity, since it does not need code compiling; the maintenance progression is extremely fast. Any bug or incompatible input will never cause a failure. Instead, the interpreter will raises an exception and if the program does not catch the exception, the interpreter prints a stack trace. The python debugger allows checking of local and global variables, validating expressions, setting breakpoints, stepping through the code line by line, and more. Even with the fast debugging the python provides, the quickest debugging is by adding a few print statements to the source.

## 2.6 Performance and Classification Evaluation

The proposed approach is expected to be faster and get better results than the sequential version. For that, we are using the following metrics to verify the text classification system efficiency and the results accuracy (Czech, 2017; Japkowicz & Shah, 2011):

### 2.6.1 Accuracy

**Accuracy** is the percentage of retrieved instances that correctly classified by the classifier.

$$\text{Accuracy} = (TP + TN) / (TP + TN + FP + FN) \qquad \textbf{(2.7)}$$

**TP** is number of positive instances that are labelled correctly by the classifier, **TN** is number of negative instances that labelled correctly by the classifier, **FP is** number of positive instances that are labelled incorrectly by the classifier, and **FN** is number of negative instances that labelled incorrectly by the classifier.

### 2.6.2 Precision

**Precision** is the percentage of predicted documents for the given topic that are correctly classified.

Precision = (TP) / (TP + FP) **(2.8)**

### 2.6.3 Recall

**Recall** is the percentage of the total documents for the given topic that are correctly classified.

Recall = (TP) / (TP + FN) **(2.9)**

### 2.6.4 F-measure

**F-measure** is a standard statistical measure used to measure the performance of a classifier based on precision and recall.

F-measure = (2 * Precision * Recall) / (Precision + Recall) **(2.10)**

### 2.6.5 Speedup

**Speedup** measures the ratio of performance to compare between two programs. For example, comparing versions of program with the same code and different number of processors or comparing two algorithms computing same result. Selecting the correct factor is the base for the comparison and is stated on a case basis. Speedup generally used to show the effect on performance after any resource enhancement and it is computed using the following formula:

$S = T_s / T_p$ **(2.11)**

where $\mathbf{T_s}$ is the execution time using only one processor and $\mathbf{T_p}$ is the execution time using **p** processors.

### 2.6.6 Parallel Efficiency

**Parallel efficiency (E)** measure how much of the available processing power is being used and commonly defined as the speedup (S) divided by the number of units of execution (P) (processors, cores, etc) as presented in the following equation:

$E = S / P$ **(2.12)**

### 2.6.7 Scalability

To test how scalable a system it is a non-functional testing. It measures the ability of a system, a network, or a process to continue to work properly when it is scaled up in size or volume in order to meet a rising need like load supported, the number of transactions, and the data volume. For example: An ecommerce site may be able to handle orders for up to 100 users at a time but scalability testing can be performed to check if it will be able to handle higher loads during peak shopping seasons.

In parallel classification systems, the scalability is estimated based on parallel efficiency, and observed by the system capability to handle work when the problem size or the number of processors are growing.

## 2.7 Summary

In this chapter, we have presented an overview of the basic theoretical and technical foundation used in this research. We have presented a brief description of Arabic language and the challenging faced while working with it, the overall stage in text classification, text classifiers, and Naïve Bayes and Multinomial LR classifiers. We also shed the light on technical foundations are used in this work namely Apache Spark, python programming language. At the end, we specified the performance metrics and classification measures to be used to evaluate the parallel classifiers accuracy.

In the next chapter, we provide an overview of related work in text classification and its parallelization.

# Chapter 3

# Related Work

# Chapter 3

# Related Work

Text classification (TC) for various languages and domains took researchers attention through various researches. There are many TC algorithms and each one of them has its structure, memory, time, and computation complexity. This chapter spots the light on some of these researches considering text language, used algorithm, applied methodology, performance and classification results. These researches are categorized into three categories: sequential TC, text pre-processing tasks affecting TC accuracy, and parallel TC.

## 3.1 Sequential Text Classification Algorithms

Mamoun and Ahmed (2014) conducted a survey on different approaches used for classifying Arabic text determining the most used algorithms and compared them in terms of corpus size, numbers of classes, used classifiers, accuracy, and other. They found a lack in Arabic text classification research, SVM, Naïve Bayesian and K-Nearest Neighbour used frequently, and SVM recommended in relatively large corpus, while C 5.0 recommended with smaller and large corpus. Based on these outcomes, it encourages using Naïve Bayesian with large-scale Arabic text.

Wahbeh and Al-Kabi (2012) conducted text classification on Arabic text collected from different websites of 1000 documents using SVM, Naïve Bayesm and C4.5 classifiers. The used documents had pass through pre-processing tasks to convert them into appropriate format for use on Weka (Hall et al., 2009) toolkit. Their work revealed that Naïve Bayes classifier achieves the highest accuracy followed by the SVM classifier, and C4.5 classifier respectively. The SVM requires the lowest amount of time to build the model needed to classify Arabic documents, followed by Naïve Bayes Classifier, and C4.5 classifier respectively. Since, the dataset size can influence the computation complexity of the classification. The used dataset size in this work was relatively small and did not provide any noticeable improvement on the computation complexity of the classification.

Al-Tahrawi (2015) investigated logistic regression to classify Arabic text for the first time in research. The experiments conducted on Alj-News dataset, which collected from Al-Jazeera Arabic news website. It consists of 1500 Arabic news documents distributed evenly among five classes: Art, Economic, Politics, Science and Sport. Each class has 300 documents 240 for training and 60 for testing. The dataset is applied to different text preprocessing tasks. Khoja stemmer is used , then chi square used as feature selection and a local policy is used to select a reduced features for building the LR classifier (only 1% of each class features). To build LR model, they used the Iteratively Reweighted Least Squares (IRLS) nonlinear optimization algorithm as a fitting procedure. The results indicated LR has very accurate classification performance in which had a precision of 96.49, a recall of 91.67 and a F1-measure of 94.0171. These results show that LR is a competitive Arabic text classifier. Moreover, that encourage using LR in our research since it can be used for classifying larger datasets in a parallelized manner to test how such datasets affects its performance.

## 3.2 Text Preprocessing Techniques affecting Classification Accuracy

In 2015, Alhutaish and Omar have studied the use of the K-Nearest Neighbour (K-NN) classifier, with an $I_{new}$, cosine, jaccard and dice similarities, in order to enhance Arabic text classification. Arabic dataset is used which is consisting of 3,172 documents, distributed into four categories: Arts, economic, politics and sport. Its text represented as non-stemmed and stemmed text, with the use of TREC-2002 light stemmer, in order to remove prefixes and suffixes. However, for statistical text representation, Bag-Of-Words (BOW) and character-level three (3-Gram) were used. In order to reduce the dimensionality of feature space; they used several feature selection methods. The Experiments showed that the K-NN classifier, with the new method similarity $I_{new}$ 92.6% Macro-F1, had better performance than the K-NN classifier with cosine, jaccard and dice similarities. Chi-square feature selection, with representation by BOW, led to the best performance over other feature selection methods using BOW and 3-Gram. This approach reduced the features but its performance accuracy can be different with large-scale Arabic text of high dimensionality and with larger number of categories. The KNN returns better results and performance with small number of categories and small size of text corpus.

In 2013, Al-Thubaity et al. studied the effect of combining five feature selection methods, namely CHI, IG, GSS, NGL and RS, on Arabic text classification accuracy. They used Naïve Bayes classification algorithm to classify a Saudi Press Agency dataset of comprised 6,300 texts divided evenly into six classes. They used for feature representation three schemas, Boolean, TFiDF and LTC. Their work showed slight improvement in classification accuracy for combining two and three feature selection methods and no improvement on classification accuracy when four or all five feature selection methods were combined. The feature selection methods can reduce the computation complexity, text dimensionality, and improve the accuracy rate. Nevertheless, this approach could not do well in the case of reducing computation complexity for classifying text documents with high dimensions. This approach reduced the features on the other hand did not do well with large-scale text of high number of features. However, we could use single method of feature selection to perform on our corpus in a way to reduce the computation and observe how to enhance the accuracy of the results.

Elhassan and Ahmed (2015) conducted Arabic text classification on full words and determine the text preprocessing efficiency on them in the accuracy of both training model and classifier. They used in house corpus of 750 documents from local and international newspaper allocated into five categories: economy, political, religion, sport and technology. Every category contains 150 documents that 105 used for training the classifier and the rest used for testing it. The documents in the corpus preprocessed by used two approaches: observation the data set and extended stop words remove. The experiments applied Sequential Minimal Optimization (SMO), Naïve Bayesian (NB) J48 and K-nearest neighbors (KNN) to build the training models. They showed that the two approaches enhanced the accuracy of the training models and indicated that the SVM algorithm outperformed all the other algorithms regard to F1, Recall and Precision measures. However, we will perform different stemming algorithms on larger Arabic datasets to investigate its effectiveness on the results accuracy.

## 3.3 Text Classification with Parallel Computing

In 2016, Shen et al. produced an improved Naïve Bayes classifier applied using MapReduce model on a hadoop cluster that the learning process of the NB classifier was executed in parallel and the training set was split and distributed among each node in the cluster to accomplish word segmentation statistics. Then start building the classifier model by calculating the probability of each word belonging to each class and overlay, and as a final step getting the probability of the various classes of the document and take the maximum as the classification results. To get better results a large number of the probability vector computations was needed and that was considered a time consuming. For that, the words one by one was processed statistically in parallel by each node. At the end, the classification results was combined. The experiments was conducted on a hadoop LAN cluster of one master node and 9 child nodes. The internet corpus Sogou data of 10 classes was used in the classification process. The tryouts showed that the processing time for the same data scale was reduced by increasing the number of nodes, thus having a better computations speed. It also showed an improvement on the efficiency of the classification with large numbers of documents in which the total recognition ratio of the improved NB classifier reached 91. 2%. This work encourages our approach in using parallelism with classifiers to enhance performance and results especially with large datasets. In spite of that, it did not test various situations like different data language as Arabic and larger number of classes with small features set in which could effect on the performance and efficiency of the classification system.

Abushab (2015) proposed a parallel approach based on the Naïve Bayes algorithm for classifying large scale Arabic text using MapReduce (Dean & Ghemawat, 2010) model. The Shamela corpus used for classification containing 101,647 text documents of eight classes. The approach was tested on a Hadoop (White, 2012) cluster of 16 machines one as name node and the rest as data nodes. The experiments used the generated pre-processed corpus under these representations ([light stemming, root stemming] -> [f, tf-id]). They showed that the parallel classification approach can process large volume of Arabic text efficiently on a MapReduce cluster and significantly improves speedup up to 12 times better than the sequential approach using the same classification algorithm. In addition, classification results showed that the proposed parallel classifier has preserved accuracy up to 97%. This work supports our approach for using clusters, but instead of using Hadoop based cluster, we will use the Apache Spark framework that is assumed to be faster than Hadoop. However, this approach has used representation as well as feature selection methods that can be considered in our approach in which feature selection reduces the size of the vector space and can affect the classification performance.

Xu, Wen, Yuan, He, and Tie (2014) and Caruana, Li, and Qi (2011) both presented a parallel SVM based on MapReduce (PSMR) model for email classification and spam filtering respectively. Traditional SVM training is an intensive computational process. Both works reduced the training time significantly, enhanced accuracy and computation time. Nevertheless, both approaches needed more testing on larger datasets.

Abu Tair and Baraka (2013) proposed a high performance parallel classifier for large-scale Arabic text based on the k-NN algorithm. They evaluated the parallel implementation on a multicomputer cluster that consists of 14 computers, using C++ programming language and the MPI library. They used OSAC Arabic corpus collected from multiple websites; the corpus includes 22,428 text documents. Each text document belongs to one of ten categories. The experimental results on the performance indicated that the parallel classifier design has very good speedup characteristics when the problem size scaled up. In addition, classification results showed that the proposed classifier has achieved accuracy, precision, recall, and F-measure with higher than 95%. This work supports our approach in terms of using parallelism in a cluster, but the volume of text documents used in corpus is small-scale compare to large volume of text documents with high dimensions.

Zhou, Wang, and Wang (2012) proposed a parallel Naïve Bayes classification algorithm based on MapReduce. They built a small cluster with three business machines (1 master and 2 slaves) on Linux. They tested efficiency and scalability of proposed parallel Naïve Bayes algorithm on seven datasets from the UCI Machine Learning repository with different size (from 178 KB to 1 MB). The proposed classifier trained the training data sets to generate the classification model, and then used the model to classify the removed category samples. The proposed model improved algorithm performance when used with large data set. Also enhanced the efficiency of the algorithm. This work supports our approach in terms of using cluster and Apache Spark, as a viable and attractive programming model for large data processing.

Chu et al. (2007) proposed a parallel implementation for many classifiers (weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, Gaussian Discriminant Analysis (GDA), EM, and Back Propagation (NN) using MapReduce model on Shared-memory system. They specify different sets of mappers to calculate them, and then the reducer sums up intermediate result to get the result for the parameters. Their experiment was on a 16 way Sun Enterprise 6000 running Solaris 10. They evaluated the average speed up on ten datasets from the UCI Machine Learning repository with different size, which made their report more convincing. The results showed linear speedup with an increase in number of processors. This work improved the computation time but there is no evidence that the accuracy of proposed parallel classifier improved in terms of dataset size and number of features.

## 3.4 Summary

The preceding researches in this chapter have presented various works covering text classification in Arabic and English languages. We went through researches conducted sequential TC like NB, and LR and we found out that NB can be suitable with Arabic text classification but it need to work more with larger datasets in which the size of a data set could influence the complexity of calculations. And LR is encourage to use with large scale datasets. Another researches we checked out mentioned that using several preprocessing tasks like stemming and feature selection could effect on the classification efficiency and the number of classes in the dataset. The last researches we reviewed applied a parallelized TC highlighted the advantages of using parallelism based on MapReduce model and how could reflect on the TC system performance and classification efficiency.

# Chapter 4

# The Proposed Parallel Classification Approach

# Chapter 4

# Parallel Classification of Arabic Text Using NB and LR

In this chapter, we present the proposed parallel approach for NB and LR classifiers. We give details in the corpuses we used and how we collect them. Then we explain the preprocessing phases of two text corpuses. First, we present the sequential text preprocessing using in house Java program, then we represent the parallel computations used for term weighting namely TF-IDF. After that, we dive in the proposed parallel classification based on MapReduce for NB and LR respectively. We intend in this chapter to explore and describe the related details of the proposed parallel classifiers approach.

## 4.1 The Proposed Parallel Classification Approach

Figure 4.1 illustrates the proposed approach workflow. We first collect and create large-scale Arabic text of various Islamic domains and perform the required text pre-processing methods to represent the text in a suitable form for the classification task.

Arabic text pre-processing is accomplished using in-house Java program. It tokenizes the text into tokens, and then normalizes each token into its standard form. For example, مدرسه – مدرسة , احمد – أحمد. Then removes Arabic stop words like الذين , هؤلاء that does not have any important meaning to reduce the dataset size. After that, derives each token to its stem or root using light1 stemmer like كاتبون - كاتب. Term weighting is performed by calculating TF-IDF terms as feature vectors to generate proper text representations using a Python program running on Apache Spark cluster. Before any term weighting computations, setting the number of features set to use per document eliminates the features numbers.

Both text representation and feature selection are applied on the pre-processed text in parallel on a standalone spark cluster in which the dataset is distributed between

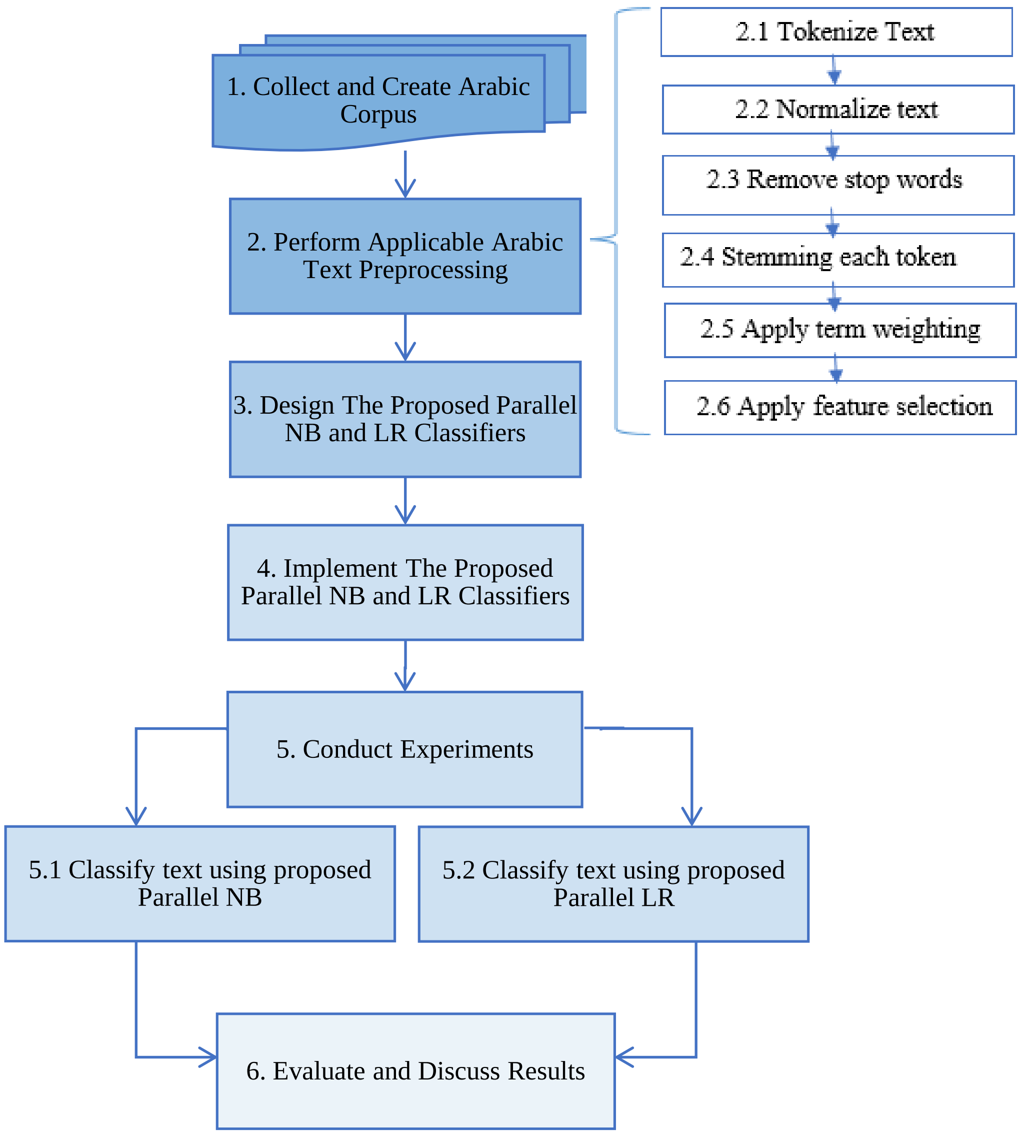
1. Collect and Create Arabic Corpus
2. Perform Applicable Arabic Text Preprocessing
3. Design The Proposed Parallel NB and LR Classifiers
4. Implement The Proposed Parallel NB and LR Classifiers
5. Conduct Experiments
5.1 Classify text using proposed Parallel NB
5.2 Classify text using proposed Parallel LR
6. Evaluate and Discuss Results
2.1 Tokenize Text
2.2 Normalize text
2.3 Remove stop words
2.4 Stemming each token
2.5 Apply term weighting
2.6 Apply feature selection

**Figure (4.1):** The Proposed Text Classification Approach Workflow

the master and worker nodes. We accomplish these steps to see how they can affect the proposed approach efficiency and accuracy.

After preparing the text corpus, we design the proposed parallel model based on MapReduce model for text classification algorithms, namely Naïve Bayes and Logistic Regression. Before implementation, the required frameworks and programming model are installed, configured and tested on each node in the working environment.

These include Apache Spark framework release 1.6.2 (Spark, 2016), Python version 3.5.1 (Rossum, 2015), Java version 1.8, and windows distribution of Hadoop version 2. The designed model for both mentioned classifiers is implemented and realized through the Apache Spark framework. The model split the used dataset into two parts: one for training and the other for testing the classifiers as shown in Figure 4.2.

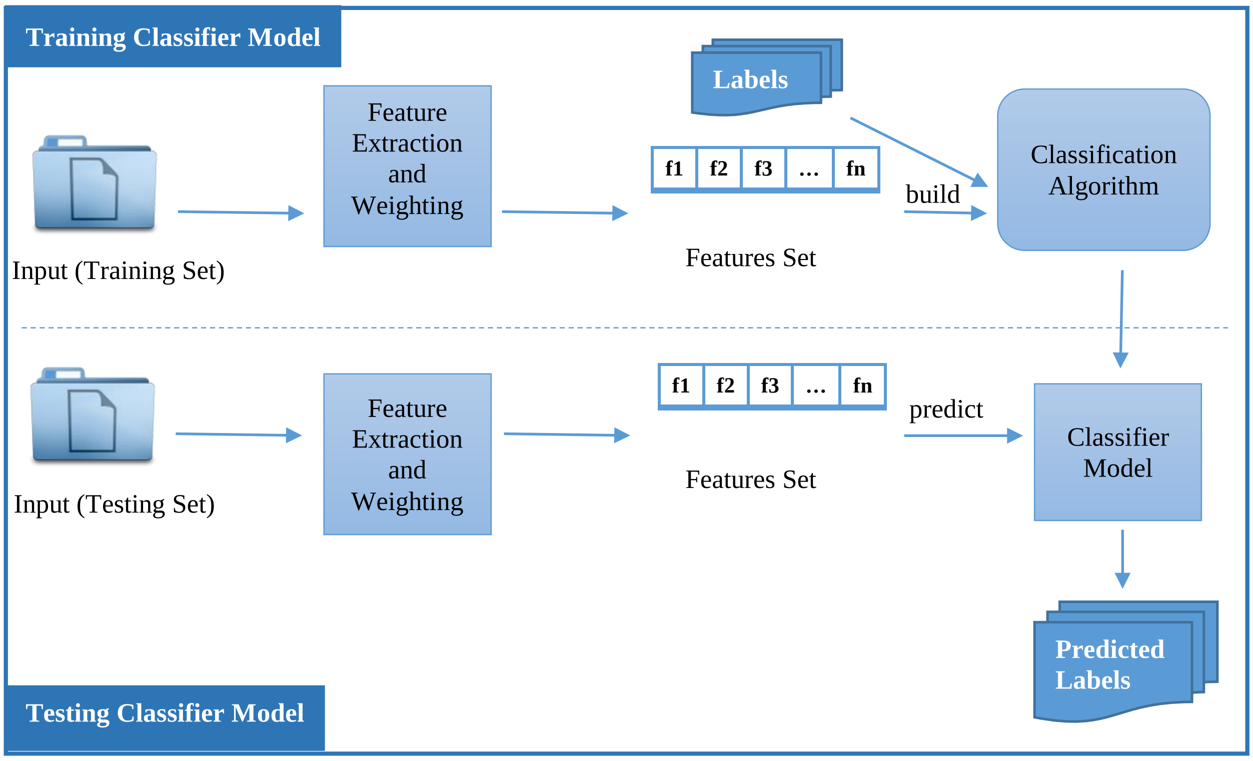

**Figure (4.2):** Classification Process on Apache Spark

Based on the implementation, we prepare the experimental environment with the pre-processed text and a computer cluster of 1, 2, 4, 8, and 16 nodes respectively. The experiments are conducted on both proposed classifiers separately in which the evaluation of the system efficiency and the results accuracy are measured using the classification and performance metrics.

The process of building the parallel Naïve Bayes and LR classifiers which are established the core of our approach includes three stages: text pre-processing, training model, and testing the generated classifier model. Next, the designed model workflow is presented and deliberated based on Figure 4.1 and 4.2.

## 4.2 Creation and Collection of Arabic Text Corpus

One of the obstacles facing this research in Arabic text classification is the lack of suitable Arabic text corpus with suitable large size for training the proposed parallel classifiers NB and LR.

Various Arabic data sets are available for text classification but most of them are not applicable for research and do not meet our experimental requirements of data size and nature for large-scale Arabic text corpus. For that reason, we choose to collect and create two real Arabic text corpuses, which differ in size, number of classes, and features dimensionality.

### 4.2.1 Shamela Corpus

It is a corpus collected by (Abushab, 2015). They collected the documents from Shamela library using tools available in Shamela library software. The process includes converting document files into text format with UTF-8 Encoding using Zilla, which is a word to text converter by software informer.

The collected Shamela corpus is considered as a large-scale dataset covers various Islamic fields in Arabic language, its size is 5 Gigabytes in total, and it is categorized into eight main topics: Aqeda, Ausol, Feqh, Al-Hadith, History, Sirah, Tafser, and Trajem.

### 4.2.2 Al-Bokhary Corpus

Since, the collected Shamela corpus has large size and features dimensionality. We need also to measure classification on a contrast situation where corpus size and features dimensionality are small but number of classes is big and that match with Sahih Al- Bokhary.

Sahih Al-Bokhary is one of the six major hadith collections and the most trusted one along with Sahih Muslim. It includes 97 sections called books. We get it as word files from Shamela Library software. Then we managed these files using in house java program that read the files and extract hadiths matan only which essential for classification and pre-processed them then write them into text files of UTF-8

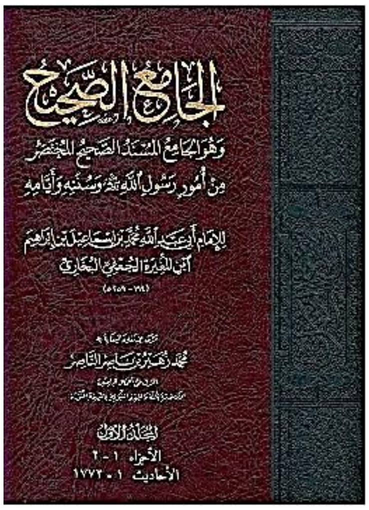
الجامع الصحيح
وهو الجامع المسند الصحيح المختصر
من أمور رسول الله ﷺ وسننه وأيامه
للإمام أبي عبد الله محمد بن إسماعيل بن إبراهيم
ابن المغيرة الجعفي البخاري
محمد زهير بن ناصر الناصر
المجلد الأول
الأجزاء ١ - ٢
الأحاديث ١ - ١٧٧٢

**Figure (4.3):** Sahih Al-Bokhary Front Cover

encoding. The generated corpus contains 21 categories with 4189 hadith in total, which each category contains more than one hundred hadiths.

## 4.3 Text Preprocessing and Term Weighting

This stage is important before building any classifier model, which can help getting better results and performance. It is applied on both corpuses Shamela and Al-Bokhary individually in two consecutive steps: First, text preprocessing which applied sequentially. Then term weighting for the preprocessed text using parallelism.

### 4.3.1 Arabic Text Preprocessing

This step accomplished using in house java program with AraNLP java library that developed by Althobaiti, Kruschwitz, and Poesio (2014) to apply various preprocessing tasks on Arabic text based on the work needs in a sequential manner then save the preprocessed text into text files of UTF-8 encoding to a given path.

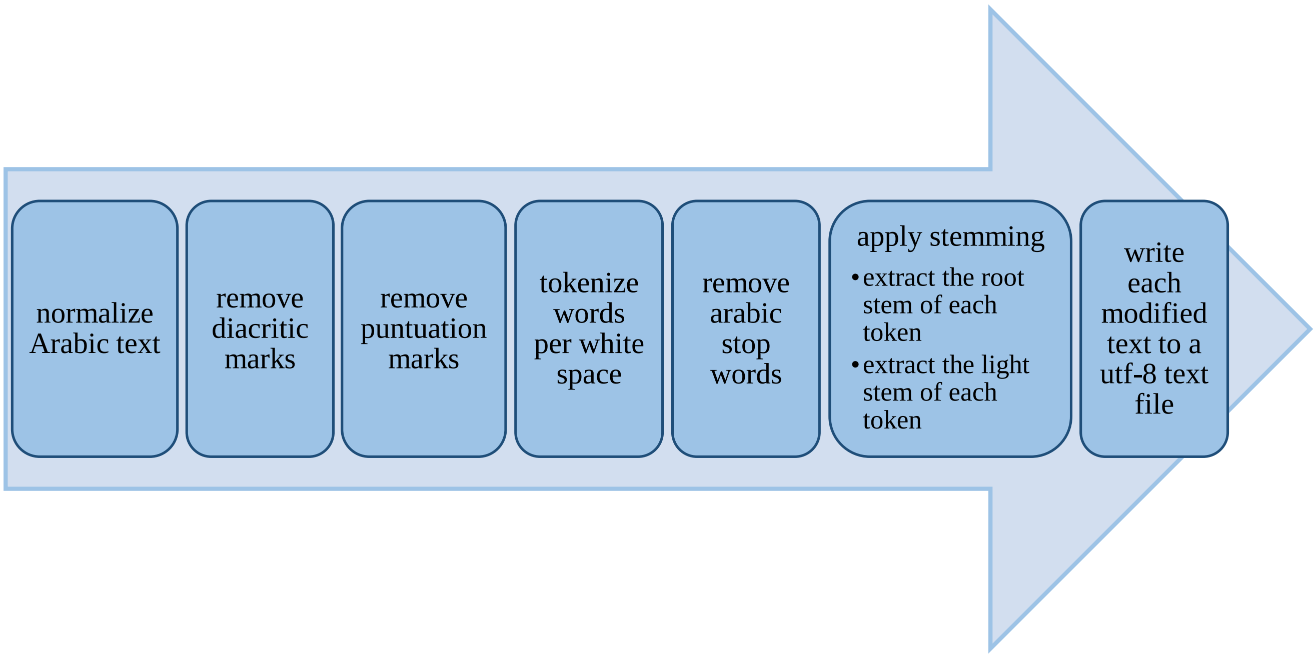
normalize Arabic text
remove diacritic marks
remove puntuation marks
tokenize words per white space
remove arabic stop words
apply stemming
•extract the root stem of each token
•extract the light stem of each token
write each modified text to a utf-8 text file

**Figure (4.4):** First Step in Text Pre-processing in a Sequential Manner

These tasks are described as presented in Figure 4.4:

- **Normalization**: normalize each token into its canonical form per line. In Arabic there are few letters are often misspelled using:
    - The Hamzated forms of Alif (أ , إ, آ) are normalized to bare Alif (ا).
    - The Alif-Maqsura (ى) is normalized to a Ya (ي).
    - The Ta-Marbuta (ة) is normalized to a Ha (ه).
    - Remove tatweel. For example: (حركـــات) to (حركات)
    - Remove numbers and special characters
    - Remove Excessive spaces.
- Remove diacritic and punctuation marks.
- **Tokenization:** is the process of breaking text to its element words. The Arabic text divided by white space into tokens.
- **Arabic Stop words removal:** remove any token considered as a stop word and it is not content bearing such as هما ,هي ,هم ,في.
- **Stemming:**
    - Derive each token to its root using root stemmer**.**
    - Derive each token to its stem using light stemmer 1.

- **Writing to files:** the preprocessed Arabic text written into text files of UTF-8 encoding. For each text file in the corpus, it is saved in three forms to its corresponding given path:
    - The pre- processed text *without stemming.*
    - The pre- processed text *with root stemming.*
    - The pre- processed text *with light stemming.*

### 4.3.2 Term Weighting

This step is executed in a parallel way as part of the proposed parallel approach using Apache Spark (pySpark API) and MLlib API for better computation performance. We use Term Frequency-Inverse Document Frequency (TF-IDF) as vector representation for term weighting on each corpus.

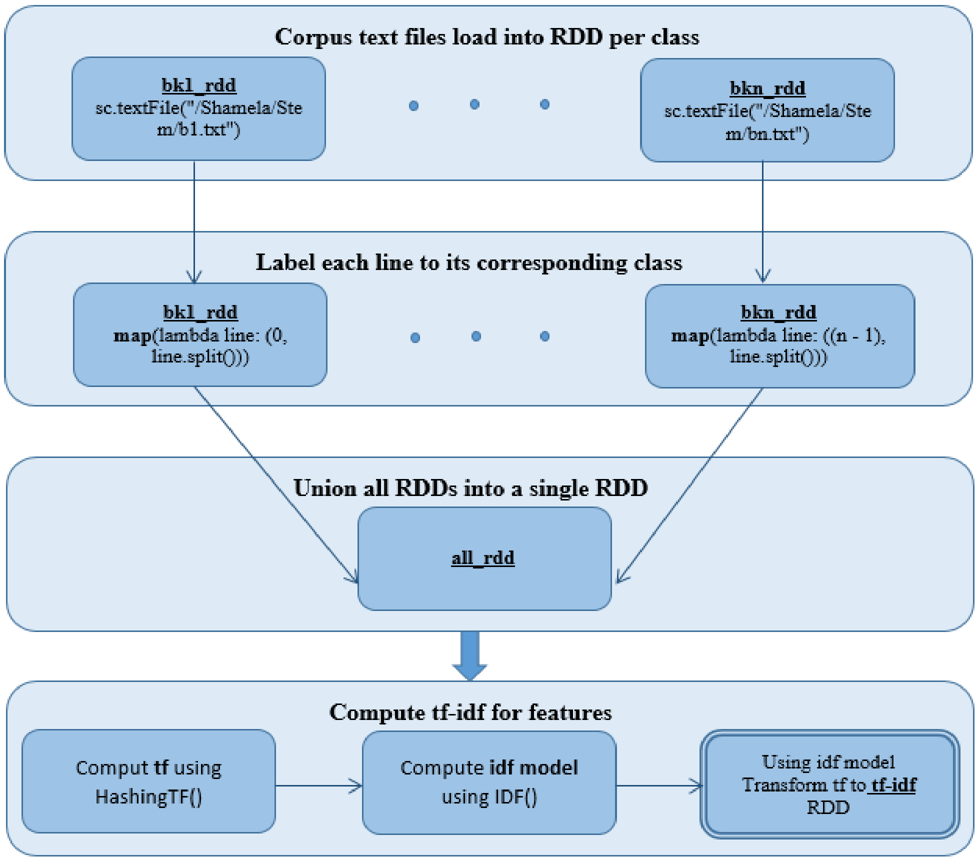
Corpus text files load into RDD per class
bk1_rdd
sc.textFile("/Shamela/Stem/b1.txt")
bkn_rdd
sc.textFile("/Shamela/Stem/bn.txt")
Label each line to its corresponding class
bk1_rdd
map(lambda line: (0, line.split()))
bkn_rdd
map(lambda line: ((n - 1), line.split()))
Union all RDDs into a single RDD
all_rdd
Compute tf-idf for features
Comput tf using HashingTF()
Compute idf model using IDF()
Using idf model Transform tf to tf-idf RDD

**Figure (4.5):** Term Weighting Steps in the Proposed Parallel Approach where **n** is the number of categories in the corpus

As a start, we read documents into a single Resilient Distributed Dataset (RDD) per category then map each document into a tuple of document category id it belongs to and array of tokens of its content. Therefore, we have eight RDDs for Shamela corpus and twenty-one RDDs for Bokhary corpus. Note that the meaning of words or their order will not make a difference in the computations and results since we use the bag of words concept.

After mapping the documents of all categories, we start computing TF-IDF values for all the read documents into a single RDD. First, we count **tf** value of each token in the documents using HashingTF class that uses the Scala native hashing. According to the computed **tf** vectors, the IDF model is generated. Then the IDF model transforms **tf** vectors into TF-IDF vectors for each document where the IDF values are the same across all documents.

Before building the classification model, the total RDD of TF-IDF vectors of all the documents in the dataset is grouped into single RDD of **LabeledPoint**s that contains corpus categories and their corresponding features, since the classifier takes input of RDD of **LabeledPoint**s. The final RDD is split into two RDDs; trainRDD for training the classifier model and testRDD for testing the generated classifier model as presented in the next sections 4.3.2 and 4.3.3.

## 4.4 Training Stage

In this research, the training stage uses the preprocessed text which represented as TF-IDF vectors in trainRDD for building the classifier models NB and LR that described in detail next for later use in prediction and evaluation as described in Section 4.5.

### 4.4.1 Naïve Bayes (NB)

Naïve Bayes (NB) is one of the machine learning algorithms commonly used in text classification which mentioned earlier in Section 2.2.1 with its needed computations. Parallel NB is applied on Apache Spark that realizes enhanced MapReduce model. For simplicity, Figure 4.6 visualizes the parallel NB using Apache Spark MlLib API for training classifier.

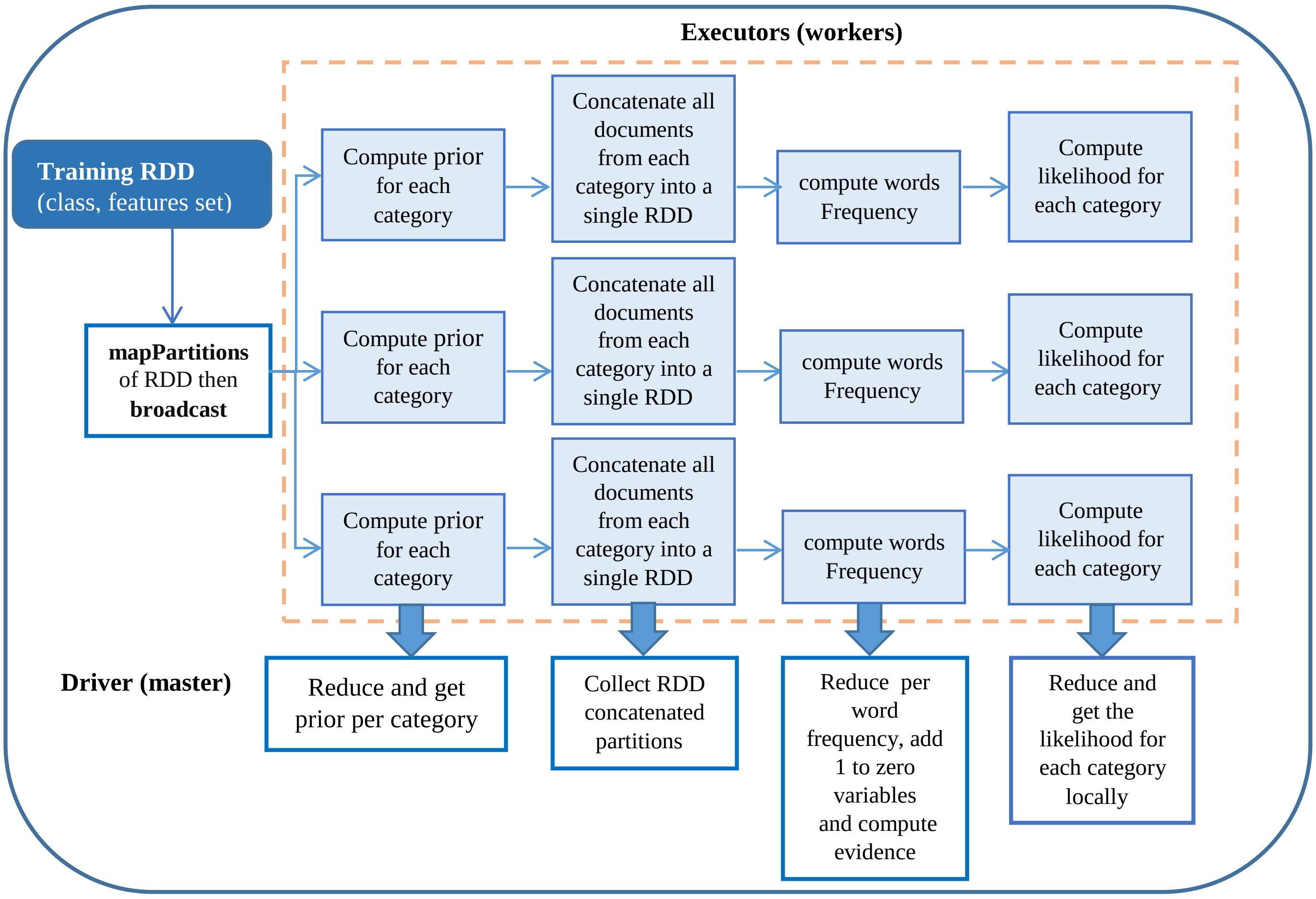
Executors (workers)
Training RDD
(class, features set)
mapPartitions of RDD then broadcast
Compute prior for each category
Concatenate all documents from each category into a single RDD
compute words Frequency
Compute likelihood for each category
Compute prior for each category
Concatenate all documents from each category into a single RDD
compute words Frequency
Compute likelihood for each category
Compute prior for each category
Concatenate all documents from each category into a single RDD
compute words Frequency
Compute likelihood for each category
Driver (master)
Reduce and get prior per category
Collect RDD concatenated partitions
Reduce per word frequency, add 1 to zero variables and compute evidence
Reduce and get the likelihood for each category locally

**Figure (4.6):** Training Naive Bayes Classifier Data Flow on Apache Spark

Parallel NB classifier inputs the training set **TrainingRDD** and divides it into partitions then distributes the partitions to the executors to execute computations on them and return the results to the master machine which runs the driver program.

First computations by the executors after caching the partitioned RDD which came from the master is counting the documents per category. Then the driver reduces from the executors and get the total documents per category and the total documents in the entire training set RDD. After that the master compute the prior probability per category existing in the corpus and cache it. If any machine memory cache is being full, the data will be written on disk instead. Then the executors concatenate all RDD partitions they have into a single RDD. At that time, the driver collects the concatenated RDDs from the executors and unions into single RDD to hold all the features set without labeling. The new merged RDD also is partitioned and distributed among the executors. The driver set task to the executors to compute each feature frequency. After that, the driver reduces each feature frequency from executors and sum the frequencies per feature and computes the evidence. Finally, the executors compute the conditional probability (likelihood) of each feature with the given category. The driver reduces and gets the likelihood per feature given category. At the end, the driver has the generated NB model and broadcast it to the executors for prediction which is described in Section 4.3.3.1.

### 4.4.2 Logistic Regression (LR)

LR known with iterative computations and that makes it suitable to be parallelized. In Figure 4.7 the LR training data flow is depicted.

The driver broadcasts the training partitions RDD and the initialized weights. After that, loops are conducted until converge and reach insignificant results. In every loop, the executors compute the loss and the gradient for each document, and sum them up locally. Then the driver reduces and getSum from executors totalLoss and totalGradient which have two parts: the model that depends on data while the loss of regularization does not depend on data.  Keep in mind that the loss and gradient of each document is independent.

After that, handle regularization using L-BFGS optimizer to find tthe next step. When the loops are finished, the final model weights are available on the driver.

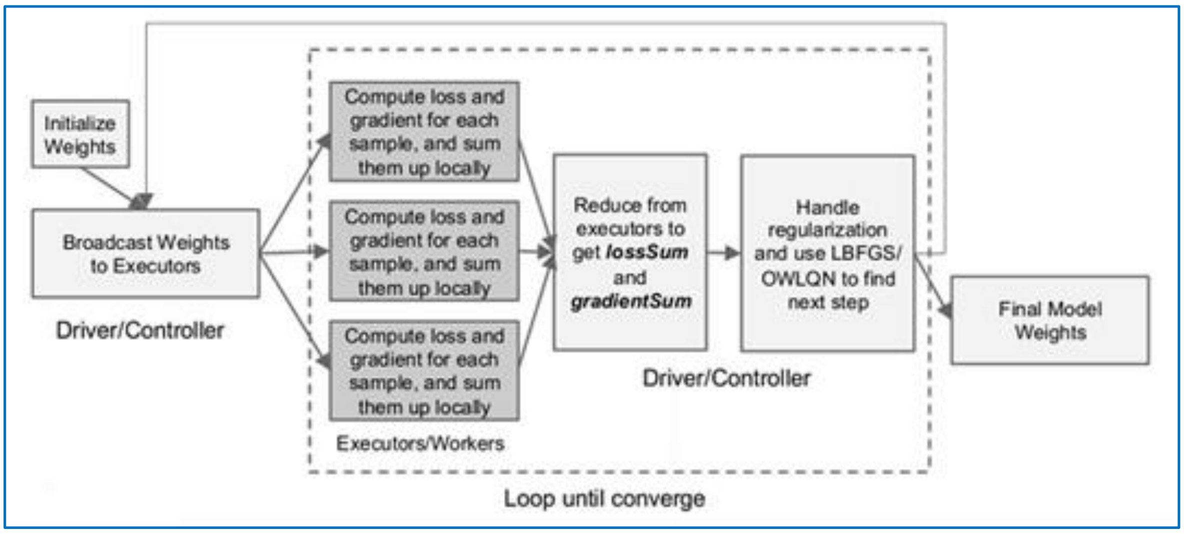
Initialize Weights
Broadcast Weights to Executors
Driver/Controller
Compute loss and gradient for each sample, and sum them up locally
Compute loss and gradient for each sample, and sum them up locally
Compute loss and gradient for each sample, and sum them up locally
Executors/Workers
Reduce from executors to get ***lossSum*** and ***gradientSum***
Handle regularization and use LBFGS/ OWLQN to find next step
Driver/Controller
Final Model Weights
Loop until converge

**Figure (4.7):** Training Logistic Regression Classifier data flow on Apache Spark

## 4.5 Testing Stage

In this stage, **testRDD** is the RDD used for prediction and is resemble 30% of the data and we also make prediction using the entire data **dataRDD** as we described is next Sections 4.5.1 and 4.5.2.

### 4.5.1 NB

Testing the NB has only one computation to estimate the document probability existence per category and to select the document category with the highest probability score. The data flow of testing the paralleized NB is shown in Figure 4.8.

Before any predictions, the trained NB model and the testing RDD are broadcasted to the executors. To overcome the zero variables, the laplacian smoothing is used through calculating the logs of the probabilities by the executors. The driver reduces and gets logs of the priors and the conditional probabilities. For each document in the testing RDD, the document probabilities of belonging to each category are calculated and saved. Then the highest probability is chosen as the predicted document category.

### 4.5.2 LR

Before making any predictions, the testRDD and LR generated model are broadcasted to the executors. For each document, iterative computations are made until reaching the maximum probability of the given category.

The document probability predictor function is the regression coefficient in which the executors compute the coefficients for each category. Then the driver reduces and sums up the results and takes the highest probability as the predicted category.

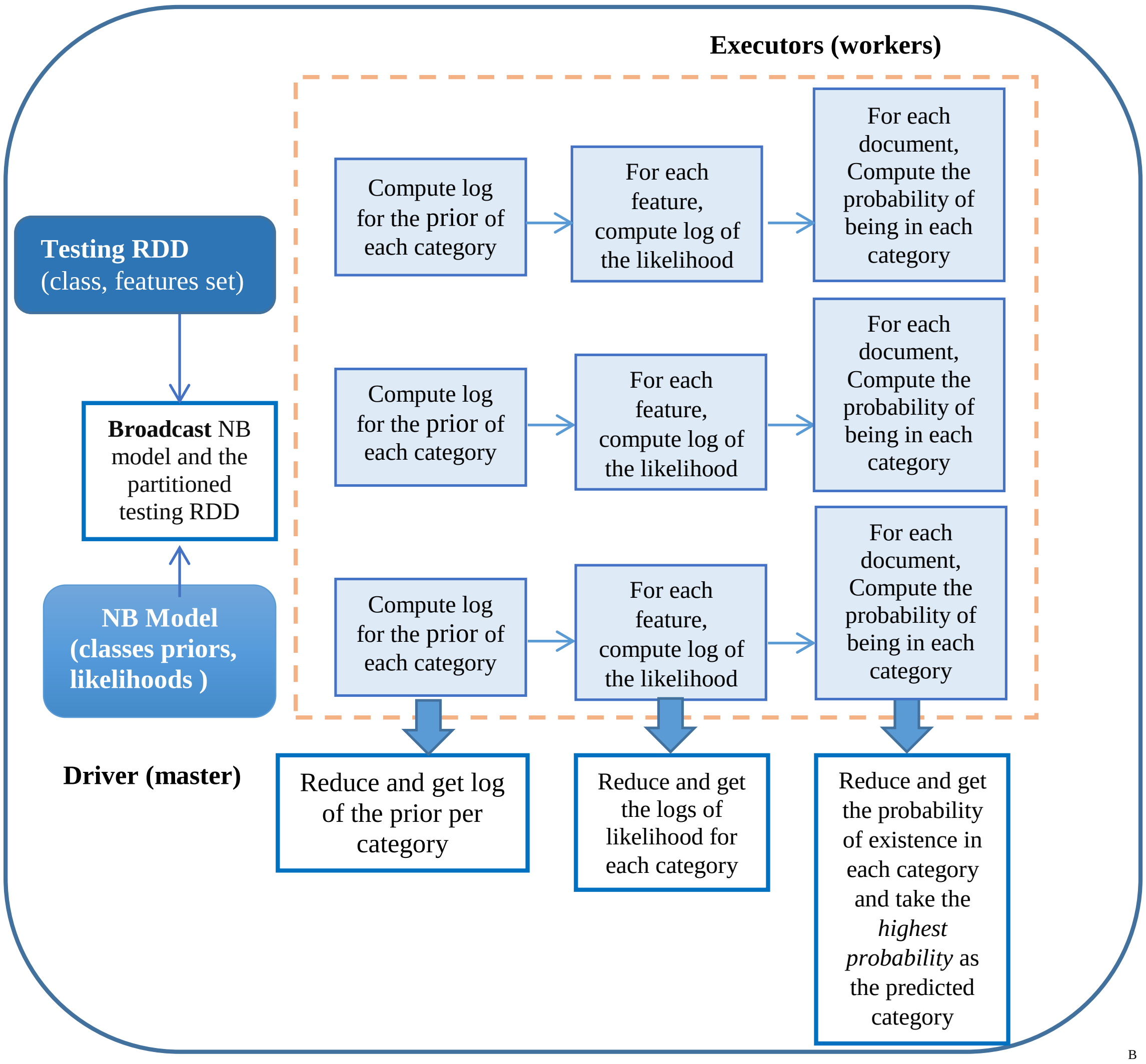
Executors (workers)
Testing RDD (class, features set)
Broadcast NB model and the partitioned testing RDD
NB Model (classes priors, likelihoods )
Driver (master)
Compute log for the prior of each category
For each feature, compute log of the likelihood
For each document, Compute the probability of being in each category
Compute log for the prior of each category
For each feature, compute log of the likelihood
For each document, Compute the probability of being in each category
Compute log for the prior of each category
For each feature, compute log of the likelihood
For each document, Compute the probability of being in each category
Reduce and get log of the prior per category
Reduce and get the logs of likelihood for each category
Reduce and get the probability of existence in each category and take the *highest probability* as the predicted category

**Figure (4.8):** Testing Naive Bayes Classifier Data Flow on Apache Spark

## 4.6 Summary

In this chapter, we have presented the proposed parallel classification approach based on MapReduce model on Apache Spark for NB and LR. We collected Shamela corpus and collected Al-Bokhary corpus to use in the experiments. We have preprocessed the two corpuses in two phases. The first phase has been accomplished by hand made Java program and has saved the new text into UTF-8 text files. Then has been computed TF-IDF in parallel manner in which term frequency is computed using hashing function of Scala, then the IDF model. At the end, IDF model has been used to transform TF-IDF to RDDs. Also, we have used two parallel MapReduce methods, one for the training stage and the other for the testing stage. Both stages have been implemented differently based on the NB and LR.

In the next chapter, we present and discuss the experimental environment settings and the results of the experiments carried out to realize and evaluate the proposed parallel classifiers NB and LR.

# Chapter 5

# Experimental Results and Approach Evaluation

# Chapter 5

# Experimental Results and Evaluation

In this chapter we present and analyse the experimental results to evaluate the proposed NB and LR parallel classifiers using Apache Spark which realizes the two algorithms as MapReduce model. To prove any enhancement on the classifiers' performance and efficiency, parallel NB and LR classifiers are used in the experiments which are provided as part of the Apache Spark MLlib library. The used corpuses in the experiments are described with their main characteristics. The experimental environment and its settings are also described. The applied steps of the implementation of the NB and LR parallel classifiers are presented together with conducting and measuring the different performance and classification metrics. Finally, the experimental results are extracted and discussed together with a comparison between the used NB and LR parallel classifiers. First we present the Corpus used in the experiments.

## 5.1 Corpus

This research required large scale dataset to evaluate the parallel classifiers NB and LR called Shamela, and Al-Bokhary which is contrast to Shamela.

### 5.1.1 Shamela Corpus

Shamela corpus has a size of almost 5 Giga Bytes with eight categories (Abushab, 2015). Preprocessing of the text corpus is performed using three different stemming types to observe its effects on the classification accuracy and save the processed corpus in utf-8 text files.

Since a large number of documents is needed for the experiments, we split each preprocessed text file into smaller text files of 30 kilobytes per file. The number of documents per category for each stemming approach is shown in Table 5.1.

**Table (5.1):** Shamela Corpus Count Documents per Category

| # | Category | Books | Documents (Without Stemming) | Documents (Light Stemming) | Documents (Root Stemming) |
|---|---|---|---|---|---|
| **1** | Aqeda | 434 | 12,427 | 9,507 | 7,388 |
| **2** | Ausol | 313 | 4,379 | 3,406 | 2,560 |
| **3** | Feqh | 672 | 40,415 | 30,786 | 24,155 |
| **4** | Hadith | 526 | 40,756 | 30,882 | 24,339 |
| **5** | History | 186 | 15,998 | 12,585 | 9,770 |
| **6** | Sirah | 372 | 8,021 | 6,251 | 4,939 |
| **7** | Tafser | 222 | 32,229 | 24,935 | 19,146 |
| **8** | Trajem | 1,004 | 26,268 | 20,629 | 16,022 |
| **-** | **Total** | **3,729** | **180,493** | **138,981** | **108,319** |

It is noticed that the stemming approach affects the number and size of documents per category as presented in Figure 5.1 in which root stemming reduces number of the original corpus documents without applying any stemming approach in total to almost 40% more than light stemming 1 which reduces to almost 23%. We also noticed that Feqh and hadith are the biggest categories in size even they have number of documents less than Trajem.

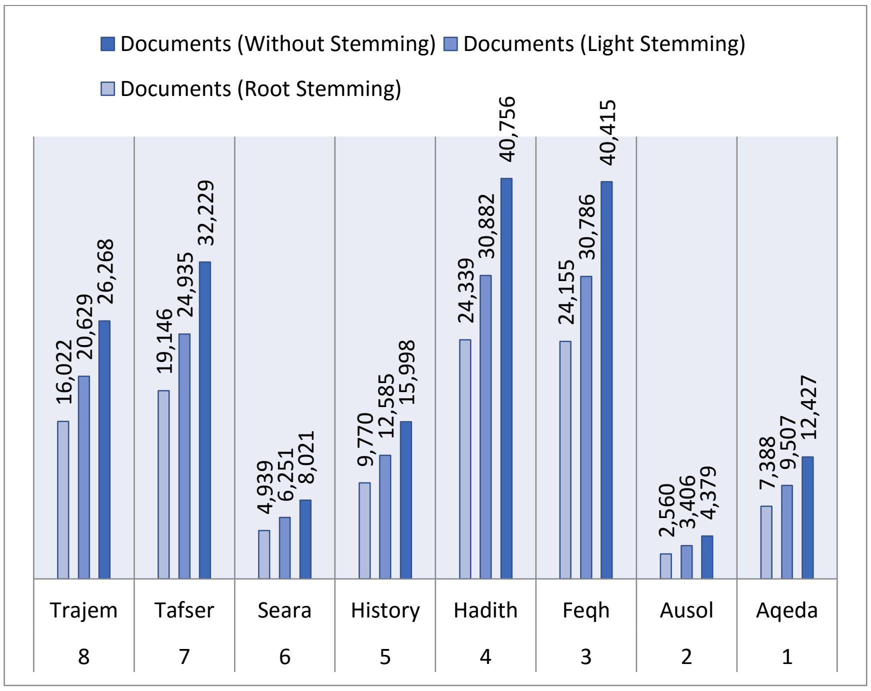

**Figure (5.1):** Stemming Effectiveness on Shamela Corpus

### 5.1.2 Al-Bokhary Corpus

We extract this corpus from Sahih Al-Bokhary with only hadith Matans as mentioned earlier in Section 4.2.2. We summarize the corpus categories and number of hadiths in Table 5.2. We need to notice that Matan text is small in size for that there is not any observations to be considered after preprocessing.

**Table (5.2):** Al-Bokhary Corpus Categories

| # | Book | الكتاب | Number of Hadith |
|---|---|---|---|
| **1** | Book4 | الوضوء | 108 |
| **2** | Book8 | الصلاة | 166 |
| **3** | Book10 | الأذان | 266 |
| **4** | Book23 | الجنائز | 156 |
| **5** | Book24 | الزكاة | 114 |
| **6** | Book25 | الحج | 241 |
| **7** | Book30 | الصوم | 110 |
| **8** | Book56 | الجهاد والسير | 286 |
| **9** | Book59 | بدء الخلق | 127 |
| **10** | Book60 | أحاديث الأنبياء | 154 |
| **11** | Book61 | المناقب | 144 |
| **12** | Book62 | أصحاب النبي ﷺ | 119 |
| **13** | Book63 | مناقب الأنصار | 170 |
| **14** | Book64 | المغازي | 464 |
| **15** | Book65 | تفسير القرآن | 479 |
| **16** | Book67 | النكاح | 180 |
| **17** | Book77 | اللباس | 181 |
| **18** | Book78 | الأدب | 252 |
| **19** | Book80 | الدعوات | 106 |
| **20** | Book81 | الرقاق | 181 |
| **21** | Book97 | التوحيد | 185 |
| | **Total** | **الاجمالي** | **4189** |

## 5.2 Experimental Environment

The experimental environment is built on an Apache Spark cluster of 16 machines as workers (executors) and a single machine as Master (driver manager). All cluster machines are Dell Laptops with 64 bit, Intel Core i3-330M 2.53GHz, 500GB HDD, and 4GB RAM. 3 GB RAM are reserved for apache spark driver and executors.

These machines are connected through local area network with speed of 10/100 Mbps. Windows 10 is the running operating system on them and set them up with Apache Spark framework release 1.6.2 (Spark, 2016), Python version 3.5.1 including *py4j* and *numpy* packages (Rossum, 2015), Java JDK version 1.8, and windows distribution of Hadoop version 2.

The proposed parallel classifier approach has been implemented on Apache Spark cluster with these predefined settings which listed and explained in Table 5.3 that placed in the *spark.defaults* config file.

**Table (5.3):** Spark Cluster Configuration Settings

| # | Option | Value |
|---|---|---|
| 1 | spark.driver.memory | 3g |
| 2 | spark.executor.memory | 3g |
| To set amount of memory to use for each node in the cl | | |
| 3 | spark.driver.extraJavaOptions | -XX:+UseCompressedOops |
| 4 | spark.executor.extraJavaOptions | -XX:+UseCompressedOops |
| This option set for master and executors machines to reduce java memory usage and garbage collections. | | |
| 5 | spark.python.worker.reuse | true |
| Reuse Python worker or not. If yes, it will use a fixed number of Python workers, does not need to fork() a Python process for every tasks. It will be very useful if there is large broadcast, then the broadcast will not be needed to transfered from JVM to Python worker for every task. | | |
| 6 | spark.network.timeout | 180s |
| set timeout for all network interactions | | |
| 7 | spark.locality.wait | 30s |
| How long to wait to launch a data-local task before giving up and launching it on a less-local node. | | |
| 8 | spark.scheduler.maxRegisteredResourcesWaitingTime | 60s |
| Maximum amount of time to wait for resources to register before scheduling begins. | | |
| 9 | spark.task.cpus | 2 |
| Number of cores to allocate for each task. | | |
| 10 | spark.executor.heartbeatInterval | 60s |
| Interval between each executor's heartbeats to the driver. Heartbeats let the driver know that the executor is still alive and update it with metrics for in-progress tasks. | | |
| 11 | spark.task.maxFailures | 50 |
| Number of individual task failures before giving up on the job. | | |
| 12 | spark.default.parallelism | 2 |
| Default number of partitions in RDDs returned by transformations like *join*, *reduceByKey*, and *parallelize* when not set by user. | | |

Naïve Bayes and Logistic Regression classifier are available in Apache Spark framework, which is highly scalable with large scale dataset. As we will describe in next Sections 5.3 and 5.4.

## 5.3 The Parallel NB and LR Classifiers Implementation in Apache Spark

The proposed parallel classifiers utilizes Apache Spark in-memory distributed data processing platform, and parallelized NB and LR classification uses Spark MLlib library as a MapReduce realization of machine learning.

We follow the steps as described (Karau et al., 2015) for building Apache Spark standalone cluster with Apache Spark version 1.6.2 and for the implementation of the parallel Naïve Bayes and Logistic Regression classifier using MLlib library. The procedure of developing the overall classification approach takes in the following steps:

1. Per category, the files are merged into a single file where each file is located in a single line in the new file (using in house python program) to reduce time spend for opening and reading from files.
2. All text preprocessing (see Section 4.3.1) is performed on Shamela and Al-Bokhary corpuses. It is saved as text file directories into the master then copied to all executors. Apache Spark reads the input Arabic text files document into data blocks RDDs. It stores the metadata of each block in the master and all the data blocks in the executors.
3. Naïve Bayes and LR work with TF-IDF vectors associated to the original text per category that have been accomplished in the last step in preprocessing phase (see Section 4.3.2.)
4. Join all TF-IDF vectors per category into single RDD, then Split it into training set and testing set. In the experiments, we selected 70%, 30% as split percentage for training and testing respectively as follows.

***Python Spark Code:***

```
training, test = dataRDD.randomSplit([0.7, 0.3], seed=0)
```

5. The training phase is conducted as a parallel NB classifier and as a parallel LR classifier separately on the training set. The output of this step is the

classifier model of NB and LR (see Appendix A.3 for the implementation source code.)

6. The testing phase is conducted to test the generated classifier model from the previous step on the testing set and record the results, analyze and discuss them (see Appendix A.3 for the implementation source code.)

## 5.4 Experimental Results and Discussion

The two parallel classifiers are evaluated through precision, recall, accuracy and f-measure based on features size.

### 5.4.1 Performance Evaluation

In the experiments, we used Shamela corpus, Shamela-More corpus (which is a copy of the Shamela corpus but with smaller splitted files) as shown in Table 5.1, and Al-Bokhary corpus. Each corpus has three groups based on applied stem approach numbered A, B, and C (A: No stem used, B: Light1 stem, and C: Root stem). We represented 900 features from Shamela and Shamela-More corpuses and 10,000 features from Al-Bokhary to train the proposed NB and LR parallel classifiers on Apache Spark Standalone cluster. We have followed the described term weighting in Section 4.3.2 to have data representations proper for the classification.

To measure the proposed NB and LR parallel classifiers, we have executed in parallel on a single machine and on many cluster nodes 2, 4, 8, 16 respectively for one to four rounds per experiment and registered execution time and calculate the speedup ratio, efficiency and system scalability as described in Section 2.6. The execution time of the proposed parallel system for building the classifiers are represented in Figure 5.2 and Figure 5.3.

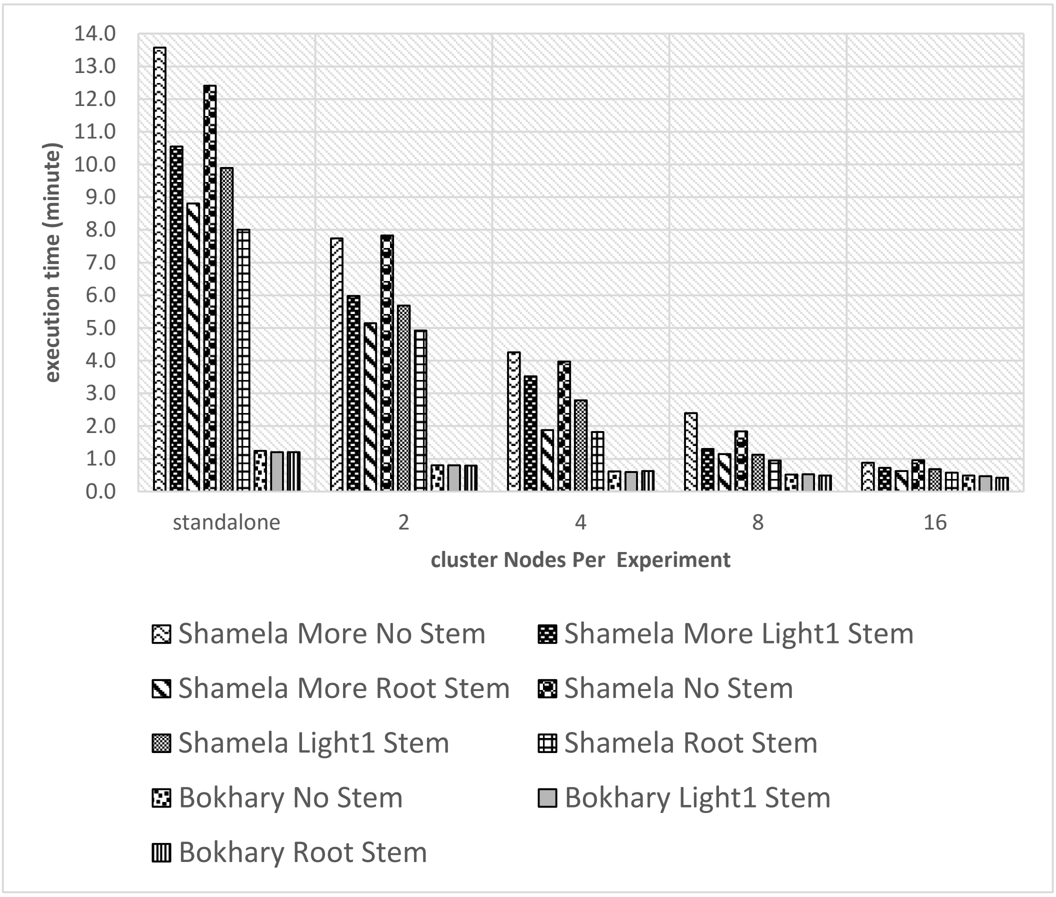

**Figure (5.2):** Execution Time of Parallel NB Classifier on Spark Cluster Nodes

We observed that by doubling the number of nodes per experiment where the execution time is decreased almost in half for large datasets. We also noticed from Table 5.4 and Table 5.5 that total number of files effects more on the execution time than size of text files in which non-stemmed Shamela (3,729 documents) has took 12.413 minutes and non-stemmed Shamela-More (180,493 document) has took 13.570 minutes on a single machine. In contrast, Al-Bokhary corpus has took 1.23 minute even it has more than 4,000 documents but they are very small in size close to 1 KB per file and that means that size of the file could affect relatively to total number of files.

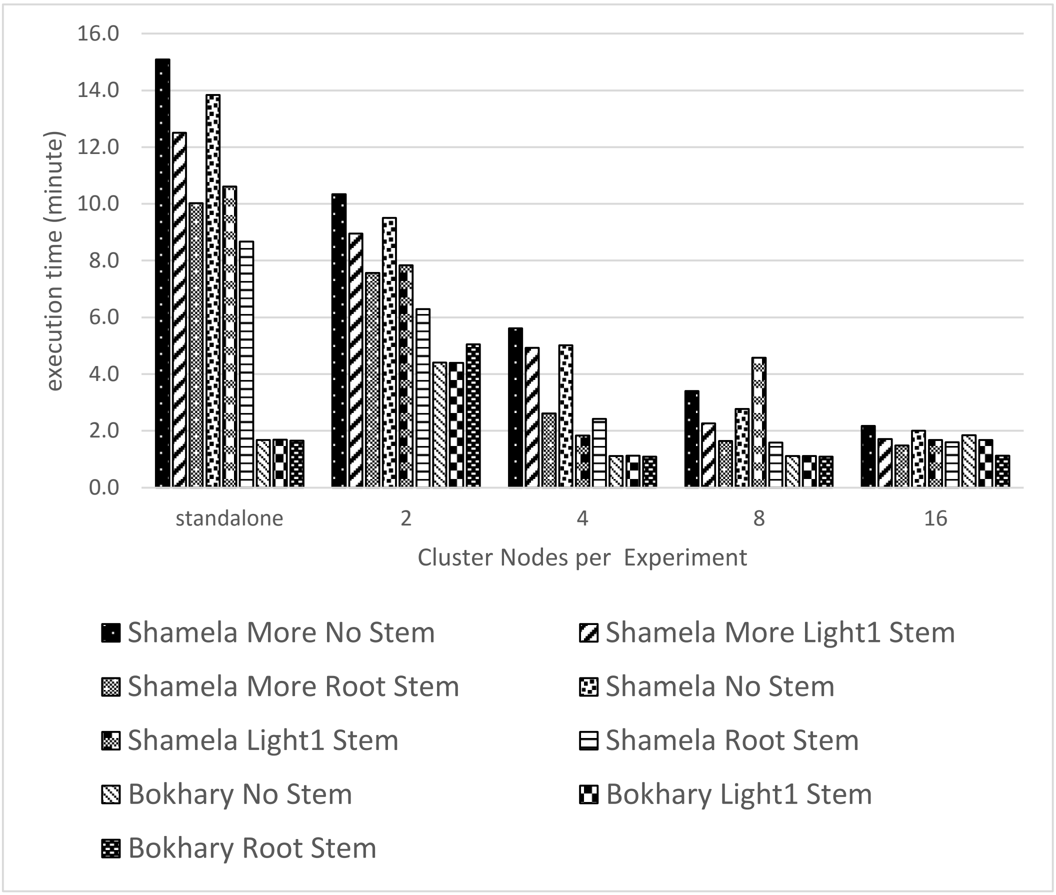

**Figure (5.3):** Execution Time of Parallel LR Classifier on Spark Cluster Nodes

The execution time of the proposed parallel classifiers is slower than usual because we used RDDs which are considered to be slower than DataFrames under Python and Apache Spark (pyspark API). When we track the execution time of both parallel classifiers NB and LR, LR has took more execution time than NB because of the iterative operations nature of LR while NB has a sequential computations.

**Table (4.4):** Execution Time (min) of Parallel NB Classifier on Spark Cluster Nodes using 70% - 30% Data Split

| **Corpus** / **# of Executors** | **Shamela -More ( 900 feature)** | | | **Shamela ( 900 feature)** | | | **Al-Bokhary ( 10000 feature)** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **A1** | **B1** | **C1** | **A2** | **B2** | **C2** | **A3** | **B3** | **C3** |
| **standalone** | 13.570 | 10.553 | 8.809 | 12.413 | 9.895 | 8.005 | 1.238 | 1.202 | 1.200 |
| **2** | 7.740 | 5.980 | 5.144 | 7.829 | 5.680 | 4.918 | 0.800 | 0.797 | 0.787 |
| **4** | 4.260 | 3.519 | 1.881 | 3.973 | 2.789 | 1.821 | 0.611 | 0.592 | 0.628 |
| **8** | 2.393 | 1.301 | 1.140 | 1.843 | 1.124 | 0.949 | 0.516 | 0.527 | 0.488 |
| **16** | 0.883 | 0.727 | 0.628 | 0.958 | 0.679 | 0.579 | 0.486 | 0.470 | 0.417 |

**Table (5.5):** Execution Time (min) of Parallel LR Classifier on Spark Cluster Nodes using 70% - 30% Data Split

| Corpus / # of Executors | Shamela -More ( 900 feature) | | | Shamela ( 900 feature) | | | Al-Bokhary ( 10,000 feature) | | |
|---|---|---|---|---|---|---|---|---|---|
| | **A1** | **B1** | **C1** | **A2** | **B2** | **C2** | **A3** | **B3** | **C3** |
| **standalone** | 15.093 | 12.505 | 10.026 | 13.838 | 10.609 | 8.669 | 1.681 | 1.689 | 1.654 |
| **2** | 10.344 | 8.954 | 7.562 | 9.503 | 7.832 | 6.296 | 4.408 | 4.392 | 5.047 |
| **4** | 5.610 | 4.925 | 2.612 | 5.014 | 4.572 | 2.415 | 1.111 | 1.125 | 1.094 |
| **8** | 3.398 | 2.258 | 1.640 | 2.772 | 1.836 | 1.590 | 1.115 | 1.114 | 1.094 |
| **16** | 2.171 | 1.711 | 1.485 | 2.001 | 1.674 | 1.599 | 1.845 | 1.682 | 1.121 |

Next, we discuss the proposed parallel system performance based on the execution time we have collected.

### *5.4.1.1 Speedup*

From the execution time, we have computed the speedup ratio that shows the improvement in the speed of execution of the proposed parallel system which executed on Apache Spark cluster with different nodes 2, 4, 8, 16 respectively and the results are illustrated in Figure 5.4 and Figure 5.5.

We note that with large datasets Shamela and Shamela-More, the parallel NB classifier is almost close to the ideal speed up ratio over the cluster nodes, while the parallel LR classifier is far away from the ideal speed up ratio after using 4 nodes which means the LR needs a lot of resources to reach the ideal speed up. On the other hand, Al-Bokhary classifier has relative slow speed up to its small size. And that means that increasing number of executors do not effect on the speedup ratio with small size datasets.

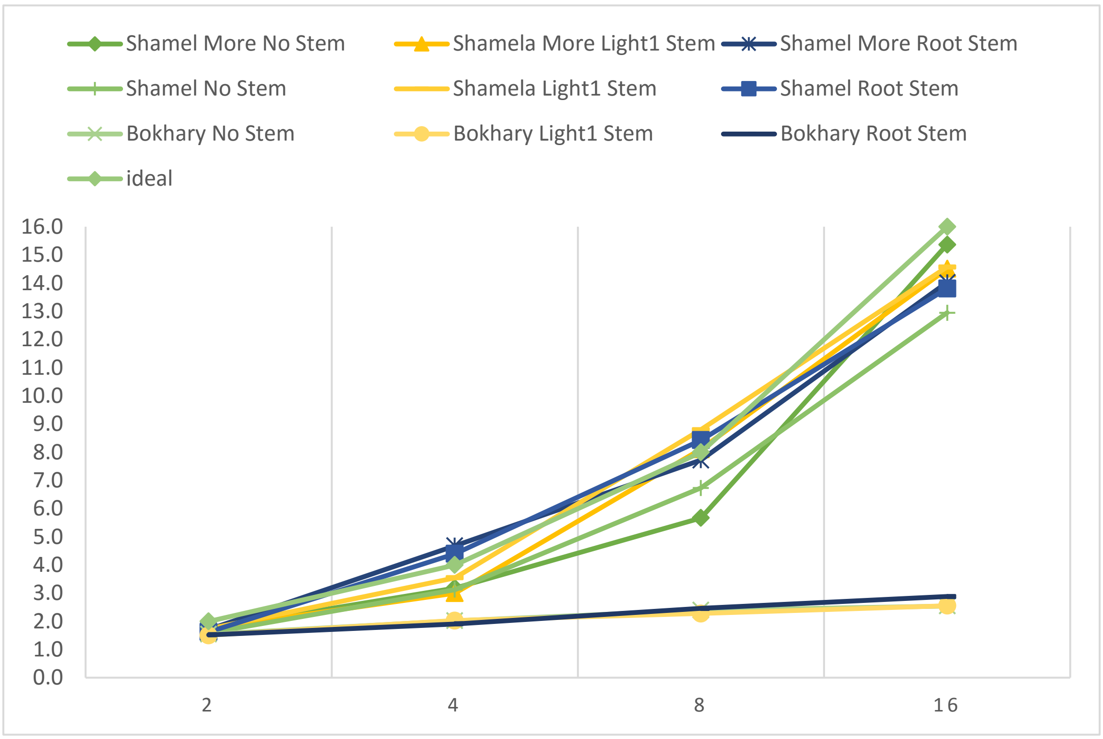

**Figure (5.4):** Speedup of Parallel NB Classifier on Spark Cluster Nodes

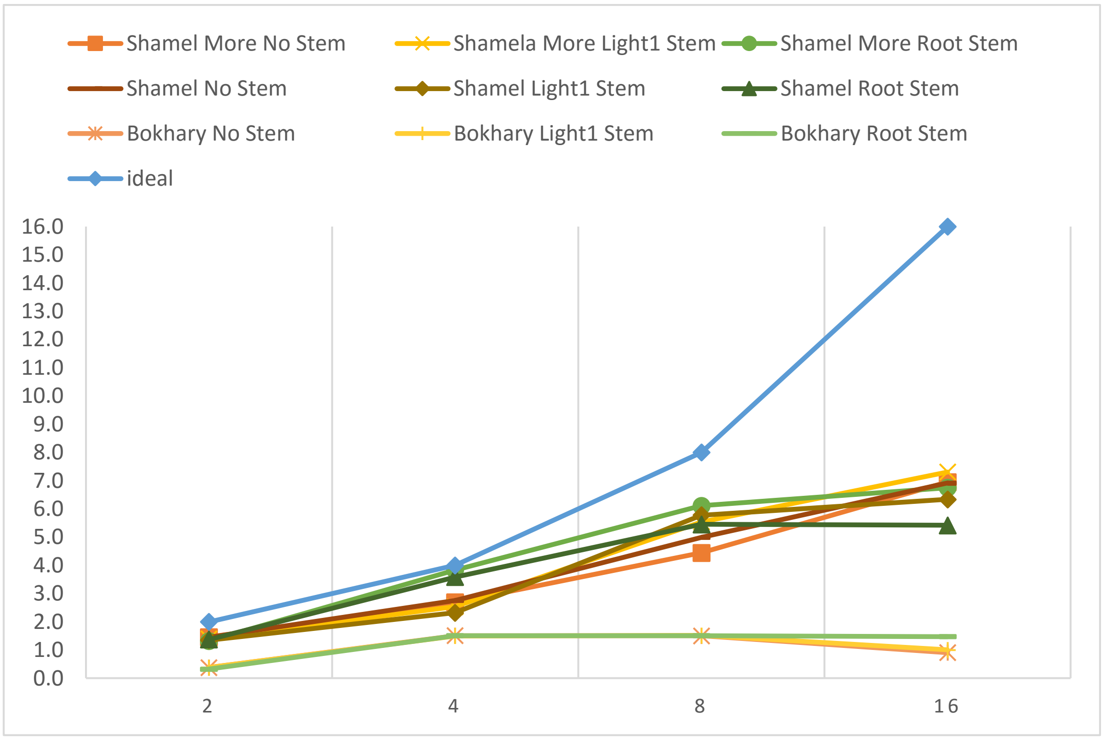

**Figure (5.5):** Speedup of Parallel LR Classifier on Spark Cluster Nodes

The calculated speedup scores for both proposed parallel classifiers are summarized in Table 5.6 and Table 5.7 which indicate that using more dataset size and more executors make the proposed parallel classifiers more efficient and the speedup getting more linearly in the parallelized NB and getting slow increasing ratio in the parallelized LR, when adding more executors to the cluster.

**Table (5.6):** Speedup of Parallel NB Classifier on Spark Cluster Nodes

| **Corpus** / **# of Executors** | **Shamela-More ( 900 feature)** | | | **Shamela ( 900 feature)** | | | **Al-Bokhary ( 10,000 feature)** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **A1** | **B1** | **C1** | **A2** | **B2** | **C2** | **A3** | **B3** | **C3** |
| **2** | 1.753 | 1.765 | 1.712 | 1.586 | 1.742 | 1.628 | 1.548 | 1.507 | 1.525 |
| **4** | 3.186 | 2.999 | 4.684 | 3.124 | 3.548 | 4.397 | 2.025 | 2.029 | 1.913 |
| **8** | 5.672 | 8.110 | 7.724 | 6.735 | 8.803 | 8.432 | 2.402 | 2.279 | 2.459 |
| **16** | 15.370 | 14.515 | 14.028 | 12.951 | 14.574 | 13.814 | 2.548 | 2.558 | 2.880 |

**Table (5.7):** Speedup of Parallel LR Classifier on Spark Cluster Nodes

| **Corpus** / **# of Executors** | **Shamela-More ( 900 feature)** | | | **Shamela ( 900 feature)** | | | **Al-Bokhary ( 10000 feature)** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **A1** | **B1** | **C1** | **A2** | **B2** | **C2** | **A3** | **B3** | **C3** |
| **2** | 1.459 | 1.396 | 1.326 | 1.456 | 1.355 | 1.377 | 0.381 | 0.385 | 0.328 |
| **4** | 2.690 | 2.539 | 3.838 | 2.760 | 2.321 | 3.589 | 1.513 | 1.501 | 1.512 |
| **8** | 4.442 | 5.537 | 6.112 | 4.992 | 5.779 | 5.451 | 1.507 | 1.517 | 1.512 |
| **16** | 6.953 | 7.308 | 6.751 | 6.916 | 6.336 | 5.420 | 0.911 | 1.005 | 1.475 |

### *5.4.1.2 Parallel Efficiency*

Parallel efficiency is measured to check how the available resources are utilized in the proposed parallel approach for both classifiers NB and LR. A system with a linear speed up rate has a parallel efficiency equal 1. A task based parallel system is more efficient than data based parallel system due to the competence use of memory cache per executor.

We measures parallel efficiency of the proposed approach from calculated speedup ratio in Table 5.6 and Table 5.7.

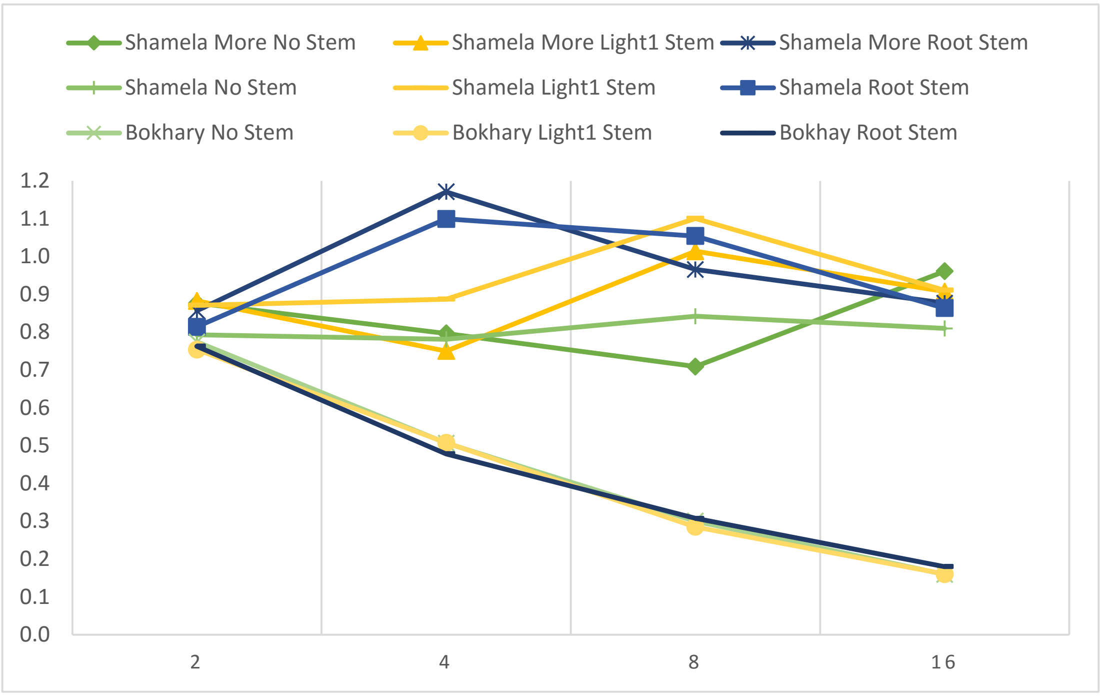

**Figure (5.6):** Parallel Efficiency of Parallel NB Classifier on Spark Cluster

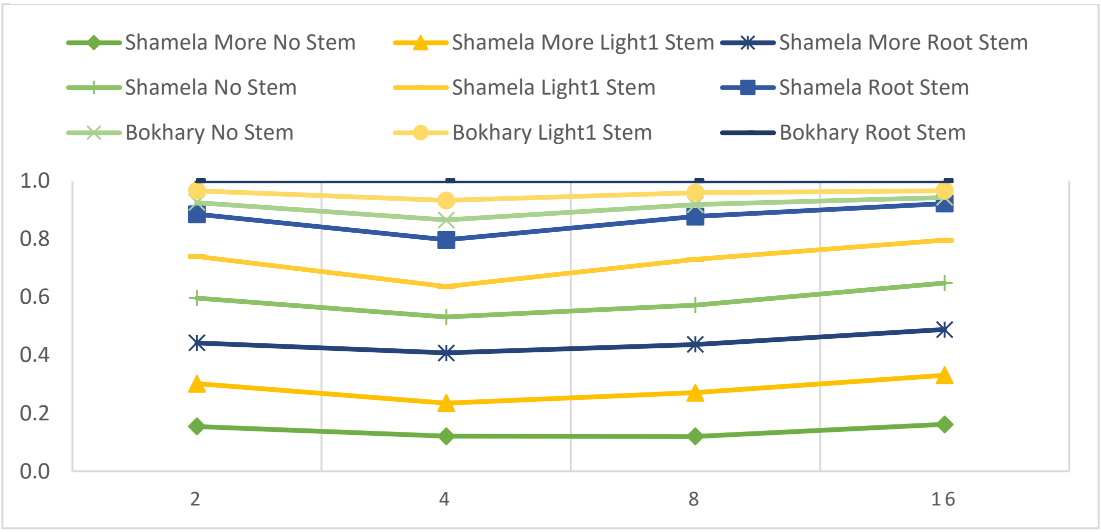

**Figure (5.7):** Parallel Efficiency of Parallel LR Classifier on Spark Cluster

Figure 5.6 shows that parallel efficiency is increasing with Shamela-More and Shamela corpuses, and decreasing with Al-Bokhary corpus in both parallel NB and LR classifiers. Because of the Apache Spark model which realized MapReduce requires a large scale data sets sizes which match with Shamela-More and Shamela data sets.

Table 5.8 and Table 5.9 summarize the parallel efficiency score for both classifiers and we have noticed that by increasing the executors the parallel efficiency was also increased particularly with large datasets in which the Shamela-More non-stemmed corpus has the highest score on running 16 executors. Al-Bokhary has the lowest scores due to its small size.

**Table (5.8):** Parallel Efficiency of Parallel NB Classifier on Apache Spark Cluster

| Corpus / # of Executors | Shamela -More ( 900 feature) | | | Shamela ( 900 feature) | | | Al-Bokhary ( 10000 feature) | | |
|---|---|---|---|---|---|---|---|---|---|
| | **A1** | **B1** | **C1** | **A2** | **B2** | **C2** | **A3** | **B3** | **C3** |
| **2** | 0.877 | 0.882 | 0.856 | 0.793 | 0.871 | 0.814 | 0.774 | 0.754 | 0.763 |
| **4** | 0.796 | 0.750 | 1.171 | 0.781 | 0.887 | 1.099 | 0.506 | 0.507 | 0.478 |
| **8** | 0.709 | 1.014 | 0.966 | 0.842 | 1.100 | 1.054 | 0.300 | 0.285 | 0.307 |
| **16** | 0.961 | 0.907 | 0.877 | 0.809 | 0.911 | 0.863 | 0.159 | 0.160 | 0.180 |

**Table (5.9):** Parallel Efficiency of Parallel LR Classifier on Apache Spark Cluster

| Corpus / # of Executors | Shamela -More ( 900 feature) | | | Shamela ( 900 feature) | | | Al-Bokhary ( 10000 feature) | | |
|---|---|---|---|---|---|---|---|---|---|
| | **A1** | **B1** | **C1** | **A2** | **B2** | **C2** | **A3** | **B3** | **C3** |
| **2** | 0.730 | 0.698 | 0.663 | 0.728 | 0.677 | 0.688 | 0.191 | 0.192 | 0.164 |
| **4** | 0.673 | 0.635 | 0.959 | 0.690 | 0.580 | 0.897 | 0.378 | 0.375 | 0.378 |
| **8** | 0.555 | 0.692 | 0.764 | 0.624 | 0.722 | 0.681 | 0.188 | 0.190 | 0.189 |
| **16** | 0.869 | 0.913 | 0.844 | 0.865 | 0.792 | 0.678 | 0.114 | 0.126 | 0.184 |

#### *5.4.1.3 Scalability*

The proposed classifier system scalability is estimated rather than calculated in which a parallel system become scalable when the parallel efficiency can be kept persistent when the number of processing units increased, or the problem size is increased (Wu, 2012). After wrapping up previous parallel performance metrics, we can say that the proposed parallelized NB and LR classifiers are scalable, where the parallel efficiency is retained steady while increasing number of executors up to 16 executor in our experiments to the Apache spark standalone cluster in addition to increasing the size of dataset where both parallel classifiers are not scalable with small size datasets.

### 5.4.2 Parallel Classification Evaluation

We have executed the proposed classifiers NB and LR in parallel on a single machine and on many cluster nodes 2, 4, 8, 16 respectively for one to four rounds per experiment and record the important metrics for measuring the quality of document classification that we have explained earlier in Section 2.6 in each round and select the average score in the rounds. We have noticed in each round we get different result score and that due to various reasons:

- When using HashingTF class for applying TF scores which uses the Scala native hashing function that lead for a different hash value per execution run and that has been solved in Apache Spark 2.
- While cluster workers execution per round, one or more of the workers executors are terminated through network connection error and re-run which could led to data loss that want be taken in the calculations and that partial solved through reset Apache spark settings as shown in Table 5.3 and through code using try-catch block in python that ask to reconnect on connection error or loss.

***Try-catch python block:***

```
try:
    // to-do code
except socket.error as error:
    if error.errno == errno.WSAECONNRESET:
        reconnect()
        retry_action()
    else:
        raise
```

- Apache Spark apply in-memory computations. For that, in heavy computations and data with small size RAM memory could raise an OutOfMemory exception where java heap space is full. Therefore, we reset the storage level to memory and disk using persist() in which if the memory is being full during execution, some memory data is going to be written on disk instead to free the memory space for the running computations.

***Reset Storage Level in Spark:***

```
dataRDD.persist(storageLevel=StorageLevel.MEMORY_AND_DISK)
```

### *5.4.2.1 NB Classifier Evaluation*

Figure 5.8 and 5.9 visualizes NB classification metrics applied on the three corpuses Shamela, Shamela-More and Al-Bokhary based on the scores listed in Table 5.10. We need to mention that we used 30% (testRDD) and 100% (dataRDD) data splits for testing.

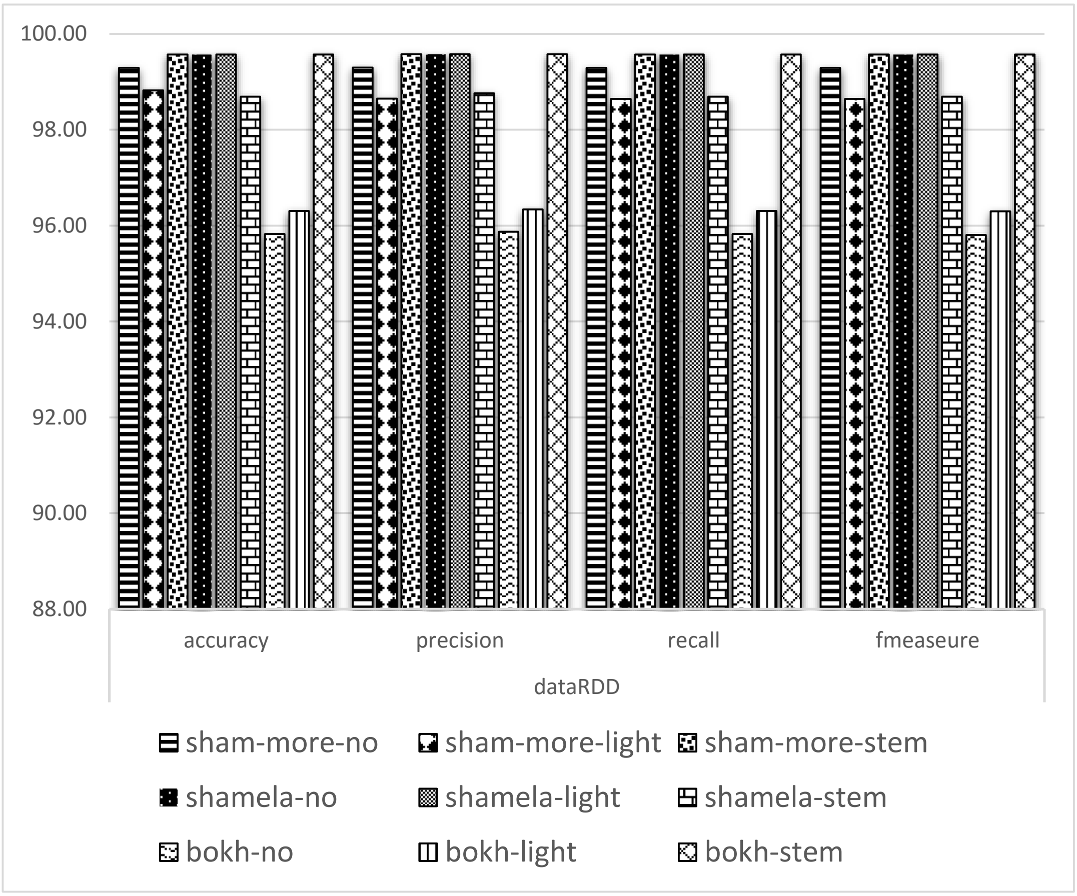

**Figure (5.8):** NB Classification Metrics using dataRDD for Shamela-More, Shamela, and Al-Bokhary Corpuses

The results shows that the highest accuracy score over 99% in Shamela-More that applied root stemmer with both dataRDD and testRDD. Light1 stemmer is used which considered bad stemmer (Otair, 2013) to test the proposed classifier in worst situation and it has performed accuracy with almost 98%. It also has scored high precision rates up to 99%. We explain that due to the large number of features is used to build the classifier were 900 feature for Shamela, Shamela-More. Al-Bokhary corpus is the opposite of Shamela where it has small feature dimensionality and size except it reaches the highest accuracy rate with root stemmer 99% and the worst fmeasure rate to 89% with both dataRDD and testRDD.

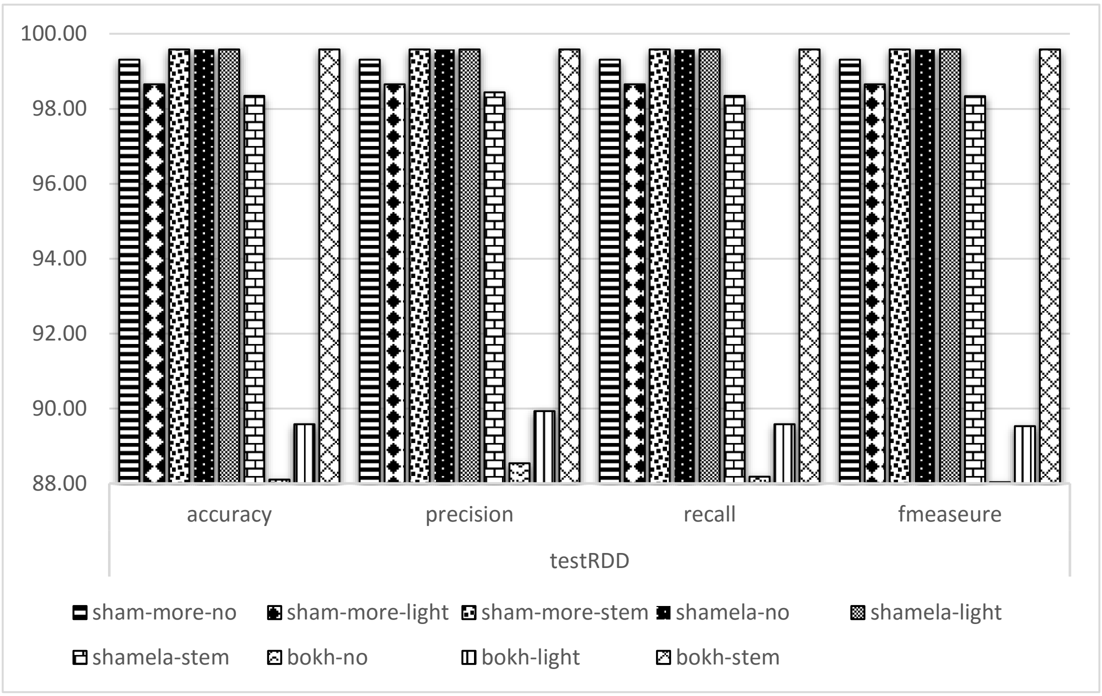

**Figure (5.9):** NB Classification Metrics using testRDD for Shamela-More, Shamel and Al-Bokhary Corpuses

**Table (5.10):** Parallel NB Classification Metrics on Shamela, Shamela-More and Al-Bokhary Corpuses

| **Test splits** | **dataRDD** | | | | **testRDD** | | | |
|---|---|---|---|---|---|---|---|---|
| **Metric / Corpus** | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **sham-more-no** | 99.29 | 99.29 | 99.29 | 99.29 | 99.31 | 99.31 | 99.31 | 99.31 |
| **sham-more-light1** | 98.82 | 98.65 | 98.64 | 98.64 | 98.65 | 98.66 | 98.65 | 98.65 |
| **sham-more-stem** | 99.57 | 99.57 | 99.57 | 99.57 | 99.58 | 99.59 | 99.58 | 99.58 |
| **Shamela-no** | 99.57 | 99.57 | 99.57 | 99.57 | 99.58 | 99.59 | 99.58 | 99.58 |
| **Shamela-light1** | 99.57 | 99.57 | 99.57 | 99.57 | 99.58 | 99.59 | 99.58 | 99.58 |
| **Shamela-stem** | 98.69 | 98.76 | 98.69 | 98.69 | 98.35 | 98.44 | 98.35 | 98.34 |
| **bokh-no** | 95.82 | 95.87 | 95.82 | 95.80 | 88.11 | 88.54 | 88.18 | 88.04 |
| **bokh-light1** | 96.31 | 96.34 | 96.31 | 96.30 | 89.58 | 89.93 | 89.58 | 89.53 |
| **bokh-stem** | 99.57 | 99.57 | 99.57 | 99.57 | 99.58 | 99.59 | 99.58 | 99.58 |

The overall classification results of the parallel NB classifier in both situations; large data with high dimensionality and small size dataset with small dimensionality returned noticeably more relevant results than irrelevant ones.

#### *5.4.2.2 LR Classifier Evaluation*

LR classifier is implemented on Apache Spark MLlib with L-BFGS optimizer and L2 regularization to avoid overfitting. Figure 5.9 represents LR classification metrics applied on the three corpuses Shamela, Shamela- More and Al-Bokhary based on the scores listed in Table 5.11. We need to remember the used 30% (testRDD) and 100% (dataRDD) data splits for testing.

The results show that the highest accuracy score over 99% in Shamela-More that applied root stemming using testRDD. We also used light1 stemmer as weak stemmer (Otair, 2013) to evaluate the proposed classifier in worst case and it has achieved accuracy with almost 98% with Shamela-More and achieved 92% with Shamela.  We noticed that Shamela-More classification measures better than Shamela and that might for larger number of samples in Shamela-More than Shamela per category. It also has scored high precision rates up to 99%. We explain that due to the large number of features is used to build the classifiers. Al-Bokhary corpus is the opposite of Shamela in feature dimensionality and size reaches highest accuracy rate with root stemmer 93.7% and the worst fmeasure rate to 81% with testRDD but it scores higher with dataRDD closes to 98%.

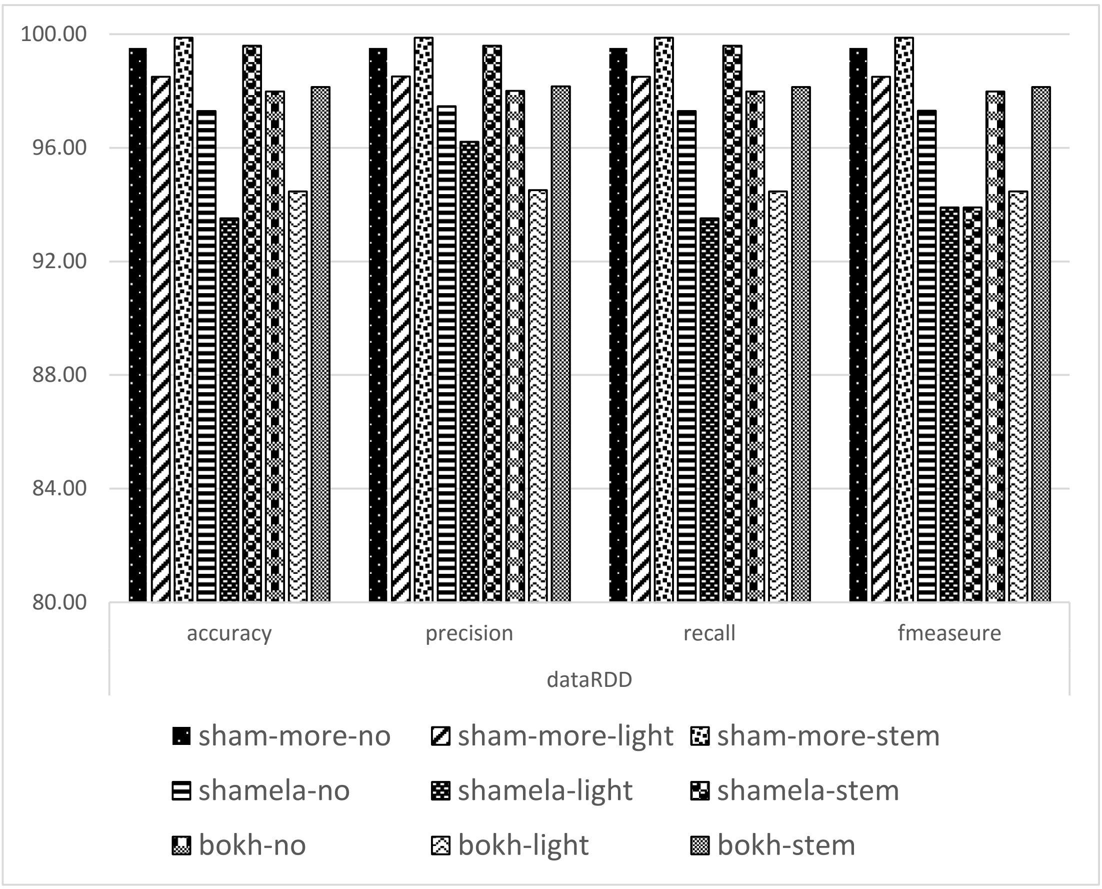

**Figure (5.10):** LR Classification Metrics using dataRDD for Shamela-More, Shamela, and Al-Bokhary Corpuses

**Table (5.11):** Parallel LR Classification Metrics on Shamela, Shamela-More and Al-Bokhary Corpuses

| Test splits | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|
| Metric / Corpus | accuracy | precision | recall | fmeaseure | accuracy | precision | recall | fmeaseure |
| sham-more-no | 99.52 | 99.52 | 99.52 | 99.52 | 99.49 | 99.49 | 99.49 | 99.49 |
| sham-more-light1 | 98.50 | 98.52 | 98.50 | 98.50 | 98.44 | 98.46 | 98.49 | 98.44 |
| sham-more-stem | 99.88 | 99.88 | 99.88 | 99.88 | 99.85 | 99.85 | 99.85 | 99.85 |
| Shamela-no | 97.29 | 97.47 | 97.29 | 97.31 | 96.28 | 96.52 | 96.28 | 96.30 |
| Shamela-light1 | 93.52 | 96.21 | 93.52 | 93.90 | 92.83 | 95.52 | 92.83 | 93.22 |
| Shamela-stem | 99.60 | 99.60 | 99.60 | 93.90 | 99.46 | 99.48 | 99.46 | 99.46 |
| bokh-no | 97.99 | 98.00 | 97.99 | 97.99 | 93.18 | 93.36 | 93.18 | 93.19 |
| bokh-light1 | 94.46 | 94.51 | 94.46 | 94.47 | 81.23 | 81.67 | 81.23 | 81.29 |
| bokh-stem | 98.15 | 98.16 | 98.15 | 98.15 | 93.72 | 93.87 | 93.72 | 93.72 |

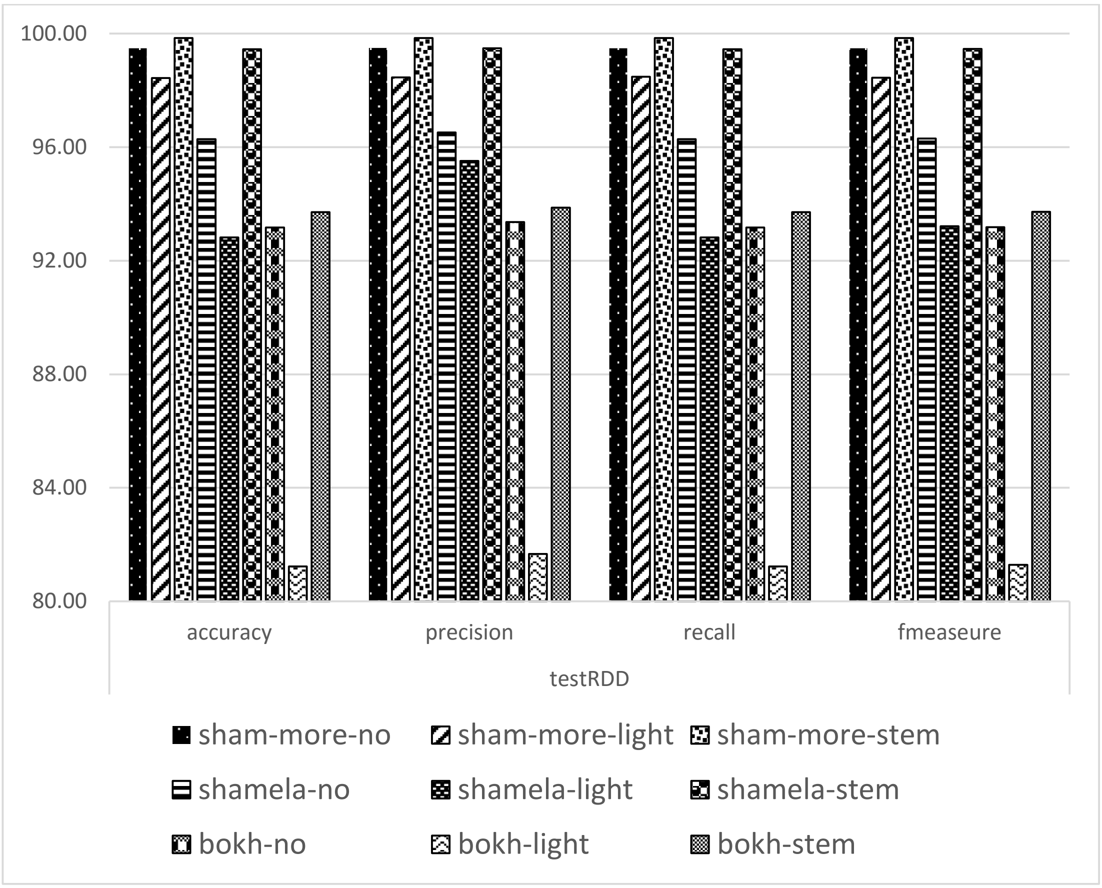

**Figure (5.11):** LR Classification Metrics using testRDD for Shamela-More, Shamela, and Al-Bokhary Corpuses

The overall classification results of the parallel LR classifier in both situations; large data with high dimensionality and small size dataset with small dimensionality returned most of the relevant results than irrelevant ones. But LR noticed to perform better with large datasets than the small one.

### *5.4.3 NB vs LR*

We have used contrasted corpuses, in size and feature dimensionality, where Shamela dataset as a large scale corpus and Al-Bokhary as a small dataset in size and feature dimensionality. In general, NB has better classification results than LR when using testRDD in prediction where NB has reach almost 99% with Shamela, Shamela-More and Al-Bokhary corpuses. LR scores reflect the effects of the used stemming approaches more than NB. This proves the superiority of LR and the simplicity of NB.

NB is considered a fast classifier with low memory requirements since it has sequential probability computations and works properly with small amounts of data, while LR cis considered to be superior algorithm but needs large amount of data to work appropriately and may cause over fitting estimations and spends large time with memory complexity due to its iterative computations.

NB assumes all the features are conditionally independent. So in case of some features dependency, it returns weak irrelevant results. LR breaks down features vector linearly, but it works in accepted rate even if some of the variables are associated to each other, i.e., with the existence of features dependency.

## 5.5 Summary

Throughout this chapter, we have covered the implementation of the proposed NB and LR parallel classifiers and the set environment settings and conducted the experiments. The results have been recorded, visualized and analysed. Then the parallel NB and LR classifiers have been evaluated based on the performance and classification metrics. Finally a comparison between the used classifiers is performed.

# Chapter 6

# Conclusion and Future Work

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

Text classification of large-scale text documents is a significant research issue in text mining and is important with the rapid growing of the Arabic text on the web. Sequential Naïve Bayes classifier is the most used machine learning for Arabic text classification, it is fast and easy to classify Arabic text documents with any size of datasets. However, it takes more time when used for classifying large scale Arabic text documents. Therefore, we proposed a parallel Naïve Bayes classifier for large-scale Arabic text document based on MapReduce. Sequential Logistic Regression is rarely used for text classification and it has an iterative nature for computations. So, it takes high time and memory complexities. Therefore, a parallelized LR classifier is proposed to overcome these shortcomings.

The proposed approach involves collecting Arabic text documents, preprocessing of this Arabic text, design of a suitable MapReduce computing model for parallel classification as Apache Spark platform, implementation of the parallel NB and LR algorithms using MLlib library over Apache Spark framework.

We tested the parallel classifiers using a large scale Shamela corpus and Al-Bokhary corpus. The experiments are performed on Apache Spark standalone cluster consisting of 16+1 nodes as workers+driver. For evaluation purposes, we have used the classification metrics: accuracy, precision, recall, and f-measure to evaluate the classification of the proposed approach and we have used the classification performance execution time, speedup, parallel efficiency and scalability to evaluate the performance.

The results showed that the proposed parallel NB classifier approach can significantly improves speedup up to 15x times better than the sequential approach using the same classification algorithm and achieves accuracy up to 99%. Also, the results showed that the proposed parallel LR classifier approach can significantly improves speedup up to 12x times better than the sequential approach using the same classification algorithm and achieves accuracy up to 99% with large scale datasets.

The result showed that the parallel NB classifier is faster than the parallel LR classifier while LR can get more accurate results than parallel NB. Parallel LR and parallel NB on apache spark need large RAMs since Apache Spark performs in-memory processing computations. The parallel NB is considered more efficient in which the speedup ratio increases almost linearly during increasing the number of executors. This is unlike parallel LR where the speedup ratio increased slowly and far away from the linear speedup.

The proposed parallel approaches can be more efficient and accurate when used to classify large scale Arabic text documents with high dimensionality.

## 6.2 Future Work

There are various research directions for improvements and future investigations. The proposed LR and NB parallel classifiers can be extended to work in larger computer clusters that have higher memory resources with larger volume of Arabic documents more of tens of Gigabytes. Also, applying other classification algorithms with our approach to investigate their effectiveness and performance with live stream data in various formats like images, videos and documents. Moreover, the proposed approach can be applied to other domains like medical analysis, weather prediction, and sentiment analysis to examine its generalization. It can also use live feed data from the web like social media and air traffics as data source for automatic classification. Additionally, the research approach can be used on different cloud-based technologies such as big data analytics and web services where data mining algorithms is needed over frameworks that realize MapReduce model to maximize the system performance and give accurate results.

# References

## References

Abu Tair, M. M., & Baraka, R. S. (2013). Design and Evaluation of a Parallel Classifier for Large-Scale Arabic Text. *International Journal of Computer Applications, 75*(3).

Abushab, M. M. (2015). *Large-Scale Arabic Text Classification Using MapReduce.* (MSc Degree), Islamic University - Gaza.

Al-Harbi, S., Almuhareb, A., Al-Thubaity, A., Khorsheed, M., & Al-Rajeh, A. (2008). Automatic Arabic text classification.

Al-Shalabi, R., Kanaan, G., & Gharaibeh, M. (2006). *Arabic text categorization using kNN algorithm.* Paper presented at the Proceedings of The 4th International Multiconference on Computer Science and Information Technology.

Al-Shalabi, R., & Obeidat, R. (2008). *Improving KNN Arabic text classification with n-grams based document indexing.* Paper presented at the Proceedings of the Sixth International Conference on Informatics and Systems, Cairo, Egypt.

Al-Tahrawi, M. M. (2015). Arabic Text Categorization Using Logistic Regression. *International Journal of Intelligent Systems and Applications, 7*(6), 71.

Al-Thubaity, A., Abanumay, N., Al-Jerayyed, S., Alrukban, A., & Mannaa, Z. (2013). *The Effect of Combining Different Feature Selection Methods on Arabic Text Classification.* Paper presented at the Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on.

Alhutaish, R., & Omar, N. (2015). Arabic Text Classification using K-Nearest Neighbour Algorithm. *International Arab Journal of Information Technology (IAJIT), 12*(2).

Alsaleem, S. (2011). Automated Arabic Text Categorization Using SVM and NB. *Int. Arab J. e-Technol., 2*(2), 124-128.

Althobaiti, M., Kruschwitz, U., & Poesio, M. (2014). AraNLP: A Java-based library for the processing of Arabic text.

Ayedh, A., Tan, G., Alwesabi, K., & Rajeh, H. (2016). The effect of preprocessing on arabic document categorization. *Algorithms, 9*(2), 27.

Bahassine, S., Kissi, M., & Madani, A. (2014). *New stemming for arabic text classification using feature selection and decision trees*.

Caruana, G., Li, M., & Qi, M. (2011). *A MapReduce based parallel SVM for large scale spam filtering.* Paper presented at the Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on.

Chu, C., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A. Y., & Olukotun, K. (2007). Map-reduce for machine learning on multicore. *Advances in neural information processing systems, 19*, 281.

Czech, Z. J. (2017). *Introduction to Parallel Computing*: Cambridge University Press.

Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM, 51*(1), 107-113.

Dean, J., & Ghemawat, S. (2010). MapReduce: a flexible data processing tool. *Communications of the ACM, 53*(1), 72-77.

Elhassan, R., & Ahmed, M. (2015). Arabic Text Classification on Full Word. *International Journal of Computer Science and Software Engineering (IJCSSE), 4*(5), 114-120.

Grishchenko, A. (2016, January 28, 2016). Spark Memory Management. from https://0x0fff.com/spark-memory-management/

Hadoop, A. (2014). The Apache Hadoop Open Source Software for Distributed Computing. from https://hadoop.apache.org/

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter, 11*(1), 10-18.

Harrag, F., El-Qawasmeh, E., & Pichappan, P. (2009). *Improving Arabic text categorization using decision trees.* Paper presented at the Networked Digital Technologies, 2009. NDT'09. First International Conference on.

Hmeidi, I., Al-shalabi, M., & Al-Ayyoub, M. (2015). *A Comparative Study of Automatic Text Categorization Methods Using Arabic Text.* Paper presented at the The International Technology Management Conference (ITMC2015).

Hmeidi, I., Hawashin, B., & El-Qawasmeh, E. (2008). Performance of KNN and SVM classifiers on full word Arabic articles. *Advanced Engineering Informatics, 22*(1), 106-111.

Japkowicz, N., & Shah, M. (2011). *Evaluating learning algorithms: a classification perspective*: Cambridge University Press.

Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). *Learning Spark: Lightning-Fast Big Data Analysis*: " O'Reilly Media, Inc.".

Krishnaveni, G., & Sudha, T. (2016). Naïve Bayes Text Classification–A Comparison of Event Models. *Imperial Journal of Interdisciplinary Research, 3*(1).

Mäkelä, M., Pauksens, K., Rostila, T. a. a., Fleming, D., Man, C., Keene, O., & Webster, A. (2000). Clinical efficacy and safety of the orally inhaled neuraminidase inhibitor zanamivir in the treatment of influenza: a randomized, double-blind, placebo-controlled European study. *Journal of Infection, 40*(1), 42-48.

Mamoun, R., & Ahmed, M. A. (2014). *A Comparative Study on Different Types of Approaches to the Arabic text classification*. Paper presented at the 1st International Conference of Recent Trends in Information and Communication Technologies, Universiti Teknologi Malaysia, Johor, Malaysia.

McCallum, A., & Nigam, K. (1998). *A comparison of event models for naive bayes text classification.* Paper presented at the AAAI-98 workshop on learning for text categorization.

Moh'd Mesleh, A. (2008). Support vector machines based Arabic language text classification system: feature selection comparative study *Advances in Computer and Information Sciences and Engineering* (pp. 11-16): Springer.

Otair, M. A. (2013). Comparative analysis of Arabic stemming algorithms. *International Journal of Managing Information Technology, 5*(2), 1.

Pacheco, P. (2011). *An introduction to parallel programming*: Elsevier.

Polamuri, S. (2017, March 14, 2017). HOW MULTINOMIAL LOGISTIC REGRESSION MODEL WORKS IN MACHINE LEARNING. from http://dataaspirant.com/2017/03/14/multinomial-logistic-regression-model-works-machine-learning/

Rossum, G. v. (2015). Python Release Python 3.5.1. Retrieved 15/05/2016, from https://www.python.org/downloads/release/python-351/

Saad, M. K., & Ashour, W. (2010). *Arabic text classification using decision trees.* Paper presented at the Proceedings of the 12th international workshop on computer science and information technologies CSIT.

Shen, P., Wang, H., Meng, Z., Yang, Z., Zhi, Z., Jin, R., & Yang, A. (2016). An Improved Parallel Bayesian Text Classification Algorithm. *Review of Computer Engineering Studies, 3*(1), 6-10.

Spark, A. (2014). Apache spark–lightning-fast cluster computing. Retrieved 23/11/2015, from http://spark.apache.org/

Spark, A. (2016). Download Apache Spark. Retrieved 26/06/2016, from http://spark.apache.org/downloads.html

Thabtah, F., Eljinini, M., Zamzeer, M., & Hadi, W. (2009). *Naïve Bayesian based on Chi Square to categorize Arabic data.* Paper presented at the proceedings of The 11th International Business Information Management Association Conference (IBIMA) Conference on Innovation and Knowledge Management in Twin Track Economies, Cairo, Egypt.

Versteegh, C. H. M., & Versteegh, K. (2014). *The arabic language*: Edinburgh University Press.

Wahba, K., Taha, Z. A., & England, L. (2014). *Handbook for Arabic language teaching professionals in the 21st century*: Routledge.

Wahbeh, A. H., & Al-Kabi, M. (2012). Comparative Assessment of the Performance of Three WEKA Text Classifiers Applied to Arabic Text. *Abhath Al-Yarmouk: Basic Sci. & Eng, 21*(1), 15-28.

White, T. (2012). *Hadoop: The definitive guide*: " O'Reilly Media, Inc.".

Wu, X. (2012). *Performance evaluation, prediction and visualization of parallel systems* (Vol. 4): Springer Science & Business Media.

Xu, K., Wen, C., Yuan, Q., He, X., & Tie, J. (2014). A MapReduce based Parallel SVM for Email Classification. *Journal of Networks, 9*(6), 1640-1647.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster Computing with Working Sets. *HotCloud, 10*(10-10), 95.

Zhou, L., Wang, H., & Wang, W. (2012). Parallel implementation of classification algorithms based on cloud computing environment. *TELKOMNIKA Indonesian Journal of Electrical Engineering, 10*(5), 1087-1092.

# Appendices

# Appendix A
# Source Code Implementation

## A.1: Arabic Text Preprocessing using Java

Text processing applied on both corpuses (Shamela and Bokhary) in a sequential manner using the AraNLP library. It adjusted to achieve our required text preprocessing methods as described in Section 4.3.1.1. We can get AraNLP library from this dropbox link https://www.dropbox.com/s/sr6pab7al9lnd28/AraNLP.zip. The source code of this work is shown below.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import utilities.AraNormalizer;
import utilities.DiacriticsRemover;
import utilities.LightStemmer1;
import utilities.PunctuationsRemover;
import utilities.RootStemmer;
import utilities.SpaceTokenizer;
public class testProcess {
public static void main(String[] args) {
//read files that need to apply text preprocessing on them
                    File folder = new File("folder files path");
```

```
        String pathLightStem = " folder files path "; String pathRootStem = " folder files path”;
        File[] listOfFiles = folder.listFiles();
        System.out.println(listOfFiles.length);
        int folderNum = 7;
        for (int i = 7; i < listOfFiles.length; i++) {
                File f = listOfFiles[i];
                folderNum++;
                String pth = pathLightStem + String.valueOf(folderNum) + "/";
                String pth1 = pathRootStem + String.valueOf(folderNum) + "/";
                // createBkDir(pth);
                // createBkDir(pth1);
                File[] listOfFiles1 = get_files(f.getAbsolutePath());
                // c+= listOfFiles1.length;
                System.out.println(listOfFiles1.length);
                int num = 500;
                for (int j = 500; j < 600; j++) {
                        File ff = listOfFiles1[j];
                        num++;
                        try {
                                BufferedReader br = readFile(ff.getAbsolutePath());
                                processText(br, num, folderNum);
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                        System.out.println(num);
                }       }       }
private static void processText(BufferedReader br, int num, int folderNum) {
        String temp = "";
        String NormalizedRoot = "", NormalizedLight = "";
        try {
```

```
                        SpaceTokenizer tok = new SpaceTokenizer(); //remove extra spacing
                        RootStemmer rs = new RootStemmer(); //get root of each token
                        AraNormalizer arn = new AraNormalizer(); //normalize each token
                        DiacriticsRemover dr = new DiacriticsRemover(); // remove Arabic Diacritics
                        PunctuationsRemover pr = new PunctuationsRemover();
                        LightStemmer1 lt = new LightStemmer1(); //get light1 stem of each token
String pathLightStem = " folder files path " + String.valueOf(folderNum) + "/"+ num + ".txt";
String pathRootStem = " folder files path " +  String.valueOf(folderNum) + "/"+num + ".txt";
                        String lightString = "", rootString = "";
                        while ((temp = br.readLine()) != null) {
                                String normalizedText = arn.normalize(temp);
                                normalizedText = dr.removeDiacritics(normalizedText);
                                normalizedText = pr.removePunctuations(normalizedText);
                                ArrayList<String> tokenss = tok.tokenize(normalizedText);
                                AraStopWords stop = new AraStopWords();
                                ArrayList<String> tokens = stop.removeStopWords(tokenss);
                                for (int x =0; x< tokens.size(); x++){//String token : tokens) {
                                        String token = tokens.get(x);
                                        String stem = rs.findRoot(token);
                                        String lstem = lt.findStem(token);
                                        // removing stop words=====
                                        rootString += stem + " ";
                                        lightString += lstem + " ";
                                }
                                NormalizedLight += lightString + "\n";
                                NormalizedRoot += rootString + "\n";
                                lightString = rootString = "";
                        } // end while
                        writeFile(NormalizedLight, pathLightStem); // save result text into text file
writeFile(NormalizedRoot, pathRootStem);
            } catch (IOException e) {
                        e.printStackTrace();
            }           }
private static void writeFile(String normalized, String path) throws IOException {
            FileOutputStream fio4 = new FileOutputStream(new File(path));
             OutputStreamWriter osw = new OutputStreamWriter(fio4,"UTF-8");
             BufferedWriter br = new BufferedWriter(osw);
             PrintWriter pw4 = new PrintWriter(br);
             pw4.println(normalized);
             pw4.close();
             osw.close();
             fio4.close();                    }
private static BufferedReader readFile(String absolutePath)
                        throws FileNotFoundException, UnsupportedEncodingException {
            File f = new File(absolutePath);
            FileInputStream fis = null;
            fis = new FileInputStream(f);
            InputStreamReader isr = null;
            isr = new InputStreamReader(fis, "UTF-8");
            return new BufferedReader(isr);      }
private static void createBkDir(String path) { //create folder for each category
            File newF = new File(path);
            if (newF.mkdir()) {
                        System.out.println(path + " was created Successfully");
            }           }
private static File[] get_files(String absolutePath) { //get list of files per category
            File folder = new File(absolutePath);
            return folder.listFiles();
            }           }
```

## A.2 Hadith Matan Extraction Java Code

Bokhary corpus contains a lot of hadiths matan that are collected and extracted manually with in-house java program. The following snippet presents this java source code where extract mattan hadith between << >> or “ ”:

```
private static String extractH(ArrayList<String> hdth) {
        String hadith = " ";
        boolean isQ = false, isSt = false, isE = false;
        int start_line = 0, end_line = 0, startH = 0, endH = 0;
        for (int i = 0; i < hdth.size(); i++) {// lines
                String line = hdth.get(i);
                for (int j = 0; j < line.length(); j++) {// per line
                if (line.charAt(j) == '«' || (line.charAt(j) == '\"' && !isQ)) {// start
                                isSt = true;
                                start_line = i;
                                startH = j + 1;
                                if (line.charAt(j) == '\"')
                                        isQ = true;
                } else if (line.charAt(j) == '»' || (line.charAt(j) == '\"' && isQ)) {// end
                                isE = true;
                                end_line = i;
                                endH = j - 1;
                                if (line.charAt(j) == '\"')
                                        isQ = false;       }
                        if (isSt && isE) {
                         if (start_line == end_line) { // start and end in the same line
                           try {
                                hadith += line.substring(startH, endH) + " ";
                                } catch (Exception e) {
                                System.out.println("= " + start_line);
                                }        } else if (start_line < end_line) {
                                int sub = end_line - start_line;
if (sub == 1) {
  hadith += (hdth.get(start_line)).substring(startH, hdth.get(start_line).length()) + " ";
  hadith += (hdth.get(end_line)).substring(0, endH + 1) + " ";
  } else if (sub > 1) {
            for (int x = start_line; x <= end_line; x++) {
              if (x == start_line) {
                  hadith+= (hdth.get(x)).substring(startH, hdth.get(x).length()) + " ";
                } else if (x == end_line) {
                      hadith += (hdth.get(x)).substring(0, endH + 1) + " ";
            } else {
            hadith += (hdth.get(x)) + " ";                                               }
  }         }
            } else if (start_line > end_line) {
            int sub = start_line - end_line;
  if (sub == 1) {
  hadith += (hdth.get(end_line)).substring(0, endH + 1) + " ";
  hadith += (hdth.get(start_line)).substring(startH, hdth.get(start_line).length()) + " ";
  } else if (sub > 1) {
              for (int x = end_line; x <= start_line; x++) {
                    if (x == end_line) {
                    hadith += (hdth.get(x)).substring(0, endH + 1) + " ";
                    } else if (x == start_line) {
                    hadith += (hdth.get(x)).substring(startH, hdth.get(x).length()) + " ";
                    } else {
                    hadith += (hdth.get(x)) + " ";
                    }       }       }       }
                    start_line = end_line = 0;
                    startH = endH = 0;
                    isSt = isE = false;
                    } // end if and   }       }
            return hadith;   }
```

## A.3 Python – Spark (pySpark) Application Code

NB and LR are the parallel classifiers covered in this research. These classifiers are applied in parallel using Apache Spark MLlib library and Apache Spark python API (see Sections 2.4 and 2.5) that the corpus files are imported to RDDs then went through few steps until generate the classifier model and apply prediction.

```
from pyspark import SparkContext

sc = SparkContext(appName="ShamelaNBClassfication")  #spark object to work with files

from time import time

file = open ("pyLog.txt", "a") #log file including execution time in milliseconds

localtime = time.asctime( time.localtime(time.time()) )

file.write("Begin at:   "+localtime+"\n")

from pyspark import  StorageLevel #storagelevel to change memory level

t_start = time().clock()

bk1_rdd = sc.textFile("category files path ").map(lambda line: (0, line.split())) #sc read files into rdd

bk1_rdd.persist(storageLevel=StorageLevel.MEMORY_AND_DISK) #save rdd into memory and disk

bk2_rdd = sc.textFile("category files path ").map(lambda line: (1, line.split()))

bk2_rdd.persist(storageLevel=StorageLevel.MEMORY_AND_DISK)

bkn_rdd = sc.textFile("category files path ").map(lambda line: (2, line.split()))

bkn_rdd.persist(storageLevel=StorageLevel.MEMORY_AND_DISK)

hadith_rdd = bk1_rdd.union(bk2_rdd) #merge rdds into one.

hadith_rdd = hadith_rdd.union(bkn_rdd)

labels = hadith_rdd.map(lambda item : item[0])  #get categories labels into single rdd

features = hadith_rdd.map(lambda item : item[1]) #get categories features into single rdd

from pyspark.mllib.regression import LabeledPoint

from pyspark.mllib.feature import HashingTF, IDF

prepare = time.clock()

tf = HashingTF(numFeatures=500).transform(features) #to computes term frequency using
hashing function

tf.persist(storageLevel=StorageLevel.MEMORY_AND_DISK)
```

```
import socket
import errno
try:
        idff = IDF().fit(tf)    #create idf model based on tf
except socket.error as error: # when socket connection cause error reconnect and rerun action
        if error.errno == errno.WSAECONNRESET:
                reconnect()
                retry_action()
        else:
                raise
tf_idf_rdd = idff.transform(tf)
dataRDD = labels.zip(tf_idf_rdd).map(lambda x: LabeledPoint(x[0], x[1]))
p_end = time.clock() - prepare
file.write("tfidf ready in "+format(round(p_end, 3))+" seconds")
training, test = dataRDD.randomSplit([0.7, 0.3], seed=0)
from pyspark.mllib.classification import NaiveBayes, NaiveBayesModel
#or from pyspark.mllib.classification import LogisticRegressionWithLBFGS, LogisticRegressionModel
model_t = time.clock()
try:
        NBmodel = NaiveBayes.train(training, 1.0)  #trainning NB model
#or     DTmodel = LogisticRegressionWithLBFGS.train(training, iterations=10, numClasses=8)
except socket.error as error:
        if error.errno == errno.WSAECONNRESET:
                reconnect()
                retry_action()
        else:
                raise
model_end = time.clock() - model_t
file.write("NBmodel was ready in  "+format(round(model_end, 3))+" seconds")
```

```
#apply prediction using the trained model

#predition using test rdd (expected, actual)

predictionAndLabel = test.map(lambda p: (p.label, NBmodel.predict(p.features)))

correct = predictionAndLabel.filter(lambda x: x[0] == x[1]).count() /float(test.count())

file.write("Accuracy = "+format(round(correct, 3))+"\n")

#predition using data rdd (expected, actual)

predictionAndLabels = dataRDD.map(lambda p: (p.label, NBmodel.predict(p.features)))

correct1 = predictionAndLabels.filter(lambda x: x[0] == x[1]).count() /float(dataRDD.count())

file.write("Accuracy (dataRDD) = "+format(round(correct1, 3))+"\ntest")

end = time().clock()-t_start

file.write("program ends in "+format(round(end, 3))+" seconds\n")

file.close()

file.write(predictionAndLabel.collect())

file.write("\ndataRDD")

file.write(predictionAndLabels.collect())
```

## A.4 (pySpark) Multiclass Evaluation Code

After training the parallel classifiers NB and LR, predictions are made with testing data then the classification metrics are measured also in parallel using MulticlassMetrics class in MLlib library. Below is *pySpark* source code for evaluation.

```
from pyspark import SparkContext

sc = SparkContext(appName="ShamelaNBClassfication")

import time

file = open ("MetricsLog.txt", "a")

localtime = time.asctime( time.localtime(time.time()) )

file.write("Begin at:   "+localtime+"\n")

#dataRDD used for prediction

dd = sc.parallelize([(0.0, 0.0), (0.0, 0.0), ...]) #prediction results

#testRDD

gg = sc.parallelize([(0.0, 0.0), (0.0, 0.0), ...]) #prediction results

from pyspark.mllib.evaluation import MulticlassMetrics

metrics = MulticlassMetrics(dd) # metrics object used for calculating the metrics

file.write("dataRDD of Merged Metrics\n")

# Overall statistics

recall = metrics.recall()

precision = metrics.precision()

f1Score = metrics.fMeasure()

file.write("Summary Stats\n")

file.write("Precision = %s\n" % format(round(precision*100, 3)))

file.write("Recall = %s\n" % format(round(recall*100, 3)))

file.write("F1 Score = %s\n" % format(round(f1Score*100, 3)))

# Statistics by class

labels = [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0]#data.map(lambda lp: lp.label).distinct().collect()
```

```
for label in sorted(labels):
    file.write("Class %s precision = %s\n" % (label, format(round(metrics.precision(label)*100, 3))))
    file.write("Class %s recall = %s\n" % (label, format(round(metrics.recall(label)*100, 3))))
    file.write("Class %s F1 Measure = %s\n" % (label, format(round(metrics.fMeasure(label,
beta=1.0)*100, 3))))
# Weighted stats
file.write("Weighted recall = %s\n" % format(round(metrics.weightedRecall*100, 3)) )
file.write("Weighted precision = %s\n" % format(round(metrics.weightedPrecision*100, 3)) )
file.write("Weighted F(1) Score = %s\n" % format(round(metrics.weightedFMeasure()*100, 3)) )
file.write("Weighted F(0.5) Score = %s\n" %
format(round(metrics.weightedFMeasure(beta=0.5)*100, 3)) )
file.write("Weighted false positive rate = %s\n" %
format(round(metrics.weightedFalsePositiveRate*100, 3)) )
file.write("==================================\n")
metricss = MulticlassMetrics(gg)
file.write("testRDD of Merged Metrics\n")
# Overall statistics
precisionn = metricss.precision()
recal = metricss.recall()
f1Scor = metricss.fMeasure()
file.write("Summary Stats\n")
file.write("Precision = %s\n" % format(round(precisionn*100, 3)))
file.write("Recall = %s\n" % format(round(recal*100, 3)))
file.write("F1 Score = %s\n" % format(round(f1Scor*100, 3)))
# Statistics by class
labels = [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0]#data.map(lambda lp: lp.label).distinct().collect()
for label in sorted(labels):
    file.write("Class %s precision = %s\n" % (label, format(round(metricss.precision(label)*100, 3))))
    file.write("Class %s recall = %s\n" % (label, format(round(metricss.recall(label)*100, 3))))
    file.write("Class %s F1 Measure = %s\n" % (label, format(round(metricss.fMeasure(label,
beta=1.0)*100, 3))))
```

```
# Weighted stats

file.write("Weighted recall = %s\n" % format(round(metricss.weightedRecall*100, 3)) )

file.write("Weighted precision = %s\n" % format(round(metricss.weightedPrecision*100, 3)) )

file.write("Weighted F(1) Score = %s\n" % format(round(metricss.weightedFMeasure()*100, 3)) )

file.write("Weighted F(0.5) Score = %s\n" % format(round(metricss.weightedFMeasure(beta=0.5)*100,
3)) )

file.write("Weighted false positive rate = %s\n" %
format(round(metricss.weightedFalsePositiveRate*100, 3)) )

file.write("===================================\n")

file.close()
```

## A.5 Python Source for Merging Files

We used custom python code to merge files per category into a single file to reduce reading time from disk.

```
def merge_file(infile, outfile, separator = ""): #merge file lines into single line in new file
   print(separator.join(line.strip("\n") for line in infile), file = outfile)
def merge_files(paths, outpath, separator = ""): #merge files into single file
   with open(outpath, 'w') as outfile:
      for path in paths:
         with open(path) as infile:
            merge_file(infile, outfile, separator)
#start main code
files = []
folders = []
import os
for (path, dirnames, filenames) in os.walk("category folder files path"):
   folders.extend(os.path.join(path, name) for name in dirnames)
   files.extend(os.path.join(path, name) for name in filenames)
merge_files(files, "mergedOutputTextFilePath")
```

# Appendix B: Experimental Results

We have executed per experiment one to four times on each cluster 2, 4, 8, 16 nodes to observe system behavior and performance. We conducted over 300 experiments. Next we list samples of the experimental results for parallel NB and parallel LR on Apache Spark standalone cluster.

## B.1 Naïve Bayes

NB is a probabilistic algorithm commonly used for classification problems. It is featured with fast execution and results which displayed in experimental results.

**Table (B.1):** NB: 70-30 sampling - 900 feature Shamela-More light1 stemming on a standalone node

| | | **dataRDD** | | | | **testRDD** | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 10.29 | 98.56 | 98.57 | 98.56 | 98.56 | 98.52 | 98.53 | 98.52 | 98.52 |
| **2** | 10.72 | 98.47 | 98.48 | 98.47 | 98.47 | 98.44 | 98.45 | 98.44 | 98.44 |
| **3** | 10.69 | 98.51 | 98.52 | 98.51 | 98.51 | 98.49 | 98.50 | 98.49 | 98.49 |
| **4** | 10.51 | 98.52 | 98.53 | 98.52 | 98.52 | 98.51 | 98.52 | 98.51 | 98.51 |
| **Avg.** | **10.55** | **98.51** | **98.52** | **98.51** | **98.52** | **98.49** | **98.50** | **98.49** | **98.49** |

**Table (B.2):** NB: 70-30 sampling - 900 feature Shamela-More root stemming on a standalone node

| | | **dataRDD** | | | | **testRDD** | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 8.75 | 99.38 | 99.38 | 99.38 | 99.38 | 99.41 | 99.41 | 99.41 | 99.40 |
| **2** | 8.76 | 99.46 | 99.46 | 99.46 | 99.45 | 99.45 | 99.45 | 99.45 | 99.45 |
| **3** | 8.85 | 99.61 | 99.61 | 99.61 | 99.61 | 99.64 | 99.64 | 99.64 | 99.64 |
| **4** | 8.87 | 99.52 | 99.52 | 99.52 | 99.51 | 99.50 | 99.50 | 99.50 | 99.50 |
| **Avg.** | **8.81** | **99.49** | **99.49** | **99.49** | **99.49** | **99.50** | **99.50** | **99.50** | **99.50** |

**Table (B.3):** NB: 70-30 sampling - 900 feature Shamela light1 stemming on a standalone node

| | | **dataRDD** | | | | **testRDD** | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 9.83 | 99.14 | 99.15 | 99.14 | 99.14 | 98.70 | 98.71 | 98.70 | 98.66 |
| **2** | 9.93 | 99.12 | 99.12 | 99.12 | 99.11 | 99.26 | 99.26 | 99.26 | 99.26 |
| **3** | 9.94 | 99.20 | 99.20 | 99.20 | 99.19 | 98.79 | 98.81 | 98.79 | 98.79 |

|  |  | dataRDD |  |  |  | testRDD |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **4** | 9.88 | 98.12 | 98.16 | 98.12 | 98.12 | 97.03 | 97.13 | 97.03 | 97.02 |
| **Avg.** | **9.89** | **98.89** | **98.91** | **98.89** | **98.89** | **98.45** | **98.48** | **98.45** | **98.43** |

**Table (B.4):** NB: 70-30 sampling - 10,000 feature Al-Bokhary light1 stemming on a 2 nodes cluster

|  |  | dataRDD |  |  |  | testRDD |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 0.80 | 96.09 | 96.13 | 96.09 | 96.08 | 88.92 | 89.37 | 88.92 | 88.88 |
| **2** | 0.79 | 95.99 | 96.03 | 95.99 | 95.98 | 88.51 | 88.90 | 88.51 | 88.48 |
| **3** | 0.80 | 96.13 | 96.19 | 96.13 | 96.12 | 89.08 | 89.70 | 89.08 | 89.05 |
| **4** | 0.79 | 96.25 | 96.29 | 96.25 | 96.24 | 89.48 | 89.77 | 89.48 | 89.59 |
| **Avg.** | **0.80** | **96.11** | **96.16** | **96.11** | **96.11** | **89.00** | **89.44** | **89.00** | **89.00** |

**Table (B.5):** NB: 70-30 sampling - 10,000 feature Al-Bokhary without stemming on a 2 nodes cluster

|  |  | dataRDD |  |  |  | testRDD |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 0.81 | 95.70 | 95.76 | 95.70 | 95.69 | 87.54 | 88.20 | 87.54 | 87.44 |
| **2** | 0.80 | 95.66 | 95.71 | 95.66 | 95.64 | 86.89 | 87.44 | 86.89 | 86.82 |
| **3** | 0.79 | 96.13 | 96.18 | 96.13 | 96.12 | 88.92 | 89.42 | 88.92 | 88.88 |
| **4** | 0.80 | 95.56 | 95.61 | 95.56 | 95.54 | 86.89 | 87.48 | 86.89 | 86.80 |
| **Avg.** | **0.80** | **95.76** | **95.82** | **95.76** | **95.75** | **87.56** | **88.14** | **87.56** | **87.48** |

**Table (B.6):** NB: 70-30 sampling - 900 feature Shamela-More light1 stemming on a 2 nodes cluster

|  |  | dataRDD |  |  |  | testRDD |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| # | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 5.80 | 98.61 | 98.61 | 98.61 | 98.61 | 98.59 | 98.59 | 98.59 | 98.59 |
| **2** | 5.78 | 98.75 | 98.76 | 98.75 | 98.75 | 98.79 | 98.79 | 98.79 | 98.79 |
| **3** | 6.83 | 98.64 | 98.66 | 98.64 | 98.65 | 98.64 | 98.65 | 98.64 | 98.64 |
| **4** | 5.51 | 98.56 | 98.56 | 98.56 | 98.56 | 98.59 | 98.60 | 98.59 | 98.59 |
| Avg. | **5.98** | **98.64** | **98.65** | **98.64** | **98.64** | **98.65** | **98.66** | **98.65** | **98.65** |

**Table (B.7):** NB: 70-30 sampling - 900 feature Shamela-More root stemming on a 4 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time (min)** | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 1.80 | 99.55 | 99.55 | 99.55 | 99.55 | 99.55 | 99.55 | 99.55 | 99.54 |
| **2** | 1.87 | 99.60 | 99.60 | 99.60 | 99.59 | 99.62 | 99.62 | 99.62 | 99.61 |
| **3** | 1.85 | 99.44 | 99.44 | 99.44 | 99.44 | 99.47 | 99.47 | 99.47 | 99.47 |
| **4** | 2.00 | 99.71 | 99.71 | 99.71 | 99.71 | 99.71 | 99.71 | 99.71 | 99.70 |
| **Avg.** | **1.88** | **99.57** | **99.57** | **99.57** | **99.57** | **99.58** | **99.59** | **99.58** | **99.58** |

**Table (B.8):** NB: 70-30 sampling - 900 feature Shamela-More light1 stemming on a 4 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time (min)** | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 3.53 | 98.52 | 98.54 | 98.52 | 98.52 | 98.53 | 98.55 | 98.53 | 98.53 |
| **2** | 3.85 | 98.59 | 98.60 | 98.59 | 98.59 | 98.55 | 98.56 | 98.55 | 98.55 |
| **3** | 3.33 | 98.54 | 98.55 | 98.54 | 98.54 | 98.61 | 98.62 | 98.61 | 98.61 |
| **4** | 3.37 | 98.70 | 98.70 | 98.70 | 98.70 | 98.69 | 98.69 | 98.69 | 98.69 |
| **Avg.** | **3.52** | **98.59** | **98.60** | **98.59** | **98.59** | **98.59** | **98.60** | **98.59** | **98.60** |

**Table (B.9):** NB: 70-30 sampling - 10,000 feature Al-Bokhary light1 stemming on a 4 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time (min)** | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 0.60 | 96.20 | 96.24 | 96.20 | 96.20 | 89.16 | 89.52 | 89.16 | 89.12 |
| **2** | 0.61 | 96.04 | 96.08 | 96.04 | 96.02 | 88.51 | 88.97 | 88.51 | 88.37 |
| **3** | 0.59 | 96.35 | 96.38 | 96.35 | 96.33 | 89.73 | 90.10 | 89.73 | 89.56 |
| **4** | 0.57 | 96.04 | 96.03 | 96.04 | 96.03 | 88.51 | 88.95 | 88.51 | 88.46 |
| **Avg.** | **0.59** | **96.16** | **96.18** | **96.16** | **96.14** | **88.98** | **89.39** | **88.98** | **88.88** |

**Table (B.10):** NB: 70-30 sampling - 900 feature Shamela root stemming on an 8 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time (min)** | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 1.04 | 97.88 | 98.11 | 97.88 | 97.92 | 97.10 | 97.63 | 97.10 | 97.22 |
| **2** | 0.94 | 99.52 | 99.52 | 99.52 | 99.52 | 99.18 | 99.20 | 99.18 | 99.18 |
| **3** | 0.91 | 98.90 | 98.93 | 98.90 | 98.89 | 98.82 | 98.85 | 98.82 | 98.80 |
| **4** | 0.91 | 98.45 | 98.48 | 98.45 | 98.43 | 98.01 | 98.09 | 98.01 | 97.99 |
| **Avg.** | **0.95** | **98.69** | **98.76** | **98.69** | **98.69** | **98.28** | **98.44** | **98.28** | **98.30** |

**Table (B.11):** NB: 70-30 sampling - 900 feature Shamela light1 stemming on an 8 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | Time **(min)** | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 1.08 | 98.87 | 98.88 | 98.87 | 98.87 | 98.61 | 98.61 | 98.61 | 98.61 |
| **2** | 1.18 | 98.31 | 98.36 | 98.31 | 98.31 | 98.24 | 98.28 | 98.24 | 98.24 |
| **3** | 1.13 | 99.01 | 99.02 | 99.01 | 99.01 | 98.42 | 98.45 | 98.42 | 98.42 |
| **4** | 1.11 | 98.12 | 98.16 | 98.12 | 98.13 | 97.40 | 97.49 | 97.40 | 97.42 |
| **Avg.** | **1.12** | **98.58** | **98.60** | **98.58** | **98.58** | **98.17** | **98.21** | **98.17** | **98.17** |

**Table (B.12):** NB: 70-30 sampling - 900 feature Al-Bokhary root stemming on an 8 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 0.50 | 98.97 | 98.98 | 98.97 | 98.97 | 97.09 | 97.15 | 97.09 | 97.08 |
| **2** | 0.49 | 99.21 | 99.22 | 99.21 | 99.21 | 97.90 | 97.93 | 97.90 | 97.89 |
| **3** | 0.48 | 98.81 | 98.81 | 98.81 | 98.80 | 97.17 | 97.25 | 97.17 | 97.14 |
| **4** | 0.48 | 98.69 | 98.70 | 98.69 | 98.69 | 96.85 | 96.95 | 96.85 | 96.86 |
| **Avg.** | **0.49** | **98.92** | **98.93** | **98.92** | **98.92** | **97.25** | **97.32** | **97.25** | **97.24** |

**Table (B.13):** NB: 70-30 sampling - 900 feature Shamela-More without stemming on a 16 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 0.89 | 98.72 | 98.73 | 98.72 | 98.73 | 98.73 | 98.74 | 98.73 | 98.73 |
| **2** | 0.86 | 99.31 | 99.31 | 99.31 | 99.31 | 99.30 | 99.30 | 99.30 | 99.30 |
| **3** | 0.92 | 99.29 | 99.30 | 99.29 | 99.29 | 99.29 | 99.29 | 99.29 | 99.29 |
| **4** | 0.86 | 99.24 | 99.24 | 99.24 | 99.24 | 99.20 | 99.20 | 99.20 | 99.20 |
| **Avg.** | **0.88** | **99.14** | **99.14** | **99.14** | **99.14** | **99.13** | **99.13** | **99.13** | **99.13** |

**Table (B.14):** NB: 70-30 sampling - 900 feature Shamela without stemming on a 16 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 0.80 | 98.82 | 98.83 | 98.82 | 98.82 | 98.09 | 98.15 | 98.09 | 98.09 |
| **2** | 0.84 | 97.16 | 97.19 | 97.16 | 97.14 | 96.19 | 96.26 | 96.19 | 96.15 |
| **3** | 0.80 | 98.04 | 98.07 | 98.04 | 98.03 | 97.10 | 97.16 | 97.10 | 97.06 |
| **4** | 1.39 | 97.96 | 97.99 | 97.96 | 97.96 | 96.82 | 96.87 | 96.82 | |
| **Avg.** | **0.96** | **98.00** | **98.02** | **98.00** | **97.99** | **97.05** | **97.11** | **97.05** | **72.82** |

**Table (B.15):** NB: 70-30 sampling - 900 feature Al-Bokhary without stemming on a 16 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 0.40 | 95.85 | 95.91 | 95.85 | 95.83 | 88.03 | 88.69 | 88.03 | 87.95 |
| **2** | 0.56 | 95.85 | 95.89 | 95.85 | 95.83 | 87.78 | 88.34 | 87.78 | 87.66 |
| **3** | 0.52 | 95.78 | 95.83 | 95.78 | 95.75 | 87.78 | 88.40 | 87.78 | 87.59 |
| **4** | 0.46 | 95.82 | 95.87 | 95.82 | 95.81 | 87.86 | 88.19 | 87.86 | 87.76 |
| **Avg.** | **0.49** | **95.82** | **95.87** | **95.82** | **95.80** | **87.86** | **88.40** | **87.86** | **87.74** |

**Table (B.16):** NB: 70-30 sampling - 900 feature Shamela-More light1 stemming on a 16 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 0.68 | 98.66 | 98.67 | 98.66 | 98.66 | 98.66 | 98.67 | 98.66 | 98.66 |
| **2** | 0.74 | 98.71 | 98.71 | 98.71 | 98.71 | 98.62 | 98.62 | 98.62 | 98.62 |
| **3** | 0.76 | 98.55 | 98.57 | 98.55 | 98.55 | 98.56 | 98.58 | 98.56 | 98.56 |
| **4** | 0.73 | 99.38 | 98.39 | 98.38 | 98.38 | 98.30 | 98.31 | 98.30 | 98.30 |
| **Avg.** | **0.73** | **98.82** | **98.59** | **98.57** | **98.58** | **98.53** | **98.55** | **98.53** | **98.53** |

## B.2 Logistic Regression

LR is a predictive analysis has difficult computation that cost time and memory resources but its results are more accurate.

**Table (B.17):** LR: 70-30 sampling - 900 feature Shamela root stemming on a standalone node

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 8.67 | 99.5 | 99.6 | 99.59 | 99.59 | 99.45 | 99.46 | 99.45 | 99.45 |
| **Avg.** | **8.67** | **99.59** | **99.6** | **99.59** | **99.59** | **99.45** | **99.46** | **99.45** | **99.45** |

**Table (B.18):** LR: 70-30 sampling - 900 feature Shamela-More root stemming on a standalone node

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 9.93 | 99.86 | 99.86 | 99.86 | 99.86 | 99.84 | 99.84 | 99.84 | 99.84 |
| **2** | 10.12 | 99.89 | 99.89 | 99.89 | 99.89 | 99.85 | 99.85 | 99.85 | 99.85 |
| **Avg.** | **10.03** | **99.87** | **99.87** | **99.87** | **99.87** | **99.85** | **99.85** | **99.85** | **99.85** |

**Table (B.19):** LR: 70-30 sampling - 10,000 feature Al-Bokhary light1 stemming on a standalone node

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 1.69 | 94.46 | 94.51 | 94.46 | 94.47 | 81.23 | 81.67 | 81.23 | 81.29 |
| **Avg.** | **1.69** | **94.46** | **94.51** | **94.46** | **94.47** | **81.23** | **81.67** | **81.23** | **81.29** |

**Table (B.20):** LR: 70-30 sampling - 10,000 feature Al-Bokhary without stemming on a 2 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 4.62 | 93.34 | 93.43 | 93.34 | 93.35 | 77.51 | 78.34 | 77.51 | 77.61 |
| **2** | 4.26 | 93.70 | 93.78 | 93.70 | 93.71 | 78.72 | 79.49 | 78.72 | 78.90 |
| **3** | 4.34 | 93.65 | 93.71 | 93.65 | 93.65 | 78.48 | 79.08 | 78.48 | 78.52 |
| **Avg.** | **4.41** | **93.56** | **93.64** | **93.56** | **93.57** | **78.24** | **78.97** | **78.24** | **78.34** |

**Table (B.21):** LR: 70-30 sampling - 900 feature Shamela root stemming on a 2 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 6.30 | 99.20 | 99.23 | 99.20 | 99.20 | 99.28 | 99.30 | 99.28 | 99.28 |
| **Avg.** | **6.30** | **99.20** | **99.23** | **99.20** | **99.20** | **99.28** | **99.30** | **99.28** | **99.28** |

**Table (B.22):** LR: 70-30 sampling - 900 feature Shamela light1 stemming on a 2 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 7.83 | 90.83 | 96.21 | 90.83 | 92.72 | 91.00 | 95.52 | 91.00 | 92.58 |
| **Avg.** | **7.83** | **90.83** | **96.21** | **90.83** | **92.72** | **91.00** | **95.52** | **91.00** | **92.58** |

**Table (B.23):** LR: 70-30 sampling - 900 feature Shamela-More without stemming on a 4 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 5.27 | 99.45 | 99.46 | 99.45 | 99.45 | 99.43 | 99.43 | 99.43 | 99.43 |
| **2** | 5.82 | 99.48 | 99.48 | 99.48 | 99.48 | 99.47 | 99.47 | 99.47 | 99.47 |
| **3** | 5.74 | 99.48 | 99.48 | 99.48 | 99.48 | 99.45 | 99.46 | 99.45 | 99.45 |
| **Avg.** | **5.61** | **99.47** | **99.47** | **99.47** | **99.47** | **99.45** | **99.45** | **99.45** | **99.45** |

**Table (B.24):** LR: 70-30 sampling - 900 feature Shamela without stemming on a 4 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 4.98 | 96.14 | 96.41 | 96.14 | 96.09 | 94.92 | 95.36 | 94.92 | 94.83 |
| **2** | 5.19 | 96.41 | 96.61 | 96.41 | 96.35 | 95.55 | 95.81 | 95.55 | 95.43 |
| **3** | 4.87 | 90.96 | 93.07 | 90.96 | 90.96 | 89.29 | 91.37 | 89.29 | 89.08 |
| **Avg.** | **5.01** | **94.50** | **95.36** | **94.50** | **94.47** | **93.25** | **94.18** | **93.25** | **93.11** |

**Table (B.25):** LR: 70-30 sampling - 10,000 feature Al-Bokhary without stemming on a 4 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | Time **(min)** | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 1.12 | 93.51 | 93.59 | 93.51 | 93.52 | 77.99 | 78.84 | 77.99 | 78.13 |
| **2** | 1.11 | 93.44 | 93.51 | 93.44 | 93.51 | 77.75 | 78.46 | 77.75 | 77.83 |
| **3** | 1.10 | 93.65 | 93.72 | 93.65 | 93.65 | 78.48 | 79.13 | 78.48 | 78.50 |

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| # | Time (min) | accuracy | precision | recall | fmeaseure | accuracy | precision | recall | fmeaseure |
| **Avg.** | **1.11** | **93.53** | **93.61** | **93.53** | **93.56** | **78.07** | **78.81** | **78.07** | **78.15** |

**Table (B.26):** LR: 70-30 sampling - 900 feature Shamela root stemming on an 8 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| # | Time (min) | accuracy | precision | recall | fmeaseure | accuracy | precision | recall | fmeaseure |
| **1** | 1.59 | 99.28 | 99.30 | 99.28 | 99.28 | 99.37 | 99.39 | 99.37 | 99.37 |
| **2** | 1.60 | 99.68 | 99.68 | 99.68 | 99.68 | 99.64 | 99.65 | 99.64 | 99.64 |
| **3** | 1.58 | 99.57 | 99.58 | 99.57 | 99.57 | 99.37 | 99.38 | 99.37 | 99.37 |
| **Avg.** | **1.59** | **99.51** | **99.52** | **99.51** | **99.51** | **99.46** | **99.47** | **99.46** | **99.46** |

**Table (B.27):** LR: 70-30 sampling - 900 feature Shamela light1 stemming on an 8 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| # | Time (min) | accuracy | precision | recall | fmeaseure | accuracy | precision | recall | fmeaseure |
| **1** | 1.84 | 89.11 | 94.90 | 89.11 | 91.20 | 89.15 | 93.98 | 89.15 | 90.94 |
| **2** | 1.84 | 89.19 | 94.02 | 89.19 | 90.75 | 88.59 | 93.44 | 88.59 | 90.24 |
| **3** | 1.83 | 90.94 | 94.44 | 90.94 | 91.93 | 90.91 | 93.91 | 90.91 | 91.73 |
| **Avg.** | **1.84** | **89.75** | **94.45** | **89.75** | **91.30** | **89.55** | **93.77** | **89.55** | **90.97** |

**Table (B.28):** LR: 70-30 sampling - 10,000 feature Al-Bokhary root stemming on an 8 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| # | Time (min) | accuracy | precision | recall | fmeaseure | accuracy | precision | recall | fmeaseure |
| **1** | 1.09 | 97.85 | 97.87 | 97.85 | 97.85 | 92.72 | 92.89 | 92.72 | 92.74 |
| **2** | 1.09 | 98.11 | 98.13 | 98.11 | 98.11 | 93.61 | 93.76 | 93.61 | 93.60 |
| **3** | 1.10 | 98.00 | 98.01 | 98.00 | 98.00 | 93.20 | 93.43 | 93.20 | 93.23 |
| **Avg.** | **1.09** | **97.99** | **98.00** | **97.99** | **97.99** | **93.18** | **93.36** | **93.18** | **93.19** |

**Table (B.29):** LR: 70-30 sampling - 900 feature Shamela-More without stemming on a 16 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| # | Time (min) | accuracy | precision | recall | fmeaseure | accuracy | precision | recall | fmeaseure |
| **1** | 2.59 | 99.45 | 99.45 | 99.45 | 99.45 | 99.41 | 99.41 | 99.41 | 99.41 |
| **2** | 1.87 | 99.54 | 99.54 | 99.54 | 99.54 | 99.50 | 99.51 | 99.50 | 99.51 |
| **3** | 2.05 | 99.43 | 99.44 | 99.43 | 99.43 | 99.40 | 99.40 | 99.40 | 99.40 |
| **Avg.** | **2.17** | **99.47** | **99.48** | **99.47** | **99.47** | **99.44** | **99.44** | **99.44** | **99.44** |

**Table (B.30):** LR: 70-30 sampling - 900 feature Shamela without stemming on a 16 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 1.85 | 94.66 | 94.99 | 94.66 | 94.57 | 93.19 | 93.58 | 93.19 | 93.01 |
| **2** | 2.31 | 96.92 | 97.20 | 96.92 | 96.96 | 95.64 | 96.13 | 95.64 | 95.72 |
| **3** | 1.84 | 93.08 | 94.88 | 93.08 | 93.42 | 91.74 | 94.01 | 91.74 | 92.15 |
| **Avg.** | **2.00** | **94.89** | **95.69** | **94.89** | **94.98** | **93.53** | **94.57** | **93.53** | **93.63** |

**Table (B.31):** LR: 70-30 sampling - 10,000 feature Al-Bokhary without stemming on a 16 nodes cluster

| | | dataRDD | | | | testRDD | | | |
|---|---|---|---|---|---|---|---|---|---|
| **#** | **Time** (min) | **accuracy** | **precision** | **recall** | **fmeaseure** | **accuracy** | **precision** | **recall** | **fmeaseure** |
| **1** | 3.29 | 93.75 | 93.79 | 93.75 | 93.74 | 78.80 | 78.23 | 78.80 | 78.75 |
| **2** | 1.12 | 93.65 | 93.73 | 93.65 | 93.66 | 78.48 | 79.40 | 78.48 | 78.65 |
| **3** | 1.13 | 94.01 | 94.10 | 94.01 | 94.01 | 79.69 | 80.60 | 79.69 | 79.76 |
| **Avg.** | **1.84** | **93.80** | **93.87** | **93.80** | **93.80** | **78.99** | **79.41** | **78.99** | **79.05** |