DESIGN AND DEVELOPMENT OF REAL-TIME BIG DATA ANALYTICS FRAMEWORKS

by

M Solaimani

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Latifur Khan, Chair

Dr. Bhavani Thuraisingham

Dr. Farokh B. Bastani

Dr. Haim Schweitzer
To my parents, and wife.
DESIGN AND DEVELOPMENT OF REAL-TIME BIG DATA
ANALYTICS FRAMEWORKS

by

M SOLAIMANI, BS, MS

DISSERTATION
Presented to the Faculty of
The University of Texas at Dallas
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS
December 2017
ACKNOWLEDGMENTS

I would like to thank a number of people in my academic, social and family life who have inspired me as I pursued this degree. My advisor, Dr. Latifur Khan, is the person I admire most for his valuable teaching, wisdom, continuous support and guidance in conducting research throughout my PhD years. His endless trust, patience, and advice has helped my academic journey and guided me towards my accomplishments. I am really grateful to my lab mates Ahsan, Iftekhar, Sayeed, Khaled, Ahmad, Swarup, Vishal, Gbadebo, and Maryum for being great friends. I really enjoyed their company, collaboration, and depth of knowledge.

I would like to thank my committee members, Dr. Bhavani Thuraisingham, Dr. Haim Schweitzer, and Dr. Farokh Bastani, for evaluating my dissertation.

I would like to thank my parents, father-in-law, mother-in-law, my wife, and my sister for their enormous support during all these years.

Finally, I would like to thank NSF and SANDIA LAB for sponsoring my research.

October 2017
DESIGN AND DEVELOPMENT OF REAL-TIME BIG DATA ANALYTICS FRAMEWORKS

M Solaimani, PhD
The University of Texas at Dallas, 2017

Supervising Professor: Dr. Latifur Khan, Chair

Today most sophisticated technologies such as Internet of Things (IoT), autonomous driving, Cloud, data center consolidation, etc., demand smarter IT infrastructure and real-time operations. They continuously generate lots of data called “Big Data” to report their operational activities. In response to this, we need advanced analytics frameworks to capture, filter, and analyze data and make quick decisions in real-time. The high volumes, velocities, and varieties of data make it an impossible (overwhelming) task for humans in real-time.

Current state-of-the-arts like advanced analytics, Machine learning (ML), Natural Language Processing (NLP) can be utilized to handle heterogeneous Big Data. However, most of these algorithms suffer scalability issues and cannot manage real-time constraints. In this dissertation, we have focused on two areas: anomaly detection on structured VMware performance data (e.g., CPU/Memory usage metric, etc.) and text mining for politics in unstructured text data. We have developed real-time distributed frameworks with ML and NLP techniques. With regard to anomaly detection, we have implemented an adaptive clustering technique to identify individual anomalies and a Chi-square-based statistical technique to detect group anomalies in real-time. With regards to text mining, we have developed a real-time framework \textit{SPEC} to capture online news articles of different languages from the web and annotated them using CoreNLP, PETRARCH, and CAMEO dictionary to gener-
ate structured political events like ‘who-did-what-to-whom’ format. Later, we extend this framework to code atrocity events – a machine coded structured data containing perpetrators, action, victims, etc. Finally, we have developed a novel, distributed, window-based political actor recommendation framework to discover and recommend new political actors with their possible roles. We have implemented scalable distributed streaming frameworks with a message broker – Kafka, unsupervised and supervised machine learning techniques and Spark.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGMENTS</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiv</td>
</tr>
<tr>
<td>CHAPTER 1   INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Real-time Data Analytics</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Popular Big Data Frameworks</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Design Challenges</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.4.1 Designing a real-time anomaly detection framework</td>
<td>4</td>
</tr>
<tr>
<td>1.4.2 Designing a real-time framework for unstructured text analysis</td>
<td>4</td>
</tr>
<tr>
<td>1.4.3 Designing a real-time framework for encoding atrocity events from unstructured text</td>
<td>5</td>
</tr>
<tr>
<td>1.4.4 Designing a real-time new political actor recommendation framework</td>
<td>6</td>
</tr>
<tr>
<td>1.5 Organization of Dissertation</td>
<td>6</td>
</tr>
<tr>
<td>CHAPTER 2   RELATED WORK</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Online Anomaly Detection Framework</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Statistical Real-time Anomaly Detection Framework</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Spark-based Event Coding</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Near Real-time Atrocity Event Coding</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Automatic Political Actor Recommendation in Real-time</td>
<td>12</td>
</tr>
<tr>
<td>CHAPTER 3   ONLINE ANOMALY DETECTION FRAMEWORK</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Background</td>
<td>17</td>
</tr>
<tr>
<td>3.1.1 Data Center</td>
<td>17</td>
</tr>
<tr>
<td>3.1.2 Distributed Resource Scheduler</td>
<td>17</td>
</tr>
<tr>
<td>3.1.3 VMware Performance Stream Data</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Real-time Anomaly Detection Framework</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1 Apache Storm</td>
<td>20</td>
</tr>
</tbody>
</table>
3.2.2 Apache Spark .......................... 21
3.2.3 Dynamic Resource Management .................. 22

3.3 Implementation Details ......................... 24
  3.3.1 Storm-based Anomaly Detection Framework ............... 24
  3.3.2 Spark-based Anomaly Detection Framework .......... 29
  3.3.3 Scalability .................................. 33
  3.3.4 Generating Anomaly .......................... 33
  3.3.5 Design Challenge .......................... 33

3.4 Experimental Result .......................... 34
  3.4.1 Dataset .................................. 34
  3.4.2 Result .................................. 34
  3.4.3 Comparison with Storm ...................... 35
  3.4.4 Experiment of Parallelism ................. 37

CHAPTER 4 STATISTICAL REAL-TIME ANOMALY DETECTION FRAMEWORK 39
  4.1 Real-time Anomaly Detection Framework ............... 41
    4.1.1 Chi-square Test .......................... 42
    4.1.2 Statistical anomaly detection ............... 42
    4.1.3 Statistical Stream Data Mining Module Implementation Using Spark 46
    4.1.4 Cluster-based Statistical Stream Data Mining Module Implementation Using Spark 51

  4.2 Case Study: Real-Time Anomaly Detection In VMware-based Data Center 53
    4.2.1 Data Center ................................ 54
    4.2.2 Dynamic Resource Scheduler .................. 54
    4.2.3 Implementation Details ...................... 55
    4.2.4 Building Statistical Training Model and Prediction .......... 56
    4.2.5 Scalability ................................ 56

  4.3 Experimental Results .......................... 57
    4.3.1 Dataset .................................. 57
    4.3.2 Results .................................. 57
3.1 Dynamic resource management using Apache Storm . . . . . . . . . . . . . . . . 23
3.2 Dynamic resource management using Apache Spark . . . . . . . . . . . . . . . . 23
3.3 Technical Approach for Storm-based Framework . . . . . . . . . . . . . . . . . . . 25
3.4 Technical Approach for Spark-based Framework Technical Approach . . . . . . . 30
3.5 Comparing average tuple processing latency during clustering . . . . . . . . . . 36
3.6 Comparing average tuple processing latency during prediction . . . . . . . . . . 36
3.7 Average tuple processing latency for different input VMs . . . . . . . . . . . . . . 37
4.1 Statistical technique for anomaly detection . . . . . . . . . . . . . . . . . . . . . 44
4.2 Statistical real-time anomaly detection framework . . . . . . . . . . . . . . . . . 45
4.3 Cluster based statistical real-time anomaly detection framework . . . . . . . . . 50
4.4 Dynamic resource management using Apache Spark . . . . . . . . . . . . . . . . 54
4.5 Average window processing latency during training . . . . . . . . . . . . . . . . 59
4.6 Comparing average window processing latency during testing . . . . . . . . . . 60
5.1 Political event coding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.2 Part-of-Speech and NER . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.3 Basic Dependency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.4 Co-reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.5 Spark-based political event coding framework . . . . . . . . . . . . . . . . . . . 73
5.6 CoreNLP parsing time over number of workers . . . . . . . . . . . . . . . . . . . 81
6.1 Atrocity event coding framework . . . . . . . . . . . . . . . . . . . . . . . . . . 85
6.2 Identifying the type of a field. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
6.3 Identifying the label of a field. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
6.4 Performance of semantic labeling victim for $\text{tf-idf}$ . . . . . . . . . . . . . . . 95
6.5 Performance of semantic labeling victim for $\text{phrase}$ . . . . . . . . . . . . . . . 95
7.1 Framework for real-time new political actor recommendation . . . . . . . . . . . 99
7.2 Technical details. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
7.3 Example Scenario for Graph Based Role Recommendation . . . . . . . . . . . . 111
7.4 Performance of Actor recommendation . . . . . . . . . . . . . . . . . . . . . . . 112
7.5 Performance of role recommendation ............................................. 113
7.6 Comparison of actor role recommendation with baseline ....................... 115
7.7 Baseline comparison with actor detection coding .................................. 117
7.8 Processing time of a burst input document ......................................... 118
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Spark Cluster</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Training of adaptive clustering using Spark</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Testing of adaptive clustering using Spark</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Training of cluster based model</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Testing of cluster based model</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Accuracy of cluster based model</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>News agencies</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>Spark Cluster Summary</td>
<td>78</td>
</tr>
<tr>
<td>5.3</td>
<td>Annotator processing time by CoreNLP</td>
<td>79</td>
</tr>
<tr>
<td>5.4</td>
<td>Event coding using PETRARCH</td>
<td>79</td>
</tr>
<tr>
<td>5.5</td>
<td>Processing time by PETRARCH</td>
<td>80</td>
</tr>
<tr>
<td>5.6</td>
<td>Total processing time by CoreNLP</td>
<td>80</td>
</tr>
<tr>
<td>5.7</td>
<td>Total processing time by PETRARCH</td>
<td>80</td>
</tr>
<tr>
<td>6.1</td>
<td>Spark Cluster Summary</td>
<td>92</td>
</tr>
<tr>
<td>6.2</td>
<td>Atrocity Filter Performance</td>
<td>93</td>
</tr>
<tr>
<td>6.3</td>
<td>Victim category distribution (each document has one category)</td>
<td>94</td>
</tr>
<tr>
<td>6.4</td>
<td>Total victim category distribution</td>
<td>94</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance metric to identify VICTIM category</td>
<td>94</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance of exact labeling victim</td>
<td>95</td>
</tr>
<tr>
<td>7.1</td>
<td>Symbol for algorithm</td>
<td>104</td>
</tr>
<tr>
<td>7.2</td>
<td>List of recommended actors with their roles</td>
<td>117</td>
</tr>
<tr>
<td>7.3</td>
<td>Spark Cluster Summary</td>
<td>118</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

In the recent era, real-time data processing and analytics with Big Data have drawn enormous attention. Nowadays, businesses are competitive and correct tactical on time decisions can make a company succeed to a great extent. Therefore, organizations must have strategies to make front-line operational decisions at once to compete in the market. Whether marketing products with promotions to customers, detecting fraudulent transactions, operating and managing a complex large system, driving autonomous cars or building applications with the Internet of Things – a real-time decision is essential. Online social media, smartphones, laptops, sensors, cameras, etc., are ubiquitous and generate a huge number of continuous streams of data. In order to extract the meaningful information from these dynamic and perishable data, an organization should act promptly to process data and take necessary actions. In response to these real-time requirements on massive data, agile, analytic, adaptive, and scalable real-time frameworks are required which combine business rules and predictive analytics to render tailored recommendations. Real-time analytics in this regard involves absorbing continuous streams of data, performing analysis on them, and initiating corrective measures if necessary. Therefore, it is a challenging task given the volume, velocity, and possibly complex nature of the input data. Failure to accomplish this task in a timely manner may lead to catastrophic consequences with a severe impact on business continuity.

1.1 Real-time Data Analytics

Real-time analytics processes and discovers meaningful patterns from incoming data streams as soon as it is collected into the system. It enables faster and time-sensitive, more precise and more effective decisions to a level of computer responsiveness that a user senses as immediate or nearly immediate. Examples of such applications are radar systems, ATM
services, etc. Near real-time analytics is not as time sensitive as real-time. It processes data in acceptable time like a few seconds to minutes. Applications such as online operational root cause analysis in a network, complete event processing may tolerate up to a few seconds or even minutes. On the other hand, batch analytics is even less time-sensitive. Sometimes, it takes hours or even days to complete. Here, data is collected before running analytics. Analytics on historical or archive data from social media, sensor data, or logs belongs to this category.

Real-time Big Data analytics discovers hidden patterns, correlations and other insights from the raw data in real-time. Big Data means a voluminous amount of structured or unstructured data of different types (e.g., video, audio, text, etc.) coming from multiple data sources [36, 35]. As data arrives really fast, analytics has to be done more quickly to deal with data “in motion” – not “in static”. The high volume of data spots system limitation and demands a scalable solution. Advanced analytics includes Natural Language Processing (NLP), Machine Learning (ML), etc., which provide accurate prediction using both static and dynamic models and are deployed as integral components of a real-time analytical system. For example, NLP techniques like sentence parsing, parts of speech tagger, dependency tree parsing, Named-entity Recognition, etc., are to process raw unstructured text data and convert them to structured data. Furthermore, supervised and unsupervised learning in ML, recommendation system, etc., improve analytics substantially.

Individually, none of the methods which are discussed above are difficult to implement, but their combination – in a real-time analytics poses a new set of requirements: computation with millisecond latency at high throughput for massive data processing, building models and prediction, distributed computations, etc. Such analytics related to Big Data must be built upon scalable frameworks which are capable of handling extremely large amounts of data with low latency.
1.2 Popular Big Data Frameworks

Big Data frameworks like Hadoop [33], MapReduce [21], HBase [37], Mahout [48], Google Bigtable [16], etc., are highly scalable, but are geared more toward batch processing. Examples of frameworks with built-in stream processing capabilities are Apache Storm [86], Apache S4 [73] and Apache Spark [84]. Spark performs in-memory analytics on stream data by running streaming computations as a series of micro-batch jobs. It makes the batch size as small as possible to achieve low latency. It also keeps the states between batches in memory so that it can recover them quickly. Spark uses an advanced DAG (Directed Acyclic Graph) execution engine that supports cyclic data flow and in-memory computing, which makes it faster than other distributed frameworks. Spark is 100 times faster than Hadoop, MapReduce in memory, and ten times faster on disk [84]. It is also faster than Storm and S4 [107, 105]. Overall, it generates low latency, real-time results. It also has instant productivity, no hidden obstacles, and an easy cluster setup procedure.

1.3 Design Challenges

Designing a real-time analytics framework in a distributed system requires solving some key challenges. First, how should the data be handled within the distributed framework in order to process it efficiently and reliably? Distributed message brokers (e.g., Apache Kafka [40], Apache Flume [24]) ship the data to the distributed framework without any data loss. A fixed time interval is used to capture and process data from the message broker. Second, how should machine learning algorithms be implemented in the distributed system? Streaming distributed frameworks like Apache Spark [84], Apache Storm [86] come with a few standard machine learning algorithms. In order to use custom algorithms, they must be implemented in such a way that they fully utilize the benefits of a distributed framework. Considering the above challenges, we have developed end-to-end real-time distributed frameworks with Machine Learning, Natural Language Processing, and Big Data tools to perform analytics.
1.4 Contributions

We have highlighted our contribution in two major areas: real-time anomaly detection and near real-time text mining for politics. In anomaly detection, we use VMware structured performance data like CPU metric (e.g., CPU/Memory usage, etc.) and in text mining for politics, we use unstructured raw text collected from online news reports.

1.4.1 Designing a real-time anomaly detection framework

Anomaly detection refers to the identification of an irregular or unusual pattern which deviates from what is standard, normal, or expected. Such deviated patterns typically correspond to samples of interest and are assigned different labels in different domains, such as outliers, anomalies, exceptions, or malware. Detecting anomalies in fast, voluminous streams of data is a formidable challenge.

We have presented novel, generic, real-time distributed anomaly detection frameworks for heterogeneous streaming data where anomalies appear either independently or as a group. We have developed an adaptive distributed clustering method to build a model from normal data and later use it to detect an individual anomaly. We have developed a distributed statistical approach to build a model and later use it to detect group anomaly. As a case study, we investigate anomaly detection for a VMware-based cloud data center, which maintains a large number of virtual machines (VMs). We have built our frameworks using Apache Spark to get higher throughput and lower data processing time on streaming data.

1.4.2 Designing a real-time framework for unstructured text analysis

Political event data have been widely used to study international politics. Previously, natural text processing and event generation required a lot of human efforts. Today we have high computing infrastructure with advanced NLP metadata to leverage those tiresome efforts.
TABARI 87 – an open source non distributed event-coding software – was an early effort to generate events from a large corpus. It uses a shallow parser to identify the political actors but ignores semantics and relations among the sentences. PETRARCH 59, the successor of TABARI, encodes event data into “who-did-what-to-whom” format. It uses Stanford CoreNLP 49 to parse sentences and a static CAMEO 14 dictionary to encode the data. To build dynamic dictionaries, we need to analyze more metadata such as the token, Named-entity Recognition (NER), co-reference, and much more from parsed sentences. Although these tools can code modest amounts of source corpora into event data they are too slow and suffer scalability issues when we try to extract metadata from a single document. The situation gets worse for other languages like Spanish or Arabic.

We develop a novel distributed framework SPEC 1 using Apache Spark, MongoDB 53, Stanford CoreNLP, and PETRARCH. It shows a distributed workflow by using Stanford CoreNLP to extract all the metadata (parse tree, tokens, lemma, etc.) from the news corpora of the Gigaword 30 dataset and storing it to MongoDB. Then it uses PETRARCH to encode events from the metadata. The framework integrates both tools using distributed commodity hardware and reduces text processing time substantially with respect to a non-distributed architecture. We have chosen Spark over existing distributed frameworks for its in-memory computation and lower processing time in both batch and stream processing. After that, we extend our framework into two different directions: near real-time atrocity coding and recommending new political actors with their role in real-time.

1.4.3 Designing a real-time framework for encoding atrocity events from unstructured text

Mass atrocities, terrorism, and political unrest have caused much human suffering. Thousands of innocent lives have been lost to these events. With the help of advanced technologies,
we can now dream of a tool that uses machine learning and natural language processing (NLP) techniques to warn of such events. Detecting atrocities demands structured event data that contain metadata, with multiple fields and values (e.g., event date, victim, perpetrator). Traditionally, humans apply common sense and encode events from news stories—but this process is slow, expensive, and ambiguous. To accelerate it, we use machine coding to generate an encoded event.

In this work, we develop a novel near-real-time supervised machine coding technique with an external knowledge base, WordNet [99], to generate a structured event. We design a Spark-based distributed framework with a web scraper to gather news reports periodically, process, and generate events. We use Spark to reduce the performance bottleneck while processing raw text news using CoreNLP.

1.4.4 Designing a real-time new political actor recommendation framework

Extracting actor data from news reports is important when generating event data. Hand-coded dictionaries are used to code actors and actions. Manually updating dictionaries for new actors and roles is costly and there is no automated method.

We propose a novel real-time framework $APART^2$ using dynamic frequency-based actor ranking algorithm with partial string matching for new actor-role recommendation, based on similar actors in the CAMEO dictionary.

1.5 Organization of Dissertation

The rest of my dissertation is organized as follows: Chapter 2 sums up all related works, Chapter 3 describes adaptive clustering based online anomaly (individual) detection framework, Chapter 4 describes statistical real-time group anomaly detection framework. Chapter 5 describes a spark-based political event coding framework. Chapter 6 discusses a near

\footnote{https://github.com/openeventdata/political-actor-recommendation}
real-time atrocity coding and Chapter 7 shows a real-time political actor recommendation framework. Finally, Chapter 8 presents the conclusion and future works.
CHAPTER 2
RELATED WORK

In this chapter, we will discuss the related works. Our related works mainly cover real-time anomaly detection frameworks and near real-time text mining. We have arranged them as follows: Section 2.1 shows existing works on online anomaly detection using machine learning, and Section 2.2 shows statistical anomaly detections in real-time. Section 2.3 reviews related works on political event coding, Section 2.4 contains atrocity event codings, and finally, Section 2.5 describes existing works on real-time political actor recommendation.

2.1 Online Anomaly Detection Framework

Anomaly detection naturally lends itself to clustering and therefore has been explored as a K-means clustering problem in the literature. However, for large datasets, the time complexity of K-means is prohibitive. Additionally, K-means’ initial centroid problem is a disadvantage. The initial centroid problem is circumvented by K-medoids \[41\] which starts with \(k\) centers, modifying the centers repeatedly and at random thereby improving the sum of squared error. Detection of outliers on stream data has been addressed by \[3\]. Their algorithm, known as AnyOut, utilizes hierarchical clustering. The deviation between an object and a cluster is determined by an outlier score. However, the structural complexity of hierarchical clustering translates to high processing time and poses a scalability challenge in case of distributed systems.

In distributed systems, anomaly detection entails processing of large-scale datasets with the aid of distributed frameworks. Among them, Hadoop \[33\] and MapReduce perform extremely well on offline data but lack the capabilities for handling real-time stream data. Similar technologies e.g., HBase \[37\], BashReduce \[26\], etc., are not designed for stream data processing. Apache Mahout \[48\], a MapReduce-based machine learning framework,
provides implementations for K-means and StreamingKMeans which only run in batch mode. Anomaly detection framework based on Hadoop and MapReduce have been proposed by [103] and [32]. Due to the inherent restrictions of the underlying frameworks, their approaches are not readily extensible to real-time processing.

The state-of-the-art in real-time data processing includes Apache Storm [86] and Apache S4 [73]. Spark being faster than Storm [105], has been used as the underlying framework of our approach. [83] performed a real-time anomaly detection framework for VMware performance data which is based on Apache Storm. [82] also presented the spark-based anomaly detection technique. Our detailed comparative analysis of storm-based and spark-based anomaly detection frameworks confirms that Spark incurs lower average tuple execution time. Traditional approaches to anomaly detection usually focus on accuracy and pay less attention to performance. However, performance is a key contributing factor to the efficacy of a real-time anomaly detector and our approach aims to find a framework that excels both in accuracy and performance. Although Spark provides a StreamingKMeans implementation, it assumes pre-domain knowledge of data. In contrast, our approach is agnostic about the VMware performance data while building the model. Therefore, we elected to use our flat incremental clustering instead of Spark’s StreamingKMeans.

### 2.2 Statistical Real-time Anomaly Detection Framework

Most of the anomaly detection techniques focus on point anomalies (Section 2.1) but we know that anomalies can form a group. [104] implemented a group-based anomaly detection system on social media. [100] proposed a Flexible Genry Model (a model that addresses both point and group data characteristic) to capture the group anomaly. All these works have used typical topic models [85] to identify group behavior. Our cluster-based model also captures group behavior but we do not consider the relationships among all the resources. For example, we experimentally find that many CPU hungry applications do not consume
higher memory & vice versa. So, we capture all the benign distributions for each resource during training within a particular time frame.

[97] implement statistical techniques to identify anomalies using a non-distributed framework. They use a multinomial goodness-of-fit test to match two unknown distributions but they assume that the two distributions have equal size widow. In practice, the window size may vary due to many reasons. In that case, we found their technique does not give good results. [81] perform real-time statistical anomaly detection for VMware performance data using Apache Spark [84] which performs well under the constraint that a certain window contains all anomalies. Moreover, it also uses a buffer to store past N models. So, an initial benign model might be identified as anomaly over time if the buffer does not contain that model. So it increases false alarms. We demonstrated that the approach proposed in this paper outperforms the one in [81] by addressing those issues. Apache Spark 1.1.0’s MLlib library [84] supports Pearson’s Chi-square goodness of fit test and independence test of paired observations of two variables.

2.3 Spark-based Event Coding

Political event data analysis has been carried out over many decades. [5] in their COPDAB project have used political event data to study foreign policy. [8] have studied political event data related to armed conflict. [34] have described a micro level study of political violence in their Global Database of Event Language and Tone (GDELT) while forecasting political conflict has been done recently by [15]

Political event data has been used to improve CAMEO dictionary. [91] scales up CAMEO dictionary to analyze categories of conflict and cooperation. [10] analyze TABARI coding system to identify actors and their relationship inside event data.

Very few frameworks exist to code political event. [79] show a complete picture of generating events from raw news text. It describes the life cycle of event generation, key pre-
processing steps, and also real-time coding with existing NLP tools. However, it does not address the scalability issue. [76] also show a framework EL:DIABLO that consists of both CoreNLP and PETRARCH. It shows the comparison between TABARI and PETRARCH. It has the same scalability problem which is not mentioned previously.

Over the past few years, distributed frameworks have become more popular. They scale up with existing commodity hardware and handle a large volume of data with lower processing time. We can divide out Big Data framework into two categories: streaming [84, 106] and non-streaming [102]. Streaming framework processes continuous data periodically. It may use machine learning techniques to analyze stream data. Spark, Storm, S4 are the key candidates to build up a streaming framework. [82] have proposed distributed real-time anomaly detection frameworks using both Spark and Storm for VMWare-based data center and also compare their processing time. Non-streaming Big data frameworks are built with Hadoop, MapReduce, HBase. They perform batch processing. Nowadays, Spark becomes popular and replaces MapReduce. [57] have shown a distributed framework to search key phrases from the web. They use MapReduce to build their framework. [2] have developed a distributed Geographic Information System Querying Framework (GISQF) to process Massive Spatial Data. They have used Spatial Hadoop [23] (extension of Hadoop) and MapReduce in their framework. Moreover, they have coded events from news text using TABARI and CAMEO.

MapReduce based frameworks are becoming obsolete and have been replaced with Spark, Storm, etc. Both Spark and Storm have lower processing time but Spark is the best for both batch and stream processing [105]. Considering all of these, we have built a framework with Spark, CoreNLP, and PETRARCH that can generate political events from the raw news text with lower processing time.
2.4 Near Real-time Atrocity Event Coding

Very few related works have been found to automatically perform atrocity event coding. [15] covers forecasting political conflict from events from Google. Recently, [74] showed a recommendation model to determine if an article reports a civil unrest event. Few frameworks exist to code political unrest events in real-time. [79] show a complete picture of generating events from raw news text. However, they do not address the scalability issue. We found Spark to be most suitable for our purposes (discussed in Section 2.3) to build a framework with CoreNLP, SEMAFOR that generates atrocity event data from regular news articles outside of our training set.

2.5 Automatic Political Actor Recommendation in Real-time

Actor dictionaries are crucial to constructing political event data. Currently, researchers manually update the CAMEO actor dictionaries when new entries are needed. This is a labor-intensive process that does not scale. Role recommendation based on surrounding keywords in the document can be considered. For example, ‘President’ keyword occurring in front of Barack Obama may help infer his role. But we may not get enough keywords like above to infer roles. Also, often they do not convey the complete role for an actor. Our above example shows a similar scenario. With the keyword ‘President’, we can imply Barack Obama as a government employee. But we cannot infer his country. Moreover, CAMEO uses short-form encoded roles (e.g., USAGOV) which requires a static mapping if we consider the above procedure.

Automatic Content Extraction (ACE) based event extraction systems use machine learning. [70] proposed TWICAL - an open domain event extraction and categorization system for Twitter. [28] showed event extraction using Wikipedia. All the above methods use classifiers to identify event type which performs well in a generic domain but not in a restricted
political domain. Therefore, we use an ontology/dictionary based event extraction method (PETRARCH) to extract political events. Distributed frameworks have become more popular for their scalability. We found Spark to be the most suitable for our purposes (discussed in Section 2.3) to build a framework with CoreNLP and PETRARCH that recommends new actors with their related roles for event data from regular news articles.
CHAPTER 3
ONLINE ANOMALY DETECTION FRAMEWORK

Real-time anomaly detection aims to capture abnormalities in system behavior in real-time. These abnormalities or anomalies may appear in the form of malicious network intrusions, malware infections, abnormal interaction patterns of individuals/groups in social media, over-utilized system resources due to design defects, etc. Anomaly detection in this regard involves absorbing continuous streams of data, performing analysis on them and initiating corrective measures if necessary. Therefore, it is a challenging task given the volume, velocity, and possibly complex nature of the input data. Failure to accomplish this task in a timely manner may lead to catastrophic consequences with a severe impact on business continuity.

The most rigorous form of real-time anomaly detection is observed in enterprise data centers which host tens of thousands of virtual machines. Dynamic resource scheduling is a crucial differentiator for the operational efficiency of these data centers. In response to varying demands for various resources, e.g., CPU and memory, the scheduler must allocate or re-allocate resources dynamically. This necessitates real-time monitoring of resource utilization and performance data for the purpose of detecting abnormal behavior.

Real-time anomaly detectors for data centers must be built upon scalable frameworks capable of handling extremely large amounts of data (so-called Big Data) with low latency. Popular Big Data frameworks, e.g., Hadoop, MapReduce, HBase, Mahout, Google Bigtable, etc., are highly scalable, but are geared more toward batch processing. Examples of frameworks with built-in stream processing capabilities are Apache Storm, Apache S4 and Apache Spark. Spark performs in-memory analytics on stream data.

by running streaming computations as a series of micro-batch jobs. It makes the batch size as small as possible to achieve low latency. It also keeps the states between batches in memory so that it can recover them quickly. Spark uses an advanced DAG (Directed Acyclic Graph) execution engine that supports cyclic data flow and in-memory computing, which makes it faster than other distributed frameworks. Spark is 100 times faster than Hadoop MapReduce in memory, and ten times faster on disk [84]. It is also faster than Storm and S4 [107]. Overall, it generates low latency, real-time results. It also has instant productivity, no hidden obstacles, and an easy cluster setup procedure.

Designing an online anomaly detection framework in a distributed system requires solving some key challenges. First, how should the data be handled within the distributed framework in order to process it efficiently and reliably? We utilize a message broker (Apache Kafka [40]) that ships the data to the distributed framework without any data loss. A fixed time interval is used to capture and process data from the message broker. Second, how should machine learning algorithms be implemented in the distributed system? Streaming distributed frameworks like Apache Spark [84], Apache Storm [86] come with a few standard machine learning algorithms. In order to use custom algorithms, they must be implemented in such a way that fully utilizes the benefits of a distributed framework. For our case study, we have implemented an adaptive incremental clustering algorithm.

Considering the above challenges, we have made the following contributions:

First, we have developed a novel generic real-time framework for multi-source stream data using Apache Spark [84] and Kafka [40] (a message broker). Kafka provides guaranteed message delivery with proper ordering. This means messages sent by a producer to a particular topic partition will be delivered in the order they are sent, and a consumer also sees messages in the order they are stored. Moreover, a Kafka cluster can be formed to lower processing latency and provide fault tolerance without loss of data. As a case study, we consider anomaly detection for a VMware-based data center. Our real-time framework is
generic so that it can handle continuous performance data (CPU load, memory usage, disk storage information, etc.) from multiple VMs at a time. This means that all VMs can send their performance data to our framework and it collects them as a data chunk. Kafka ships the continuous multi-source stream data from all VMs to a Spark cluster. Inside Spark, a machine learning technique is applied to analyze the data.

Second, we have also developed a novel real-time framework using Apache Storm and Kafka. Kafka ships the continuous performance stream data from all VMs to Storm. Inside Storm, a machine learning technique is applied to analyze the data. Our real-time framework monitors a continuous flow of VMware CPU usage statistics and analyzes it by applying an unsupervised incremental clustering technique. The framework has an option to accommodate other VMware performance data like memory usage, storage statistics, etc.

Third, an unsupervised flat adaptive incremental clustering technique has been implemented in both frameworks to model benign data in an online manner. Clusters are derived from benign training data only. The number of clusters is not fixed in advance, but are generated dynamically.

Fourth, we have also experimentally established the accuracy and low latency of our real-time frameworks for anomaly detection. The overall processing time has been reduced due to the reduction in prediction time.

Finally, we experimentally compared the average processing latency of a tuple during training and testing for both the Spark and Storm-based implementation. We found that Spark outperforms Storm substantially.

The rest of the chapter is organized as follows: Section 3.1 describes key concepts that are used throughout the chapter. Section 3.2 shows the anomaly detection framework in further detail. Section 3.3 shows more technical details regarding our novel Storm-based and Spark-based anomaly detection frameworks. Section 3.4 describes the experimental results with discussion.
3.1 Background

In this section, we will briefly describe data center and its dynamic scheduling using some machine learning techniques.

3.1.1 Data Center

A data center centralizes an organization’s information technology operations, infrastructure, and data storage. It houses computer systems and associated components, such as telecommunications and storage systems. Although data center designs are unique, they typically include a cloud appliance with VMware, OpenStack, Microsoft, etc., and resource management software for managing the storage infrastructure. Some advanced data centers manage their resources dynamically to adapt to their customers’ extra demands.

3.1.2 Distributed Resource Scheduler

VMware’s Distributed Resource Scheduler (DRS) balances the computational workloads with available resources in a virtualized environment. It allows users to define the rules for allocating physical resources among virtual machines. DRS uses resource pools which can be easily added, removed or reorganized. If the workload among virtual machines changes drastically, DRS reschedules and redistributes the virtual machines among the physical servers. It uses affinity and anti-affinity rules to improve the scheduling. In order to do real-time dynamic scheduling, machine learning techniques (unsupervised, and supervised learning, etc.) can be applied in the scheduler to analyze stream data. Machine learning is necessary because the behavior/nature of data is not known a priori.

For unsupervised learning, data instances associated with a “normal” load must first be gathered. Then, a clustering algorithm (e.g., k-means, incremental clustering, etc.) is applied to these instances. Finally, to classify new, unseen instances (e.g., a “normal” VM or
a “resource intensive” VM), an outlier detection approach can be applied. For example, for each test instance, if it is inside any cluster, then it can be treated as a VM with “normal” load. Otherwise, it is classified as a “resource intensive” VM. An instance is considered inside a cluster if the distance to its centroid is smaller than its radius.

For supervised learning, first, training data is collected for classes of interest namely, “normal” and “resource intensive” categories. Next, a classification model (or rules) is built from these training data using a supervised learning algorithm (e.g., support vector machines, decision trees, k-nearest neighbor). Finally, testing (unseen) instances are predicted using the classification model.

3.1.3 VMware Performance Stream Data

The performance metrics of VMware reflect its resources usage. If a user runs a resource intensive application (e.g., CPU or memory intensive), more resources must be allocated to the task in order for it to complete properly. This resource increase would be reflected in the associated counters in the performance metrics. Our real-time distributed framework should diagnose this potential issue and reschedule the appropriate resources dynamically.

VMware ESXi server [96] has an administrative command [ESXTOP] which provides access to the performance data. Periodically, this raw data is captured and processed into a feature vector that acts as a data point for our framework. [ESXTOP] gives several performance statistics of CPU, memory, and disk storage. For our case study, only CPU data is considered. The performance metrics contain average CPU load, percentage CPU usage of the physical cores (%Processor Time), percentage utilization of the logical cores (%Util Time), and percentage utilization of the physical cores (%Core Util Time).
3.2 Real-time Anomaly Detection Framework

A real-time anomaly detection framework consists of data preprocessing, model training, prediction, model updating, and resource scheduling. The stream data comes in a raw format, which needs to be processed in order to be analyzed. After processing, it is converted to multi-dimensional time series data, such as the CPU usage, CPU load of user programs, etc., in a data center over time. Sometimes preprocessing needs data normalization and noise elimination. The real-time framework applies machine learning algorithms to analyze stream data. It uses unsupervised clustering algorithms (e.g., \(k\)-means, incremental clustering) to build clusters from training data. Furthermore, the clustering applies only to benign data. After clustering, it predicts data using its cluster information. If any data fails to fall in any cluster, it is considered an anomaly; otherwise, it is considered benign. As the stream data evolve in nature, the training model should be updated periodically. Finally, the framework sends a request to the resource scheduler to update its resources if it detects any abnormal behavior.

Algorithm 1 describes the basic flow of the real-time framework. Initially, the model is built from the incoming stream data (line 2-3). After building the model, it predicts (line 4-5) data using that model. During prediction, if the framework finds any deviation from usual behavior, it notifies (line 6) the resource scheduler for rescheduling. It is important

Algorithm 1 Real-time framework

1: procedure IDENTIFYANOMALY(dataChunk, operation)
2: if operation == MODEL_BUILD then
3: \( m \leftarrow \text{BuildModel}(dataChunk) \)
4: if operation == PREDICT_DATA then
5: \( \text{hasAnomaly} \leftarrow \text{PredictData}(dataChunk, m) \)
6: \( \text{NotifyManagerForScheduling}(\text{hasAnomaly}) \)
7: if operation == MODEL_UPDATE then
8: \( m \leftarrow \text{UpdateModel}(dataChunk) \)
to periodically update the training model (line 7-8) to adapt to the dynamic nature of the
data, which hopefully reduces the potential for false alarms and missed anomalies.

Data center automation [95] (e.g., dynamic resource management) may require analyzing
the performance data in real-time to identify anomalies. As the stream data comes contin-
uously and is quite voluminous, a scalable distributed environment is required for analysis.
However, many distributed solutions (e.g., Hadoop, MapReduce, etc.) run in batch mode,
which makes it difficult to handle real-time data. Fortunately, there are a few distributed
frameworks capable of handling data in real-time. Two such distributed systems are Apache
Storm and Apache Spark.

### 3.2.1 Apache Storm

Apache Storm has a simple architecture consisting of three sets of nodes:

i) Nimbus node - a master node similar to the Hadoop [33] JobTracker. It distributes jobs
and launches workers across the cluster. It also monitors jobs and reallocates workers
as needed.

ii) ZooKeeper node [108] - communicates and coordinates the Storm cluster.

iii) Supervisor node - communicates with Nimbus through Zookeeper. It starts and stops
workers according to Nimbus.

Storm also consists of Spout, Bolt, and Topology. Spout provides an input stream into
Storm and is capable of reading external data sources. A Bolt is a processor unit that can
process any number of input streams and produce any number of output streams. Crucial
analysis logic is typically written in Bolt. Finally, Topology is a network of Spouts and Bolts.
It is a complex multi-stage stream computation that runs indefinitely when it is deployed. In
a Storm cluster, the master node coordinates and communicates with all the worker nodes.
3.2.2 Apache Spark

Apache Spark [84] is an open-source distributed framework for data analytics and also has simple architecture. It uses Hadoop [33] for the distributed file system and is capable of working on top of a next-generation Hadoop cluster called YARN [102]. Spark avoids the I/O bottleneck of the conventional two-stage MapReduce programs by providing in-memory cluster computing that allows a user to load data into a cluster’s memory and query it efficiently. This architecture increases the performance up to 100 times over that of Hadoop MapReduce [84].

Spark has two key concepts: Resilient Distributed Dataset (RDD) and directed acyclic graph (DAG) execution engine.

1. Resilient Distributed Dataset (RDD) [106] - a distributed memory abstraction. It allows in-memory computation on large distributed clusters with high fault-tolerance. Spark has two types of RDDs: parallelized collections that are based on existing programming collections (like list, map, etc.) and files stored on HDFS. RDD performs two kinds of operations: transformations and actions. Transformations create new datasets from the input or existing RDD (e.g., map or filter), and actions return a value after executing calculations on the dataset (e.g., reduce, collect, count, etc.). Transformations are the lazy operations that only define a new RDD, while actions perform the actual computation and calculate the result or write to external storage.

2. Directed acyclic graph (DAG) execution engine - whenever the user runs an action on an RDD, a directed acyclic graph is generated that considers all the transformation dependencies. This eliminates the traditional MapReduce multi-stage execution model and also improves the performance.

Spark also has a streaming [105] implementation. It is highly scalable and fault-tolerant. It uses a micro-batch technique which splits the input stream into a sequence of small chunks.
of data. It then processes these small chunks as a batch and sends its results to the next batch process.

Spark Streaming has two types of operators:

1. Transformation operator - maps a new DStream \[107\] from one or more parent streams. It has both a stateless (independent on each interval) and stateful (share data across intervals) property.

2. Output operator - an action operator that allows the program to write data to external systems (e.g., save or print a DStream).

Like MapReduce, map is a transformation operation that takes each dataset element and returns a new RDD/DStream. On the other hand, reduce is an action operation that aggregates all the elements of the RDD/DStream and returns the final result (reduceByKey is an exception that returns an RDD) as an RDD/DStream. Spark Streaming inherits all the transformation and action operations like map, reduce, groupBy, and join, etc. These operators are used to build a model for prediction. In the prediction unit, transformation and action operations are also performed to identify the anomaly from the input stream data. Machine learning algorithms are implemented here to analyze the data. After analysis, the framework reports the results to the resource manager for the virtual center. Finally, the resource manager will allocate the resources to the overused VMware machines dynamically.

### 3.2.3 Dynamic Resource Management

Figure 3.1 and 3.2 show the basic data flow for VMware dynamic resource management using Storm and Spark. The resource manager collects the VMware performance data periodically and sends it to the Spark cluster model. A cluster model is built in the training phase. After that, it predicts the incoming data. If it finds that any testing instance deviates from its built-in model, it identifies the testing instance as an anomaly. It informs the resource
manager when an anomaly is detected. Finally, the resource manager sends the data analysis information to the resource pool to dynamically allocate resources if necessary.
3.3 Implementation Details

3.3.1 Storm-based Anomaly Detection Framework

Our VMware cluster consists of 5 VMware ESXi [90] (VMware hypervisor server) 5.5 systems. Each of the systems has an Intel(R) Xeon(R) CPU E5-2695 v2 2.40 GHz processor, 64 GB DDR3 RAM, a four TB hard disk and dual NIC card. Each processor has 2 sockets and every socket has 12 cores. Therefore, there are 24 logical processors in total. Each ESXi system contains three virtual machines. Each of the virtual machines is configured with eight virtual CPUs, 16 GB DDR3 RAM and one TB Hard disk. As all the VMs are sharing the resources, performance may vary during runtime. Linux Centos v6.5 64 bit OS with the JDK/JRE v 1.7 is installed on each VM. Our real-time framework is divided into two components: a message broker and a data stream mining module.

Message Broker

We are continuously monitoring the performance data of each VMware ESXi server. We use VMware utility tool EXSTOP. ESXTOP continuously writes the performance data to a CSV (Comma Separated Values) file. We read those CSV files and transport the data through Message broker. We have chosen Kafka because it is stable and also compatible with Apache Spark. Kafka creates a dedicated queue for message transportation. It supports multiple sources and sinks on the same channel. It ships (in Figure 3.3) those performance data to Spark’s streaming framework. It provides guaranteed message delivery with proper ordering and we can form a Kafka cluster to handle large volumes of data.

Stream Data Mining Module

As in Figure 3.3 we have implemented a clustered network using Storm. Storm has input sources called Spout. It has also working nodes called Bolt and the whole network is called
Figure 3.3. Technical Approach for Storm-based Framework

Topology. Storm’s Spout receives the VMware performance data as tuples through Kafka. Kafka ensures guaranteed tuples delivery. It transports data to Storm’s Spout. We are monitoring only the VMware CPU performance data. Spout emits those tuples to Bolts. We have developed a Bolt called Clustering Bolt for building the cluster model and several Predicting Bolts to predict anomaly.

(i) Training

Initially, the system has no information about the input data. The Clustering Bolt first builds the training model. The Predicting Bolt will discard all tuples coming from Spout until generating a training model. As Storm has stateless property, all the clusters generated in training should be kept in memory. Furthermore, it does not store any tuples. Here, we have used a flat incremental clustering algorithm for the training model. We have not fixed the number of clusters. For each data, if it fits into any cluster then we update the cluster. Otherwise, create a new cluster with that point. So the cluster size is not fixed and it may vary according to the training data. Line 5-11
Algorithm 2 Training for adaptive clustering

1: procedure Training(trainingData)
2:     for each n in trainingData do
3:         BuildCluster(n)
4: 
5: procedure BuildCluster(n)
6:     totalCluster ← 0
7:     fitClusterIndex ← FitsInAnyCluster(n)
8:     if fitClusterIndex == −1 then
9:         CreateCluster()
10:        totalCluster ← totalCluster + 1.
11:     else
12:         UpdateClusterInfo(fitClusterIndex, n)
13: 
14: procedure FitsInAnyCluster(n)
15:     maxSimilarity ← MAX_NEG
16:     maxSimilarClusterIndex ← −1
17:     for each cluster k do
18:         distance ← EuclideanDistance(n, centroid(k))
19:         similarity ← \( \frac{1}{1 + distance} \)
20:         if similarity \( \geq \) TH_SIM then
21:             if similarity \( \geq \) maxSimilarity then
22:                 maxSimilarity ← similarity
23:                 maxSimilarClusterIndex ← k
24:     return maxSimilarClusterIndex
25: 
26: procedure UpdateClusterInfo(clusterIndex, dataPoint)
27:     cluster ← FindCluster(clusterIndex)
28:     CalculateWeightedAvgCenter(cluster, dataPoint)
29:     CalculateNewRadius(cluster)
Algorithm 3 Testing anomaly

1: procedure Testing(testingData)
2:     for each n in testingData do
3:         for each cluster k do
4:             distance ← EuclideanDistance(n, centroid(k))
5:             if distance ≤ Radius(k) then
6:                 n is benign
7:                 break
8:         if n doesn’t fall into any cluster then
9:             n is anomaly

in Algorithm 2 shows the building of the cluster. Moreover, we have used Euclidean Distance to measure the distance of two data points and our similarity function is as follows:

$$\text{Similarity} = \frac{1}{1 + \text{Distance}}$$  \hspace{1cm} (3.1)

The above similarity measure function has the lowest value 0 (when Distance = ∞) and the highest 1 (when Distance = 0). We also see that the most distanced points have lower similarity and the closest points have higher similarity. We also consider a threshold of 70% when we assign points to a cluster. Furthermore, we take a weighted average when we calculate the centroid of the cluster. Lines 13-24 in Algorithm 2 show how a point fits into a cluster and lines 26-29 show the weighted centroid modification of a cluster.

(ii) Testing

When the clusters are built, the Clustering Bolt sends the clusters information to all Predicting Bolts. The Predicting Bolt now performs anomaly detection. If any tuple does not fit to any cluster, it is considered as an anomaly, otherwise, it is benign. If the distance between the tuple and the centroid of the clusters is less than the cluster’s radius, then we consider the tuple falling inside the cluster. During prediction the
Clustering Bolt does nothing and it discards all the tuples coming from Spout. The details of the algorithm are given in line 1-8 in Algorithm 3.

Each source or executor in Storm can be configured with multiple threads to increase parallelism. For this system, we have configured five parallel Spouts and eight parallel Bolts. The reason behind using Storm is first handling the stream data and also minimizing the execution time. As we generate the cluster model in a single Bolt, we may not be able to reduce the model building time, but we can easily reduce the prediction time. Furthermore, \texttt{FitsINAnyCluster}(n) method takes \(O(n)\) to find the nearest cluster for a data point and practically we do not have a large number of clusters. So the execution time in Clustering Bolt heavily depends on the number of input data points that it handles during training.

**Update Training Model**

Stream data is dynamic in nature. We cannot always continue predicting with the initial training model. We should also periodically update our training model. At a fixed time interval, the Predicting Bolt signals the Clustering Bolts to update the model. When the Clustering Bolt receives the signal, it takes the incoming tuples to modify its clustered model. It uses its previously clustered model. It updates the cluster centroids or it may add new clusters.

**Scalability**

Our framework is more generic. We can build a message broker cluster using Kafka [40]. It is highly available and also scalable. This helps us to add more VMware sources to collect their performance data. Moreover, we can add more executors/workers in Storm’s Spout and Bolt to increase parallelism. In this way, we can accommodate these large volumes of data.
3.3.2 Spark-based Anomaly Detection Framework

Our anomaly detection framework uses VMware cluster setup. The cluster consists of 5 VMware ESXi [96] (VMware hypervisor server) 5.x systems. Each of them has Intel(R) Xeon(R) CPU E5-2695 v2 2.40GHz processor, 64 GB DDR3 RAM, 4 TB hard disk and dual NIC card. Each processor has 2 sockets and every socket has 12 cores. So there are 24 logical processors in total. Each of the ESXi systems contains 3 virtual machines. Each of the virtual machines is configured with 8 vCPU, 16 GB DDR3 RAM and 1 TB Hard disk. As all the VM’s are sharing the resources, performance may vary in runtime. We have installed Linux Centos v6.5 64 bit OS in each of the VM along with the JDK/JRE v1.7. We have designed a real-time outlier detection system on this distributed system using Apache Spark. We have installed Apache Spark version 1.0.0. We have also installed Apache Hadoop NextGen MapReduce (YARN) [102] with version Hadoop v2.2.0 and formed a cluster. Apache Spark uses this YARN cluster for a distributed file system.

Our framework has two components. The first one is Message Broker and the second one is Stream Data Mining module.

Message Broker

Our framework continuously monitors each VMware ESXi server [96] performance data. VMware ESXi has utility tool [EXSTOP]. [ESXTOP] continuously writes the performance data of its ESXi server in a CSV file. We read those CSV files and transport the data through message broker. Several message brokers (e.g., Apache Kafka [10], RabbitMQ [66], etc.) are available to integrate with Spark. We have chosen Kafka because it is stable and also compatible with Apache Spark. Kafka creates a dedicated queue for message transportation. It supports multiple sources and sinks on the same channel. It ships (in Figure 3.4) those performance data to Spark’s streaming framework. It provides guaranteed message delivery with proper ordering and we can form a Kafka cluster to handle large volumes of data.
Stream Data Mining Module

As in the Figure 3.4, we have implemented a clustered network using Apache Spark. The VMware performance data is transported through Kafka queue continuously as Stream. It is split into small micro batches called DStream. Several transformation and action operations are performed on these DStreams to build a training model and also predict anomalies. At each micro-batch processing, it stores the immutable dataset in memory and propagates the result to the next micro-batch processing. The framework builds the model from the benign stream. After that, it predicts the incoming stream data to identify anomalies.

i Training

The Spark framework builds the initial training model. Algorithm 4 shows that it has two components: map and reducedByKey. The map function (lines 3-8) takes each instance of the DStream called tuple and extracts data point from this tuple. Later, it finds the nearest cluster for that data point. If there is no nearest cluster, then it uses the data point as a new cluster’s centroid. It uses this nearest cluster’s centroid (line
**Algorithm 4** Training for adaptive clustering

1: `procedure Training(InputDStream)`
2: `trainingModel ← InputDStream.map(Tuple t){`
3: `n ← t.getDataPoint()`
4: `closestCluster ← findNearestCluster(n)`
5: `if closestCluster == Empty then`
6: `return Map(n, n)`
7: `else`
8: `return Map(closestCluster.Centroid, n)`
9: `}.reducedByKey(centroid, [dPoint1, dPoint2..])`
10: `index ← FitsInAnyCluster(centroid)`
11: `if index == −1 then`
12: `CreateCluster(centroid)`
13: `else`
14: `Update(index, [dPoint1, dPoint2..])`
15: `

**Algorithm 5** Testing anomaly

1: `procedure Testing(InputDStream)`
2: `testingModel ← InputDStream.map(Tuple t){`
3: `n ← t.getDataPoint()`
4: `for each cluster k do`
5: `distance ← EuclideanDistance(n, centroid(k))`
6: `if distance ≥ Radius(k) then`
7: `n is anomaly`
8: `else`
9: `n is benign`
10: `
11: `testingModel.print()`
8) or new centroid (line 6) as a key and the data point as a value. The reducedByKey acts as a reducer. It receives the centroid as a key and a list of data points as a value. If the centroid does not belong to any cluster, then it creates a new cluster with that centroid. Otherwise, it updates the cluster. Lines 9-15 describes it. All the data points with the same centroid will go to a reducer. Data points with different centroids will go to different reducers. In the \texttt{Update} function (line 16-19), we fetch the cluster from its index. After that, we update the cluster’s centroid by taking weighted average using the list of data instances.

As Spark has stateful property, all the clusters generated in training should be kept in memory. Furthermore, it does not store any data point. Here, we have used the flat incremental clustering algorithm for training model. We have not fixed the number of clusters. For each data point, if it fits any cluster, then we update the cluster. Otherwise, we have created a new cluster with that point. So, the number of clusters is not fixed and it may vary according to the training data. Moreover, we have used Euclidean Distance to measure the distance of two data points and our similarity function is defined before in equation 3.1.

\textbf{ii Testing}

During testing, it takes each instance of the DStream called tuple and extracts data point from this tuple. If any data point does not fit to any cluster, it is considered anomalous, otherwise benign. If the distance between the data point and the centroid of the clusters is less than the cluster’s radius, then we consider it falling inside the cluster. A detailed algorithm is given in line 1-11 in the algorithm 5.

\textbf{Update Training Model}

The stream data may evolve in time. So, we have to update our training model over time periodically. We can update the training model at a fixed time interval. The interval might
be set experimentally. While updating, we can use the previous clustered model and modify the cluster’s centroids for existing clusters and add new clusters for new instances. We can also save the whole cluster model information to HDFS for backup. In our experiment, we have not updated the training model periodically, but it can be implemented by periodically updating the cluster.

### 3.3.3 Scalability

We have designed a generic framework. We can build a message broker cluster using Kafka\cite{40}. It is highly available and scalable. It helps us to add more VMware sources to collect their performance data. Moreover, we can add more threads to Spark executors/workers to increase parallelism. Thus, we can accommodate these large volumes of data.

### 3.3.4 Generating Anomaly

VMs become CPU intensive when their CPU metrics increase. We have programmatically increased CPU load to simulate the anomalous behavior of data. For example, a program with infinite loop may increase CPU load to 100\%. Again, we have done some expensive database read/write operations which also increases the value of several counters in CPU metrics (e.g., CPU usage, processor time, core utility time, etc.).

### 3.3.5 Design Challenge

Several key challenges like integrating a stable message broker into the framework, handling message synchronization and message ordering, etc., came into the picture while we designed and developed our framework.
3.4 Experimental Result

3.4.1 Dataset

We have used two different datasets for our experiment. For the first dataset D1, we run several jobs on Spark cluster in VMs. We then monitor the real-time CPU performance metrics from all the VMs. We capture the stream and build the benign model. We then programmatically increase the CPU metrics for generating anomalous stream data.

For the second dataset D2, we have installed the Yahoo Cloud Serving Benchmark (YCSB)\cite{19} framework to generate extensible workload. We run the cloud service benchmark with different workloads and runtime properties continuously to capture stream data and build our benign model. The database system is a simple MySQL database with billions of rows inserted. Data is loaded in the load phase of the YCSB run. Later, transactions such as read and update are run on the database system to increase the workload. The client data is the performance data that captures the network, I/O, CPU and memory usages during the workload run. We have used millions of read/write operations on the database to generate anomalous stream data.

3.4.2 Result

We have presented the accuracy of our real-time framework in this section. Table 3.1 shows the Apache Spark cluster setup.

<table>
<thead>
<tr>
<th>Component</th>
<th>Number of parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker for emitting tuples</td>
<td>05</td>
</tr>
<tr>
<td>Worker for clustering</td>
<td>08</td>
</tr>
<tr>
<td>Worker for prediction</td>
<td>08</td>
</tr>
</tbody>
</table>

As data come continuously, we have taken a time window to collect the training instances for building the model. We run the flat incremental clustering algorithm on 10,000 data
instances. At the end, we get 63 clusters for dataset 1 (D1) and 134 for dataset 2 (D2). These clusters are generated dynamically. Table 3.2 shows the training data model.

<table>
<thead>
<tr>
<th>Number of data points</th>
<th>D1</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of clusters</td>
<td>63</td>
<td>134</td>
</tr>
</tbody>
</table>

Table 3.3 shows the testing of adaptive clustering using Spark.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>TPR</th>
<th>FNR</th>
<th>TNR</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>98.0%</td>
<td>2.00%</td>
<td>99.83%</td>
<td>0.17%</td>
</tr>
<tr>
<td>D2</td>
<td>99.20%</td>
<td>0.80%</td>
<td>99.06%</td>
<td>0.94%</td>
</tr>
</tbody>
</table>

Table 3.3 shows the accuracy of the framework. We have taken a time window for data prediction. Total 3,500 testing instances have been taken for both datasets. 3,000 of them are benign and 500 are anomalies. Our framework correctly predicts 2,995 benign data out of 3,000 and also identifies 490 anomaly data out of 500 for dataset 1 (D1). It also correctly predicts 2,972 benign data out of 3,000 and also identifies 496 anomaly data out of 500 for dataset 2 (D2). Our framework has higher accuracy in anomaly detection. In this table, TPR is the proportion of actual anomalies which are correctly identified and TNR is the proportion of actual benign which are correctly identified. Moreover, FNR is the proportion of actual anomalies which are misclassified as benign and FPR is the proportion of actual benign which are misclassified as anomalies.

3.4.3 Comparison with Storm

We have also implemented this framework using Storm with the same environment and also the same data.

Figure 3.5 shows the average process latency of a tuple for clustering during training for both Spark and Storm implementations. We have taken the average process latency of a
tuple after a certain number of training instances have been processed and plot them. The graph shows that the clustering takes almost 5.85 ms (first dataset D1) and 5.30 ms (second dataset D2) on average for Storm and almost 0.6 ms (D1) and 0.4 ms (D2) on average for Spark. From the result, we see that for dataset 1, Spark is almost 10 times faster than Storm and for dataset 2, Spark is almost 13 times faster than Storm.
Figure 3.6 shows the average process latency of a tuple for predicting anomalies during testing for both Spark and Storm implementations. We take the average process latency of a tuple after a certain number of training instances have been processed and plotted them. The graph shows that the prediction takes almost 0.78 ms (first dataset D1) and 0.55 ms (second dataset D2) on average for Storm (when the graph reaches saturation) and almost 0.038 ms (D1) and 0.03 ms (D2) on average for Spark. From the result, we see that for dataset 1, Spark is almost 20 times faster than Storm and for dataset 2, Spark is almost 18 times faster than Storm.

![Figure 3.7. Average tuple processing latency for different input VMs](image)

3.4.4 Experiment of Parallelism

We vary the number of input VMs when we collect data. From Figure 3.7, we see that the average process latency of a tuple during clustering does not increase although we increase the number of input VMs. For single VM, the latency is almost 0.61 ms and then it decreases to almost 0.52 ms for double and triple VMs. The average process latency during prediction is almost 0.04 ms for all three cases.

The limitation of this framework is using the appropriate threshold. We have experimentally selected the similarity threshold at 70% during clustering. Finally, as we do not
fix the number of clusters, there might be a possibility to generate many clusters with a single point. But they are not influential during testing because they have the only centroid without radius. Unless exact matching to these data points occurs, no testing data points will fall inside these clusters. We can do the post-pruning of these single point clusters.
Anomaly detection refers to the identification of an irregular or unusual pattern which deviates from what is standard, normal, or expected. Anomalies can be appeared individually (discussed in Chapter 3) or as a group. Detecting anomalies in fast, voluminous streams of data is a formidable challenge.

This chapter presents a novel, generic, real-time distributed anomaly detection framework for heterogeneous streaming data where anomalies appear as a group. We have developed a distributed statistical approach to build a model and later use it to detect an anomaly. As a case study, we investigate group anomaly detection for a VMware-based cloud data center, which maintains a large number of virtual machines (VMs). We have built our framework using Apache Spark to get higher throughput and lower data processing time on streaming data. We have developed a window-based statistical anomaly detection technique to detect anomalies that appear sporadically. We then relaxed this constraint with higher accuracy by implementing a cluster-based technique to detect sporadic and continuous anomalies. We conclude that our cluster-based technique outperforms other statistical techniques with higher accuracy and lower processing time.

We have identified some key challenges in our real-time anomaly detection framework: (i) Identifying a data point as anomalous may increase the false alarm when, for example, a sudden spike in any resource usage due to temporal hardware malfunctioning in a data center may not indicate its anomalous or over-usage condition. A simple extension can be detecting anomalies on windowed data instead of individual data points. As data comes in a stream, we set a time window to collect data from the stream periodically. This approach

---

requires setting a time period for the window and collecting data within that window. Later, a non-parametric histogram has been used to generate the distribution for the window. If a window’s distribution substantially deviates from the model of longer-term benign distributions, the window is flagged as an anomaly. (ii) Heterogeneous data co-exist in the data center. For example, each VM in the data center generates separate statistics for each of its resources (e.g., CPU, memory, I/O, network usages statistics). Their balanced usages exhibit the data center’s operational efficiency. When demands for various resources go high, their balanced usages will no longer exist. Therefore, we capture these balanced usage distributions. Later, we build a separate model of distributions for each resource from them. We have observed that there exists no relation among distributions of different resources. For example, higher CPU usage may not increase the memory usage & vice versa. So, we considered no relation among different resources when we build models. (iii) Synchronized data communication and efficient parallel design are the main focuses for designing distributed real-time anomaly detection framework [6, 13]. We use a message broker (Apache Kafka [40]) to transfer the data to the distributed framework without any loss of data. We set a fixed time interval to capture and process data from the message broker.

Considering the above challenges, we have made the following contributions:

- We have developed a novel, generic, window-based statistical real-time framework for heterogeneous stream data using Apache Spark [84] and Kafka [40].

- We have developed a novel cluster-based statistical framework for heterogeneous stream data using Apache Spark and Kafka. We have set a time window to capture data and generated a non-parametric distribution for each of the feature in data. We have built a model of benign distributions for each feature in training and used that to identify anomaly during testing. We will consider a window as an anomaly if any of its feature’s distribution fails to fit its corresponding model during testing.
We have implemented the distributed two-sample Chi-square test in Apache Spark where each distribution has an unequal number of data instances.

We have compared our cluster-based approach with our previous window-based approach [81] and also another online statistical method proposed by Chengwei et al. [97]. It uses a multinomial goodness of fit test where each window has a fixed number of data instances. We use Chi-square two sample test where the number of data instances varies. Moreover, in our scenario, anomalies may be distributed across multiple windows. Our cluster-based approach outperforms both approaches with higher TPR (true positive rate) because they assume that anomaly will be distributed within a window.

The rest of the chapter is organized as follows: Section 4.1 describes our statistical real-time anomaly detection frameworks in detail. Section 4.2 shows our case study of real-time anomaly detection on VMware performance data using Apache Spark with implementation details. Section 4.3 presents our experimental results with our framework’s accuracy.

### 4.1 Real-time Anomaly Detection Framework

In real-time anomaly detection, preprocessing of raw stream data serves as a precursor to machine learning or data mining algorithms. K-means is a popular unsupervised data mining technique for stream data analytics that builds clusters of data points from training data with the assumption that the clusters are benign. If a test data point falls into any one of the clusters, it is labeled as benign, otherwise, it is flagged as anomalous. In order to accommodate the evolution of stream data, the training model needs to be updated periodically.

Data center automation [95] (e.g., dynamic resource management) may require analyzing the performance data in real-time to identify anomalies. As the stream data come continuously and the volume of data is also huge, we require a scalable distributed framework. To
address the scalability issue, we can use a distributed solution like Hadoop or MapReduce. Hadoop runs on batch mode and cannot handle real-time data. As we are looking for real-time data analysis in a distributed framework, we have decided to use Apache Spark [84], which is fault-tolerant and supports distributed real-time computation system for processing fast, large streams of data.

Apache Spark [84] is an open-source distributed framework for data analytics. It avoids the I/O bottleneck of the conventional two-stage MapReduce programs. It provides the in-memory cluster computing that allows a user to load data into a cluster’s memory and query it efficiently. This increases its performance faster than Hadoop MapReduce [84].

4.1.1 Chi-square Test

Given two binned data sets, let $R_i$ be the number of items in bin $i$ for the first data set and $S_i$ be the number of items in bin $i$ for the second data set. The Chi-square statistic is:

$$
\chi^2 = \sum_i \left( \sqrt{S/R} \times R_i - \sqrt{R/S} \times S_i \right)^2 / (R_i + S_i)
$$

where $R \equiv \sum_i R_i$ and $S \equiv \sum_i S_i$.

It should be noted that the two data sets can be of different sizes. A threshold $T$ is computed against which the test statistic $\chi^2$ is compared. $T$ is usually set to that point in the Chi-squared cumulative distribution function (CDF) with $N_{bins}$ degrees of freedom (for data sets of unequal sizes) that corresponds to a confidence level of 0.95 or 0.99 [97]. If $\chi^2 < T$ the data sets follow the same distribution.

4.1.2 Statistical anomaly detection

A window $W$ can be defined as a data container which collects data $d_1$, $d_2$, ..., $d_n$ from a stream periodically at a fixed time interval $t$ from multiple VMs (virtual machines) in a
The length of a Window is the total number of data instances that it contains within that time interval.

A feature \( f \) is a property to describe a physical computing hardware or resource used in a data center. For example, CPU usage, memory usage, I/O block transfer, etc.

Clustering techniques aim to classify a single data point. In cases of stream data, it seems more appropriate to consider a point as part of a distribution and determine its characteristics (anomaly/benign) with reference to the remaining points in the distribution. Statistical approaches are a natural fit for this kind of scenario, since they offer ways to compare two distributions. Parametric statistical techniques assume prior knowledge of the data distribution and verify whether the test distribution conforms to the known model. Non-parametric techniques, on the other hand, make no assumptions on the probability distribution of data. Hence, non-parametric techniques are less restrictive than parametric ones and have wider applicability compared to their parametric counterparts [38].

\[
\textbf{Algorithm 6 Statistical Anomaly Detection}
\]

1: \textbf{procedure} \textsc{detectanomalies}(windows, TH, \( C_{TH} \))
2: \hspace{1em} \( w_i \leftarrow \text{dataOf } i^{th} \text{TimeInterval} \)
3: \hspace{1em} AddToWindows(windows, \( w_i \))
4: \hspace{1em} status \leftarrow \text{anomalous}
5: \hspace{1em} for \( j \leftarrow i - 1 \text{ to } i - N + 1 \) do
6: \hspace{2em} \( T_j \leftarrow \text{ComputeChi-SqrStatistic}(w_i, w_j) \)
7: \hspace{2em} \( T \leftarrow T \cup T_j \)
8: \hspace{2em} \( T_{min} = \arg\min_j \{ j \mid T_j \in T \} \)
9: \hspace{1em} if \( T_{min} < \text{TH} \) then
10: \hspace{2em} \( c_j \leftarrow c_j + 1 \)
11: \hspace{2em} if \( c_j > c_{TH} \) then
12: \hspace{3em} status \leftarrow \text{benign}

In Algorithm 6, we provide an outline of the anomaly detection process by a non-parametric statistical method. It periodically collects data from an incoming stream. Data collected at the \( i^{th} \) time interval will be stored in the \( i^{th} \) window \( w_i \). Each window has a score that represents the number of times this window matches with another window. \( w_i \) is
Figure 4.1. Statistical technique for anomaly detection

compared against past $N - 1$ windows using a Chi-square test (line 6); the test statistics from this test are stored in a list $T$. If the list is empty, window $w_i$ is an anomaly. Otherwise, we choose the window that has minimum test statistics $T_{min}$ (line 8). If the test statistic $T_{min}$ is below a pre-defined threshold, the score $c_j$ of the $j^{th}$ window $w_j$ is incremented. If $c_j$ exceeds a pre-defined threshold $C_{TH}$, $w_i$ is declared as benign. $w_i$ is flagged as anomalous if we do not find any window whose score exceeds $C_{TH}$ (line 9-12). In Figure 4.1, a sliding buffer always stores the recent $N$ windows. Each window has a distribution. When a window comes at time $t$, it is inserted as a most recent window $w_n$. It is matched with distributions of the past $n - 1$ windows. A Chi-square two sample test has been used to check if two distributions are same or not. It gives the Chi-square statistics. The lower statistic indicates the closest match. If the window $w_n$ matches window $w_j$, then its score $c_j$ is incremented. The score defines the number of window matches. This technique assumes that the stream data will not change abruptly. So, anomalies will appear rarely and will not come consecutively. If
a window has a high score, then it represents a benign window. Moreover, a new window that matches this high scored window will also be considered as a benign window. We set a threshold $C_{TH}$. If $c_j > C_{TH}$, then $w_n$ is benign, else it is anomaly. At time $t + 1$, a window $w_n$ comes and if it does not find any closest distribution from the past $n - 1$ windows, it is immediately termed as an anomaly.

In the real word, consecutive anomalous windows may come frequently. Moreover, because of using the past $N$ windows, some benign windows may be interpreted as anomalous, increasing the false positive rate. To overcome these, we have introduced a cluster-based statistical technique, where we have applied unsupervised clustering on benign distributions during training to build our model. Later, we use this model to detect anomaly while testing. We use a Chi-square two-sample test to compare two distributions. During testing, if the distribution of a window fails to match with any distribution in the training model, we will consider it an anomaly.

![Figure 4.2. Statistical real-time anomaly detection framework](image)

Figure 4.2. Statistical real-time anomaly detection framework
4.1.3 Statistical Stream Data Mining Module Implementation Using Spark

Figure 4.2 shows the statistical real-time framework for anomaly detection using Apache Spark. Each virtual machine in the network continuously sends its performance data through Kafka. The Spark framework receives the stream as small micro batches called a DStream. After that, it uses several transformation and action operations to build the statistical model. Here, we have focused more on a set of data points in a window rather than an individual data point. An anomalous window carries more information and represents a pattern or distribution of abnormal characteristics of data. After receiving a window, the framework generates a non-parametric distribution. It stores the most recent distribution. Later anomalies will be detected by matching the current window’s distribution with its stored distributions.

In algorithm 7, DetectAnomaly illustrates our distributed statistical procedure for anomaly detection using Spark. It describes a general procedure to find anomalies for each feature. In this case, we say a window is benign if it is not anomalous for all features; otherwise, it is anomalous.

i Generating Non-parametric Distribution

As we do not have any prior knowledge of data, we have used a bin-based histogram technique to derive this non-parametric distribution by using the following formula.

\[ \text{Bin}_{\text{current}} = \left\lceil \frac{d - d_{\text{min}}}{d_{\text{max}} - d_{\text{min}}} \times \text{totalBin} \right\rceil \]  \hspace{1cm} (4.2)

where \( d_{\text{min}} \) and \( d_{\text{max}} \) are the minimum and maximum value of the data \( d \). These can be calculated experimentally. We have used percentage of resource usage like CPU / memory usage. So, their minimum and maximum values are 0 and 100.

GETDISTRIBUTION in algorithm 8 describes how to generate distribution. Here, we have used Spark’s map and reduce functions. The map function (lines 5-8) takes an index \( i \) as
Algorithm 7 Anomaly detection

1: procedure DetectAnomaly
   Inputs
   InputDStream D (n data, l features)
   Window W
4: models ← {}            ▷ empty models collection
5: bin ← GetDistribution(InputDStream, totalBin)
6: ml,c ← createModel(bin, score = 1)
7: models.insert(ml,c)      ▷ insert current model to models collection
8: if models.size = 1 then
9: current first model is benign
10: else
11: chi ← MODELSWITHCHIVALUE(bin, models)
12: (ml,c, (ml,match,chi)) ← GETMATCHEDMODEL(chi, models, ml,c)
13: W ← benign
14: for j ← 1 to l do
15: if m.j,match.isEmpty() then
16: current model m.j,c is anomaly
17: W ← anomaly
18: else
19: if m.j,match.getScore() > Sc_TH then
20: current model m.j,c is benign
21: else
22: current model m.j,c is anomaly
23: W ← anomaly
24: return W

Algorithm 8 Generate non-parametric distribution

1: procedure GetDistribution
   Inputs
   InputDStream D (n data, l features)
2: totalBin
   Outputs
   List (bin, f)                ▷ f = frequency
5: Map (i, Di), where i = 1 to n
6: for j ← 1 to l do
7: bin ← \left[\frac{D_{ij} - D_{ij_{min}}}{D_{ij_{max}} - D_{ij_{min}}} \times totalBin\right], where k ∈ 1 .. totalBin
8: collect(bin, 1)
9: Reduce (bin, [f1, f2, ...]), where k = 1 to totalBin, j = 1 to l
10: return sum([f1, f2, ...])
key and each instance of the DStream $D_i$ as value. Later it uses equation 4.2 to generate appropriate bin for each feature and finally emits (bin, frequency) as output. The reduce function (lines 9-10) takes bin as key and frequency list as a value. It counts the total number of frequencies for each bin of the corresponding feature. Finally, it returns a list of bins with its frequency.

### Algorithm 9 Models with Chi-square value

1: procedure ModelsWithChiValue

Inputs

2: List (bin, $f$) $\triangleright f =$ frequency
3: Models $M_{l,n}$ ($l$ models, each has $n$ distributions)

Output

4: Chi − Models $M_{l,n-1}$ ($l$ models, each has $n-1$ chi values)
5: Map (bin$_{k,j}$, $f_{k,j}$), where $k = 1$ to totalBin, $j = 1$ to $l$
6: $m_{j,\text{current}} \leftarrow M_{j,n}$ $\triangleright$ current model for feature $j$
7: $RT \leftarrow m_{j,\text{current}}.\text{getTotalFreq}$
8: $R \leftarrow f_{k,j}$ $\triangleright$ frequency of bin$_{k,j}$

9: for $i \leftarrow 1$ to $n - 1$ do
10: $ST \leftarrow M_{j,i}.\text{getTotalFreq}$
11: $S \leftarrow M_{j,i}.\text{getBinFreq}(\text{bin}_{k,j})$
12: $\text{chi}_sqr \leftarrow \frac{\left(\sqrt{ST/RT} \times R - \sqrt{RT/ST} \times S\right)^2}{RT + ST}$
13: collect($M_{j,i}$, $\text{chi}_sqr$)

14: Reduce ($M_{j,i}$, [$c_1, c_2, ...$]), $\quad$ where $j = 1$ to $l$, $i = 1$ to $n-1$
15: return sum([$c_1, c_2, ...$]), $\triangleright c =$ chi-square value

ii Chi-square Test

The Chi-square Test is the ideal non-parametric technique to match two unknown distributions. Here, we are collecting data as a window. VMs send their data as a stream. Due to network delay or packet loss, some data might not reach the receiving end at their scheduled time. So, the window size of the Spark framework may vary. Therefore, we use Chi-square test for matching two distributions where the number of instances is not equal.
Figure 4.2 shows the matching of the current distribution with its recent $N - 1$ distributions. After generating the distribution, we store it as a model (line 5-6 in algorithm 7) for each feature. This is our current model and we score it as 1. We assume our first model as benign. If the current model is not the first model, then we compute Chi-square value for the rest of the models and list them (line 11 in algorithm 7). In algorithm 9 ModelsWithChiValue computes and lists the model for each feature with their Chi-square values. The map function (lines 5-13) takes bin and frequency as a key-value pair. It extracts each bin of a feature and its frequency from the tuple. Later, it computes the frequencies of that bin for that feature from all the past $N - 1$ models. After that, it calculates the partial Chi-square statistics for that bin of the corresponding model. Finally, it emits the (model, Chi-square value) as a tuple list. The reduce function (lines 14-15) receives model as a key and Chi-square value list as a list of values and sums up the Chi-square statistics for that model. At the end, we will get Chi-square statistics for all models.

Now, we have calculated all the feature model’s Chi-square statistics and we have to filter them by using a Chi-square threshold TH-CHI. GetMatchedModel (line 12 in algorithm 7) finds each feature’s model which has the lowest Chi-square value. Here, we can use any statistical tool like R [65] to find out this threshold. R has a Chi-squared test of comparing two distributions which assumes they have an equal number of data instances but in our case, we have an unequal number of data instances. qchisq(cf, df) in R gives the threshold, where $cf =$ confidence level and $df =$ degrees of freedom. Here, $df =$ Number of bins. In algorithm 10 GetMatchedModel, the map function (lines 5-8) takes a model and Chi-square value as a key-value pair. It also knows the current model. It finds all the possible closest models using chi-square threshold and emits the current model with minimum matching Chi-square value as a key-value pair. The reduce function (lines 7-8) receives a current model with a list of Chi-square values as a key-value
list and returns the model which has minimum Chi-square statistic. When we match our current model with past models, several models have lower Chi-square statistics than the threshold. So, we have to select the model which has the lowest statistics. In algorithm 11, MINMODEL (line 1-12) selects the model which has the lowest Chi-square statistics. If several models have the same lowest Chi-square statistics, then it selects the model which has the highest score. The score means how many times it matches a model. Initially, all models have a score = 1. If any past model is matched by the current model, then we increase the score of that past model.

iii Detecting Anomaly

The current window has multiple distributions of different features. Therefore, the window will be flagged as benign if all its distribution is benign, otherwise, it is an anomaly. Lines 13-24 in algorithm 7 describe this.

Figure 4.3. Cluster based statistical real-time anomaly detection framework
Algorithm 10 Find closest model

1: procedure GetMatchedModel

Inputs
2: \( Chi \) – Models \( M_{l,n-1}(l \models, each \ has \ n-1 \ chi \ values \ c) \)
3: CurrentModel \( m_{l,c}(l \ models) \)

Outputs
4: \( ModelsWithLowestChiValue (m_{l,min}, c_{l,min}) \), where \( j = 1 \ to \ l \)
5: Map \( (M_{j,i}, c_{j,i}) \), where \( j = 1 \ to \ l, i = 1 \ to \ n-1 \)

6: if \( c_{j,i} \leq TH - CHI_j \) then \( \triangleright \) Calculated in R for 95% confidence
7: collect \( (m_{j,c}, (M_{j,i}, c_{j,i})) \)
8: Reduce \( (m_{j,c}, [(M_{1,1}, c_{1,1}), ...]) \),
9: \( (m_{j,min}, c_{j,min}) \leftarrow MINMODEL([(M_{1,1}, c_{1,1}), ...]) \)
10: \( m_{j,min}.incrementScore() \)
11: return \( (m_{j,min}, c_{j,min}) \)

Algorithm 11 Model with minimum Chi-square value

1: procedure MinModel

Inputs
2: \( [(m_1, c_1), (m_2, c_2) ..] \) (List of models with Chi – square value)

Outputs
3: \( (m_{min}, c_{min}) \) (matched model with min with Chi – square value)
4: \( c_{min} \leftarrow \infty \)
5: for each \( (m, c) \) in \( [(m_1, c_1), (m_2, c_2) ..] \) do
6: \( sc \leftarrow m.getScore() \)
7: if \( c < c_{min} \) then
8: \( (c_{min}, m_{min}, sc_{high}) \leftarrow (c, m, sc) \)
9: else
10: if \( c = c_{min} \ and \ sc > sc_{high} \) then
11: \( (m_{min}, sc_{high}) \leftarrow (m, sc) \)
12: return \( (m_{min}, c_{min}) \)

4.1.4 Cluster-based Statistical Stream Data Mining Module Implementation Using Spark

In Figure [4.3], we show the cluster based statistical real-time framework for anomaly detection using Apache Spark. Each virtual machine in the network continuously sends its performance
data through Kafka. We cluster on the benign distributions and build a model, which will be used later to detect an anomaly.

Algorithm 12 Training
1: procedure Training for building a model of distributions
   Inputs
2:   InputDStream D (n data, l features)
3:   models_l ← {}
4:   bin_k,l ← GETDISTRIBUTION(InputDStream, totalBin)
5:   m_l,c ← createModel(bin_k,l, score = 1)
6:   chi − models_l ← MODELSWITHCHIVALUE(bin_k,j, models_l)
7:   (m_l,c, (m_l,match, chi)) ← GETMATCHEDMODEL(chi − models_l, m_l,c)
8:   if m_l,match.isEmpty() then
9:      models_l.insert(m_l,c)
10: else
11:      m_l,match.incrementScore()

Algorithm 13 Testing anomaly
1: procedure Testing
   Inputs
2:   InputDStream D (n data, l features)
3:   Window W
4:   models_l ← {}
5:   bin_k,j ← GETDISTRIBUTION(InputDStream, totalBin)
6:   m_new ← createModel(bin_k,l, score = 1)
7:   chi − models_l ← MODELSWITHCHIVALUE(bin_k,j, models_l)
8:   (m_new, (m_l,match, chi)) ← GETMATCHEDMODEL(chi − models_l, m_l,c)
9:   W ← benign
10: for j ← 1 to l do
11:    if m_l,match.isEmpty() then
12:       current model m_j,new is anomaly
13:       W ← anomaly
14:    else
15:       if m_l,match.getScore() > ScTH then
16:          current model m_j,new is benign
17:       else
18:          current model m_j,new is anomaly
19:          W ← anomaly
20: return W
In algorithm 12, we illustrate our distributed statistical procedure for anomaly detection using Spark. It uses a semi-supervised technique for training unknown distributions. We assume that all VMs are in stable state and their performance data do not contain anomalies in training. We have taken a time window to collect all the data and build non-parametric distribution for each feature. During clustering, we build a training model for each feature. If a distribution does not fit to any cluster, it starts with a new cluster. If it fits a cluster, then we increment the score of that cluster. The score describes the confidence level of the cluster. TRAINING illustrates the details.

During testing, we take a time window to collect data and generate the unknown distribution for each feature. Later, if the distribution does not fit to any clusters of the relevant feature, then it is called anomalous, otherwise, it is benign. We can say a window is benign if it is not anomalous for that feature; otherwise, it is called anomalous. In algorithm 13, TESTING illustrates the details.

4.2 Case Study: Real-Time Anomaly Detection In VMware-based Data Center

We have developed a generic framework using a statistical technique. We can implement this framework in a variety of fields like in social media, real-time traffic analysis, intrusion detection in sensor data, and so on. In this case, an anomaly corresponds to over-utilization of the resources in a data center. More precisely, we can say that an anomaly corresponds to an abnormal distribution of its resources usage. For example, when CPU or memory intensive applications are running, the overall CPU or memory usage goes high and these generate abnormal distributions, which can be considered as anomalous.

In this section, we will describe data centers and dynamic resource scheduling and later our framework implementation details.
4.2.1 Data Center

A data center is the storehouse of data. It provides computer systems and associated components, such as telecommunications and storage systems. It offers dynamic resource management software to manage the storage infrastructure and also provides cloud facilities with the help of VMware, OpenStack, Microsoft, etc.

4.2.2 Dynamic Resource Scheduler

In Figure 4.4, we have shown the data flow for VMware dynamic resource management. The resource manager periodically reads the VMware performance data and sends it to the Spark cluster model to analyze it. The resource manager then sends the data analysis information to the resource pool to dynamically allocate resources if necessary.
4.2.3 Implementation Details

VMware Cluster Setup

Our VMware cluster consists of 5 VMware ESXi systems (VMware hypervisor server) 5.5 systems. Each of the systems has Intel(R) Xeon(R) CPU E5-2695 v2 2.40GHz processor, 64 GB DDR3 RAM, 4 TB hard disk and dual NIC card. Each processor has 2 sockets and every socket has 12 cores. So there are 24 logical processors in total. All of the ESXi systems contain 3 virtual machines. Each of the virtual machines is configured with 8 vCPU, 16 GB DDR3 RAM and 1 TB Hard disk. As all the VMs are sharing the resources, performance may vary in runtime. We have installed Linux Centos v6.5 64 bit OS in each of the VM along with the JDK/JRE v1.7. We have designed a real-time outlier detection system on this distributed system using Apache Spark. We have installed Apache Spark version 1.0.0. We have also installed Apache Hadoop NextGen MapReduce (YARN) with version Hadoop v2.2.0 and formed a cluster. We ran multiple MapReduce jobs on Hadoop cluster to put extra load so that we can monitor the performance data of the VMware-based cluster.

VMware Performance Stream Data

It is imperative for Data Center operations team to be able to track resource utilization of various hardware and software in order to maintain high quality of service and client satisfaction. The performance counters generated by the data center operation management system needs to be of sufficient granularity to facilitate the detection of anomalies.

In this experiment, we have used a percentage of CPU usage and percentage of memory usage to build our statistical model. Our framework can accommodate further attributes as well. We resorted to top, a Linux tool which reports CPU related statistics, vmstat for memory related statistics and we integrated their output with that of the vSphere Guest SDK using Kafka API.
Message Broker

We are continuously monitoring each VMware performance data using the vSphere Guest SDK \[93\]. We have also integrated top \[58\] to get the current percentage of CPU usage and vmstat \[98\] to get the current percentage of memory usage. We have chosen Kafka because it is stable and also compatible with Apache Spark. Kafka creates a dedicated queue for message transportation. It supports multiple sources and sinks on the same channel. It ships (in Figure 4.2) those performance data to Spark’s streaming framework. Kafka ensures that the messages sent by a producer to a particular topic partition are delivered to the consumer in the order they are sent. In addition to high-speed, large-scale data processing, Kafka clusters offer safeguards against message loss due to server failure. Kafka is a natural fit for our real-time anomaly detection initiative where it serves as an intermediary between multiple VMware virtual machines and the Spark cluster, transferring multi-source stream data from the VMs to Spark.

4.2.4 Building Statistical Training Model and Prediction

Our framework captures CPU and memory data periodically and generates distributions for each of them. It then clusters the distributions separately for both CPU and memory usages during clustering. While testing, it again generates distributions for both CPU and memory usages. If any of the distribution does not fit into the cluster of distributions, then we consider them as an anomaly. More details are described in Section 4.1.

4.2.5 Scalability

Our framework can easily adapt to an increasing number of VMware virtual machines. For each new VM, we simply plug in one more data collection instance to Kafka \[40\]. Furthermore, we can increase computational parallelism by adding more threads to Spark executors/workers.
4.3 Experimental Results

4.3.1 Dataset

We continuously monitor 12 VMs’ performance data. Each VMware sends data to Kafka server at twelve-second intervals. Apache Spark periodically collects stream data from Kafka server at two minute intervals. The length of the window is not fixed here. The number of data points in each window varies from 95-105. In this experiment, we have trained our model with clustering by 200 windows. So, 200 x 100 (average) = 20,000 data points have been considered during training. We collect 400 windows during testing both for our cluster-based statistical method and window-based method. So, 400 x 100 (average) = 40,000 data points have been considered during testing.

We have compared our result with a non-distributed, online statistical approach [97]. For this, we dump our real-time testing data to a CSV file. It uses a fixed window and it uses multinomial goodness-of-fit [20] for statistical analysis while we are using a Chi-square two-sided test.

In this experiment, we consider any VM resource hungry if its usage goes beyond 95%. For example, if a VM has more than 95% CPU usage, then it is called CPU hungry.

4.3.2 Results

Table 4.1 shows the training statistics. We use 200 windows for training and each window has 100 data points on an average. We have found seven clusters for CPU usage and five clusters for memory usage.

<table>
<thead>
<tr>
<th>Number of Windows</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of data points</td>
<td>20,000</td>
</tr>
<tr>
<td>Number of clusters</td>
<td>7(CPU), 5 (Memory)</td>
</tr>
</tbody>
</table>
Next, we have used our data set for both non-distributed and distributed experiments during testing. The online distributed method is used to detect anomaly for a cluster-based statistical model and window-based model. The non-distributed method is used for online multinomial goodness-of-fit based model. Table 4.2 shows the details.

Table 4.2. Testing of cluster based model

<table>
<thead>
<tr>
<th></th>
<th>Distributed</th>
<th>Non-distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Windows</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>Number of data points</td>
<td>40,000</td>
<td>40,000</td>
</tr>
</tbody>
</table>

Table 4.3. Accuracy of cluster based model

<table>
<thead>
<tr>
<th>Data set</th>
<th>TPR</th>
<th>FNR</th>
<th>TNR</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multinomial goodness-of-fit based model</td>
<td>20.00%</td>
<td>80.00%</td>
<td>82.16%</td>
<td>17.84%</td>
</tr>
<tr>
<td>Window-based model</td>
<td>60.00%</td>
<td>40.00%</td>
<td>99.80%</td>
<td>0.20%</td>
</tr>
<tr>
<td>Cluster-based model</td>
<td>96.00%</td>
<td>3.00%</td>
<td>95.67%</td>
<td>4.32%</td>
</tr>
</tbody>
</table>

Our framework has a higher accuracy to identify the anomaly. In table 4.3, TPR is the proportion of actual anomaly which is correctly identified and TNR is the proportion of actual benign which are correctly identified. Moreover, FNR is the proportion of actual anomaly which is misclassified as benign and FPR is the proportion of actual benign which are misclassified as an anomaly. We take 20 bins for building histogram/distribution and 400 windows out of which 30 are anomaly. We calculate the Chi-square threshold \( TH_{chi} = 31.41 \). We have used the function \( qchisq(0.95, df = 20) = 31.41 \) in R, where confidence level = 95% and degrees of freedom = number of bins. Here, we assume that an anomaly can be spread across multiple windows. During testing, we found that our framework can successfully detect most of the injected anomalies. Only 1 of them is misclassified and the remaining 29 are correctly classified. Moreover, it also detects 354 benign windows as benign and misclassifies only 16 as an anomaly.

On the other hand, our window-based statistical model correctly identifies 18 anomaly out of 30 although it correctly classifies 369 benign out of 370. The reason for its low
accuracy is that it assumes anomaly will come abruptly and spread within a window and if it comes across multiple windows, then it identifies the first one but misclassifies the rest of the anomalous windows. The online Multinomial goodness of fit test based approach has low TPR rate. We take only 400 windows and 30 of them are anomalous. It detects only 06 of them and misclassifies 24. Moreover, it detects 304 benign windows as benign and misclassifies 66 anomalies. It assumes equal window length which may increase some false alarm.

Figure 4.5. Average window processing latency during training

Figure 4.5 shows the average execution time per window for 200 windows while training. It has a consistent execution time (on average 1.75ms).

Figure 4.6 shows the comparison of execution time per window of three approaches during testing. We plot the average execution time for window against the total number of windows. Here, we can see that our cluster-based model and Chi-square-based technique have a lower execution time. It varies from 1.22-2.28ms for cluster-based model and 1.33-2.45ms for window-based model. The baseline approach has average execution time between 3.08-1.49 ms. Initially, the Multinomial test based approach has higher execution time because it operates in batch mode. It reads the whole file into memory before doing the computation. So, it has higher I/O time. Therefore, it will affect the average execution time of a window when we have billions of data.
Figure 4.6. Comparing average window processing latency during testing
CHAPTER 5
SPARK-BASED POLITICAL EVENT CODING

Modern social and political conflicts typically emerge in geographically constrained regions before propagating through the larger international system. Such propagation occurs through both spatial proximity and complex inter-dependencies among actors and related phenomena. For instance, it is not uncommon for emerging threats to include elements of political conflict, criminal violence, and disease that are each geographically and causally interconnected. No social science dataset currently exists that provides accurate structured data on political and social events around the globe, with historical coverage, drawn from multiple language sources, and is freely available within hours of the events occurring.

Political event data are records of interactions among political actors using common codes for actors and actions, allowing for aggregate analysis of political behaviors. These include both material and verbal interactions between political entities, such as bombings, protests, or diplomatic consultations. These data are collected at the event level, the smallest aggregation possible, allowing for both low-level and aggregate analysis of political behaviors. As such, they enable us to quantify the propagation of an event or a series of events through space and time, while also capturing the complex interactions of the actors involved.

Human coding was the standard method to generate event data before the 1990s, and remains the gold standard for validating machine-coded output. Unfortunately, human capabilities have their limits as researchers cannot be counted on to do this task with sufficient speed and accuracy [51, 50]. Humans are capable of providing high-quality gold standard cases for reference purposes, but that is extremely laborious and cross-validated gold standard cases can only be generated at a rate of three to six events per hour, which cannot be applied to near-real time production-level event generation involving millions of records.

Human coders who are not working in a gold-standard environment - that is, they are simply trying to generate event data as efficiently as possible - appear to have roughly the same accuracy level as machine-coding systems (70% to 80%) and in some documented instances are far worse (25% and 40% is documented) [22]. Thus, for event data with daily updates, machine coding is the only option.

Developments in open source Natural Language Processing (NLP) software such as Stanford CoreNLP [49], PETRARCH [59] allow for more accurate content extraction from each sentence and support text analysis in multiple languages, facilitating for the first time the analysis of foreign language texts without requiring time-intensive word-by-word translation of each news story. These developments add computational complexities and hence increase the overall processing time of standalone applications. Due to this, standalone applications are not suitable when processing a huge corpus of data since they suffer from scalability issues. There is, hence, a demand for scalable frameworks capable of handling Big Data with low latency. Popular Big Data frameworks, e.g., Hadoop [33], MapReduce [21], HBase [37], Mahout [48], Google Bigtable [16], MongoDB [53], etc., are highly scalable and more geared towards batch processing. There are some frameworks that have stream processing capabilities like Apache Storm [86], Apache S4 [73] and Apache Spark [84]. Among these, Spark performs best for both batch and stream processing [84].

To address these scalability issues in political event coding using existing applications, we have come up with a distributed framework using Apache Spark which comprises of two workflows. The first workflow extracts all the metadata information from the documents using Stanford CoreNLP and stores them in MongoDB. The reason for storing all the metadata as-is in the MongoDB is that even though PETRARCH needs only the sentences and their parse trees, the extra metadata information can be used to dynamically add actors to the CAMEO dictionary. The second workflow uses the extracted metadata information to generate a political event code data using PETRARCH. By running these two workflows...
inside Apache Spark worker jobs, we are able to achieve near real-time political event coding for large corpus of data.

In this chapter, we have the following contributions:

- We develop a novel distributed framework using Apache Spark, MongoDB, Stanford CoreNLP and PETRARCH.
- We show a distributed workflow that extracts all the metadata (parse tree, tokens, lemma, etc.) from the news source, stores it to MongoDB, and later generates events.
- We compare the processing time of distributed versus non-distributed frameworks with a set of experiments.

The rest of the chapter is organized as follows: Section 5.1 gives a background on the various open source tools and concepts used in this chapter. Section 5.2 talks about the non-distributed version of political event coding. Section 5.3 explains how we have implemented the political event coding in a distributed manner using Apache Spark. Section 5.4 shows the experimental result.

5.1 Background

The following concepts are pertinent to our framework:

5.1.1 Apache Spark

Apache Spark is an open-source distributed framework for data analytics. It avoids the I/O bottleneck of the conventional two-stage MapReduce programs. It provides the in-memory cluster computing that allows a user to load data into a cluster’s memory and query it efficiently. This increases its performance faster than Hadoop MapReduce.
5.1.2 MongoDB

MongoDB [53] is an open source NoSQL document database. MongoDB stores data in the form of a BSON (Binary JSON-like) document. This makes its integration of data in certain types of applications easier and faster. MongoDB has a key with a complex data structure called as a document. A set of documents forms a collection.

5.1.3 Stanford CoreNLP

Stanford CoreNLP [49] is a useful framework for annotating text. Stanford CoreNLP includes tools like part-of-speech (POS) tagger, named entity recognizer (NER), parser, co-reference resolution system, sentiment analysis, and bootstrapped pattern learning tools. It supports models for other languages. It has natural language analysis tool embedded in it and takes the raw text as input and gives the tokens, Parts-of-Speech, whether they are names of companies, people, etc., normalized dates, times, and numeric quantities, and marks up the structure of sentences in terms of phrases and word dependencies, indicates which noun phrases refer to the same entities, indicates the sentiment, etc. It has flexible and extensible interface and helps to ease the application of linguistic analysis tools from texts. It is also simple and easy to use.

CoreNLP is formed by two classes: Annotation and Annotator. Annotations are the data structures that hold the results of Annotators. Annotations are maps, e.g., the parse, the part-of-speech tags, or named entity tags. Annotators are functions and operate over Annotations, e.g., tokenize, parse, or NER tag sentences. Both Annotators and Annotations are glued by Annotation Pipelines to create generic Annotators.

5.1.4 TABARI

Textual Analysis By Augmented Replacement Instructions (TABARI) [87] – is an open-source program. It is a successor to KEDS (Kansas Event Data System) [29]. TABARI
extracts event data from the stream of text using sparse pattern recognition techniques [27]. It generates event data as a semi-formalized structure like (date, source, target, event). It uses shallow parsing procedures which identify the constituents of sentences. However, it does not specify the internal semantics and their role in the sentence like Stanford CoreNLP.

5.1.5 PETRARCH

PETRARCH (A Python Engine for Text Resolution And Related Coding Hierarchy) [59] is a natural language and processing tool to produce machine-coding events data. It takes fully-parsed text summaries in Penn Tree format from Stanford CoreNLP and generates event like ‘who-did-what-to-whom’ format. PETRARCH uses CoreNLP to parse sentences and generate the events. It is a successor to TABARI and handles the noun/verb/adjective disambiguation that accounts for much of the size of the TABARI dictionaries. It uses synonym sets from WordNet [99] to identify actors even if they are not in the dictionaries.

5.2 Political Event Coding

Advances in NLP improve the fidelity of machine-coded events and can be done in near real time with high accuracy. Coding open-source news articles into structured text, analysts can create rich baselines and forecasts of fast-moving political events. Developments in Stanford’s CoreNLP software allows for more accurate content extraction from each sentence and supports text analysis in multiple languages, facilitating the analysis of foreign language texts, and also the geolocation of events. The language extensions flow from those coded in Stanford CoreNLP. While PETRARCH was developed using English language texts, the coding engine relies exclusively on the Penn TreeBank tags that are produced in the CoreNLP markup, and these are standardized across languages.

The existing event coding software, PETRARCH, requires hand-crafted actor dictionaries for coding the subject and objects of the actions or statements from news reports. Man-
ually creating these dictionaries is a laborious task. We, therefore, need to automatically expand actor dictionaries using NLP techniques. Our goal for automatic expansion of actor dictionaries takes as input (1) a set of existing actor dictionaries that are to be expanded, (2) the event dictionaries employed by PETRARCH, and (3) a large raw corpus containing the documents of interest. From the event dictionaries, we first extract all the event words of interest. Then, we identify all occurrences of these event words from the raw corpus. After that, we use the Stanford CoreNLP tool to identify the subject and object of each of these event mentions. These noun phrases are the actors that we will use to expand the actor dictionaries. To increase coverage, we, therefore, propose to train a sequence model for extracting additional actors from the corpus. Modifying PETRARCH to identify new actors should be primarily an issue of dictionary development. Stanford CoreNLP (and thus PETRARCH) can employ fully parsed sentences with other metadata information (NER, lemma, etc.) to build this dynamic dictionary.

Figure 5.1 shows the political event coding flow. It starts with collecting the political news stories. Open source web scraper (e.g., Goose [31]) allows collecting arbitrary news pages. Instead of collecting arbitrary news pages, a whitelist of RSS (Rich Site Summary) [72] feeds is utilized for gathering relevant news stories.

The news stories are in raw format and cannot be fed into a standard parser. It requires filtering and preprocessing to convert into structured data. We make use of the Gigaword dataset [30] which is a collection of structured news data. It is SGML (Standard Generalized Markup Language) format and has a collection of news stories from multiple source languages.

The next crucial part is to extract metadata from the structured news stories. Stanford CoreNLP allows us to parse each sentence and collect metadata accurately. It gives us Part-of-Speech, tokens, lemma (a canonical form of a set of words), NER (Named-entity Recognition), parse tree, dependencies, co-reference, sentiment, and so on. These metadata are extremely important to improve dictionaries.
Figure 5.1. Political event coding

In the following example, we have shown how to generate metadata using CoreNLP. Figure 5.2 visualizes the Part-of-Speech tagging and NER of the first sentence of the paragraph. Figure 5.3 has the dependency between them. Figure 5.4 shows the co-reference of the four sentences of the paragraph. Besides that, we also show the token, lemma, parse tree (Penn TreeBank format) and sentiment of the first sentence of the paragraph.

The event data generated from PETRARCH is of the form ‘who-did-what-to-whom’. It outputs the Source, the Target and a CAMEO code [14] for the political event along with other information like the date of an event, news source, etc.

“Officials said that, in response to the attacks in Paris, the administration was seeking renewed global commitment to that intensified military action, and to a negotiated settlement of Syria’s civil war.
France’s retaliation came as Obama held talks with allied leaders and with Russian President Vladimir Putin at the summit being held in this Turkish Mediterranean resort city.

Obama vowed again on Sunday to help France hunt down the perpetrators of the attacks.”

Figure 5.2. Part-of-Speech and NER

Figure 5.3. Basic Dependency

"Officials said that, in response to the attacks in
Paris, the administration was seeking renewed global commitment to that intensified military action, and to a negotiated settlement of Syria’s civil war.”

Token

Officials, said, that, in, response, to, the, attacks, in, Paris, the, administration, was, seeking, renewed, global, commitment, to, that, intensified, military, action, and, to, a, negotiated, settlement, of, Syria,’s, civil, war,.

Lemma

official, say, that, in, response, to, the, attack, in, Paris, the, administration, be,
seek, renew, global, commitment, to, that,
intensify, military, action, and, to, a,
negotiate, settlement, of, Syria, 's, civil,
war,.

Parse

(ROOT
  (S
    (NP (NNS Officials))
    (VP (VBD said)
      (SBAR (IN that)
        (S (, ,)
          (PP (IN in)
            (NP
              (NP (NN response))
              (PP (TO to)
                (NP
                  (NP (DT the) (NNS attacks))
                  (PP (IN in)
                    (NP (NNP Paris)))))))))
    (, ,)
    (NP (DT the) (NN administration))
    (VP (VBD was)
      (VP (VBG seeking)
        (NP
          (NP (VBN renewed) (JJ global) (NN commitment))
          (PP
            (PP (TO to)
              (NP(DT that)(JJ intensified)(JJ military)(NN action)))
            (, ,)
          )
        )
      )
    )
  )
)
MongoDB stores all the metadata and later they are fed as input to PETRARCH to generate events. MongoDB is a NoSQL database and stores all the data in BSON (Binary JSON) format. PETRARCH uses CAMEO dictionary to identify political events and discards other events from the news data. PETRARCH gives the output as ‘who-did-what-to-whom’ format and it is what we refer to as event data. More specifically, event data consists of three
components, \{\text{SOURCE\_ACTOR}, \text{ACTION\_TYPE}, \text{TARGET\_ACTOR}\} as well as general attributes \{\text{DATE\_TIME}, \text{LOCATION}\}.

A sample DB record is given below.

MongoDB record

```
{
    type: story,
    doc_id: WP\_ENG\_20151115.0017R,
    head_line: France launches..,
    date_line: ANTALYA, Turkey, Nov 15
    sentences: [
        {
            sentence_id: 1,
            sentence: Officials..., 
            parse_sentence: ..,
            dependency_tree: ..,
            token: ..,
            lemma: ..,
            ner: ..,
            corref: ..,
            sentiment: ..
        }
    ]
}
```

5.3 Spark-based Political Event Coding

Stanford CoreNLP and PETRARCH are used to code political event. These tools perform accurately but suffer from slow processing time when running on huge news corpus. A signif-
icant bottleneck comes from the Stanford CoreNLP because each annotator needs sufficient amount of time to process a document. Adding more annotators makes it much slower. As a result, a single document with a couple of sentences requires more than a few minutes, so processing a large number of news stories in real time is a challenge. Also, PETRARCH is slower than its successor TABARI [76]. Considering this, we have no choice but to develop a distributed framework that will integrate both CoreNLP and PETRARCH in a parallel manner and perform the event coding in close to real time.

Motivated by these ideas, we have designed a distributed framework with Apache Spark. Spark is our obvious choice over existing distributed frameworks like MapReduce [33], Storm [86], etc., because of its in-memory computation and lower execution time [105, 82]. In our two-part framework, we show a workflow for extracting metadata from news stories using CoreNLP and also coding event from them using PETRARCH. First, it has a distributed
program that executes CoreNLP in each Spark’s worker nodes in parallel to extract metadata from news story and stores it to MongoDB. Second, it runs another distributed program that executes PETRARCH in parallel inside each worker node to generate political events from data stored in MongoDB. Figure 5.5 describes our Spark-based political event coding framework. We will discuss it thoroughly in the following subsections.

5.3.1 News Story Collection and Preprocess

News stories are collected using web scrappers, filtered, and stored in structured format (SGML). Each SGML file consists of multiple thousands of news documents. For example, we have following sample file in Gigaword dataset.

```xml
<DOC id="WP_ENG_20151115.0017R" type="story">
  <HEADLINE>France launches..</HEADLINE>
  <DATELINE>ANTALYA, Turkey, Nov 15</DATELINE>
  <TEXT>
    <P>Officials said that, in response to the attacks in Paris, the administration was seeking renewed global commitment to that intensified military action, and to a negotiated settlement of Syria’s civil war.
    </P>
    <P>...
    </P>
  </TEXT>
</DOC>
```

```xml
<DOC>
  ...
</DOC>
```
These documents require further processing before feeding them into our framework, after which we store each file in HDFS (Hadoop Distributed File System) where each line of the file contains a document.

5.3.2 Metadata Extraction

We are using CoreNLP to parse sentences and generate all the metadata for a document. After that, we store the extracted metadata in MongoDB. We have run the CoreNLP parsing inside each worker in the Spark job. Algorithm 14, \textsc{ExtractMetaData}, describes the procedure for metadata extraction. We read all the files from HDFS and store them in RDD (line 2). The RDD now contains all the documents. We apply the mapping on RDD. The map function (line 3-5) is run on worker node in parallel. Inside map, we process each document. For each document, we parse the sentences with the required annotator using CoreNLP and obtain a list of metadata information (line 4). Once metadata are obtained, a MongoDB object for that document is created using that list (line 5). So, the map function gives us the list of MongoDB objects. Each of them is a JSON-like object that contains all the metadata for that document. After getting the object list, \textsc{saveRDDtoMongoDB} stores all DB objects to MongoDB (line 6). We have used the Hadoop-based MongoDB connector API [17] to make the Spark cluster communicate with MongoDB. It has a function (\textsc{saveAsNewAPIHadoopFile}) that stores data to MongoDB in parallel like reducer in MapReduce.

5.3.3 Event Coding

MongoDB stores all the information to code events. PETRARCH generates the political event codes. For each document in the MongoDB, it is encoded by passing it as input to
Algorithm 14 Extract Metadata

1: procedure ExtractMetaData
2: $hrdd \leftarrow \text{hadoopRDD.readFiles}$
3: $hrdd.map$
4: $list \leftarrow \text{Generate metadata using CoreNLP}$
5: $\text{MongoDBObject.create(list)}$
6: $}.saveRDDtoMongoDB()$

PETRARCH, which is run inside a Spark worker and political event codes are generated. Algorithm 15 CodingEvent describes the overall process. We use the Hadoop MongoDB connector to read documents from MongoDB (line 2). These documents are read in parallel and stored in RDD. We apply the mapping on RDD. The map functions are run in parallel in each worker where we use PETRARCH to code the events. Inside the map function (line 3-4), we pass the documents with their sentences and parse trees to the PETRARCH and generate the political event codes. The generated political events are stored (like reducer in MapReduce) in a file in HDFS (line 5).

Algorithm 15 Coding Event

1: procedure CodingEvent
2: $mongordd \leftarrow \text{mongoRDD.readFromMongoDB}$
3: $mongordd.map$
4: $events \leftarrow \text{Generate events using PETRARCH}$
5: $}.saveRDDtoHDFS()$

5.3.4 Design Challenges

Parsing with CoreNLP is a time consuming and memory intensive job. It occupies almost the entire memory of a Spark worker node when processing a large corpus of text. As a result, the Spark jobs are frequently halted when the Spark’s heap memory exceeds its allocated limit. An immediate mitigation would be to increase the worker node heap memory when this happens.
Table 5.1. News agencies

<table>
<thead>
<tr>
<th>News agency</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agence France-Presse, English Service</td>
<td>AFP</td>
</tr>
<tr>
<td>Associated Press World stream, English Service</td>
<td>APW</td>
</tr>
<tr>
<td>Central News Agency of Taiwan, English Service</td>
<td>CNA</td>
</tr>
<tr>
<td>Los Angeles Times/Washington Post News wire Service</td>
<td>LTW</td>
</tr>
<tr>
<td>New York Times Newswire Service</td>
<td>NYT</td>
</tr>
<tr>
<td>Xinhua News Agency, English Service</td>
<td>XIN</td>
</tr>
</tbody>
</table>

5.4 Experiments

5.4.1 Dataset

We have chosen English Gigaword dataset [30] for our experiment. It has news stories from six different news agencies which is shown in Table 5.1.

It has a total of 722 files with more than 7 million documents. Each document contains document id, headline, dateline and news type and news story. The news comes from multiple news types like stories, multi (summary of a story), advis (advisory), and other. In our experiment, we select documents from AFP and CNA and consider only story type news articles.

5.4.2 Cluster Setup

Our VMware cluster consists of 5 VMware ESXi [96] (VMware hypervisor server) 5.5 systems. Each of the systems has Intel(R) Xeon(R) CPU E5-2695 v2 2.40GHz processor, 64 GB DDR3 RAM, 4 TB hard disk and dual NIC card. Each processor has 2 sockets and every socket has 12 cores. So there are 24 logical processors in total. All of the ESXi systems contain 3 virtual machines. Each of the virtual machines is configured with 8 vCPU, 16 GB DDR3 RAM and 1 TB Hard disk. As all the VM’s are sharing the resources, performance may vary in runtime. We have installed Linux Centos v6.5 64 bit OS in each of the VM along with the JDK/JRE v1.8. We have installed Apache Spark version 1.5.0. We have also installed
Apache Hadoop NextGen MapReduce (YARN) with version Hadoop v2.6.0 and formed a cluster. Table 5.2 has the summary of the Spark cluster.

<table>
<thead>
<tr>
<th>Total Master node</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total slave node</td>
<td>12</td>
</tr>
<tr>
<td>vCPU (core) in each node</td>
<td>8</td>
</tr>
<tr>
<td>Memory in each node</td>
<td>16GB</td>
</tr>
<tr>
<td>Storage in each node</td>
<td>1TB</td>
</tr>
</tbody>
</table>

### 5.4.3 Result

As parsing with CoreNLP is time-consuming, it is not feasible to conduct a non-distributed experiment over all the documents. However, it is also important to compare the execution time of non-distributed and distributed frameworks. That is why we have two distinct sets of experiments. First, we are comparing the two frameworks. We have chosen a document set which has 12,484 documents of story type news for this experiment. Second, we run our distributed framework for CNA and AFP documents and try to compare with the non-distributed with estimation.

Table 5.3 shows the comparison of document processing times using CoreNLP. It shows also scale-up by using following equation:

$$Scale\ up = \frac{Non\ distributed - Distributed}{Non\ distributed} \times 100 \quad (5.1)$$

We know that each annotator of CoreNLP takes extra time to generate metadata, so we list them individually and show the time comparison between distributed and non-distributed parsing. Table 5.3 shows us that the distributed framework requires less time than a non-distributed framework for each annotator. Furthermore, if we consider all of the annotators,
then distributed framework only requires 15.34 minutes to process 12,484 documents whereas a non-distributed framework need 17.60 hours.

<table>
<thead>
<tr>
<th>Annotator</th>
<th>Distributed</th>
<th>Non-distributed</th>
<th>Scale up (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>token</td>
<td>51.96s</td>
<td>08.03m</td>
<td>89.00</td>
</tr>
<tr>
<td>lemma</td>
<td>60.14s</td>
<td>08.60m</td>
<td>88.00</td>
</tr>
<tr>
<td>ner</td>
<td>11.76m</td>
<td>10.12h</td>
<td>98.00</td>
</tr>
<tr>
<td>parse</td>
<td>09.71m</td>
<td>07.82h</td>
<td>97.00</td>
</tr>
<tr>
<td>dependency</td>
<td>10.45m</td>
<td>8.00h</td>
<td>98.00</td>
</tr>
<tr>
<td>dcoref</td>
<td>14.72m</td>
<td>17.33h</td>
<td>98.00</td>
</tr>
<tr>
<td>sentiment</td>
<td>10.08m</td>
<td>07.80h</td>
<td>97.00</td>
</tr>
<tr>
<td>All of the above</td>
<td>15.34m</td>
<td>17.60h</td>
<td>98.00</td>
</tr>
</tbody>
</table>

Table 5.3 lists the document statistics when 12,484 documents are used for political event generation using PETRARCH. Out of 12,484 documents, only 2,127 documents are considered for political event generation and the rest are discarded. This is expected because PETRARCH only encodes political events and discards any other news events. A total of 2,134 political event codes are generated from these 2,127 documents.

Table 5.5 compares the distributed framework for event coding with the non-distributed one. The distributed framework requires only 13.22 milliseconds of processing time whereas the non-distributed frameworks take 1.26 minutes.

Table 5.4. Event coding using PETRARCH

<table>
<thead>
<tr>
<th>Skipped documents</th>
<th>1,556</th>
</tr>
</thead>
<tbody>
<tr>
<td>No event documents</td>
<td>8,801</td>
</tr>
<tr>
<td>Event documents</td>
<td>2,127</td>
</tr>
<tr>
<td>Total documents</td>
<td>12,484</td>
</tr>
<tr>
<td>Total events</td>
<td>2,134</td>
</tr>
</tbody>
</table>

Table 5.6 shows the CoreNLP processing time of the distributed framework versus its non-distributed counterpart. Here, we consider all the annotators listed in Table 5.3. We have conducted our experiment on CNA and AFP news. In both cases, we have estimated
time for a non-distributed framework. CNA has 95,777 documents and our distributed framework takes only 2.9 hours to process all the documents whereas the estimated time of a non-distributed framework is 16 days. In the same way, AFP’s 0.9 million records require only 17.37 hours, whereas the non-distributed framework is estimated to run for a couple of months.

<table>
<thead>
<tr>
<th>Total events</th>
<th>Distributed</th>
<th>Non-distributed</th>
<th>Scale up (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,134</td>
<td>13.22ms</td>
<td>1.26m</td>
<td>99.00</td>
</tr>
</tbody>
</table>

Similarly, we show the event coding time comparison for AFP and CNA news documents. We can see from table 5.7 that CNA has 16,541 events and takes 102.47 milliseconds for event code generation, and AFP has 14,8738 events and takes 921.42 milliseconds in a distributed framework whereas in a non-distributed framework it is estimated to take a couple of minutes.

<table>
<thead>
<tr>
<th>Total documents</th>
<th>Distributed</th>
<th>Non-distributed (Estimated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNA 95,777</td>
<td>02.19h</td>
<td>16.00d</td>
</tr>
<tr>
<td>AFP 9,00,000</td>
<td>17.371h</td>
<td>5.20mo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total events</th>
<th>Distributed</th>
<th>Non-distributed (Estimated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNA 16,541</td>
<td>102.47ms</td>
<td>9.76m</td>
</tr>
<tr>
<td>AFP 1,48,738</td>
<td>921.42ms</td>
<td>87.82m</td>
</tr>
</tbody>
</table>

Figure 5.6 illustrates the variation in the processing time of CoreNLP with the number of workers in Spark. Adding more workers reduces the parsing time significantly.
Figure 5.6. CoreNLP parsing time over number of workers
CHAPTER 6
NEAR REAL-TIME ATROCITY EVENT CODING

Political conflicts typically emerge in geographically constrained regions before propagating through the larger international system. Such propagation occurs through both spatial proximity and complex inter-dependencies among political actors. Often these phenomena result in the deliberate killing of non-combatant people/civilians. This is defined as an atrocity in our context, e.g., ‘A suicide bomb attack killed 5 civilians in Baghdad’. Atrocity event data capture the detailed interaction among political actors who engage in political violence. These data are collected from multiple news reports, at the event level, the smallest aggregation possible allowing for both low-level and aggregate analysis of political violence. As such, they enable us to quantify the propagation of an event or a series of events through space and time, while also capturing the complex interactions of the actors involved.

Human coding has been widely used to record atrocities and remains the gold standard for validating machine-coded events. However, human coding has its limitations, like insufficient speed and accuracy. Human coded events can be used for reference purposes but the process is extremely laborious and cross-validated gold standard cases can only be generated at a rate of three to six events per hour, which cannot be applied to near real-time production-level event generation involving millions of records. On the other hand, machine-coding tries to generate event data quickly and as efficiently as possible [22], making it more acceptable for event data generation with daily updates.

We envision a near real-time framework for event coding. It collects articles periodically from different news portals all over the world using a web scraper. Later, we filter only atrocity-related articles and send them to a back-end server. In the back-end, we use Stanford

---

CoreNLP to extract the necessary information like the parse trees, lemma, tokens, etc., for event coding and store them in MongoDB. Finally, we used a supervised multi-class classification technique to code the features of the atrocity events. During coding, we first identify the feature field types and later suggest a label for that field using external knowledge base, WordNet.

Open source Natural Language Processing (NLP) software such as Stanford CoreNLP allows more accurate content extraction from the news stories. But the software itself is computationally intensive and, hence, increases the overall processing time of standalone applications. Thus, these real-time applications face scalability issues and become unsuitable for processing huge corpora of data. This creates a need for scalable frameworks capable of handling Big Data with low latency. Our preference is Apache Spark over other popular Big Data frameworks, e.g., Hadoop, HBase, Mahout, Apache Storm, or Apache S4. This is due to the superior performance of Spark in both batch and stream mode for processing data.

In this chapter, we have the following contributions:

- We develop a new near real-time distributed framework using web scraper with an atrocity filtering, Apache Spark, MongoDB, Stanford CoreNLP to store the possible atrocity reports.

- We develop an innovative multi-classification based machine coding technique to generate structured atrocity event data.

- We develop a novel technique to identify a field type of an event and suggest possible semantic labels for that feature using WordNet synonyms set and word-sense disambiguation.

The rest of the chapter is organized as follows: Section 6.1 gives a background on the various open source tools and concepts used in this chapter. Section 6.2 explains the overall framework. Section 6.3 shows the experimental results.
6.1 Background

The following concepts are pertinent to our framework:

**Atrocity Event Coding** records an atrocity-related event [78]. Each record has multiple attributes, e.g., type, perpetrator, victim, weapon, number of deaths, place, and so on.

**Web Scraper** [60] works for collecting news articles and extracting the main content of the news. The web-scraper has been integrated into the rest of the framework and runs periodically (every 20 minutes). It collects news stories from about 400 RSS (Rich Site Summary) Feeds [61] (on average, each containing 15-20 links to news articles). For the periodic nature of collecting and processing data, the system is called near real-time.

**Apache Spark** is an open-source distributed framework for data analytics. It supports both batch processing and streaming data [105].

**MongoDB** is an open source NoSQL document database. MongoDB stores data in the form of a BSON (Binary JSON-like) document. A set of documents forms a collection.

**Stanford CoreNLP** [49] is an useful framework for annotating text. Stanford CoreNLP includes tools like part-of-speech (POS) tagger, named entity recognizer (NER), parser, co-reference resolution system, sentiment analysis, and bootstrapped pattern learning tools. It supports models for other languages.

**SEMAFOR** is an NLP tool that describes the frame-semantic structure of English text. It uses a lexical resource called FrameNet [25]. It groups predicates in a hierarchy of structured concepts, known as frames. Each frame defines several named roles corresponding to that concept. These are then used to extract the features of the atrocities event data, such as the type of victim.

**WordNet** [99] is a lexical database. It groups words into sets of synonyms called synsets. It provides short definitions and usage examples and records a number of relations among these synonym sets or their members.
**Word-sense disambiguation** (WSD) [56] is a problem of identifying which sense of a word (i.e. meaning) is used in a sentence when the word has multiple meanings. Algorithm like lesk [44], and adaptive lesk [7] are widely used to solve this.

### 6.2 Near Real-time Atrocity Event Coding Framework

![Figure 6.1. Atrocity event coding framework](image)

Figure 6.1 shows our near real-time atrocity coding framework. It collects news stories periodically through web scrapers and then filters only the news related to atrocities. After that it uses Stanford CoreNLP to extract necessary information from the documents like parse tree, tokens, lemmas, etc., and stores it in a NoSQL database like MongoDB. We have used Apache Spark streaming with Kafka [40] to collect all scrapped data periodically. We used CoreNLP parsing inside Spark worker node to accommodate scale up and also to minimize parsing time. We implemented a supervised classification technique to code an event into a structured format that works on the parsed data.

The next subsections describe the above in details.
6.2.1 Atrocity Filter

After collecting the news articles from web scrapers, we pass them through the atrocity filter which works at two levels. At the first level, it discards any news article that does not contain the word(s) from the following list: killed, kills, massacre, bomb, bombing, bomber, beheaded, or a mass grave. If any of those words are found in the articles, then it is searched for another set of words (i.e., earthquake, accident, etc.) which classifies the reported event as non-violent. If none of the words in the second set is found, the article is considered for second stage filtering that involves a model-based text classification. We train the model on documents that report atrocities and those do not. We use a Multinomial Naive Bayes model as our classifier. The classifier outputs whether a given article reports an atrocity or not.

6.2.2 Near real-time data processing with CoreNLP and Spark

We collect news stories using a web scraper. The text of the reports is then filtered into a JSON form, such as,

```
{
    "AGENCY": "NGA",
    "DATE": "1 September 2015",
    "DOCUMENT": "AFNWS00020150901eb91000j9"
    "STORY": "Suspected Boko Haram gunmen on horseback have shot dead nearly ...",
    "NEWS_AGENCY_DESC": "All Africa",
    "AUTHOR": "Al Jazeera",
    "HEADLINE": "'Boko Haram Fighters On Horseback' Kill Villagers"
}
```
CoreNLP is used to extract valuable information from the text (like the parse trees, lemma, tokens, etc.). We use these features, applying machine learning techniques for the atrocity and feature classification of additional texts. CoreNLP performs accurately for English sentences but suffers from slow processing time when running on large news corpora. A significant bottleneck comes from the CoreNLP because each annotator needs a sufficient amount of time to process a document. Adding more annotators to the CoreNLP analysis makes it much slower. This means that a single document with several sentences to be processed requires more than a few minutes. Processing a large number of news stories in real time is, therefore, a challenge. Given these costs, we develop a distributed framework that will integrate both CoreNLP in a parallel manner, and perform the event coding tasks in close to real time.

Motivated by this, we use a distributed framework, Apache Spark. Spark is our obvious choice over existing distributed frameworks like Hadoop [33], Storm, etc., because of its in-memory computation and lower execution times [105] [82].

We read all the news documents coming into the system as a stream using Apache Kafka. We then filter them and store them in MongoDB using Spark Streaming. We apply a map function to each of document and then parse the sentences with the required annotator using CoreNLP to obtain a list of attributes and store them in MongoDB.

6.2.3 Atrocity event coding

The Political Instability Task Force Worldwide Atrocities Dataset [78] [64] is a hybrid of machine-filtered news reports that are then human-coded to create a dataset that documents global atrocity events. Each event is represented by a meta-data about the news reports that are then coded into a series of numeric (e.g., number killed) and textual variables (e.g., type of victim). The main goal here is to use NLP and machine learning to create a version of this dataset.
We studied the human-coding guidelines in the Codebook. The atrocities event data contains the following fields:

- **Fields having direct values from text.** These fields in the meta-data are straightforward like date, news agency, latitude/longitude, etc.

- **Fields having binary values.** These fields can take only two values like ‘YES/NO’, e.g., event type, perpetrator state role, etc. A binary classification technique can be used to code these.

- **Fields with multiple values.** These fields are very complex and need special attention. After analysis of the atrocity data, we have found four main categories of these fields: victim, perpetrator, related tactic, and Data source. Each of these may belong to multiple categories. Likely victim categories can be a religious victim, social victim, random victim, etc.

Considering the above findings, the goal is then to learn from the existing human-coded atrocities how to code streaming new atrocity reports into events and extract the relevant fields without human intervention. This requires training a new model that implements the following strategies:

**Feature extraction**

We employ the frame-parser, SEMAFOR, to extract the features. It generates semantic parsing with Key-Value pairs for each sentence. It includes a VICTIM key. We use this key to get all possible victims from a document. Other atrocities dataset fields require different keys from SEMAFOR. For example, the ‘WEAPON’ key in SEMAFOR can be used for coding ‘related tactic’ field in the machine-code version of the atrocity data.

After getting all the candidates of a particular field, we generate a feature vector. Later we apply supervised classification to identify the encoded category of the field.
Feature vector generation

While generating the feature vector, we consider two tasks:

- We take the SEMAFOR output for an event as a feature (bag-of-phrase) and generate a feature vector.
- We apply tf-idf with n-grams (uni-grams, bi-grams) on the SEMAFOR output and generate a feature vector.

Classification to identify the category

Classifying a field with multiple categories is a complex task. To know more about the data, we run some sample tests to gather statistics because some fields have multiple categories. Considering the statistics, we conclude that the following classification techniques are suitable for event coding.

Multi-class classification - [69] assumes that each training point belongs to one of \( N \) different classes. The goal is to construct a function which, given a new data point, will correctly predict the class to which the new point belongs. For example, classify a set of images of fruits which may be orange, apple, or pear. A fruit can be either an apple or a pear but not both at the same time.

We use a One vs. Rest SVM [63, 69] classifier for the above two classifications techniques. Here we build \( N \) different binary SVM classifiers. For the \( i^{th} \) classifier, let the positive examples be all the points in class \( i \), and let the negative examples be all the points not in class \( i \). Let \( f_i \) be the \( i^{th} \) classifier which we then classify with the following objective function:

\[
f(x) = \arg \min_i f_i(x)
\]

(6.1)
Figures 6.2 and 6.3 describes the procedure of coding a data field (which require machine learning) in the atrocities data. We apply SEMAFOR on the documents and extract relevant features for a particular field. After that, we apply bag-of-phrase or tf-idf on these features to generate feature vectors. For an event’s variables, we first identify the type of each field and then predict its value. Since we are using supervised classification, we split data into training and testing. We build a classifier to predict a field type for a testing instance. After that, we suggest a label for that testing instance. For a given field type, we find the closest match from the training data and take the label of the field type. This will be the suggested/predicted label of that field for the testing instance.
Resolving word sense disambiguation using WordNet

Performance of the event coding demands human coding. Human coding is expensive and time-consuming. So, we take a subset of the data where we have human coding and perform classification and calculate performance metrics. Moreover, it is typical that the predicted label of a field will not be an exact match to its original label, but may match a synonym. For a given atrocity victim, the predicted label may be ‘people’, but its original label was ‘civilian’, then the classifier will normally mispredict this. As a result, we will get very low precision and recall in exact labeling.

Instead of requiring exact matches across a large set of possible labels for victim categorization, we introduce semantic labeling, where we will suggest the semantic labels of the field. For example, ‘people’ and ‘civilian’ have semantic similarity and would be considered a match in most cases. We used external knowledge from WordNet for making these synonym sets. Instead of taking the exactly predicted labels, we use all the related sets of synonyms that fit with the SEMAFOR features for variables. SEMAFOR gives us a list of phrases for a particular field during feature extraction. When we take related synonyms set, we remove all the ambiguity using simple leks and adapted leks word sense disambiguation. We use a similarity threshold $\text{SIM}_{th}$ to find the closest synonyms sets.

Algorithm 16 Algorithm for Accuracy Metric

1: procedure ACCURACY(Original label: $O$, Predicted label: $P$, SEMAFOR list: $S$)
2:  \hspace{1cm} $O_{list} \leftarrow \text{StemmedTokens}(O)$
3:  \hspace{1cm} $P_{tok} \leftarrow \text{StemmedTokens}(P)$
4:  \hspace{1cm} $P_{syn-list} \leftarrow \text{TakeSynsetwithWSD}(P_{tok}, S)$
5:  \hspace{1cm} $match \leftarrow \text{Set}(O_{list}) \cap \text{Set}(P_{syn-list})$
6:  \hspace{1cm} $\text{precision} \leftarrow |match|/|P_{syn-list}|$
7:  \hspace{1cm} State $\text{recall} \leftarrow |match|/|O_{list}|$

Algorithm 16 shows how we generate prediction labels with WordNet synonym sets. Initially, we stem the original and predicted label and make a list of the token in lines 2
and 3. In line 4, we take the WordNet synonym sets applying word sense disambiguation.
Finally, we calculate precision and recall in lines 6–7.

6.3 Experimental Results

6.3.1 Training Dataset and Development Environment:

The Atrocities Dataset employed for training here [78] is a collection of recent news reports. These reports are machine filtered news reports of atrocities and mass killings in several locations in the world. Data from these reports are extracted by human coders who annotate information such as the event’s victims, perpetrators, etc., from multiple news reports.

For the experiments to compare the training data with the predicted labels for the variables, we used our own Spark-based cluster with streaming data. Table 6.1 shows the configuration of the cluster.

<table>
<thead>
<tr>
<th>Total Master node</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total slave node</td>
<td>12</td>
</tr>
<tr>
<td>vCPU (core) in each node</td>
<td>8</td>
</tr>
<tr>
<td>Memory in each node</td>
<td>16GB</td>
</tr>
<tr>
<td>Storage in each node</td>
<td>1TB</td>
</tr>
</tbody>
</table>

6.3.2 Performance of the initial Atrocities Filter

We trained the atrocity filter which separates news reports and their features that do and do not include such events over a previously collected dataset, containing 397 documents that report atrocities, and 223 documents that do not. We split the data randomly, using 70% of the data to train the classifiers and 30% to test. The different supervised classification techniques (listed with accuracy in the Table 6.2) are considered. The Multinomial Naive Bayes model shows the highest level of accuracy. Ten-Fold cross-validation with Multinomial Naive Bayes shows a similar level of performance.
Table 6.2. Atrocity Filter Performance

<table>
<thead>
<tr>
<th>Approach</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multinomial Naive Bayes</td>
<td>98.35%</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>93.96%</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>91.76%</td>
</tr>
<tr>
<td>Linear Support Vector Machine</td>
<td>96.70%</td>
</tr>
</tbody>
</table>

6.3.3 System Scalability

Table 7.3 shows our cluster set up. We performed the system scalability by taking sample documents and ran in distributed and non-distributed system and found almost 98% scalability. We took 12,484 documents as a performance comparison. Our Spark-based system required 15.34 minute to process 12,484 documents whereas a non-distributed framework needs 17.60 hour [80].

6.3.4 Atrocity Event Coding Results

Atrocity event meta-data contains few dozens of fields [64]. We describe it in section 6.2.3. As a case study, we show our experimental result for ‘VICTIM’. Our near real-time framework generates news documents but they don’t come with human event coding. All our classification based techniques require gold standard or ground truth. We take a subset of our dataset with human event coding (treated as ground truth) and use them to measure classifiers’ performance.

According to atrocity event coding standard [64], ‘VICTIM’ field’s categories and their corresponding distribution in dataset are shown in table 6.3 and 6.4. For our experiment, we consider the documents that have exactly one category.

Furthermore, Table 6.3 shows some categories (e.g., Combat, Ethnic, etc.) are fewer in number and it indicates that we have an imbalanced dataset. As a result, we apply oversampling and undersampling [18] during training to make our dataset balanced.
Table 6.3. Victim category distribution (each document has one category)

<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>Number of document</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>victimpoli</td>
<td>Political</td>
<td>360</td>
<td>19.40</td>
</tr>
<tr>
<td>victimcomb</td>
<td>Combat</td>
<td>21</td>
<td>1.13</td>
</tr>
<tr>
<td>victimreli</td>
<td>Religious</td>
<td>215</td>
<td>11.60</td>
</tr>
<tr>
<td>victimsoci</td>
<td>Social</td>
<td>195</td>
<td>10.51</td>
</tr>
<tr>
<td>victimrand</td>
<td>Random</td>
<td>908</td>
<td>48.94</td>
</tr>
<tr>
<td>victimethn</td>
<td>Ethnic</td>
<td>156</td>
<td>8.40</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>1855</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Table 6.4. Total victim category distribution

<table>
<thead>
<tr>
<th>Total category per document</th>
<th>Total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>88</td>
<td>4.22</td>
</tr>
<tr>
<td>1</td>
<td>1855</td>
<td>89.02</td>
</tr>
<tr>
<td>2</td>
<td>133</td>
<td>6.38</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>0.38</td>
</tr>
<tr>
<td>Total</td>
<td>2084</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Table 6.5. Performance metric to identify VICTIM category

Table 6.5 shows the performance result of identifying VICTIM category. It shows that both bag-of-phrase and tf-idf based oversampling techniques have good performance result but bag-of-phrase oversampling outperforms all in terms of precision, recall, F2, and accuracy.

Figure 6.4 and 6.5 show the performance of semantic labeling using WordNet. Both have similar precision, recall, and F2 while table 6.6 shows exact labeling and it is very poor. During semantic labeling, we vary threshold $SIM_{th}$ and report the result.
Figure 6.4. Performance of semantic labeling victim for tf-idf

Figure 6.5. Performance of semantic labeling victim for phrase

Table 6.6. Performance of exact labeling victim

<table>
<thead>
<tr>
<th></th>
<th>Precision (%)</th>
<th>Recall (%)</th>
<th>F2 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tf-idf (n-gram=1,2)</td>
<td>6.3</td>
<td>9.8</td>
<td>7.7</td>
</tr>
<tr>
<td>Bag-of-phrase</td>
<td>2.5</td>
<td>4.2</td>
<td>3.1</td>
</tr>
</tbody>
</table>
CHAPTER 7
AUTOMATIC POLITICAL ACTOR RECOMMENDATION IN REAL-TIME

Political event data are encoded from news reports. This can be accomplished manually by humans or via machine coding. Events are categorized based on a set of dictionaries for the actions and actors. The typical format to create these data is to determine ‘who-did/said-what to whom’. While the set of actions or verbs is finite and can be matched to a fixed ontology, the set of source or target nouns is quite large in political terms. To transform news reports into useful data to study international relations and civil conflict, identifying these political actors along with their roles is important. These include both material and verbal interactions between political entities, such as bombings, protests, or diplomatic consultations. These data are collected at the event level, the smallest aggregation possible, allowing for both low-level and aggregate analysis of political behaviors. They enable us to quantify the propagation of an event or a series of events through space and time, while also capturing the complex interactions of the actors involved.

Human coding was the standard method to generate political event data before the 1990s and remains the gold standard for validating machine-coded output. As a manual process, humans were responsible for both encoding the events and updating the dictionaries used for coding. Modern technologies offer efficient Natural Language Processing (NLP) techniques that ease the human efforts in political event coding. PETRARCH is a tool to encode events. It encodes the events in ‘who-did-what-to-whom’ format, supported by CAMEO dictionaries. It uses the dictionaries for actors to identify them and assign appropriate roles (i.e., Government Employee, Media Person, etc.) based on a timeline. However, if a source or target actor has not been entered by a human in the relevant actor dictionary,

\[^{1}\]© 2017 Springer International Publishing AG. Portions Adapted, with permission of Springer, from M. Solaimani, S. Salam, L. Khan, P. T. Brandt, and V. D’Orazio, APART: Automatic Political Actor Recommendation in Real-time, SBP-BRiMS, Pp. 342-348, 2017
PETRARCH will ignore the event related to the new actor. These events are lost because of infrequent dictionary maintenance conducted by human curators, largely due to the cost of human involvement.

Designing a framework for recommending actors in real-time poses several challenges. First, actors may come with multiple alias names, e.g., ‘Barack Hussein Obama’, ‘Barack Obama’, ‘President Obama’, etc. Currently, humans include a dictionary entry for each of these alias in CAMEO actor dictionary. Second, the role of an actor changes over time. For example, ‘Shimon Peres’ served multiple political roles in Israel during his lifetime. Finally, processing a large volume of news articles across the world demands scalable, distributed computing to detect these changes.

Considering these challenges, we develop a real-time, distributed recommendation framework for the identification of potential actors and their associated roles. We develop this tool to reduce human efforts in the dictionary update and development process. We gather news articles from the all-around-the world, pre-process the articles with Stanford CoreNLP and PETRARCH to extract the information needed about the actors. Then we apply our unsupervised ranking algorithm for recommending probable new actors with roles. Our actor recommendation uses PETRARCH for political event coding. We compare it with a proprietary event coder BBN ACCENT™ and show the performance. Moreover, we implement a graph-based actor role recommendation technique using weighted Label propagation [46] as a baseline and compare it with our role detection method. This chapter highlights the following contributions.

First, we have proposed a novel time window-based, unsupervised new actor recommendation technique with possible roles. In particular, we have designed a frequency-based actor ranking algorithm with alias actor grouping from news articles.

Second, we have developed a distributed real-time framework to address the scalability issue of event coding over political news articles.
Finally, we have compared our proposed approaches with state of the art (e.g., BBN ACCENT event coding) and shown the effectiveness of our work.

The rest of the chapter is organized as follows: Section 7.1 gives the background on the various open source tools and concepts. Section 7.2 explains the overall framework. Section 7.3 describes our new actor with role recommendation technique in details. Section 7.4 shows the experimental results for new actor detection.

7.1 Background

Stanford CoreNLP [49] is a useful tool for annotating text with part-of-speech (POS) taggers, named entity recognition (NER), parser, co-reference resolution system, sentiment analysis, and bootstrapped pattern learning tools. It supports models for multiple languages.

CAMEO (Conflict and Mediation Event Observations) is an event coding framework to record political events (both material and verbal interactions) in a structured ‘who-did-what-to-whom format’.

PETRARCH (A Python Engine for Text Resolution And Related Coding Hierarchy) [59] uses natural language processing output to produce machine-coded event data. It takes fully-parsed text summaries in Penn Tree format [89] from Stanford CoreNLP and generates events in a ‘who-did-what-to-whom’ format.

Apache Spark [105] is an open-source distributed framework for data analytics with both batch processing and streaming data.


Word2Vec [52] is a two-layer neural net text processing tool. It takes a text corpus as input and its output is a set of vectors: feature for words in that corpus. Word2vec groups
the vectors of similar words together in vector space and detects similarities mathematically. Word2vec creates vectors that are distributed numerical representations of words as features.

7.2 Framework

Figure 7.1 shows our new political actor recommendation framework. It collects news stories periodically through web scrapers and then filters only the news related to political events. Later, it uses CoreNLP to extract meta-data from the documents like parse trees, tokens, lemmas, etc., and stores it in the NoSQL database, MongoDB. After that, it uses PETRARCH to generate a political event code from the meta-data. We used Apache Spark streaming with Kafka [42] to collect all the scrapped data, periodically. We use CoreNLP parsing inside a Spark worker node to scale up the process, and also to minimize overall parsing time. We implement an unsupervised frequency-based actor ranking technique to recommend potential new actors with their related roles using event codes and associated meta-data. Finally, human expert validates the recommended actors with roles and updates the dictionary. We describe our framework in detail in the following subsections.
7.2.1 News Scraper

We use a web-scraper to collect news articles and extract the main content of the news. The web-scraper has been integrated into the rest of the framework and runs periodically (e.g., every 2 hours). It collects news stories from about 400 RSS (Rich Site Summary) Feeds [61] (on average, each containing 15-20 links to news articles). The news articles are shipped through Apache Kafka and sent to an Apache Spark-based data processing module [80].

7.2.2 Meta-data extraction

Inside our data processing unit, CoreNLP parses all the news documents and extracts meta-data like Parts-Of-Speech (POS) tagging, Parse Tree, Named Entity Recognition (NER), LEMMA, etc., and stores them into MongoDB. For our framework, we use the Parse Tree & NER.

- **Parse Tree** [89] - is an ordered, rooted tree that represents the syntactic structure of a sentence in some context-free grammar. CoreNLP generates Penn Tree-banks using Part-of-speech tagging.

- **Named-entity recognition (NER)** [55] locates and classifies named entities in text into pre-defined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc., or our framework, we use persons and organizations to detect new actors. We use Stanford CoreNLP as Named Entity Recognition tool because it has higher accuracy [71, 4] than other similar tools like Alchemy [39], OpenCalais [68], OpenNLP [62], etc.

7.2.3 Event coding

PETRARCH generates political event codes [80] for each sentence in a news article using parse tree, CAMEO dictionaries. It produces ‘who-did-what-to-whom’ like event coding. The following example illustrates this in detail:
“Obama vowed again on Sunday to help France hunt down the perpetrators of the attacks.”


7.2.4 New Actor Recommendation

PETRARCH provides political events. Each of them consists of actors which are looked up from CAMEO dictionary. We use PETRARCH to catch the political events where actors are missing in CAMEO dictionary. These actors are possible news actors. We filter them with NER and generate a score based on their frequency. We detect an actor with alias names and group them together. We introduce a window based actor ranking where we select the top $N$ actors for each window. We recommend an actor as a new actor if he/she appears in most of the windows. New actors’ role can be inferred from their related existing actors’ roles. We describe them in detail in section 7.3.
7.2.5 Human Feedback

We provide a graphical user interface and dashboard to the end users/human expert. It shows all the new recommended actors with their role. Users can easily view them and update the actor dictionary (e.g., CAMEO) accordingly.

7.3 Recommending a New Actor and Role

Figure 7.2. Technical details.

Figure 7.2 describes the technical details of our real-time actor recommendation technique. We use NER to filter out all possibles actors from a news article. After that, we calculate a score or ranking for each actor across all the documents and select the top $N$ with their roles. Here we have maintained a buffered time window $W$ of length $L$. Each window contains the top $N$ actors, which are merged with the previous window’s actors list and updated the rank statistics. After $L$ windows, we recommend new actors with their role,
if their occurrences in the $L_{th}$ buffered window exceed a certain threshold $TH$. We describe in detail about this in forthcoming subsections starting with the following example.

**Scenario** - Donald Trump is the new President of United States. Current CAMEO actor dictionary does not have any entry of him. So, PETRARCH will not generate any event code from news articles related to Donald Trump unless human updates dictionary manually. On the other hand, our real-time framework periodically captures political news and from them, it identifies new probable political actors. As many news highlight Donald Trump, he has a high chance of being an actor and we identify and recommend him as a political actor. Moreover, we suggest his role from his surrounding existing political actors.

### 7.3.1 New Actor Discovery

We use a frequency-based ranking algorithm to find the top actors and their roles. Each news article/document contains raw text related to a political event. CoreNLP parses this raw text and outputs NER for each sentence. Using CoreNLP’s parse tree output, PETRARCH generates event codes for each sentence. These event codes contain existing actors (sources/targets) with their roles. For example, *President Obama* has role *USAGOV*. NER gives us all actors (i.e Person, Organization) in a sentence. A news article contains multiple sentences. We will get NER list and existing actors’ list (from PETRARCH) for each of the sentences. We filter potential new actors by removing existing actors from NER actor list. Sometimes, an actor may come in different name/partial name across the article. For example, *Barack Hussein Obama* may come as *Obama* or *Barack Obama* in many sentences. As a result, we need to group these actors with all possible alias names and make a single actor name (e.g., *Barack Hussein Obama*). We use several similarity measures like Levenshtein Distance (edit distance) $[45]$, MinHash $[12]$ to group alias actors. All of these methods need a similarity threshold $sim_{th}$. After grouping, we calculate the term-frequency of each actor inside the news article. For an alias actor, we take the maximum count as
its term-frequency. For example, the term-frequency of Barack Hussein Obama will be the maximum frequency of Barack Hussein Obama, Obama, and Barack Obama. Similarly, we group aliases for actors across multiple documents. Finally, for each actor $a$, we calculate a document frequency $df$ and use the following equations to generate a score:

We use term frequency $tf(a,d) = \text{count}_{a \in d}(a)$ to show how frequent an actor $a$ is in a document $d$. Equation [7.1] shows the rank calculation of an actor $a$, where $df(a,D) = \frac{|\{d \in D: a \in d\}|}{|D|}$ is document frequency and it shows how frequent an actor $a$ comes across all news articles/document in the set $D$.

$$rank(a) = \sum_{d \in D} tf(a,d) \times df(a,D)$$ (7.1)

### 7.3.2 Actor role discovery

PETRARCH gives us the event codes for a sentence in a news article. We collect all event codes which contain source and target actors with their roles from an article. These are looked up in the existing CAMEO dictionary during coding. Each actor may have multiple roles in different time and we capture them. Here, we assume that new actors’ political roles will be related to their most frequent co-occurred actors. So, we collect the roles and assign them to new actors as possible roles. We keep a map of each role with its occurrence. We update actors’ maps with a role for all news articles. When an actor appears in two news
Algorithm 17 New actor discovery

```
1: procedure actorDiscovery (DocumentSet D)
2:     Actor Map M ← {}
3:     for each d ∈ D do
4:         m ← CoreNLP(d)
5:         Ed, Acurrent ← PETRARCH(d)
6:         Aall ← m.NER()
7:         Ad ← Aall − Acurrent
8:     R_d ← E.Roles()
9:     for each a ∈ Ad do
10:         tf ← count_a∈d(a) ▷ calculate tf
11:         for each a_i ∈ Ad − a do
12:             if Match(a, a_i) > sim_th then
13:                 a ← Merge(Ad, a, a_i)
14:                 tf ← max(tf, count_a_i∈d(a_i))
15:                 k ← arg max_k {Match(M(k), a) > sim_th}
16:                 if k ≠ empty then
17:                     k ← Merge(M, k, a_i)
18:                 M.insert(k, [(tf, d, R_d)])
19:         else
20:             M.insert(a, [(tf, d, R_d)])
21:     M_rank ← {}
22:     for each k ∈ M.keys() do
23:         df ← |k ∈ d : d ∈ D|/|D| ▷ calculate df
24:         rank ← ∑_tf∈M(k) (tf) × df
25:         M_role ← {}
26:     for each R_d ∈ M(k) do
27:         for each r ∈ R_d do
28:             M_role(r) ← M_role(r) + 1
29:         M_rank.insert(k, (rank, M_role))
```

articles, we include all roles in both articles and increase common roles’ occurrence. While we have all ranked actors, we also have their role maps.

Algorithm 17 shows the new actor discovery algorithm with ranking. It takes the news document set $D$ as input and gives actor map with rank $M_{rank}$ as output. $M_{rank}$ contains each actor as key with its ranking and role map $M_{role}$. It takes each document $d ∈ D$ and extracts new actor with roles $R_d$ (lines 4-8). After that, it groups all aliases for actors inside
Algorithm 18 Real-time new actor recommendation

1: procedure actorInRealTime($T H_{\text{min}}, T H_{\text{max}}, N, L$)
2: New Actor Map $M_A \leftarrow \{\}$
3: New Actor Role Map $M_R \leftarrow \{\}$
4: for $t \leftarrow t_1$ to $t_L$ do
5: $D \leftarrow \text{getDocuments}(t)$
6: $M_{\text{rank}} \leftarrow \text{ACTORDISCOVERY}(D)$
7: $M_{\text{topN}} \leftarrow M_{\text{rank}}.\text{top}(N)$
8: for each $<a,(\text{rank}, M_{\text{role}})> \in M_{\text{topN}}$ do
9: \hspace{1em} $k \leftarrow \text{arg\,max}_k\{\text{Match}(M_A(k), a) > \text{sim}_{th}\}$
10: \hspace{1em} if $k \neq \text{empty}$ then
11: \hspace{2em} $k \leftarrow \text{Merge}(M,k,a)$
12: \hspace{2em} $M_A(k) \leftarrow M_A(k) + 1$
13: \hspace{1em} else
14: \hspace{2em} $M_A(a) \leftarrow 1$
15: \hspace{1em} for each $r \in M_{\text{role}}$ do
16: \hspace{2em} $M_R(k) \leftarrow M_R(k) + 1$
17: \hspace{1em} for each $a \in M_A$ do
18: \hspace{2em} if $M_A(a) \geq T H_{\text{max}}$ then
19: \hspace{3em} $M_A(a) \leftarrow "\text{new actor}"$
20: \hspace{3em} $M_R(a) \leftarrow "\text{new actor’s role}"$
21: \hspace{1em} if $M_A(a) \leq T H_{\text{min}}$ then
22: \hspace{2em} $M_A(a) \leftarrow "\text{discard actor}"$
23: \hspace{2em} $M_A\.\text{remove}(a)$
24: \hspace{2em} $M_R\.\text{remove}(a)$

a document with counting term-frequency $tf$ (lines 10-16). Next, it groups actors’ aliases across multiple documents and updates the temporal actor map $M$ which contains an actor as a key and a list of the tuple $<tf,d,R_d>$ as a value (lines 17-24). After that, for each actor in $M$, it calculates the document frequency $d$, and rank score using equation 7.1 (lines 28-29). It then creates a role map $M_{\text{role}}$ for each actor which contains all the possible roles with their total occurrences across the document set $D$ (lines 31-35). Finally, we insert the actor as key and a tuple of rank, and role map $M_{\text{role}}$ as value into $M_{\text{rank}}$. So, $M_{\text{rank}}$ contains all possible new actors with their rank, and possible roles.
7.3.3 Recommending new actors in real-time

Our framework gets political news articles periodically and possesses a buffered time window \( W \) of length \( L \). At each time window, it scans all the news articles and gets the possible actor list with rankings and possible roles. It is well known that new political actors will always be highlighted in media and people will always talk about them and in print or online media. Here we assume that, if an actor comes in the top \( N \) ranking in multiple time windows, he or she has a high probability of being a new political actor.

Algorithm 18 describes in details about the recommendation procedure. For each time window, we receive the actor list with ranking \( M_{\text{rank}} \). We take the top \( N \) actors from the list (lines 5-7). We update a new actor Map \( M_A \) and its role map \( M_R \). These are incrementally updated during each time window. For each actor in \( M_{\text{rank}} \), we find the list of co-occurred actors in \( M_A \). If there is no actor in \( M_A \), we insert it in \( M_A \) with occurrence 1 and its role in \( M_R \). If we find the closest match, then we merge the actor name if required. Later, we increment its occurrence in \( M_A \). In the same way, we update the role of that actor in \( M_R \) (lines 8-20). As we increase the actors in \( M_A \) and \( M_R \), their size will always be increased. So, we introduce a \((\min, \max)\) threshold \( TH_{\min} \) and \( TH_{\min} \). After the \( L^{th} \) time window, if the total occurrence of an actor is greater than \( TH_{\max} \), we consider him/her as a new actor and recommend top \( N \) roles from \( M_R \) also. On the other hand, if total occurrence is below \( TH_{\min} \), we discard the actor from \( M_A \) and \( M_R \) (lines 21-31). After \( L^{th} \) time window, we will recommend new actors with possible related roles. Human experts will verify this and update CAMEO dictionary accordingly.

7.4 Experiments

7.4.1 Setup and Dataset

To evaluate our framework, we take 9-time windows at 2 hours interval. In each time window, we scrap newly published online news articles listed in 400 RSS feeds around the world. This
gives us 9 windows for the experiment. We process these articles/documents with CoreNLP running in Apache Spark. Finally, the processed articles are used for recommendation-related tasks.

7.4.2 Threshold Estimation

We are using similarity measure for grouping actors with alias names. This uses partial string matching, like edit distance, Min-Hash, etc. In the experiments, we use both Min-Hash and edit Distance and show their performance. Min-Hash calculates the Jaccard similarity between two strings and the edit distance tries to find the similarity based on how much editing is necessary to convert one string to another. Both approaches give a numeric value between 0 and 1, where the lower the value, the less is the similarity and vice versa. For our purpose, we need to set the similarity threshold value for considering whether two strings represent the same actor using these methods. When the similarity value between two actor names becomes greater than the threshold, we regard them as the same entity. To have an estimate of the threshold, we consider similarity values for actors already in the CAMEO actor dictionary. As an example, consider the following snippet from the actor dictionary for Kofi Annan and Boutros Boutros Ghali:

```
KOFI_ANNAN_
+SECRETARY-GENERAL_KOFI_ANNAN
+UNITED_NATIONS_SECRETARY_GENERAL_KOFI_ANNAN
+SECRETARY-GENERAL_KOFI_ANNAN
+K._ANNAN_
+K_ANNAN_
+KOFI_ATTA_ANNAN_
```
Both of them are followed by multiple aliases (lines starting with a ‘+’). We calculate the similarity between these aliases and the aliases of different actors. In the first case, we observe higher values (close to 1). For the later one, we found smaller values. Then we set a value that will minimize the number of false positives (i.e., reporting high similarity when they are not). By this empirical study, we estimate thresholds for edit distance and min-hash based methods to be 0.75 and 0.45 respectively.

7.4.3 Baseline methods

We note that building an automated system to recommend new actors with roles is not a task for which there is existing literature or benchmarks. However, there are works related to our core components of our framework like discovering new actors and recommending their roles. We take the following two methods as baselines.

BBN ACCENT™ [11] event coder automatically extracts event data from news reports from around the world. It is based on BBN SERIF™, a natural language analysis engine that extracts structured information (e.g., entities, relationships, and events) from
texts. Its events are coded according to the Conflict and Mediation Event Observations (CAMEO) ontology. It is a proprietary political event coding tool developed by Raytheon BBN Technologies Corp [67]. BBN encodes political event which has actor, agent, and action (CAMEO event type) but it does not generate the role of an actor. In our experiment, we use BBN ACCENT as an alternate of PETRARCH and draw a comparison for discovering new actors only. As BBN is proprietary and its used dictionaries are encrypted, so our comparison is based on new actor recommendation with human expert validation, i.e., precision comparison. This means that our framework will recommend new actors using PETRARCH and BBN separately and a human expert will validate how many of them are really new actors for each of the event coding tools.

**Graph-based Role detection technique.** We introduce a graph-based technique to suggest roles for recommended actors. Roles of a new actor will be influenced by existing actors with whom he/she has mostly interacted with. Therefore, we use weighted label propagation technique[46] to infer possible roles from existing related political actors. We formulate a graph $G = (V, E)$ that implies the interaction between actors. Here, $V$ is the set of actors (contains both existing and recommended actors). For two actors, $u$ and $v$, $(u, v) \in E$ represents those actors who are mentioned in the same news article. We also assign a weight function $w(u, v)$ that implies how frequent $u$ and $v$ are co-occurred in the news articles.

$$w(u, v) = \text{co-occurrence count of } u \text{ and } v$$

After that, we assign labels to the nodes which are the roles found in the dictionary for existing actors. For recommended actors, we simply put an empty label. We use $\text{label}(x)$ to denote the label for actor $x$.

$$\text{label}(x) = \begin{cases} \text{roles from dictionary} & \text{if } x \text{ is an existing actor} \\ \text{empty} & \text{otherwise} \end{cases}$$
Now we run our iterative label propagation algorithm, where we assign weight to each possible role for a recommended user. Roles are selected from neighboring actors. Weight is assigned for each role based on a weight associated with the edge of interaction graph, $G$. We repeat the process $N$ times or until the label assignments become stable. The weighting function for roles works as follows,

$$\text{role-weight}(u, \text{role}) = \sum_{v \in \text{Neighbors}(u)} T(v, \text{role}) \times w(u, v)$$

where $T(v, r)$ is used as indicator variable that simply returns 1 if $v$ has ‘$r$’ in its list of roles, 0 otherwise. In Figure 7.3, we present a scenario of how role recommendation works.

![Graph for Role Recommendation](image)

Figure 7.3. Example Scenario for Graph Based Role Recommendation

Here we are going to determine roles for DONALD TRUMP. He is associated with both existing actors (gray nodes) and new political actors. All the existing actors have their roles associated with their nodes. For example, BARACK OBAMA has USAELI and USAGOV as his roles. Now, we use our role-weight method to calculate the role. For example,

$$\text{role-weight(DONALD TRUMP, USAGOV)} = 4 \times 1 + 9 \times 1 + 3 \times 0 + 2 \times 0 = 12$$

After we calculate weight for all roles from neighbor, we find USAGOV, USAELI, RUSGOV as the top 3 possible roles for DONALD TRUMP.
Figure 7.4. Performance of Actor recommendation

(a) Deleted actor = 5

(b) Deleted actor = 10

(c) Deleted actor = 15
Figure 7.5. Performance of role recommendation
7.4.4 Experiment 1: Performance Evaluation

This experiment evaluates the framework’s correctness. To do this, we eliminate some well known actors from existing CAMEO actor dictionary and try to retrieve them using our framework over time. We start with removing 5, 10 and 15 actors. PETRARCH will not generate events involving those deleted actors. So the deleted actors are considered for a recommendation (as if they are newly discovered). We calculate how many of them are successfully retrieved (i.e., recommended by our algorithm) and this gives us the recall. We can not compute precision because all the recommended actors are political actors.

From Figure 7.4, we see that if we increase the top $N$ actors (i.e., how many actors are recommended at each time window), it will increase the possibility of retrieval of a deleted actor.

In the following experiment, we suggest the possible roles for identified actors in the previous experiment. As a reference, we have their roles (R1) listed in the CAMEO dictionary. Now using our frequency-based role recommendation algorithm we predict appropriate roles (R2) for them. Then we calculate the intersection of the two sets, R1 and R2. If the output is non-empty set then we consider it as a success. For the purpose of the experiment, we keep the size of R2 as 5. The roles are selected based on frequency. Top 5 frequent roles are presented as possible actor roles. Again we vary the number of deleted actors. We also vary the top $N$. For the experiment, we reported the ratio of the number of success and number of actors retrieved. Figure 7.5 shows the results. We see that edit distance has a slight advantage while actor recommendation. We use word set in MinHash calculation and it uses Jaccard similarity. So, for close matched actor aliases like ‘Donald Trump’ and ‘Donald J Trump’, edit distance gives higher similarity than MinHash. For similar reason, we see recall variation in 7.5 for role recommendation. Roles of an actor is a short text. Edit distance performs better when top $N$ increases, whereas MinHash and Exact matching have similar level of performance (worse than edit distance). Because for edit distance, we
consider partial matching. For example, USAMILGOV and USAMIL are similar using edit distance method, but hugely different when MinHash (Jaccard similarity) is used.

**Discovering roles with Word2Vec** - We use Word2Vec to find the related roles, which requires a training model. We take possible new actors with related existing actors from news articles. We collect them across 1 to $L^{th}$ (algorithm 18) time windows to train the model. Later, for each new actor, we find the top $N$ similar existing actors (in the CAMEO dictionary) using Word2Vec’s `similar_word` method and recommend their roles. Unfortunately, we get less than 5% recall. This is obvious because Word2Vec does not apply alias actor grouping. As a result, it fetches less similar actors than our frequency-based approach.

![Figure 7.6. Comparison of actor role recommendation with baseline](image)

**Comparison of Role Recommendation Techniques** In this experiment, we first delete some well-known actors from the existing CAMEO dictionary. Then we try to retrieve their roles by frequency based and graph based role detection techniques. For this experiment, we fix the values of a number of recommended actors per window and number of deleted actors to 15. We do not vary those parameters for graph-based approach because it caused the resulting graph to be very sparse. That, in turn, makes the role inference hard with label propagation technique. For example, if we use $N = 5$ and number of deleted...
actors=15, we found only 40 actors with 56 edges between them in the interaction graph. That is only 7% of the edges possible in a complete graph with the same number of nodes. So we consider all the actors suggested by the actor recommendation algorithm and their interactions to formulate the graph. Figure 7.6 shows the statistics for the experiment where \( N = 15 \), and deleted actors are 15. Here, frequency based approach outperforms graph-based approach for all the similarity measurement techniques because in the frequency based approach we consider roles from existing actors in the CAMEO dictionary. But in case of graph-based approach, we consider roles from neighbors who can be either existing or new actors. In that case, the error with one role assignment can propagate to others. The output contains 91 existing actors with 694 possible new actors. That means label propagation algorithm has less number of labels to propagate and the edges between new actors will be higher because they have the larger possibility to co-occur in a document.

7.4.5 Experiment 2: Recommendation of New Actors with Roles

This experiment represents the output of our framework. We list the possible newly recommended political actors based on a threshold. The threshold is how many times an actor appeared in the time windows. We set number of time window \( L = 9 \) and max, min frequency threshold \((TH_{\text{max}}, TH_{\text{min}}) = (5, 2)\) in Algorithm 18. So if any actor appears in 5 (more than 50% window length), he/she will be suggested as a potential new political actor. In case of roles, we list the most probable roles from their co-occurred existing political actors. Table 7.2 shows the list of recommended actors after all windows. We use both MinHash and edit distance based string similarity and list the output side-by-side. We found that both the approaches detect similar roles for identical actors, but the recommended user list varies.

**Compare with BBN ACCENT** We use BBN ACCENT as event coder in our framework and run an experiment to recommend new actors. Human expert validates recommended new actors and we calculate precision based on that. We run the same experiment
using PETRARCH in our framework and calculate precision. We vary the number of top
N recommended actors. Figure 7.7 shows the comparison. It shows that actor recommend-
dation with PETRARCH coding has higher precision than BBN ACCENT coding. BBN
ACCENT cannot separate political and non-political actors. If it does not find any actor
in the dictionary, it generates an empty event. On the other hand, PETRARCH separates
political and non-political events with actors. For a political event, if it does not find any
actor in the dictionary, it catches that event with empty actors.

![Graph showing comparison between PETR and BBN for Top N actors recommendation](image)

**Figure 7.7. Baseline comparison with actor detection coding**

**Table 7.2. List of recommended actors with their roles**

<table>
<thead>
<tr>
<th>Actor</th>
<th>Top 3 roles</th>
<th>Actor</th>
<th>Top 3 roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>DONALD TRUMP</td>
<td>USA, USAELI, LEG</td>
<td>SEDIQ SEDIQQI</td>
<td>AFG, PPL, UAF</td>
</tr>
<tr>
<td>SEDIQ SEDIQQI</td>
<td>UAF, AFG, PPL</td>
<td>DONALD TRUMP</td>
<td>USA, USAELI, LEG</td>
</tr>
<tr>
<td>AMIR SHEIK SABAH AL-AHMAD</td>
<td>SAUMED, SAUGOV, KWTMEDGO</td>
<td>AMIR SHEIK SABAH AL-AHMAD</td>
<td>SAUMED, SAUMEDGOV, KWTMEDGOV</td>
</tr>
<tr>
<td>AMIR SHEIK SABAH AL-JABER AL-SABAH</td>
<td>SAUMED, SAUGOV, KWTMEDGOV</td>
<td>AMIR SHEIK SABAH AL-JABER AL-SABAH</td>
<td>SAUMED, SAUMEDGOV, KWTMEDGOV</td>
</tr>
<tr>
<td>WILLIAM HUTCHINSON</td>
<td>USA, MIL, REB</td>
<td>BUKOLA SARAKI</td>
<td>USAGOVLEG, GOV, NGAELIGOV</td>
</tr>
<tr>
<td>LYNNE O’DONNEL</td>
<td>AFG, AFGUAF, PPL</td>
<td>RODRIGO DUTERTE</td>
<td>PHILGOV, GOV, PHL</td>
</tr>
<tr>
<td>RODRIGO DUTERTE</td>
<td>PHILGOV, GOV, PHL</td>
<td>ANTONIO CARPIO</td>
<td>JUD, PHL, CRM</td>
</tr>
</tbody>
</table>
7.4.6 Experiment 3: Scalability Test

We used our own Spark-based cluster with streaming data. Table 7.3 shows the configuration of the cluster.

<table>
<thead>
<tr>
<th>Table 7.3. Spark Cluster Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Master node</td>
</tr>
<tr>
<td>Total slave node</td>
</tr>
<tr>
<td>vCPU (core) in each node</td>
</tr>
<tr>
<td>Memory in each node</td>
</tr>
<tr>
<td>Storage in each node</td>
</tr>
</tbody>
</table>

We are using CoreNLP which is computationally expensive [80]. Each CoreNLP instance runs inside a Spark worker node and extracts meta-data from news articles. Figure 7.8 shows average document processing time of 5776 news articles in Spark. The framework processes news articles quickly when we increase the number of worker nodes. Spark fetches the news articles at 10 minutes intervals, whereas the scraper sends news at 2 hours interval. For this test, we take 5776 news articles in a burst to observe actual performance. Normally, the scraper fetches 5000-6000 online news article at the beginning of a run. Later, it scrapes only
the updated news reports which are fewer than initial. We collect the first news batch for evaluating scalability. Spark internally creates micro-batches to process these 5776 articles. So, we take average processing time per article. If we have only one worker, then its average processing time is almost 0.376 sec which is reduced to 0.047 when the total number of worker reaches to 10 (maximum load). It scales up almost linearly.
CHAPTER 8
CONCLUSION AND FUTURE WORK

In this chapter, we will draw a conclusion and discuss the future scope of our works. We will start with our real-time anomaly detection frameworks and later we will focus on our political textual mining frameworks.

8.1 Online Anomaly Detection Framework

We presented a real-time anomaly detection framework based on Apache Spark. Powered by Spark’s in-memory computing, parallel processing and fault tolerance, our framework can process large amounts of stream data in real time. We have also experimentally shown that Spark outperforms Storm substantially because of its in-memory computation and quick data processing. Moreover, our real-time framework can be integrated with VMware’s dynamic resource management system as well as various other monitoring systems to measure operational performance. Another important use case would be the analysis of sensor data from embedded devices.

In the future, we will aspire to implement other machine learning algorithms in our framework and compare them. We will explore the feasibility of incorporating other performance metrics, namely, memory usage, storage statistics, etc., and perform a correlation analysis to detect anomalies without any degradation of accuracy or performance. Moreover, the training data may contain noise. We will build an ensemble-based model to minimize the noise effect during training.

Furthermore, we may dig the root cause analysis of anomalies of heterogeneous data which come from a variety of sources and in a variety of formats. For example, unstructured data from sensor devices may need to be combined with structured data. From a data center perspective, resource usage data from different system components, e.g., CPU and memory,
differ in format and semantics. So, we can design a framework that can reconcile these two different sources and also easily accommodate itself to higher levels of heterogeneity.

8.2 Statistical Real-time Anomaly Detection Framework

We have developed a novel statistical real-time anomaly detection framework with Apache Spark that uses a Chi-square test as a distance metric. We showed online buffer-based technique and also a cluster-based technique to identify anomaly and also showed the comparison. Our cluster-based framework segments performance data streams of VMware virtual machines into windows of variable lengths and performs window-based comparisons against a trained model of benign windows. Our experiments confirm that it is capable of detecting anomalous CPU and memory utilizations of VMware virtual machines that span across multiple windows collected at different time intervals. Our framework is generic and can be adapted to a wide array of use cases.

In the future, we intend to apply it to a complex heterogeneous streaming environment where each feature of the heterogeneous data interrelated and we can explore either statistical approaches or graphical models to capture anomaly.

8.3 Spark-based Event Coding

We have presented a novel distributed framework for political event coding. Our framework has a complete workflow of extracting metadata using CoreNLP, storing them into MongoDB, and feeding them to PETRARCH to generate events. All of these are run in parallel and our framework reduces the time substantially.

In the future, we will improve our framework to code event data in several non-English languages. Moreover, we can use the stored metadata to automatically update actor dictionary using a novel class detection technique. Furthermore, we can extend our work to identify news location using machine learning techniques.
8.4 Near Real-time Atrocity Event Coding

We have addressed the problem of near real-time atrocity event coding. We have proposed a Spark-based framework with supervised classification techniques that work on news articles collected on a periodic basis to generate event data about atrocities. Our experimental results resemble similarity with the level of human coding accuracy.

In the future, we expand our framework to code events for other domains like sociology, international politics, etc.

8.5 Automatic Political Actor Recommendation in Real-time

Dictionaries of political actors are an integral component of political event data generation. We have addressed the problem of detecting and recommending new political actors and their roles in real-time. We have proposed a Spark-based framework with unsupervised ranking techniques with new actor aliases grouping that works on news articles collected on a periodic basis to recommend new actors. Our experimental results evaluate the framework’s performance.

Currently, we limit ourselves to finding new political actors but this approach can be extended to recommend new political actions in CAMEO verb dictionary which narrates political interactions among actors. Moreover, an existing political actor’s role can change over time. In the future, we will use our framework to update the CAMEO dictionaries to capture new Action/Verb sequences and also the new roles of actors. In addition, we will extend this to build CAMEO dictionaries for other languages (e.g., Spanish, Arabic).
REFERENCES


Masud, M. M., C. Woolam, J. Gao, L. Khan, J. Han, K. W. Hamlen, and N. C. Oza (2012). Facing the reality of data stream classification: coping with scarcity of labeled data. *Knowledge and information systems 33*(1), 213–244.


M Solaimani received his B.Sc. in Computer Science & Engineering from Bangladesh University of Engineering & Technology. He is a Ph.D. student in the Department of Computer Science at The University of Texas at Dallas. He has been working in the Big Data Analytics and Management Lab as a Research Assistant since Fall 2012. His research interests include anomaly detection using Big data frameworks, forecasting political unrest, atrocity data mining, etc. He developed real-time anomaly detection frameworks for a VW-ware-based data center to monitor its resource over usages. These frameworks were developed with statistical and machine learning techniques and also cutting-edge Big data tools like Apache Spark, Storm, MapReduce, Kafka, etc. Currently, he has developed a real-time Spark-based distributed framework using Stanford CoreNLP and PETRARCH for political event coding. All of his research works are funded by prestigious Government agencies like Sandia National Lab, USA, National Science Foundation, USA, etc.

In addition to working as a Research Assistant, he also has industrial experience through his internship at Reliance Jio Infocomm, USA and NEC Laboratories America, Inc.
CURRICULUM VITAE

M Solaimani
October 24, 2017

Contact Information:
Department of Computer Science Voice: (469) 358-9547
The University of Texas at Dallas Email: mxs121731@utdallas.edu
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Educational History:
B.S., Computer Science and Engineering, Bangladesh University of Engineering & Technology, 2007
M.S., Computer Science, University of Texas at Dallas, 2015
Ph.D. candidate, University of Texas at Dallas, 2017

Design and Development of Real-time Big Data Analytics Framework
Ph.D. Dissertation
Computer Science Department, UTD
Advisors: Dr. Latifur Khan

Publications:
"Designing a real-time log analysis framework” USA, Patent No: 16055 with NEC Laboratories America, Inc
“Online Anomaly Detection for Multi-source VMware Using a Distributed Streaming Framework,” Software:Practice and Experience
“Statistical Technique for Online Anomaly Detection Using Spark Over Heterogeneous Data from Multi-source VMware Performance Data,” IEEE BigData 2014, Washington DC, USA.
“Spark-based Anomaly Detection Over Multi-source VMware Performance Data In Real-time,” IEEE SSCI 2014, Orlando, Florida, USA.
“Host-Based Anomalous Behavior Detection Using Cluster-Level Markov Networks,” IFIP WG 11.9 International Conference on Digital Forensics held January 8-10, 2014 in Vienna, Austria.
“RFID Based Biometrics Analysis Over Cloud,” *Workshop on Data Science for Social Good, KDD 2014, NY, USA.*

“Host-Based Anomaly Detection Using Learning Techniques,” *ICDM Workshops 2013, Dallas, TX, USA.*

**Employment History:**

Research Assistant at University of Texas at Dallas, USA. August 16, 2017 - present.


Teaching Assistant at University of Texas at Dallas, USA. September 1, 2016 - May 15, 2017.


Research Assistant at University of Texas at Dallas, USA. September 1, 2015 - May 12, 2016.

Big Data Intern at Reliance Jio Infocomm USA, Inc. May 12, 2015 - August 21, 2015.

Research Assistant at University of Texas at Dallas, USA. September 1, 2013 - May 12, 2015.


Senior System Engineer at Bangla Phone Ltd., Bangladesh. August 1, 2007 - December 31, 2009.

**Professional Recognitions and Honors:**

ECS Graduate Fellowship, UTD, 2017

ECS Graduate Fellowship, UTD, 2016

Board Scholarship of Peoples Republic of Bangladesh, 2002

**Professional Memberships:**

Institute of Electrical and Electronics Engineers (IEEE), 2013–present