# AN APPROACH FOR SCALABLE FIRST-ORDER RULE LEARNING ON TWITTER DATA

A Dissertation
IN
Computer Science
and
Telecommunications and Computer Networking

Presented to the Faculty of the University
of Missouri–Kansas City in partial fulfillment of
the requirements for the degree

DOCTOR OF PHILOSOPHY

by
MONICA SENAPATI

B. S., College of Engineering and Technology, Bhubaneswar, India, 2014

Kansas City, Missouri
2021

AN APPROACH FOR SCALABLE FIRST-ORDER RULE LEARNING ON TWITTER DATA

Monica Senapati, Candidate for the Doctor of Philosophy Degree

University of Missouri–Kansas City, 2021

## ABSTRACT

Scalable Rule Learning (SRLearn) is a scalable divide-and-conquer approach with graph-based modeling of social media data, to scale up first-order rule learning through Markov Logic Networks on a commodity cluster on large scale Twitter data. SRLearn takes advantage of distributed systems to partition large-scale data into smaller but meaningful partitions based on user interaction and incorporates a gradient boosting approach with a tool called BoostSRL for first-order rule mining. We show how this scalable solution on first order predicates is more accurate and efficient than existing systems, such as ProbKB (a scalable system to construct probabilistic knowledge base) and XGBoost (extreme gradient boosting) on relational data.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Graduate Studies, have examined a dissertation titled "An Approach for Scalable First-Order Rule Learning on Twitter data," presented by Monica Senapati, candidate for the Doctor of Philosophy degree, and hereby certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Praveen Rao, Ph.D., Committee Chair
Department of Computer Science & Electrical Engineering

Baek-Young Choi, Ph.D., Co-Discipline Advisor
Department of Computer Science & Electrical Engineering

Ghulam Chaudhry, Ph.D.
Department of Computer Science & Electrical Engineering

Yugyung Lee, Ph.D.
Department of Computer Science & Electrical Engineering

Lein Harn, Ph.D.
Department of Computer Science & Electrical Engineering

CONTENTS

ABSTRACT . . . . . iii

ILLUSTRATIONS . . . . . vii

TABLES . . . . . x

ACKNOWLEDGEMENTS . . . . . xii

Chapter

1 INTRODUCTION . . . . . 1

2 BACKGROUND . . . . . 8

2.1 Apache Spark . . . . . 8

2.2 Hadoop Distributed File System . . . . . 9

2.3 First-Order Logic . . . . . 11

2.4 Markov Logic Networks . . . . . 13

2.5 Relational Dependency Networks . . . . . 15

2.6 Statistical Relational Learning . . . . . 19

3 RELATED WORK . . . . . 25

3.1 Rule Mining Techniques . . . . . 26

3.2 Scalable Tools for Rule Mining . . . . . 31

4 SRLearn . . . . . 33

4.1 Ground Predicates from Tweets . . . . . 34

4.2 Ground Predicate Graph . . . . . 38

4.3 User-centric Graph . . . . . 41
4.4 Graph Partitioning . . . . . 46
4.5 Regroup Ground Predicates and Rule Mining . . . . . 49
5 EXPERIMENTATION AND EVALUATION . . . . . 59
5.1 Hardware Environment . . . . . 59
5.2 Datasets . . . . . 61
5.3 Evaluation and Results . . . . . 62
6 CONCLUSION AND FUTURE WORK . . . . . 92
REFERENCE LIST . . . . . 95
VITA . . . . . 103

ILLUSTRATIONS


| Figure | | Page |
|---|---|---|
| 1 | Elon Musk's Twitter account hacked for Bitcoin scam | 3 |
| 2 | Roblox hacked to spread political propaganda | 4 |
| 3 | First-order logic fundamentals | 13 |
| 4 | First-order logic fundamentals | 15 |
| 5 | Example (a) data graph and (b) model graph | 17 |
| 6 | SRLearn architecture | 34 |
| 7 | Example of ground predicate graph | 39 |
| 8 | Example of user-centric graph | 43 |
| 9 | Distribution of ground predicates with varying preconfiguration on 32 partitions | 49 |
| 10 | Distribution of vertices (user IDs) with varying preconfiguration on 32 partitions | 50 |
| 11 | Distribution of grouped predicates, i.e. partitions amongst the nodes in the cluster | 52 |
| 12 | Distribution of ground predicates with varying preconfiguration on 32 partitions (2M tweets) | 64 |
| 13 | Distribution of vertices (user IDs) with varying preconfiguration on 32 partitions (2M tweets) | 65 |

14 Distribution of ground predicates with varying preconfiguration on 64 partitions (2M tweets) . . . . . 66
15 Distribution of vertices (user IDs) with varying preconfiguration on 64 partitions (2M tweets) . . . . . 67
16 Distribution of ground predicates with varying preconfiguration on 32 partitions (4M tweets) . . . . . 67
17 Distribution of vertices (user IDs) with varying preconfiguration on 32 partitions (4M tweets) . . . . . 68
18 Distribution of ground predicates with varying preconfiguration on 64 partitions (4M tweets) . . . . . 69
19 Distribution of vertices (user IDs) with varying preconfiguration on 64 partitions (4M tweets) . . . . . 70
20 Quality of graph partitioning by KaHIP . . . . . 70
21 AUC PR curve for 2M 32 partitions . . . . . 82
22 AUC PR curve for 2M 64 partitions . . . . . 82
23 AUC PR curve for 3M 32 partitions . . . . . 83
24 AUC PR curve for 3M 64 partitions . . . . . 83
25 AUC PR curve for 4M 32 partitions . . . . . 84
26 AUC PR curve for 4M 64 partitions . . . . . 84
27 AUC ROC curve for 2M 32 partitions . . . . . 85
28 AUC ROC curve for 2M 64 partitions . . . . . 85
29 AUC ROC curve for 3M 32 partitions . . . . . 86

30 AUC ROC curve for 3M 64 partitions . . . . . 86
31 AUC ROC curve for 4M 32 partitions . . . . . 87
32 AUC ROC curve for 4M 64 partitions . . . . . 87

# TABLES


| Tables | | Page |
|---|---|---|
| 1 | Example of a first-order knowledge base and MLN. `Fr()` is short for `Friends()`, `Sm()` for `Smokes()`, and `Ca()` for `Cancer()` | 14 |
| 2 | Set of first-order predicates generated from Twitter data based on CWA (denoted by *) and OWA | 35 |
| 3 | Social relationships between users | 44 |
| 4 | Runtime for different preconfiguration on 2 million tweets' graph partitioned into 32 partitions on 16 nodes | 48 |
| 5 | Number of ground predicates in each dataset | 62 |
| 6 | Graph dimensions for evidence datasets considered | 63 |
| 7 | Runtime for different stages of SRLearn prior to structure learning | 71 |
| 8 | Sample of rules learnt by SRLearn coupled with Alchemy | 71 |
| 9 | Class imbalance, i.e. ratio of negatives to positives in 2M, 3M and 4M datasets | 72 |
| 10 | Time taken to train models by BoostSRL for target predicate *isPossiblySensitive* | 73 |
| 11 | AUC ROC for both algorithms of BoostSRL for target predicate *isPossiblySensitive* | 74 |
| 12 | Graph dimensions for evidence datasets considered | 76 |

13 The schema closure graph used for rule mining . . . . . 77
14 Spark-submit configuration . . . . . 80
15 XGBoost Classifier parameters . . . . . 81
16 Time taken to learn models for XGBoost and SRLearn . . . . . 81
17 Comparision of total runtime for SRLearn and XGBoost . . . . . 88
18 AUC PR for target *isPossiblySensitive*: SRLearn vs Competitive methodologies . . . . . 88
19 AUC ROC for target *isPossiblySensitive*: SRLearn vs Competitive methodologies . . . . . 89
20 Time taken to train models by SRLearn for target predicate *malicious* . . 90
21 AUC PR for target *malicious*: SRLearn vs Competitive methodologies . . 90
22 AUC ROC for target *malicious*: SRLearn vs Competitive methodologies . 91

ACKNOWLEDGEMENTS

The journey at UMKC, as well as the path to the dissertation, has been one of the most eventful 5 years of my life. First and foremost, I would like to thank my research advisor *Dr. Praveen Rao*, because of whom this dissertation came into being. Thank you for believing in me and mentoring me to be a better person. I sincerely appreciate all the time and effort you put towards me, to help complete this dissertation; your constant encouragement, especially when I needed it. I am deeply honored to have gotten the opportunity to work with you, as well as be part of the Big Data Lab.

I would like to thank all the committee members for the research experience. *Dr. Ghulam Chaudhry*, thank you for the support, both for research as well as teaching opportunities during my time at UMKC. I greatly appreciate it. Thank you *Dr. Baek-Young Choi*, *Dr. Yugyung Lee* and *Dr. Lein Harn* for being part of the committee, for your time and advising, as well as for teaching me.

This research has been possible due to the gracious financial support provided by several organizations at different stages. I am thankful for the financial support provided by the UMKC SGS Travel Grant, UMKC SAFC Travel Grant, and UMKC Women's Council Graduate Assistance Fund (2018-2019) that helped me present my work in front of an international audience at the 35th IEEE International Conference on Data Engineering (ICDE 2019) at Macau, China. I would also like to extend my gratitude to Air Force Research Laboratory (AFRL) Visiting Faculty Research Program and Extension Grant Program, NSF Grant No. IIP-1841752, UMKC CSEE matching funds for Advance

Big Data Project (Provost's Strategic Initiative), UMKC Funding For Excellence (FFE), and the Mahatma Gandhi Scholarship by the School of Graduate Studies (SGS) for the valuable monetary support for this research.

As an international student, living miles away from home to pursue my degree had its fair share of challenges. Therefore, having a good support system is of utmost importance. I was fortunate to have had the opportunity to work with very smart and motivated colleagues, whose friendship I shall cherish forever. To name a few, *Daniel E. Lopez Barron*, *Dr. Khulud Alsultan* and *Nouf Alrasheed* have been the most supportive and encouraging group of friends one could ask for. It has been a great honor to have worked with them. A big thank you to *Dr. Mayanka Chandrashekar* for being a constant pillar of support, wisdom, and inspiration, as well as next to a family in this country.

Most important of all, I am forever grateful and indebted to my family, because of whom, I got the best of everything. I am thankful to God to have blessed me with a family, who truly understands the value of quality education for women. I am forever thankful to my parents *Jashodhara Senapati* and *Suvendu Senapati*, to have raised me and my sister into strong, independent women, who face the challenges with a brave face and dignity. A very special shout out to my sister *Dr. Samikshya Senapati*, who always stood by me. She deserves a special mention for playing the most critical role in helping me keep calm. Last but not the least, my deepest gratitude to my husband *Sambit Tripathi*, for having stood by me both virtually and in-person during my times of highs and lows. I am grateful for his never-ending support, even when he has his classes and research to take care of. Thank you for believing in me. Always. Also, kudos to *Sambit Tripathi* for

surviving his wife's dissertation and now I hope to find the strength to survive his.

I would like to dedicate this dissertation to my grandparents *Benudhar Senapati* and *Pramila Senapati*. They have practically raised me to be the woman I am today. Even though they are not around any longer, their values and teachings are instilled in me for eternity. Being known as their granddaughter has and shall always be my biggest honor. I would also like to dedicate this to my great maternal grandmother *Satyabhama Senapati* and my maternal grandfather *Prafula Senapati*, for being an unwavering source of love and support.

# CHAPTER 1

# INTRODUCTION

Social media has been on the rise for the past few decades, connecting people from all corners of the world. With platforms like Facebook[1], Instagram[2], Twitter[3], LinkedIn[4] etc., people are now spoilt with multiple options on how we choose to communicate with the world. They rely on these platforms not just to communicate with friends and family, both near and far, but also to reach out to a much wider audience. People rely on these channels for a quick mode of communication in terms of conveying personal life updates, pictures or videos from the latest vacation, political debates on the different world problems, send out news update, even send out distress signals in hopes to find help, or even reach out to a larger market for their small businesses. Over the course of time social media has evolved along with how people choose to communicate. This exchange of messages can be in various formats: text messages, shared URLs, images, etc.

Despite it's numerous advantages, social media also evolved into an unreliable place on the web. Due to constant attacks on social media, the spread of wrong information has become very common. Cyber criminals leverage personal information of users, such as contacts, locations, personal and/or business activities to can either target specific advertisements towards users or lead to criminal activities in virtual and/or real world. [32]

[1] www.facebook.com
[2] www.instagram.com
[3] www.twitter.com
[4] www.linked.com

Furthermore, identity theft is also on the rise where cyber criminals hack into people's accounts. A Stratecast study showed that 22% of social media users have been victims of security related attacks. Pony botnet, a controller that is a code for credential theft in Windows machines, affected Facebook, Google, Yahoo and other social media users, stealing more than 2 million users' passwords. Facebook estimated that it sees 50-100 million fake duplicate accounts from it's monthly active users.

Another kind of social media attack is the "false flag" attack that tricks users into revealing private and personal information about themselves. Upon initiating a password change, the cyber criminals can immediately access the security information of these users.

It has been time and again seen that misinformation has the potential to create disruption in an otherwise peaceful society. Continuous investigation is carried out to understand the motives behind these attacks as well. One of the many trends seen recently have been the increase in attacks on celebrity and government officials' social media accounts by 43% since 2019. [49]. The cyber criminals mostly intend to spread malware and misinformation through these attacks. In May 2016, the attack on LinkedIn exposed 117 million credentials. Vevo encountered a phishing attack in 2017 and 3.12 terabytes of sensitive company data was leaked. Slack was hacked in 2017 because of which 0.5 million in Ether coins were stolen. Twitter was hacked in July 2020 and some of the most influential accounts were used in a bitcoin theft.

Attackers get a financial gain by targeting high-profile individuals. In July 2020, Twitter employees were targeted through a phishing attack, because of which accounts of


Figure 1: Elon Musk's Twitter account hacked for Bitcoin scam
https://www.theverge.com/2020/7/15/21326200/elon-musk-bill-gates-twitter-hack-bitcoin-scam-compromised

influential people like Barack Obama and Elon Musk were compromised to be used for bitcoin theft.(Fig. 1).

Because of the global reach of social media, it cybercriminals benefit from it by targeting a massive audience at once. In early 2020, Roblox[5] was hacked and the attackers took over gaming accounts to spread pro-Trump election propaganda. (Fig. 2). Similary Reddit[6] was under a similar attack in 2020, leading to spread of misinformation about the US presidential election.

With the global outreach, social media platforms also have the potential to become a new battleground for cyber warfare. Nation-states and state-backed cybercriminals exploit social media to influence the audience and disrupt political peace in foreign

[5]https://www.roblox.com/
[6]https://www.reddit.com/

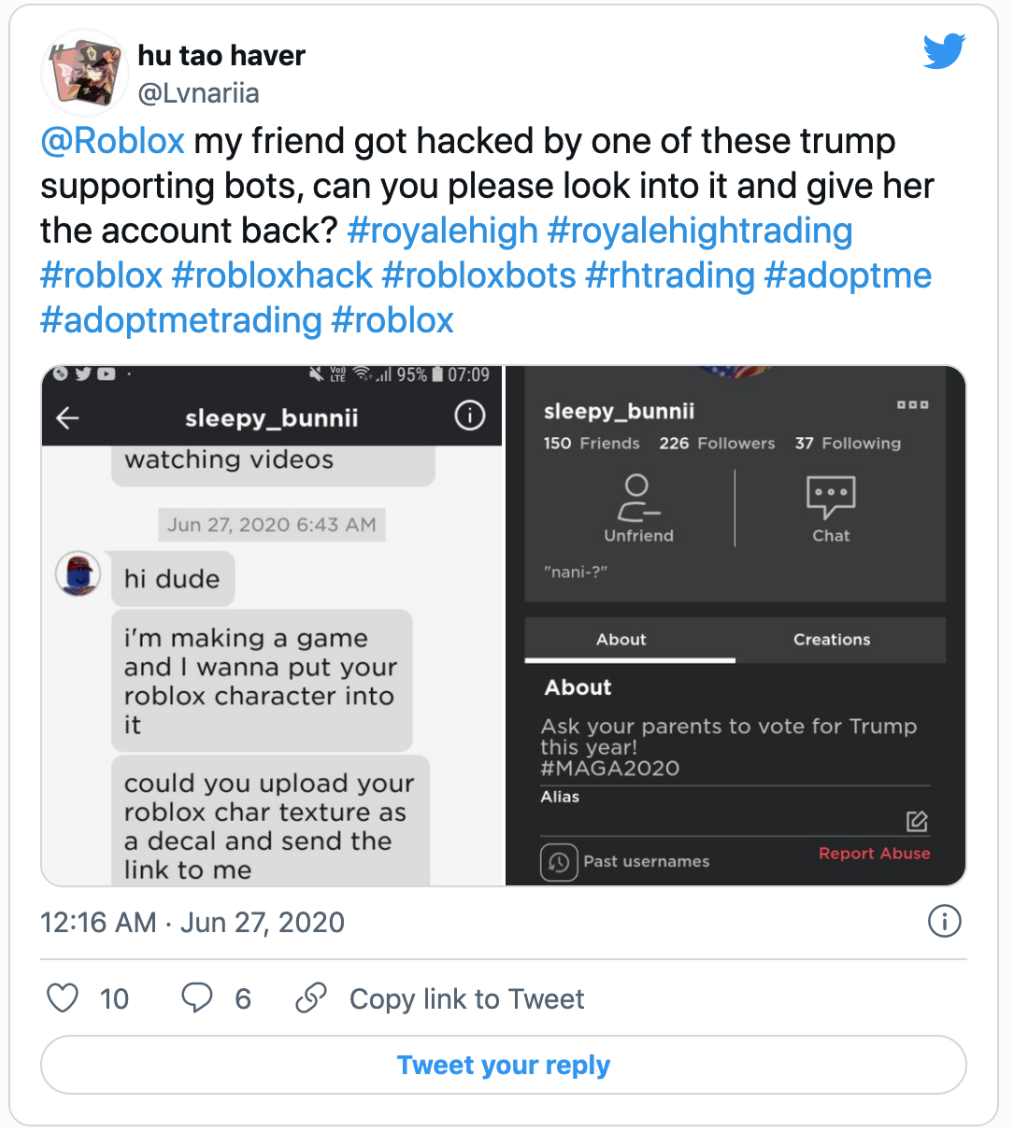



Figure 2: Roblox hacked to spread political propaganda
https://www.forbes.com/sites/davidthier/2020/07/05/hackers-are-spreading-trump-propaganda-through-roblox/?sh=537dac0c6aa7

governments. US government threat intelligence agencies and organizations reported evidence of Russia interfering with the 2016 US presidential election [49] by spreading misinformation. According to Microsoft, Russia state-backed threat actors are actively engaging in attacks against campaigns and parties for the 2020 election as well. Vice Chairman of the Senate Select Committee on Intelligence, Mark Warner, stated at a cyber security conference that Russia tried to exacerbate the political parties' divisions to create disparity by creating fake accounts on social media. Another example of cyber warfare was in Iran [50],where the killing of Major General Qasem Soleimani, led to sponsoring social media disinformation campaigns. In December 2019, the National Cybersecurity Authority of Saudi Arabia declared that Iran was employing Dustman, a new data-wiping malware, against Bapco, Bahrain's national oil company. Dustman can disrupt computer processes and overwrite data on targeted computers. Iran's abilities to research and target energy companies in the Middle East, as well as develop new wiper malware, demonstrate Iran can conduct sophisticated cyber attacks.

Thus, we have seen time and again that the social media platforms are constantly under attack, it leads to question the truthfulness behind the posts on the platform. One can also question the path in which this information can travel to reach the targeted audiences. This leads to a system that is capable of navigating the intricacies of social media to identify such potential threats, before they are conceived in reality. This is a big umbrella problem, and this dissertation aims to make a small but significant contribution in addressing the many challenges of social media's truthfulness.

Furthermore, over billions of users on these platforms, it leads to the increase in

the amount of data being generated each minute. With the continuous attack on social media users, as well as the constant scaling up of data in consideration, this creates a need for a scalable system that can process the ever increasing size data. Therefore, we start with one such popular social media, Twitter to address this issue.

In statistical relational learning, a Markov logic network (MLN) [42] is regarded as one of the most flexible representations as it combines first-order logic and probabilistic graphical models. First-order logic enables the complexity of the data to be modeled; and probability allows expressing the uncertainty in the data. An MLN is a knowledge base (KB) and can model the complex, diverse nature of Twitter posts (a.k.a. tweets) containing 100+ attributes. Once the KB is learned, which includes the first-order rules (or formulas) and their weights, probabilistic inference can be conducted on the KB to reason about the content in the posts and users' behavior. For example, one can compute the marginal probability of a URL being malicious, a tweet being sensitive, a user account being an adversary or social bot [46], and so on [45]

Therefore, taking advantage of the popular big data processing tools as well as the prior work done in the field of first order rule learning, we propose SRLearn(**S**calable **R**ule **Learn**ing) [45] a scalable scales up first order rule learning on large scale Twitter[7] data. The novelty of our approach comes from the way we model the large scale data based on user interaction on social media. We rely on this logic to create small but meaningful partitions with the intent of maximum retention of data, implying increase in chances of learning interesting rules from the data.

[7]www.twitter.com

SRLearn comprises of the following steps:

- divide-and-conquer approach to first-order rule learning through parallel and distributed graph-based modelling of users and the related ground predicates,
- parallel graph-partitioning to create a balanced set of partitions in order to minimize the chances of missing out on interesting rules, and
- employing first order rule learning through BoostSRL [27] on these partitions.

The remaining chapters describe in details the evolution as well as the implementation of the system in the following order.

- Data collection: Information regarding the size, domain and additional details,
- Evidence DB generation: Modification of Twitter data into a knowledge base to be used by SRLearn,
- Experimental setup: System set up for our proposed approach along with the popular rule learning techniques,
- SRLearn: Our proposed approach with rule learning,
- Rule learning in the absence of SRLearn,
- XGBoost: A very popular machine learning approach suggested in literature that relies on gradient boosting as well as supports distributed systems, and
- ProbKB: A scalable rule mining method that also supports distributed systems.

# CHAPTER 2

# BACKGROUND

In this chapter, we provide a background on the big data tools that enable large-scale data processing, the concepts behind first-order logic, Markov Logic, and Markov Logic Networks.

## 2.1 Apache Spark

The necessity to analyze the ginormous amounts of data consumed and produced by industries and research alike led to the development of various big data tools catering to cluster programming needs. Apache Spark [54] is one such tool that aims at providing a unified platform for large-scale data processing. Using a common engine, Spark caters to the diverse workloads such as streaming services through Spark Streaming, SQL queries through Spark SQL, machine learning capabilities through Spark MLlib, and graph processing using GraphX, making it easier and efficient to use. Users are thus able to take advantage of distributed storage like Hadoop, distributed systems management tools like Mesos, NoSQL databases like Cassandra, or even relational database management systems like MySQL from a single API.

As opposed to prior systems of MapReduce [7], which requires data to be brought

in and out of the storage to be processed, Spark provides an increased run-time, by allowing processes to manipulate data in memory. Along with the unification of multiple models, Spark allows writing applications through programming languages like Java, Scala, Python, R, and SQL.

Each Spark application consists of a driver program that begins with the main function, triggering parallel processing on a cluster. Spark provides data abstraction mainly in the form of *resilient distributed dataset*(RDD), *dataframes*, and shared variables such as *broadcast variables*, that are shipped across tasks in memory and *accumulators*, like counters and sums. RDDs are a collection of elements distributed across nodes on a cluster. Spark follows lazy computation technique, where the RDDs are transformed from one form to another using *transformation functions*, but the RDD that users want to work with, is created when the *action function* is called. DataFrames are also a distributed collection of data that are equivalent to relational databases or data frames in Python. Thus this data abstraction enables to run SQL-like queries through Spark SQL in a distributed environment.

## 2.2 Hadoop Distributed File System

Apache Hadoop [17] is an open-source distributed framework that allows the processing of large data across a cluster and is capable of handling application failures. The different project components of Hadoop are as follows.

- HDFS: Distributed file system
- MapReduce: Distributed computation network

- HBase: Column-oriented distributed database system
- Pig Latin: Data querying language and parallel execution framework
- Hive: Data warehousing tool
- ZooKeeper: Distributed coordination service
- Chukwa: Data management collection system
- Avro: Data serialization system

Hadoop Distributed File System [47](HDFS) is a component of Apache Hadoop originally developed by Yahoo!. As the name suggests, it is a distributed UNIX-like file system providing reliable storage. HDFS is capable of scaling from a single computer to thousands of computers, enabling different industrial use cases and research, without worrying about node failure or data loss.

HDFS stores data in the form of metadata on a single NameNode, and the actual data on one or many servers called as DataNodes. To ensure the reliability of the data, the DataNodes contain replications of the application data. The aggregated file content in HDFS usually is split into blocks of data, each of which is stored in DataNode. The NameNode in the HDFS architecture contains information pertaining to the namespace tree and the mapping of file blocks onto the DataNodes. The NameNode is made aware of all the participating DataNodes at the time of startup through a *handshake* mechanism. It is also aware of the DataNodes in operation at any given point of time, through the *heartbeats* that the DataNodes send to the NameNode. If the NameNode does not receive

a *heartbeat* from DataNode for ten minutes, it considers this DataNode to be no longer in service and initiates the creation of new replicas of the blocks that were on the said unavailable DataNode.

## 2.3 First-Order Logic

First-order logic [13] [42] evolved from propositional logic as a more expressive mechanism of knowledge representation to describe real-world entities and their relationship with each other. As opposed to propositional logic, first-order logic is not restricted to the facts but also assumes the following.

- **Objects**: Represent *constants* (e.g. people: `Alice`, `Bob`, `Charles`) and *variables* (e.g. `x`, `y`, `z`) in a domain.
- **Relations**: Also known as *predicates*, represent relationships among objects or their attributes (e.g. `Father`, `Friend`, `Artist`).
- **Function**: Represent mappings of one object onto another or other tuples of objects (e.g. `FatherOf`, `RelatedTo`).

An *atomic* sentence is a fundamental sentence in first-order logic, formed using predicate and one or a group of objects enclosed within parenthesis. Atomic sentence to represent the relation *Alice is a friend of Bob* is given by `Friend(Alice, Bob)`. Atomic sentence to represent the relation *Alice is an artist* is given by `Artist(Alice)`. Furthermore, a positive literal is an *atom* and a negative literal is a *negated atom*. A *ground predicate* is a predicate that does not have any object/term associated with it.

First-order formula is formed using the atomic sentences connected with logical connectives and/or quantifiers like *conjunction*, *disjunction*, *negation*, *implication*, *equivalence* and *universal and existential quantifiers*. They are constructed recursively. A set of first-order formulas is referred to as *first-order knowledge base* (KB). Fig. 3 shows an example of predicates, domain and the first-order KB that can be formulated from this predicates. A formula or a predicate can make a *closed-world assumption* (CWA), i.e. all ground predicates that are not known to be true are be false, or an *open-world assumption* (OWA), i.e. if unknown, then it may be assumed to be either true or false.

It is not possible to have a complete inference on first-order logic. Therefore, knowledge bases are usually constructed with only a desirable subset of first-order logic, in the form of Horn clauses. The first-order logic examples in Fig. 3 are some of the simplest examples and do not necessarily cover all possible use-cases in that domain. Often, it is not possible to come up with all possible formulas in a given domain, because a formula may be true for a subset of the data, and contradict completely for the remaining subsets. In other words, there exists no measure of gauging the truth of a formula in a given domain. Therefore, despite the flexibility and ease of expressiveness of first-order logic, its applicability is limited to practical AI applications. The solution to this problem is addressed through the probabilistic nature of the Markov Logic Networks described in the next section.

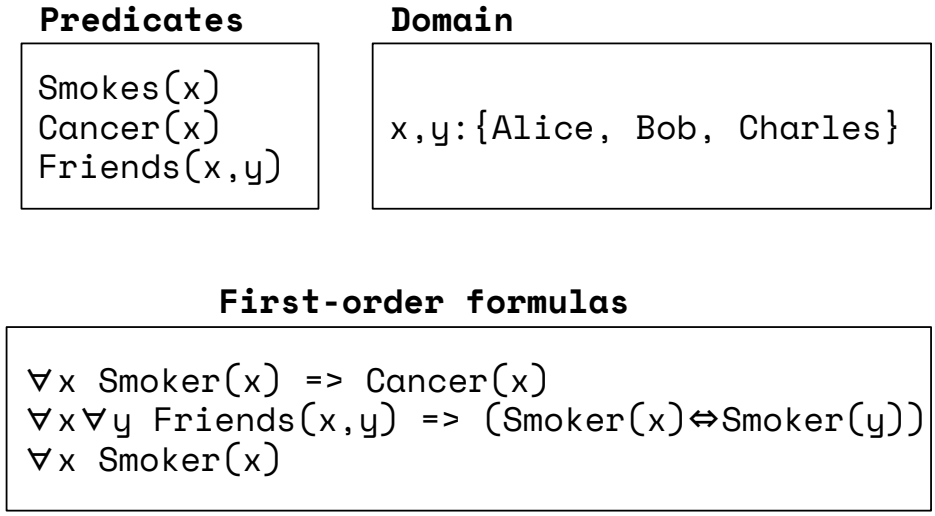



Figure 3: First-order logic fundamentals

### 2.4 Markov Logic Networks

While first-order KB can be understood as hard constraints in a domain, Markov Logic Networks (MLN) [42] provide a more practical representation of these constraints through a numeric weight. In other words, when a domain subset contradicts a given first-order formula, the formula becomes less probable. The more domain subsets that exist validating with the first-order formula, the more probable it becomes.

Richardson and Domingos [42], and Domingos and Lowd [9] define a Markov Logic Network as a first-order knowledge base with a weight attached to each formula (or clause). Together with a set of constants representing objects in the domain, it specifies a ground Markov network containing one feature for each possible grounding of a first-order formula in the KB, with the corresponding weight.

Table 1: Example of a first-order knowledge base and MLN. `Fr()` is short for `Friends()`, `Sm()` for `Smokes()`, and `Ca()` for `Cancer()`

| English | First-order logic | Weight |
|---|---|---|
| Friends of friends are friends | $\forall x \forall y \forall z Fr(x,y) \wedge Fr(y,z) \Rightarrow Fr(x,z)$ | 0.7 |
| Friendless people smoke | $\forall x(\neg(\exists y Fr(x,y)) \Rightarrow Sm(x))$ | 2.3 |
| Smoking causes cancer | $\forall x Sm(x) \Rightarrow Ca(x)$ | 1.5 |
| If two people are friends, either both smoke or neither does | $\forall x \forall y Fr(x,y) \Rightarrow (Sm(x) \Leftrightarrow Sm(y))$ | 1.1 |

**Definition 1.** A Markov logic network $L$ is a set of pairs ($F_i$,$w_i$), where $F_i$ is a formula in first-order logic and $w_i$ is a real number. Together with a finite set of constants $C = \{c_1, c_2, ..., c_{|C|}\}$, it defines a Markov network $M_{L,C}$ (Equations 1 and 2) as follows.

1. $M_{L,C}$ contains one binary node for each possible grounding of each predicate appearing in $L$. The value of the node is 1 if the ground atom is true, and 0 otherwise.

2. $M_{L,C}$ contains one feature for each possible grounding of each formula $F_i$ in $L$. The value of this feature is 1 if the ground formula is true, and 0 otherwise. The weight of the feature is the $w_i$ associated with $F_i$ in $L$.

An MLN can be viewed as a template for constructing Markov networks. Given different sets of constants, it will produce different networks, and these may be of widely varying size, but all will have certain regularities in structure and parameters, given by the MLN (e.g., all groundings of the same formula will have the same weight). We call each of these networks a ground Markov network to distinguish it from the first-order MLN. The graphical structure of $M_{L,C}$ follows from Definition 1: there is an edge between two nodes of $M_{L,C}$ iff the corresponding ground atoms appear together in at least one

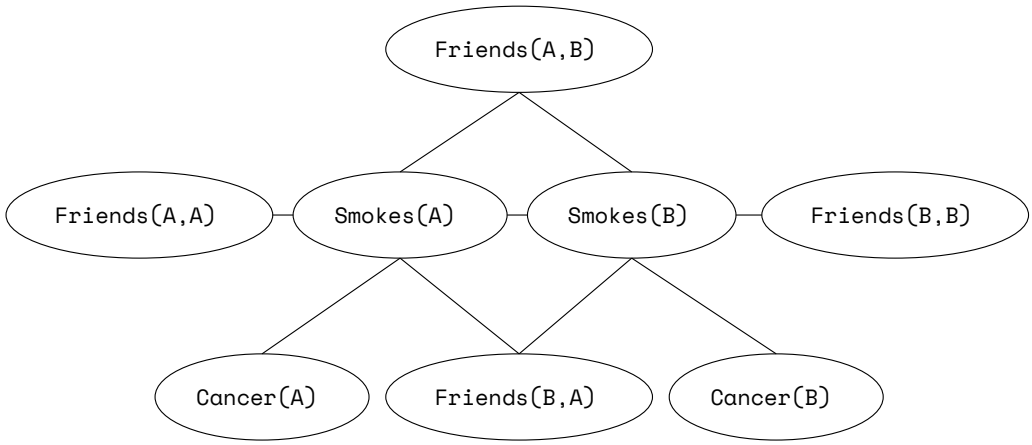



Figure 4: First-order logic fundamentals

grounding of one formula in $L$.

Fig. 4 shows the graph of the ground Markov network defined by the last two formulas in Table 1 and the constants `Anna` and `Bob`. Each node in this graph is a ground atom (e.g., `Friends(Anna,Bob)`). The graph contains an arc between each pair of atoms that appear together in some grounding of one of the formulas. $M_{L,C}$ can now be used to infer the probability that Anna and Bob are friends given their smoking habits, the probability that Bob has cancer given his friendship with Anna, and whether she has cancer, etc.

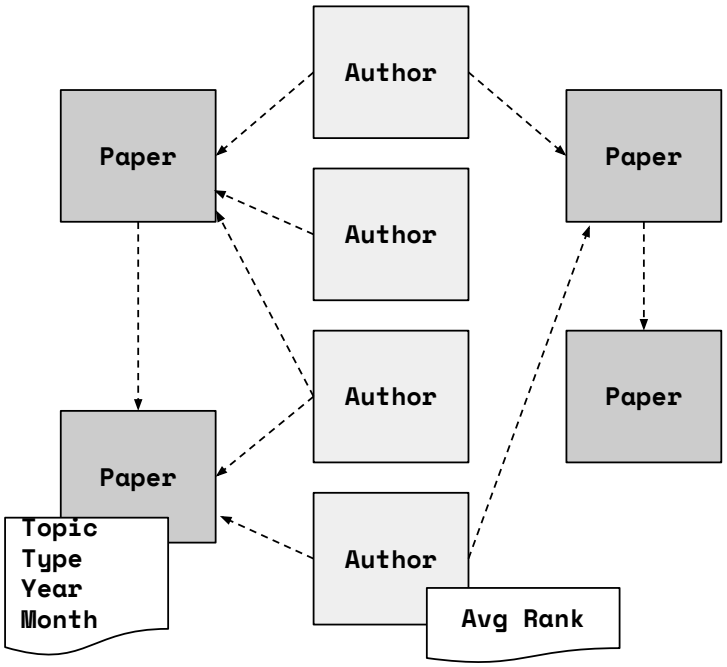
### 2.5 Relational Dependency Networks

Most relational data sets have instance dependencies. For example, in citation data [38] there are dependencies among the topics of a paper's references, and in genomic

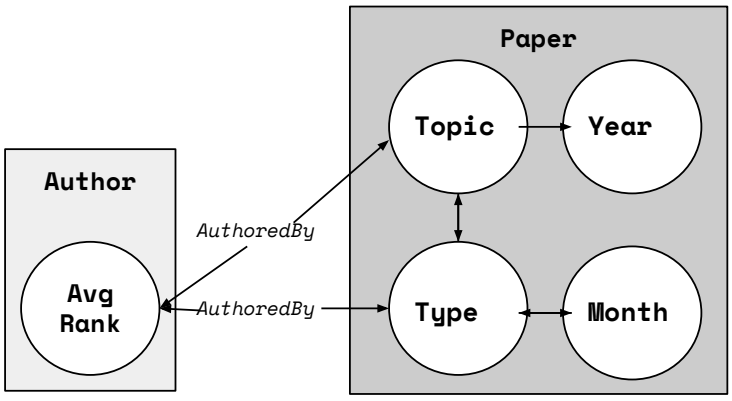
data, there are dependencies among the functions of interacting proteins. This is where the use of Relational Dependency Networks (RDNs) comes into the picture. RDNs [38] are graphical models that are capable of expressing and reasoning with such dependencies in relational data. RDNs can represent and reason with the cyclic dependencies required to express and exploit auto correlation dependencies. They also offer a relatively simple method for structure learning and parameter estimation, which results in models that are easier to understand and estimate.RDNs approximate the full joint distribution and thus are not guaranteed to specify a consistent probability distribution. The quality of approximation will be determined by the data available for learning.

Dependency networks [38] are particularly desirable for modeling relational data. First, learning a collection of conditional models offers significant efficiency gains over learning a full joint model. This is generally true, but it is even more pertinent to relational settings where the feature space is very large. Second, networks that are easy to interpret and understand aid analysts' assessment of the utility of the relational information. Third, the ability to represent cycles in a network facilitates reasoning with autocorrelation, a common characteristic of relational data. In addition, whereas the need for approximate inference is a disadvantage of DNs for propositional data, due to the complexity of relational model graphs in practice, all PRMs use approximate inference. RDNs are an extension of dependency networks [19] for relational data.

When modelling relational data, it has three associated graphs: the *data graph* $G_D$, the *model graph* $G_M$ and the *inference graph* $G_I$ First, the relational data set is represented as a typed, attributed data graph $G_D$ = ($V_D$,$E_D$). For example, consider the




(a)




(b)

Figure 5: Example (a) data graph and (b) model graph

data graph in Fig. 5a. The nodes $V_D$ represent objects in the data (e.g., authors, papers) and the edges $E_D$ represent relations among the objects (e.g. author-of, cities). Each node $v_i \in V_D$ and edge $e_j \in E_D$ is associated with a type, $T(v_i) = t_{v_i}$ and $T(e_j) = t_{e_j}$ (e.g., paper, cited-by). Each item type $t \in T$ has a number of associated attributes $X^t = (X^t{}_1, ..., X^t{}_m)$ (e.g., topic, year). Consequently, each object $v_i$ and link $e_j$ is associated with a set of attribute values determined by their type, $X_{v_i}^{t_{v_i}} = (X_{v_i}^{t_{v_i}}{}_1, ..., X_{v_i}^{t_{v_i}}{}_m)$ and $X_{e_j}^{t_{e_j}} = (X_{e_j}^{t_{e_j}}{}_1, ..., X_{e_j}^{t_{e_j}}{}_m)$.

The dependencies among attributes are represented in the model graph $G_M$ = ($V_M$,$E_M$) in Fig. 5b. Attributes of an item can depend probabilistically on other attributes of the same item, as well as on attributes of other related objects or links in $G_D$. For example, the topic of a paper may be influenced by the attributes of the authors that wrote the paper. The relations in $G_D$ are used to limit the search for possible statistical dependencies, thus they constrain the set of edges that can appear in $G_M$. In $G_M$, each item type is represented by a plate, and each attribute of each item type is represented as a node. Edges characterize the dependencies among the attributes at the type level. The representation uses a modified plate notation. Dependencies among attributes of the same object are represented by arcs within a rectangle; arcs that cross rectangle boundaries represent dependencies among attributes of related objects, with edge labels indicating the underlying relations. For example, *month*$_i$ depends on *type*$_i$, while *avgrank*$_j$ depends on the *type*$_k$ and *topic*$_k$ for all papers *k* written by author *j* in $G_D$.

The graphical representation illustrates the qualitative component ($G_D$) of the RDN. It does not depict the quantitative component ($P$) of the model, which consists

of the Conditional Probability Distributions (CPDs) that use aggregation functions. Although conditional independence is inferred using an undirected view of the graph, directed edges are useful for representing the set of variables in each CPD. For example, in Fig. 5b the CPD for *year* contains *topic* but the CPD for *topic* does not contain *year*. This represents any inconsistencies that result from the RDN learning technique.

A *consistent* RDN specifies a joint probability distribution $p(\mathbf{x})$ over the attribute values of a relational data set from which each CPD $\in P$ can be derived using the rules of probability. There is a direct correspondence between consistent RDNs and relational Markov Networks. It is similar to the correspondence between consistent DNs and Markov Networks [19], but the correspondence is defined with respect to the template model graphs $G_M$ and $G_D$.

## 2.6 Statistical Relational Learning

The vast majority of work in learning has focused on propositional data which consists of identically structured entities that are assumed to be independent. However, many real-world data sets are relational. Relational data consists of different types of entities where each entity is characterized with a different set of attributes. Relational data are more complex and better suited with our surroundings where examples are given as multiple related tables. The structure of relational data provides an opportunity for objects to carry additional information via their links and enables the model to show correlations among objects and their relationships. [25]

On one hand, statistical learning addresses the uncertainty in the data. On the

other hand, relational learning addresses the complexity of relations in domains. The two types of learning have been combined to be implemented with real-world data, and also due to the growth of research interest in these areas. Statistical relational learning (SRL) combines both types of learning. Two of the vital steps in this learning mechanism are *structure learning* and *parameter learning*. Structure learning learns the nodes and edges of a graphical structure, whereas parameter learning learns the prior and conditional probability distributions [34]. SRL models predominantly belong to the following categories.

- **Probabilisitic Relational Models (PRMs)**: PRMs [16] are a rich representation language and one of the successful models for statistical learning. PRMs extend the concept of Bayesian networks and that of objects to represent relational data logically with probabilistic semantics on the directed graphical models.

- **Relational Dependency Network (RDN)**: RDNs [38], an extension of Dependency Networks (DNs) [19] are a class of graphical models that approximate a joint distribution using a bidirected graph with conditional probability tables for variables. The RDNs are thus able to represent cyclic dependencies, employ simple methods for parameter estimation with efficient structure learning. The reason RDNs are popular is due to the use of pseudo-likelihood [1] learning algorithm that estimates an acceptable approximation of joint distribution.

- **Bayesian Logic Programming (BLP)**: Bayesian Logic programs [23] are a model based on Bayesian networks. BLPs use logic programming [35] to unify Bayesian networks with logic programming. This unification overcomes the propositional

character of Bayesian networks and logical programs. BLPs use Bayesian clauses that use a conditional probability table to present the distribution of the head of the clause conditional on its body and use combining rules to unite the information on a single literal that is the head of several clauses. BLPs are implemented in a software called BALIOS [23].

- **Markov Logic Networks (MLNs)**: As described in the prior section, syntactically MLNs extend first-order logic and put weight for each formula. Semantically, they can represent a probability distribution over possible worlds using formulas and their corresponding weights. [25]

One such type of statistical relational learning is through probabilistic graphical models like Bayesian Networks or Markov Logic Networks. Learning with Bayesian Networks involves a combination of Prolog language with Bayesian networks. [15] [24]. In order to avoid cycles in learning with Bayesian networks, Markov networks provide an alternative to use undirected graphical models. [16] In the following section we describe the details of structure learning of MLNs and RDNs.

### 2.6.1 Structure Learning in Markov Logic Networks (MLN)

In this subsection, we describe in detail the learning and inferencing techniques used in the case of Markov Logic Networks.

#### 2.6.1.1 Structure Learning

MLN structure learning [28] can start from an empty network or from an existing KB. Richardson and Domingos (2004) [42] and, Kok and Domingos (2005) [28] have found it useful to start by adding all unit clauses (single predicates) to the MLN. The weights of these captures (roughly speaking) the marginal distributions of the predicates, allowing the longer clauses to focus on modeling predicate dependencies. This phase is able to learn first-order formulas and not just horn clauses

Kok and Domingos (2005) [28] have defined the weighted pseudo-log-likelihood (WPLL) as

$$logP_w(X = x) = \sum_{r \in R} c_r \sum_{k=1}^{g_r} logP_w(X_{r,k} = x_{r,k} | MB_x(X_{r,k})) \tag{2.1}$$

where $R$ is the set of first-order predicates, $g_r$ is the number of groundings of first-order predicate $r$, and $x_{r,k}$ is the truth value (0 or 1) of the $k$th grounding of $r$. The choice of predicate weights $c_r$ depends on the user's goals.

MLNs use the close-world assumption [12] that if a ground atom is absent in the database, it is assumed to be false.

#### 2.6.1.2 Inference

Inference has two main phases in MLNs. In the first phase, a minimal subset of $G_I$ the ground Markov network is selected. Many predicates that are independent of the predicates of the query may be filtered in this phase. As a result inference carried out over a smaller Markov network. In the second phase inference is performed on the Markov network using Gibbs sampling [2] where the evidence nodes are observed and are set to

their values. Gibbs sampling first randomly initialize and orders unobserved variables in the network, i.e. $\{X_1 = x_1 ... X_n = x_n\}$, and then iterate through the variables. In step 1 a new value $x_1'$ for variable $X_1$ is sampled conditional on all the other variables $P(X_1|X_2 = x_2 ... X_n = x_n)$, and in step $i$ a new value $x_1'$ for variable $x_i$ is sampled conditional on all the other variables $P(X_i|X_1 = x_1' ... X_{i-1} = x_{i-1}', X_{i+1} = x_{i+1} ... X_n = x_n)$. . Conditional independence eases the computation as $V_i$ conditional on its immediate neighbors, Markov Blanket, is independent of all other variables.

### 2.6.2 Structure Learning in Relational Dependency Networks (RDN)

In this subsection, we describe in detail the learning and inferencing techniques used in the case of RDN

#### 2.6.2.1 Learning

RDNs [25] extend the learning of DNs to a relational setting. The set of the conditional probability tables CPTs describes both the structure and the parameters of the model. RDNs use pseudo-likelihood techniques [1] to avoid the complexities of estimating the partition function. Instead of optimizing the log-likelihood of the joint distribution, RDNs optimize the pseudo-likelihood for each node separately conditioned on all its neighbors. Equation 2.2 computes the pseudo-likelihood for each node in $G_M$ separately, where $\theta_{GM}$ is the parameters for $G_M$, $D(v)$ is the domain of values for variable $v$, and $D(p(v))$ is the domain of values for parents of $v$.

$$Pl(\theta_{GM}|G_D, G_M) = \sum_{v \in V_M} \sum_{k \in D(v)} \sum_{u \in D(pa(v))} P(v = k|pa(v) = u) \tag{2.2}$$

where, $P(v = k|pa(v) = u)$ is computed by two main relational learners

- Relational Bayesian Classifier (RBC) is a non selective model that treats heterogeneous relational subgraphs as a homogeneous set of attribute multisets. The classifier assumes that each value in the multiset is drawn independently from the same multinomial distribution. For example, the *difficulty* of the courses taken by a student form a multiset Hard, Hard, Easy, Medium. RBC selects values independently from the multiset distribution.

- Relational probability trees are a selective model that extends traditional classification trees to relational settings. Relational probability trees also treat heterogeneous relational subgraphs as a set of attribute multisets; however, instead of treating the values as an independent set, Relational probability trees use aggregation functions to map the set of values into a single value.

#### 2.6.2.2 Inference

RDNs use Gibbs sampling for inference on $G_I$ . The values of unobserved variables are initialized with their prior distribution and are iteratively relabeled using the current state on the model and the CPT of the node. Gibbs sampling is generally an inefficient approach to estimate the joint probability of the model, however, it is reasonably fast to estimate conditional probabilities for each node given its parents.

# CHAPTER 3

# RELATED WORK

In statistical relational learning, a Markov logic network (MLN) [42] is regarded as one of the most flexible representations as it combines first-order logic and probabilistic graphical models. First-order logic enables the complexity of the data to be modeled, and probability allows expressing the uncertainty in the data. An MLN is a knowledge base (KB) and can model the complex, diverse nature of Twitter posts (a.k.a. tweets) containing 100+ attributes. Once the KB is learned, which includes the first-order rules (or formulas) and their weights, probabilistic inference can be conducted on the KB to reason about the content in the posts and users' behavior. For example, one can compute the marginal probability of a URL being malicious, a tweet being sensitive, a user account is an adversary or social bot [46], and so on.

So far in the literature, significant work has been done in the rule mining domain which has led to the development of state-of-the-art techniques. However, those techniques are not suitable for the large-scale data that gets generated on social media platforms. We describe in further detail in the following sections the related work done on first-order rules and/or scalable systems.

## 3.1 Rule Mining Techniques

Extensive work has been done in the field of first-order rule learning and MLN structure learning in literature. One approach is to use handcrafted first-order rules based on prior observations or knowledge [41]. This approach has shown good accuracy but the rules need to be handcrafted by a domain expert. Therefore, this is not the most viable option for rule learning on large-scale data, especially highly volatile social media data.

Alternatively, the first-order rules can be learned over the data, which is appealing when the significance of a rule changes over time. Although a number of methods have been proposed for structure learning for first-order rules, we highlight a subset of them. Li et al. [31] developed an unsupervised data repairing system that performs structure learning on a real-world dataset but through Bayesian Networks. Alps [10] on the other hand implements first-order logic to perform relational representation learning. This tool has also been tested on small-scale data so far. Dhami et al. address the problem of scalable structure learning through relational learning using first-order rules for Gaifman models [8] as opposed to MLNs. Another recent framework called Probabilistic Logic Graph Attention Network (pGAT) [18] combines MLNs and graph attention networks, for the task of link prediction in knowledge graphs and aims to achieve knowledge graph completion. Another preliminary study has been done in the field of scalable spatial probabilistic graphical modeling (SPGM) using MLNs [43]. This system exploits MLN to scale up the performance of SPGM techniques, however, the evaluation has been preliminary has not been tested for scalability. Numerical Markov Logic Network (NMLN) [55] is yet another probabilistic framework for hybrid knowledge inference. However, the model proposed

relies on hybrid knowledge rules which consist of first-order logic and mathematical expressions.

Despite extensive work being done in the field of MLNs and first-order rule learning, we believe the following work has more relevance with respect to the niche of problems we are attempting to address. Several state-of-the-art MLN structure learning techniques have been implemented over time to learn first-order rules [14, 20, 26, 29, 30, 48] have been tested only on small datasets using a single machine. Ontological pathfinding [4] is a recent work on scalable first-order rule mining using cluster computing. However, it learns only Horn clauses [4], whereas MLNs use general first-order rules, which are more expressive than Horn clauses. Thus, our goal is to scale MLN structure learning on large-scale Twitter data by (a) leveraging cluster computing and (b) exploiting the capability of existing structure learning tools [30] for learning rules on small datasets.

In the following sections, we describe in detail specific frameworks that have contributed significantly to the area of first-order rule learning using MLNs.

### 3.1.1 SocialKB

SocialKB [41] is a one-of-its-kind unified modeling framework specializing in social media data. It focuses on modeling the social media data and reasoning the reliability of the content. The goal of this framework is to discover suspicious users and malicious content on social media, building upon statistical relational learning on knowledge representation.

This framework learns over a knowledge base to capture the complex user behavior on social media. The knowledge base further contains multiple entities pertaining to the user, their posts, as well their interactions on social media. Using the KB, one can efficiently reason about the veracity of the social media posts and users' behavior to flag suspicious content/activity in a timely manner. [41] This method of modeling the social media information into a knowledge base and relying on user behavior forms the foundation of the logic behind SRLearn.

SocialKB, however, relies on hand-crafted social media rules, that can either be created by people with a basic understanding of how a particular social media platform might function, or by domain experts of user behavior. Nevertheless, given the size of social media data and the rate at which data evolves in today's scenario, hand-crafted rules are likely not to be efficient enough to capture interesting patterns to design the rules from. This gives rise to the need for a system that can automatically learn interesting rules, possibly through probabilistic inferencing.

### 3.1.2 Alchemy

Alchemy[1] is a software package system that provides a set of algorithms for statistical relational learning as well as probabilistic inference on Markov Logic Networks. Kok and Domingos [29, 30] have developed a codebase that performs structure learning via MLNs on a given knowledge base.

They proposed MSL [30] that learns the structure of MLNs from relational databases

[1] http://http://alchemy.cs.washington.edu/

by performing a shortest-first search of the clausal space through optimized pseudo-likelihood weights. It begins by adding all possible clauses to an MLN while calculating the score improvement upon each addition. It keeps adding $n$ clauses with the highest improvement in its weighted pseudo-log-likelihood (WPLL) (Equation 2.1) to the prior maintained MLN set. This step continues when no more clauses can be added, and the algorithm restarts all over again. This is an overall greedy approach that generates candidate clauses and is susceptible to local optima.

Kok and Domingos went on to further address this problem by proposing an approach that directly utilizes the data in constructing candidate clauses [29]. This approach is referred to as Learning via Hypergraph Lifting (LHL). LHL uses a relational pathfinding technique through *lifting* hypergraphs by creating clusters of constants and then finding paths in them. It, therefore, learns MLNs through a more compact data modeling, rather than the greedy approach of MSL, and therefore surpasses MSL in performance.

The above approaches are supported by Alchemy's umbrella package, even though are versatile with a wide variety of knowledge bases, they function well with a small set of data. The systems fail to scale up when the size of knowledge bases increases significantly. In the case of social media, given the amount of data that is being generated on a regular basis, Alchemy fails to run on such scaled-up data.

### 3.1.3 BoostSRL

Boosting for Statistical Relational Learning (BoostSRL) [2] is a gradient-boosting based approach developed by the Statistical Artificial Intelligence and Relational Learning (StARLinG) Lab at UT Dallas to learning different types of SRL models. This approach learns models in the form of a series of relational regression models based on relational data. BoostSRL also operates on data in the form of predicate logic and outputs first-order regression trees, thus relying heavily on probabilistic graphical models.

This approach considers Relational Dependency Networks (RDNs) and tries to address the prior problem of learning a single probability tree per variable, through gradient-based boosting of a series of relational function-approximation problems [37]. BoostSRL learns both the structure and parameters of RDN models simultaneously while achieving better predictive performance over comprehensibility. Moreover, the regression trees used by BoostSRL are much richer in the case of representation. It implements an algorithm for learning RDNs called *RDN-B*. *RDN-B* iterates over all predicates, by generating examples for the regression tree learner to get a new regression tree. This process again repeats for $M$ iterations. During each iteration, for a new predicate considered, the algorithm computes the probability as well as the gradient. It further calculates the regression values based on the current example's groundings. It then assigns the gradient as the weight of the example. In this manner, the algorithm is able to learn the structure of the relational data through sets of regression trees learned and learns the parameters through the set of leaves of this conditional distribution.

[2] https://starling.utdallas.edu/software/boostsrl/wiki/

Yang, Khot, Kersting, Kunapuli, Hauser, and Natarajan in 2014 [51] also proceeded to address a prevalent problem of class imbalance in relational data sets. Class imbalance is a very common problem of learning probabilistic models, where the number of negative examples exceeds the number of positive examples by a large difference. They proposed Cost-sensitive Statistical Relational learning (CSSRL) that includes a soft margin to trade-off between the false positives and false negatives, in the objective function of learning. This makes sure the positive examples and the negative examples are weighted fairly, rather than being weighted equally.

BoostSRL holds a lot of potential with its highly predictive capabilities. It also considers the fact that we need a large number of clauses to express a relational model, as opposed to the traditional learners that select a single model from the data. It is one of its kind that implements boosting for relational probabilistic models, even though it does not really consider MLNs. Furthermore, the tests done with this algorithm have been limited to concise data, and not that of practical big data that one can expect in a real-world scenario, with high unpredictability as well as noise.

## 3.2 Scalable Tools for Rule Mining

### 3.2.1 ProbKB

**PROB**abilistic **K**nowledge **B**ase [4] [5] [52] (ProbKB) is a scalable system that aims at constructing a web scale probabilistic knowledge base. It involves mining first order rules and inferring implicit knowledge. ProbKB combines numerous parallelization and optimization techniques that enable it to scale to web scale knowledge bases.

### 3.2.2 XGBoost

Tree boosting is a highly effective and widely used machine learning method. XGBoost [3] is a scalable end-to-end tree boosting system, which is used widely by data scientists to achieve state-of-the-art results on many machine learning challenges. It is available as an open-source package[3]. Furthermore, it also integrates well with Spark and Hadoop to take advantage of distributed computing

The most important factor behind the success of XGBoost is its scalability in all scenarios. The system runs more than ten times faster than existing popular solutions on a single machine and scales to billions of examples in distributed or memory-limited settings. The scalability of XGBoost is due to several important systems and algorithmic optimizations. These innovations include: a novel tree learning algorithm is for handling sparse data; a theoretically justified weighted quantile sketch procedure that enables handling instance weights in approximate tree learning. Parallel and distributed computing make learning faster which enables quicker model exploration. More importantly, XGBoost exploits out-of-core computation and enables data scientists to process hundreds of millions of examples on a desktop. Finally, it is even more exciting to combine these techniques to make an end-to-end system that scales to even larger data with the least amount of cluster resources.

[3] https://github.com/dmlc/xgboost

# CHAPTER 4

# SRLEARN

In this chapter, we present our approach called **SRLearn** [45] (**S**calable **R**ule **Learn**ing) to scale the learning of relevant first-order rules over large scale social media data (Twitter[1]) using a commodity cluster. The original design of SRLearn was proposed by Rao et.al [39] and [40]. However, it was not implemented or evaluated on real-world datasets. The salient features of SRLearn are as follows.

a. A *divide-and-conquer* approach to rule learning by first employing graph-based modeling of ground predicates and users, and applying graph partitioning to create partitions of ground predicates to minimize the chance of missing interesting rules;

b. Exploiting data parallelism at different stages during rule learning using a commodity cluster (CloudLab[2]);

c. Leveraging the power of Alchemy's structure learning [30] as well as BoostSRL's structure learning using RDNs [36] on small datasets.

The architecture of the SRLearn is shown in Fig. 6. We describe the details of each of the above features in the subsequent sections, simultaneously describing the architecture in the following stages.

a. Ground predicates

[1]www.twitter.com
[2]https://www.cloudlab.us/

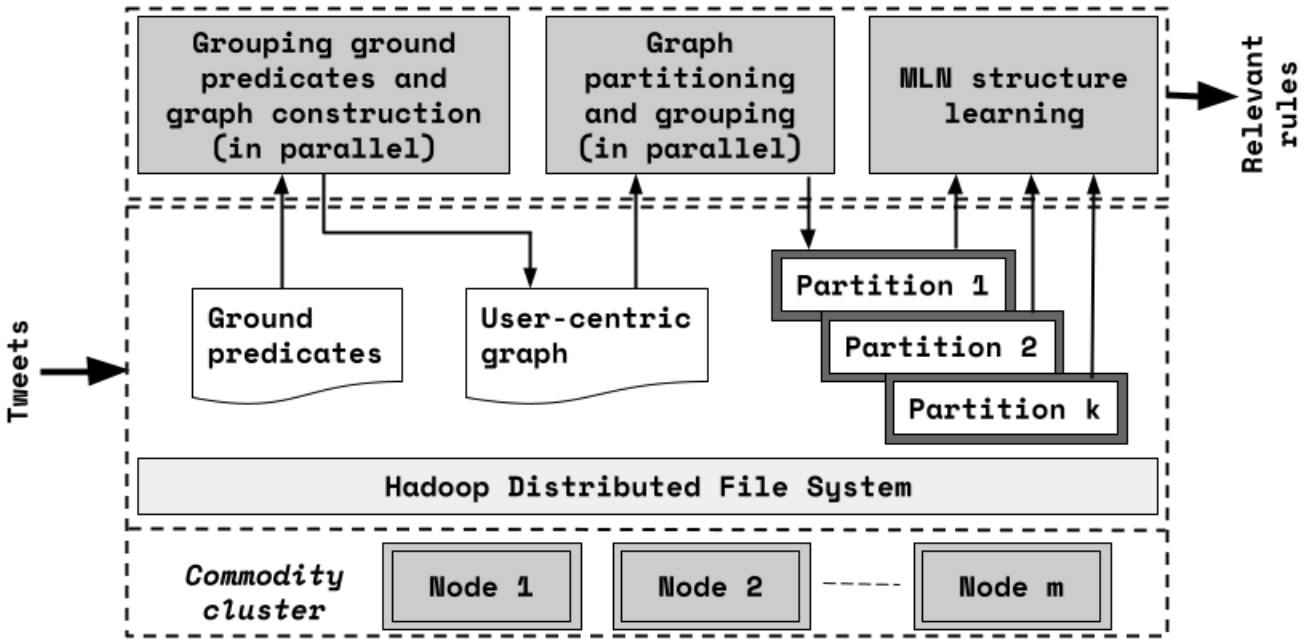



Figure 6: SRLearn architecture

b. Ground predicate graph

c. User-centric graph

d. Graph partitioning

e. Regroup ground predicates and Rule Mining

### 4.1 Ground Predicates from Tweets

Twitter is a popularly used social media platform, with over 300 million active users across the globe per month. The platform is estimated to generate 6000 tweets per second each day.

Table 2: Set of first-order predicates generated from Twitter data based on CWA (denoted by *) and OWA

| | |
|---|---|
| *tweeted(userID, tweetID) | *containsLink(tweetID, link) |
| *containsHashtag(tweetID, hashtag) | malicious(link) |
| *verified(userID) | attacker(userID) |
| *mentions(tweetID, userID) | *retweeted(userID, tweetID) |
| isPossiblySensitive(tweetID) | *retweetCount(tweetID, count) |
| friend(userID1, userID2) | isFollowedBy(userID1, userID2) |
| *friendsCount(userID, count) | *followersCount(userID, count) |
| *statusesCount(userID, count) | *favouritesCount(userID, count) |

Tweets [45] are short messages created by users on Twitter. Each tweet collected from Twitter is assigned a unique ID; each user account is also assigned a unique ID. There are attributes whose values embed the actual text of a tweet, the URLs contained in a tweet, hashtags used in a tweet, users mentioned in a tweet, who retweeted a tweet, and so on. There are attributes that provide counts about the number of friends of a user, the number of followers of a user, and the number of posts of a user (i.e., statuses count). For our evaluation purpose, we collect the tweets from Twitter API, using a given set of keywords. We refer to the retweet associated with the tweet object as the ”inner tweet” for that tweet. Each tweet, including the inner tweet, has an associated non-zero tweet ID generated by Twitter.

We map the tweets collected from the Twitter API into first-order predicates as shown in Table 2 to build the evidence KB. These predicates are a representation of the rich and complex relationships in the tweet entities. A predicate can make a closed-world assumption (CWA) or an open-world assumption (OWA). CWA assumes that what is not known to be true must be false. On the other hand, OWA assumes that what is not known

may or may not be true.

We derive most of the first-predicates directly from the elements as is from JSON object. A tweet does not contain the list of friends or followers of a user. These pieces of information, however, can be obtained using Twitter APIs. The predicate *tweeted(userID, tweetID)* states whether a user posted a particular tweet or not; *containsLink(tweetID, link)* states whether a tweet contains a particular URL or not; *containsHashtag(tweetID, hashtag)* states whether a tweet contains a particular hashtag or not; *mentions(tweetID, userID)* states whether a particular user is mentioned in a tweet (using the @ symbol) or not; *retweeted(userID, tweetID)* states whether a user retweeted a particular tweet or not; finally, *verified(userID)* states whether a user has been verified or not. Twitter independently verifies user accounts that are of public interest in domains such as government, fashion, music, politics, sports, etc. For the OWA predicates, The predicate *malicious(link)* states whether a URL is malicious or not; *friend(userID1, userID2)* states whether a user denoted by *userID1* has a friend denoted by *userID2* or not. Twitter defines a friend as someone who a user is following. The predicate *attacker(userID)* indicates whether a user is a suspicious user or not; *isFollowedBy(userID1, userID2)* indicates whether a user denoted by *userID1* is followed by another user denoted by *userID2* or not; and finally, *isPossiblySensitive(tweetID)* indicates whether a tweet is possibly sensitive or not. Twitter flags a tweet as possibly sensitive based on users' feedback and by monitoring if the tweet follows Twitter's rules and policies[3]. The count predicates model the friends count/followers count/statuses count of a user, the retweet count of a tweet,

[3]https://help.twitter.com/en/rules-and-policies/media-policy

and the number of tweets a user has ”liked”. These predicates are based on a CWA. [41] *friendsCount(userID, count)* denotes the number of friends a user denoted by *userID* has. *followersCount(userID, count)* indicates the number of followers a user has. We derive the predicates such as *malicious*, *attacker*, *friend*, *isFollowedBy* through additional Twitter API calls or from external sources. We make additional calls to the Twitter API to collect *friend* and *isFollowedBy* predicates. For evaluation purpose, we retain only those *friend* and *isFollowedBy* predicates, whose user IDs are present in the set of *tweeted* predicates. In other words, for each *friend* and *isFollowedBy* predicate, we retain the predicate, if for both *userID1* and *userID2*, *tweeted(userID1, tweetID1)* and *tweeted(userID2, tweetID2)* are present in the evidence data, else we refrained to consider the predicate for further evaluation. The intention is to have friends and followers for only those users, whose tweets had been used to build the evidence KB. For the *malicious* predicates, we use VirusTotal[4] to determine whether a given URL is malicious or not. We acquire academic keys to stream URLs via the VirusTotal API and check each URL from the set of unique URLs present in *containsLink* predicate, with the VirusTotal database. We add the URL to the evidence KB, mapped as *malicious(url)* if at least one of the engines in the VirusTotal server flagged the URL as malicious. We match URLs with VirusTotal at two levels.

a. Level 1: Direct matching of the original URL.

b. Level 2: Indirect matching of the expanded form of the original URL. In other words, If the original URL was shortened, then we expanded it. We perform direct match (Level 1 matching) on this expanded URL.

[4]https://www.virustotal.com/gui/

We curate these ground predicates for each set of N number of tweets we collect for multiple experiments. This forms the primary input on which the SRLearn system operates. We store this input set of ground predicates in the Hadoop Distributed File System (HDFS) set up on CloudLab, a commodity cluster.

## 4.2 Ground Predicate Graph

The first step of the data modeling is to group the ground predicates for each user. In the process we create a *ground predicate graph* [45]. Suppose a user posted two tweets containing a link flagged as malicious and a hashtag that is trending. Fig. 7 shows an example ground predicate graph constructed on 19 predicates.

We define a ground predicate graph for a user, as a graph representing all the ground predicates associated with the user, either directly or indirectly. Each user node is connected to all the tweet ID nodes, and their associated ground predicates with the tweet ID, or user ID, or both. The ground predicates containing the user ID as one of the variables are connected to the corresponding user ID node. The remaining ground predicates, that do not have the user ID as one of the variables, are connected to the tweet nodes and are then associated with the user ID node as in Fig. 7.

We use a divide-and-conquer approach using a distributed system on CloudLab, as well as Spark's MapReduce [7] capabilities to map the list of first-order ground predicates into a ground predicate graph. The algorithm for this transformation is shown in Algorithms 1, 2 and 3.

First, following Algorithm 1, we build an associative array containing (tweet ID,

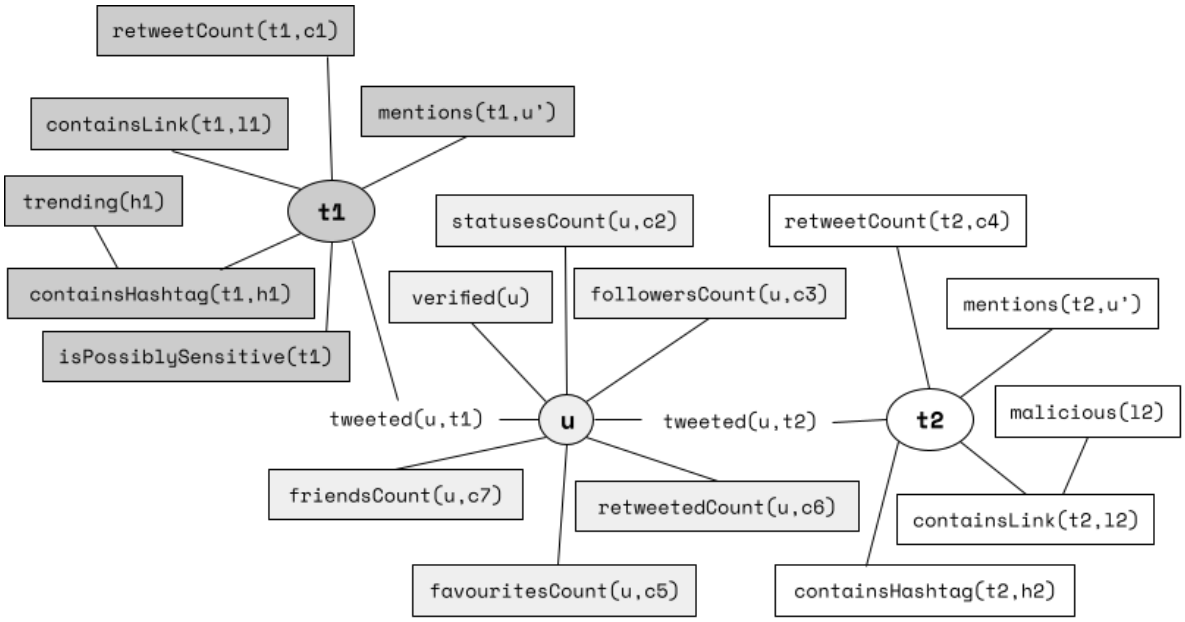



Figure 7: Example of ground predicate graph

user ID) pairs on the input data set. Then we broadcast to all workers on the nodes in the distributed cluster (Line 3). This is required for associating ground predicates that do not contain user ID as a variable, to the right user. We then implement the map (Lines 5-6) and reduce (Lines 7-8) functions.

In the map phase (Algorithm 2), on each worker node, each block of ground predicates is processed by invoking *MapPredicates* function and appending the mapped ground predicate into a distributed collection $G$ initialized in Line 2 of Algorithm 1. This collection is in the format $(userID, groundpredicate)$.The predicates which have user ID as the first parameter are associated with the user ID and then added to $G$. We process those

**Algorithm 1:** Grouping the ground predicates by users

1: Let $F$ denote the HDFS file containing ground predicates
2: Let $G$, $H$, and $I$ denote distributed collections
3: Using map and reduce operations on $F$, construct an associative array $A$ containing key-value pairs $(t, u)$, where $t$ is tweetID and $u$ is userID
4: Broadcast $A$ to all workers on nodes
{/* **Map phase in algorithm 2** */}
5: **for** each block $B_i$ of $F$ in HDFS **do**
6: Invoke `MapPredicates`$(B_i)$
{/* **Reduce phase in algorithm 3**/}
7: **for** each $(K, V)$ in $G$ **do**
8: Invoke `ReducePredicates`$(K, V)$
9. Run a reduce-by-key operation to process $H$ by grouping the tuples by userID and store the result into $I$
10: return $I$

predicates whose first argument is not of the type *user ID* (e.g., *malicious(l)*, *containsHashtag(t, h)*) (Lines 5-14 in algorithm 2) specially in order to associate them with the right user in the reduce phase. We add ground predicates *malicious(l)*, *trending(hashtag)* to $G$ in the form of *(malicious(l), "EXISTS")* and *(trending(hashtag), "EXISTS")*, respectively. For *containsLink(tweet ID, link)*, we add *(malicious(link), tweet ID)* to $G$. Simultaneously, we find the user ID corresponding to this tweet ID from the associative array $A$, and append this to $G$ in the form *(user ID, containsLink(tweet ID, link))* (Lines 7-9). For *containsHashtag(tweet ID, hashtag)*, we add *(trendingHashtag(link), tweet ID)* to $G$. Simultaneously, we find the user ID corresponding to this tweet ID from the associative array $A$, and append this to $G$ in the form *(user ID, containsHashtag(tweet ID, hashtag))* (Lines 10-12). Similarly, for predicates *mentions(tweet ID, user ID1)*, *isPossiblySensitive(tweet ID)* and *retweetCount(tweet ID, count*, we find the tweet ID from $A$

and append the predicate with the user ID corresponding to the tweet ID to $G$ (Lines 13-14). Finally, for *retweeted(user ID, tweet ID*, we append *(user ID1,retweetedBy(tweet ID, user ID)* to $G$ if there exists *(tweet ID, user ID1)* in $A$ (Lines 15-16).

The reduce phase (Algorithm 3) invokes *ReducePredicates* function and generates key-value pairs where user ID is the key, and the value is a group of predicates belonging to the ground predicate graph of that user. Here we use distributed collection $H$, initialized in Line 2 of Algorithm 1. We append this $(K, V)$ pair to $H$. At this stage, there can be more than one key-value pair with the same key. We also, take care of the *(malicious(l), "EXISTS")* and *(trending(hashtag), "EXISTS")* generated in Algorithm 2, and append them to $H$ with the corresponding tweet ID from $A$. Finally, we execute a reduce-by-key operation group of all the ground predicates for a particular user into a key-value pair (Line 9 in algorithm 1).

### 4.3 User-centric Graph

The second step of the data modeling is to group the ground predicate graphs for each user ID. In the process, we create a *user-centric predicate graph*, where we group the ground predicate graphs from the previous section around each user ID. Fig. 8 shows an example of user-centric graph. Each edge in the user-centric graph is a representation of the interaction of users on Twitter.

**Algorithm 2:** Map function for grouping the ground predicates by users

1: **begin func** `MapPredicates`(block $B$)
2: **for** each ground predicate $p$ in $B$ **do**
3: **if** $p$ is a predicate with userID $u$ as the first parameter **then**
4: Output $(u, p)$ to $G$
5: **if** $p$ is type of malicious(link) **or** trending(hashtag) **then**
6: Output ($p$,"EXISTS") to $G$
7: **if** $p$ is of type containsLink(t, link)) **then**
8: Output ("malicious(link)", t) to $G$
9: Find $(t, \bar{u})$ in $A$; Output $(\bar{u}, p)$ to $G$
10: **if** $p$ is of type containsHashtag(t, hashtag)) **then**
11: Output ("trending(hashtag)", t) to $G$
12: Find $(t, \bar{u})$ in $A$; Output $(\bar{u}, p)$ to $G$
13: **if** $p$ is of type mentions(t, u') **or** isPossiblySensitive(t) **or** retweetCount(t, c) **then**
14: Find $(t, \bar{u})$ in $A$; Output $(\bar{u}, p)$ to $G$
15: **if** $p$ is of type retweeted(u, t) and $((t, \bar{u})$ exists in $A$ **then**
16: Output ($\bar{u}$,"retweetedBy(t,u)") to $G$
17: **end func**

**Algorithm 3:** Reduce function for grouping the ground predicates by users

1: **begin func** `ReducePredicates`($K$,$V$)
2: **if** $K$ is a userID **then**
3: Output ($K$,$V$) to $H$
4: **else if** any item $V$ ="EXISTS" **then**
5: **for** each item $t' \neq$ "EXISTS" in $V$ **do**
6: Find ($t'$,$\bar{u}$) in $A$; Output($\bar{u}$,$K$) to $H$
7: **end func**

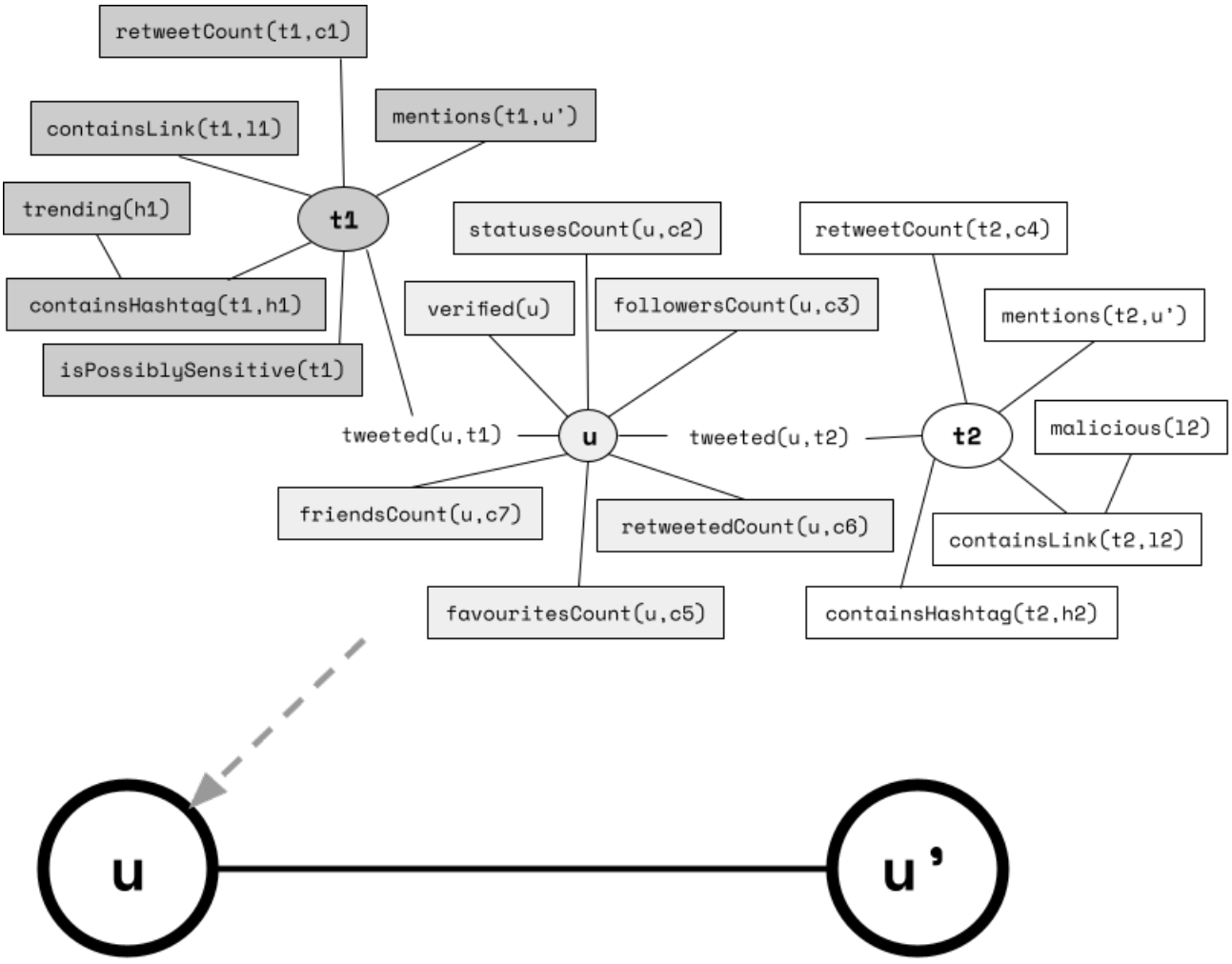



Figure 8: Example of user-centric graph

**Algorithm 4:** Constructing the user-centric graph

1: Let $I$ denote the output of Algorithm 1
2: Let $J$ and $L$ denote distributed collections
{/* **Map phase in algorithm 5** */}
3: **for** each ($K$,$V$) in $I$ **do**
4: Invoke `MapWts`($K$,$V$)
{/* **Reduce phase in algorithm 6**/}
5: **for** each $(K, V)$ in $J$ **do**
6: Invoke `ReduceWts`$(K, V)$
7: Store $L$ as a file to HDFS
8: **return**

Table 3: Social relationships between users

| Condition | Edge(u, u') |
|---|---|
| tweeted(u, t) $\wedge$ mentions(t, u') | TRUE |
| tweeted(u', t') $\wedge$ mentions(t', u) | TRUE |
| friend(u, u') $\vee$ friend(u', u) | TRUE |
| isFollowedBy(u, u') $\vee$ isFollowedBy(u', u) | TRUE |
| tweeted(u, t) $\wedge$ retweeted(u', t) | TRUE |
| tweeted(u', t) $\wedge$ retweeted(u, t) | TRUE |

Algorithm 4 shows the steps involved in constructing the user-centric graph. It has a map phase (Lines 3-4) and a reduce phase (Lines 5-6). In the map phase (Algorithm 5), we use the output of Algorithm 1 as the input for the function $MapWts$. On each $(K, V)$ we invoke $MapWts$ function, where $K$ is a user ID and $V$ denotes all the ground predicates for that user ID.

We determine the vertex weights for the corresponding vertex ID. (Lines 2-3 of Algorithm 5). The vertex weight is the number of predicates in $V$. In other words, each predicate has a unit weight contribution towards the total vertex weight of the vertex.

**Algorithm 5:** Map function for constructing the user-centric graph

1: **begin func** `MapWts`($K$,$V$)
2: Let $c$ denote the no. of predicates in $V$ excluding retweetedBy
3: Output $(K, c)$ to $J$
4: $edgeList \leftarrow \emptyset$
5: **for** each $p$ in $V$ **do**
6: **if** is of type friend(u,u') **or** isFollowedBy(u,u') **or** mentions(t,u') **or** retweetedBy(t,u') **then**
7: $x \leftarrow min(u, u'); y \leftarrow max(u, u')$
8: Create edge $(x, y)$
9: **if** $(x, y) \in edgeList$ **then**
10: Increase edge weight $w_{x,y}$ by 1
11: **else**
12: Add $(x, y)$ to $edgeList$ with $w_{x,y} = 1$
13: **for** each $(x, y)$ in $edgeList$ **do**
14: Output $((x, y), w_{x,y})$ as the edge to $J$
15: **end func**

Next, we determine the social interaction between two users based on the rules in Table 3. We define an edge between two users $u$ and $u'$ in the graph if (a) $u$ mentions $u'$ or vice-versa, (b) $u$ is friends with $u'$ or vice-versa, (c) $u$ is followed by $u'$ or vice-versa, or (d) $u$ retweets a tweet of $u'$ or vice-versa. Each of the above conditions defines an edge with unit weight. We compute partial edge weights and output them along with the edge (Lines 4-14 in Algorithm 5).

**Algorithm 6:** Reduce function for constructing the user-centric graph

1: **begin func** `ReduceWts`($K$,$V$)
2: **if** $K$ is a userID **then**
3: Output $(K, V)$ to $L$ {/* weighted vertex */}
4: **else**
5: Let $\bar{w}$ denote the sum of all wt. values in $V$
6: Output $(K, \bar{w})$ to $L$ /* weighted edge */
7: **end func**

We combine the partial edge weights for the same edge from different map operations to reduce phase by $ReduceWts$ function (Algorithm 6). Finally, we store the weighted graph (with both vertex weights and edge weights) in HDFS (Line 7 in Algorithm 4). The output graph is in the form of a METIS graph [22]. Thus, we store the user-centric graph $G = (V, E)$ has $n$ vertices and $m$ edges in a single text file, containing $n+1$ lines. Line 1 holds the information in the format: *number of vertices, number of edges, type of graph*. The type of graph in our case is 011, implying it is both edge-weighted and vertex-weighted. The subsequent lines 2 through $n+1$ contain the adjacency lists for each vertex. Each line represents a vertex in the user-centric graph along with its vertex weight, neighboring vertex, and the edge weight associated with the edge between two vertices.

### 4.4 Graph Partitioning

At this stage, we have successfully modeled the tweets into a user-centric graphical structure. In other words, we have grouped the ground predicates in a systematic manner with their corresponding users, as well as established a connection among the users based on their social relationships for a given data set. Algorithm 7 shows the divide-and-conquer approach of SRLearn. The user-centric graph is partitioned into $k$ number of partitions by minimizing the total weight of the cut edges (Line 4). The intuition is to group predicates of users that have a high degree of social interaction, in the same partition so that interesting rules can be discovered between them. Alternatively, users with a low degree of social interaction can be placed in different partitions without

missing interesting rules spanning across partitions.

We use an existing graph partitioning technique to partition this single graph into smaller partitions. This results in the partitioning of vertices (user IDs) along with their neighbors (user IDs) and corresponding predicates into these partitions. For our experimentation, we have tested with both the Karlsruhe Fast Flow Partitioner Evolutionary (KaFFPaE) and Parallel High Quality Partitioning (ParHIP) algorithms from the Karlsruhe High Quality Partitioning[5] (KaHIP) [44] family of graph partitioning algorithms. With the KaFFPaE, we use a "strongsocial" preconfiguration on a single machine due to the smaller size of the graph built from 20,000 tweets. The "strongsocial" configuration is used to achieve the highest quality of graph partitioning possible on a social network or web graph.

Since KaFFPaE is localized to one machine, it forms a bottleneck when scaling up. Therefore, we implement the parallel and memory distributed ParHIP algorithm [33], which is specifically designed to process large complex networks like social networks or web graphs. It parallelizes and adopts the label propagation technique to the coarsening and refinement phase of multilevel graph partitioning. The algorithm claims to be more scalable, efficient and of higher quality than state-of-the-art systems like ParMetis [21].

The user-centric graph from the previous section carries both edge weights as well as vertex weights. With the help of the authors of ParHIP, we were able to obtain a private version of the algorithm that is compatible with a graph with both vertex weights and edge weights, i.e. the user-centric graph. The available options for preconfiguration

[5]https://kahip.github.io/

Table 4: Runtime for different preconfiguration on 2 million tweets' graph partitioned into 32 partitions on 16 nodes

| Preconfiguration | Runtime |
|---|---|
| strong | 17h 33m 29s |
| eco | 3m 16s |
| fast | 28s |
| ultrafast | 16s |

in this tool are: *strong*, *fast*, *eco*, *ultrafast*. We retrieve the user-centric graph obtained in the graph-construction phase from HDFS and move it to a shared file system on the cluster. Using this graph partitioning tool, we create $n$ partitions of the user-centric graph on the distributed cluster (Lines 1-4 of Algorithm 7). It is recommended to use *strong* preconfiguration if the quality of graph partitioning is of utmost importance. The preconfiguration *eco* is recommended if a tradeoff between partitioning quality and run time is acceptable. The preconfigurations *fast* and *ultrafast* are recommended for fast run time, despite a poor quality of graph partitioning. To finalize this knob in the pipeline, we have tested the available preconfigurations for ParHIP. We use the user-centric graph built from 2 million tweets and partition it into 32 partitions using a 16-node cluster. Using *strong* preconfiguration, despite the highest run time (Table 4), ensures the most balanced partitioning of a graph possible (Fig. 9 and Fig. 10). This is followed by *eco* preconfiguration, which not just provides a balanced partitioning outcome, but also has a significantly reduced runtime. Therefore, with the aim to build a fast and scalable system, we choose to use *eco* preconfiguration for the graph partitioning in our pipeline.

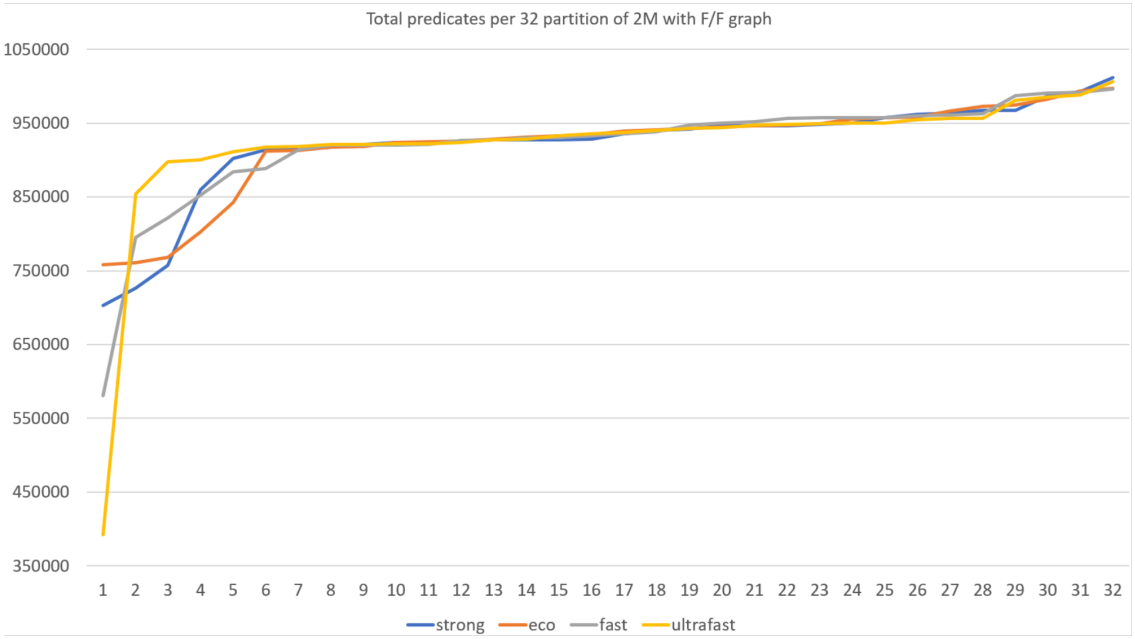



Figure 9: Distribution of ground predicates with varying preconfiguration on 32 partitions

### 4.5 Regroup Ground Predicates and Rule Mining

Once we partition the graph (with $K$ edges and $V$ vertices) into $k$-number of parts, we push the resulting partition file back to HDFS. This partition file contains $V$ number of lines denoting the vertex or user ID that went into the corresponding partition. We use the predetermined map of user IDs with their set of ground predicates, to place the ground predicates associated with a user ID into the corresponding partition bucket. We then distribute the $k$ partitions containing the ground predicates from HDFS across all nodes of the cluster in a round-robin fashion. Then we initiate the rule mining process on all the partitions across the nodes. (Lines 5-14 of Algorithm 7)

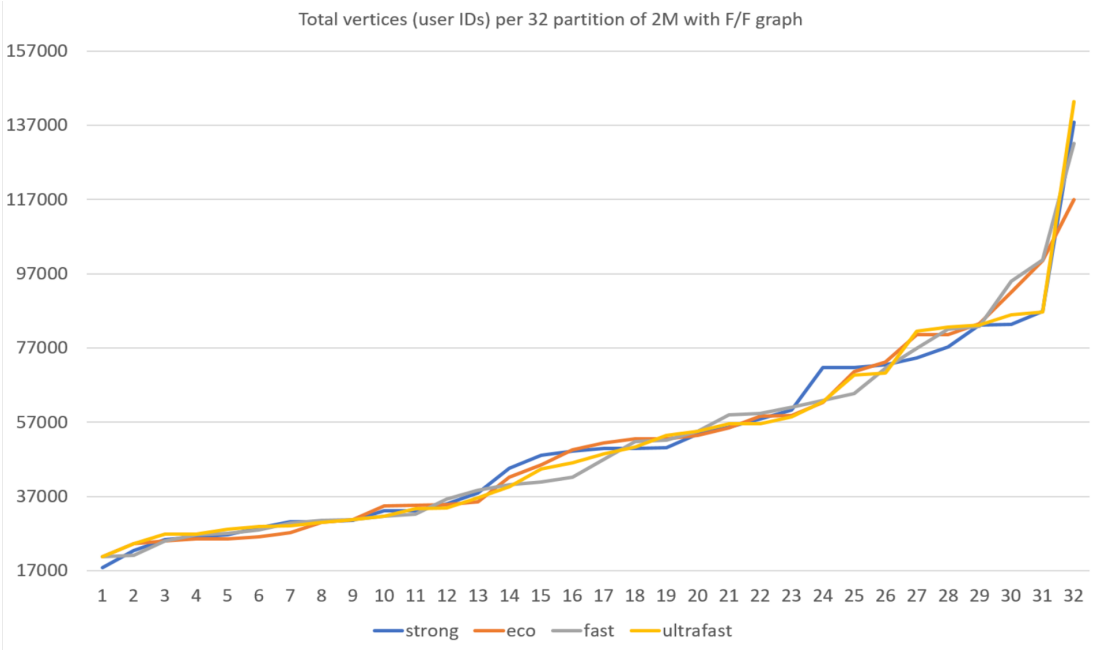



Figure 10: Distribution of vertices (user IDs) with varying preconfiguration on 32 partitions

**Algorithm 7:** Graph partitioning and rule learning in parallel

Let $U$ denote the user-centric graph file output by Algorithm 4
Let $m$ denote the number of required partitions
Let $n$ denote the number of machines
Run graph partitioning (in parallel) on $U$ to generate partitions $p_1, ..., p_m$ of the ground predicates (in $F$) by minimizing the total weight of the cut edges in $U$
Assign partitions in a round-robin fashion to the $n$ machines
In parallel, invoke `LearningAndInferencing()` on each node
Wait for all the nodes to finish
Let $R$ denote the union of all predicted probabilities to calculate the AUC
Calculate the global AUC ROC on $R$
**return** global AUC ROC
**begin func** `LearningAndInferencing()`
**for** each partition $p_i$ on the node **do**
    Run BoostSRL on $p_i$
**end func**

### 4.5.1 Set up for rule mining

In order to assess the effectiveness of the model, as well as to ensure a fair training process, we implement a process similar to 5-fold cross-validation. The idea is to group the ground predicates into a training set and testing set for all 5 folds of cross-validation and compare the AUC across all folds. This makes sure, we avoid any bias or overfitting. Fig. 11 shows the distribution of partitions in a round-robin fashion across all folds. We initiate BoostSRL training and inferencing for rule mining on the partitions after the completion of this setup process.

Once we place these $k$ partitions on the $n$-node cluster, we sort the respective partitions on each node, according to their size, and group them into $m$-subgroups in a round-robin order. This ensures that each subgroup gets the largest partition and smallest partition, with an even grouping of predicates. For example, let us consider a graph divided into 128 partitions, on a 16-node cluster. In this case, $k$=128 and $n$=16. On each node, we have 8 partitions on each node. Now we group these partitions together according to the number of subgroups. For an experiment with the *number of subgroups* parameter set to 4, we group the largest and smallest partition into sub-group 1, the next largest and smallest partition into sub-group 2. Similarly, we group the remaining partitions for sub-groups 3 and 4.

To factor in the 5-fold cross validation, we create 5 folders representing 5 folds of cross validation. BoostSRL takes the following files as input:

1. facts, positive examples, negative examples as training set;

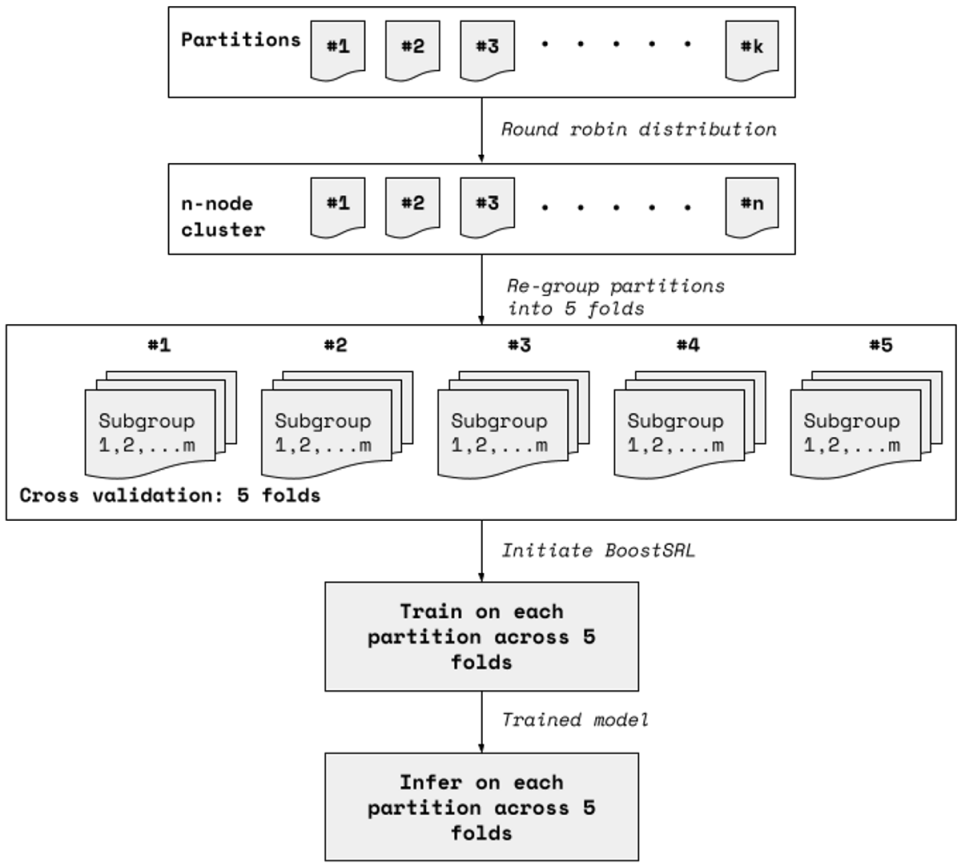



Figure 11: Distribution of grouped predicates, i.e. partitions amongst the nodes in the cluster

2. facts, positive examples, negative examples as test set; and

3. the background file providing context on the ground predicates.

Facts file contains all ground predicates in the evidence data, except the target ground predicates *isPossiblySensitive* and *malicious*. For each fold of cross-validation, these facts file remain constant. We vary the ground predicates set in the positive and negative examples files for training and testing.

To build the positive examples of training data for the target *isPossiblySensitive* predicate, we extract the tweet IDs from the *tweeted* predicates in the facts file in the k-th fold of cross-validation and that particular inner sub-group. Similarly, for the target *malicious* predicates, we place the URLs present in *containsLink* predicates in the facts file as positive examples of training, if they were identified as malicious by VirusTotal. We achieve this by providing a copy of a file with a list of all URLs in the evidence db that was flagged as malicious. This was done to make sure if a URL, say $url1$ is present in *containsLink* on node 1 and node 3 (having been tweeted by two different users), they are still flagged as malicious or not malicious on the respective nodes for the respecting BoostSRL instances. For determining the negative examples for training set, we take the subset of the tweet IDs from *tweeted* and URLs from *containsLink* predicates in question that were neither flagged as *isPossiblySensitive* nor *malicious*, respectively.

We follow a similar approach for the input files of the test set. Furthermore, to determine the distribution of predicates into the cross-validation folder, we split the positive and negative examples files into 5 parts (parts 1,2,3,4,5). For fold 1 of cross-validation, we placed the first 4 parts (parts 1,2,3,4) into training and the 5th part into testing (part 5).

For fold 2, we place the next 4 parts (parts 2,3,4,5) into training and the 1st part into testing (part 1). Fig. 11 denotes how we organize the rest of the parts, along with an overall summary of the arrangement of predicates into test and training data.

### 4.5.2 Initiation of rule mining: Learn and Infer

We start the structure learning phase one the prerequisites[6] of BoostSRL, in terms of the training and test data sets that have been established. Here we have experimented with the default BoostSRL, i.e. RDN-Boost[7], which does not take into account the class imbalance, as well as the CSSRL that considers a soft margin to fairly deal with the class imbalance in our data sets.

For the default RDN-Boost, we fork each instance into further 10 instances and trigger the jar file using the command:

```
java -jar BoostSRL.jar -l -train train/
-target isPossiblySensitive
```

In the above command:

- `-l` enables training (learning);
- `-train <Training directory>` denotes the path to the training directory;
- `-target <target predicates>` are the comma-separated list of predicates to be learned/inferred upon.

[6] facts, positive examples and negative examples of training and test data for each fold of cross-validation and each inner sub-group

[7] https://starling.utdallas.edu/software/boostsrl/wiki/rdn-boost/

Once the jar for the learning phase has been forked, we pick the instance that finishes training the earliest, and use the model learned in that instance to infer on the target predicates using the following command:

```
java -jar BoostSRL.jar -i -model train/models/
-test test/ -target isPossiblySensitive
-aucJarPath /mydata/ -testNegPosRatio -1
```

In the above command:

- `-i` enable testing (inference);
- `-test <Testing directory>` denotes the path to the testing directory;
- `-target <target predicates>` are the comma-separated list of predicates to be learned/inferred upon;
- `-aucJarPath <path to auc.jar>` denotes the path to the auc.jar that is used to compute the AUC ROC and AUC PR values at the end of learning and inferencing;
- `-testNegPosRatio <Negative/Positive ratio>` : denotes the ratio of negatives to positive for testing. It is set to -1 to disable sampling.

Unlike the learning phase, where we use multiple cores through the creation of multiple forks of BoostSRL's jar, we infer using only one core. Although we attempted using multiple cores to speed up the inferencing further, it resulted in an out-of-memory error for certain instances during the inference phase.

Taking into account the class imbalance in our data set, we repeat the learning and inferencing using the CSSRL algorithm of BoostSRL family. Furthermore, with the aim of improving our system further, we coupled with SRLBoost[8], which is smaller in size than BoostSRL and significantly faster than BoostSRL. For the CSSRL, we use only one instance per partition, instead of forking as we did above. We thus run the following command to initiate the structure learning:

```
java -jar srlboost-0.1.0-jar-with-dependencies.jar -l
-train train/ -target isPossiblySensitive
-softm   -alpha {0,40} -beta 40
```

In the above command:

- `-l` enables training (learning);
- `-train <Training directory>` denotes the path to the training directory;
- `-target <target predicates>` are the comma-separated list of predicates to be learned/inferred upon;
- `-softm` is set to activate the CSSRL learning;
- `-alpha` is to assign the weight on false negative examples. This is set to 0 and 40 one after the other;
- `-alpha` is to assign the weight on false positive examples. This is set to 40 both times.

[8]https://github.com/srlearn/SRLBoost

Once the jar for learning phase has been forked, we pick the instance that finishes training the earliest, and use the model learnt in that instance to infer on the target predicates using the following command:

```
java -jar srlboost-0.1.0-jar-with-dependencies.jar -i
-model train/models/ -test test/
-target isPossiblySensitive  -aucJarPath /mydata/
-softm  -alpha {0,40} -beta 40
```

In the above command:

- `-i` enable testing (inference);
- `-test <Testing directory>` denotes the path to the testing directory;
- `-target <target predicates>` are the comma-separated list of predicates to be learned/inferred upon;
- `-aucJarPath <path to auc.jar>` denotes the path to the auc.jar that is used to compute the AUC ROC and AUC PR values at the end of learning and inferencing;
- `-softm` is set to activate the CSSRL learning;
- `-alpha` is to assign the weight on false negative examples. This is set to 0 and 40 one after the other;
- `-beta` is to assign the weight on false positive examples. This is set to 40 both times.

Unlike the learning phase, where we use multiple cores through the creation of multiple forks of BoostSRL's jar, we infer using only one core. Although we attempted using multiple cores to speed up the inferencing further, it resulted in an out-of-memory error for certain instances during the inference phase.

Once the BoostSRL learning and inferencing is complete on all instances, we gather all the results together onto one node for further evaluation. For the learning time of the k-th fold of cross-validation, we consider the maximum learning time across all nodes for each target. Similarly, we record the inference time. Further, we calculate a global run-time for learning on each target by averaging the times for each fold of cross-validation for each target. Since AUC-ROC and AUC-PR are considered as the evaluation metrics for the classification models, we record the global AUC-ROC and global AUC-PR values in a similar fashion, by averaging the AUC-ROC and AUC-PR values respectively, for 5 folds of cross-validation. In order to calculate the individual AUCs for each fold, for target *isPossiblySensitive*, we accumulate the predicted results in one single location and calculate the AUC values.

# CHAPTER 5

# EXPERIMENTATION AND EVALUATION

In this chapter, we elaborate on the details of experimentation done on SRLearn, as well as its competitive tools ProbKB and XGBoost. We also compare our results with a scenario, where the SRLearn methodology is not present, i.e. Vanilla SRLearn.

For each of the methodologies, we describe the following aspects:

- Hardware environment
- Datasets
- Evaluation Metrics
- Results

## 5.1 Hardware Environment

CloudLab[1] [11] is the primary platform for conducting the experiments. CloudLab is a ”meta-cloud” - that is, it is not a cloud itself; rather, it is a facility for building clouds. It provides bare-metal access and control over a substantial set of computing, storage, and networking resources; on top of this platform, users can install standard cloud software stacks, modify them, or create entirely new ones. The current CloudLab deployment consists of more than 25,000 cores distributed across three sites at the University of

[1]https://www.cloudlab.us/

Wisconsin, Clemson University, and the University of Utah. CloudLab interoperates with existing testbeds including GENI[2] and Emulab[3], to take advantage of hardware at dozens of sites around the world.

We set up a cluster of 16 physical nodes. We used Clemson and Wisconsin data centers based on the availability of machines. The machines used had Ubuntu 16.04.1 LTS as the operating system. At the Clemson data center, each machine had a 10-core Intel processor with 256 GB RAM and two 1 TB SATA HDDs. At the Wisconsin data center, each machine had a 10-core Intel processor with 160 GB RAM, 480 GB SSD, and two 1.2 TB HDDs. We ran our experiments for all the proposed methodologies on the two datasets comprising of COVID-19 related tweets: (1) 2 million (2M) tweets with 27,888,395 ground predicates and (2) 4 million (4M) tweets with 62,076,958 ground predicates.

Scala 2.11.8, Apache Spark 2.4.6[4], and Apache Hadoop 2.7.6[5] were the primary software tools used throughout the entire experimentation. We also used Python 2.7.12 and Bash scripting for additional automation of processes and intermediary data transformation. We built and ran the SRLearn experiment pipeline through a series of Bash scripts for ease of experimentation and evaluation.

[2]https://www.geni.net/
[3]https://www.emulab.net/portal/frontpage.php
[4]http://spark.apache.org
[5]http://hadoop.apache.org

## 5.2 Datasets

Twitter is the primary source of data for all our experimentation. We collected tweets from Twitter API on Covid-19 topic. We used the following keywords to collect the tweets. ”Corona virus”, ”Coronavirus”, ”coronavirus”, ”corona virus”, ”bats”, ”COVID-19”, ”COVID”, ”COVID19”, ”coronavirususa”, ”COVID19US”, ”CoronaVirusUpdates”, ”Corona virus outbreak”, ”Coronavirus travel risk”, ”Wuhan”, ”lockdown”, ”pandemic”, ”fatalities”, ”quarantine”, ”Quarantine”, ”positive cases”, ”stimulus package”, ”social distancing”, ”social-distancing”, ”PhysicalDistancing”, ”physical distancing”, ”StayHome”, ”stayhome”, ”stay at home”, ”StayAtHome”, ”hospitalization”, ”virus” and ”Virus”. We curated two sets of input data in JSON format: (1) 2 million tweets, (2) 3 million tweets and 4 million tweets. The 2 million tweets comprise tweets dated from March 17, 2020, 9:18:30 AM to March 17, 2020, 8:25:14 PM. The 4 million tweets dataset was made by concatenating another 2 million tweets, streamed from Jul 27, 2020, 9:01:05 PM to Jul 28, 2020, 8:08:51 AM with the former 2 million tweets. The 3 million tweets were a fresh dataset collected in early 2021.

We then created evidence data each from these 3 JSON files, consisting of our previously defined ground predicates. The 2 million evidence dataset consists of 27,888,395 ground predicates (excluding *retweetedBy* predicates, which are introduced in the later stages of graph construction). The 3 million dataset consists of a total of 36,545,836 such ground predicates (excluding *retweetedBy* predicates). The 4 million dataset consists of a total of 62,076,958 such ground predicates (excluding *retweetedBy* predicates). Table 5 denotes the constituents of the evidence datasets, i.e. number of each type of ground

Table 5: Number of ground predicates in each dataset

| # of ground predicates | 2M | 3M | 4M |
|---|---|---|---|
| tweeted(userID,tweetID) | 2,061,498 | 3,000,000 | 4,482,693 |
| containsLink(tweetID,link) | 300,604 | 609,983 | 763,882 |
| containsHashtag(tweetID,hashtag) | 508,692 | 1,799,455 | 1,077,197 |
| mentions(tweetID,userID) | 1,917,996 | 2,868,590 | 4,057,902 |
| friendsCount(userID,count) | 1,999,772 | 3,000,000 | 3,999,772 |
| followersCount(userID,count) | 1,999,772 | 3,000,000 | 3,999,772 |
| statusesCount(userID,count) | 1,999,772 | 3,000,000 | 3,999,772 |
| retweetCount(tweetID,count) | 1,586,013 | 701,237 | 3,103,096 |
| retweeted(tweetID,userID) | 1,586,013 | 1,942,390 | 3,103,096 |
| verified(userID) | 26,529 | 58,082 | 58,165 |
| friend(userID1,userID2) | 7,857,534 | 8,094,294 | 18,004,700 |
| isFollowedBy(userID1,userID2) | 4,494,404 | 7,199,702 | 12,765,116 |
| isPossiblySensitive(userID2) | 20,062 | 30,950 | 32,870 |

predicate present in the evidence dataset.

## 5.3 Evaluation and Results

We have used the three evidence datasets (one each from 2M, 3M, and 4M tweets) for evaluating the performance of SRLearn and comparing it with Vanilla SRLearn, ProbKB, and XGBoost in the following sections.

### 5.3.1 SRLearn

We implemented the algorithms 1 and 4 to build the user-centric graphs using the 2 million tweets data set (2M), 3 million tweets data set (3M), and 4 million tweets data set (4M), simultaneously. Table 6 shows the dimensions of the user-centric graphs built. In the subsequent subsections, we elaborate on the experimentation results at various stages.

Table 6: Graph dimensions for evidence datasets considered

| Dataset | Number of vertices | Number of edges |
|---|---|---|
| 2million tweets | 1,655,350 | 12,171,014 |
| 3million tweets | 1,705,926 | 14,017,453 |
| 4million tweets | 2,653,154 | 26,444,708 |

#### 5.3.1.1 Graph partitioning

Using the ParHIP graph partitioning tool, we created 32 and 64 partitions of each of the 2M, 3M, and 4M tweets using "eco" preconfiguration. 2 cores per node in the cluster were used for this tool. Thus we now have 4 different data sets to be executed by BoostSRL in the later stages: (a) 2 million tweets with 32 partitions, (b) 2 million tweets with 64 partitions, (c) 3 million tweets with 32 partitions, (d) 3 million tweets with 64 partitions, (e) 4 million tweets with 32 partitions and (d) 4 million tweets with 64 partitions

For the 2M and 4M datasets, that were part of our former evaluation, we calculated the vertex distribution (Fig. 13 and Fig. 15 for 2M data set and, Fig. 17 and Fig. 19 for 4M data set) to gain an understanding of vertex distribution for all partitioning techniques. We also calculated the total number of ground predicates per partition (Fig. 12 and Fig. 14 for 2M data set and, Fig. 16 and Fig. 18 for 4M data set) to visualize how large or small the partitions were in terms of the total number of ground predicates associated with them. This evaluation was essential to understand how balanced or sparse the graph partitioning technique is. Furthermore, this was a learning curve for us to understand and implement this tool on the weighted social graphs we were working with.

This confirms our prior observation that *eco* preconfiguration gives a comparable

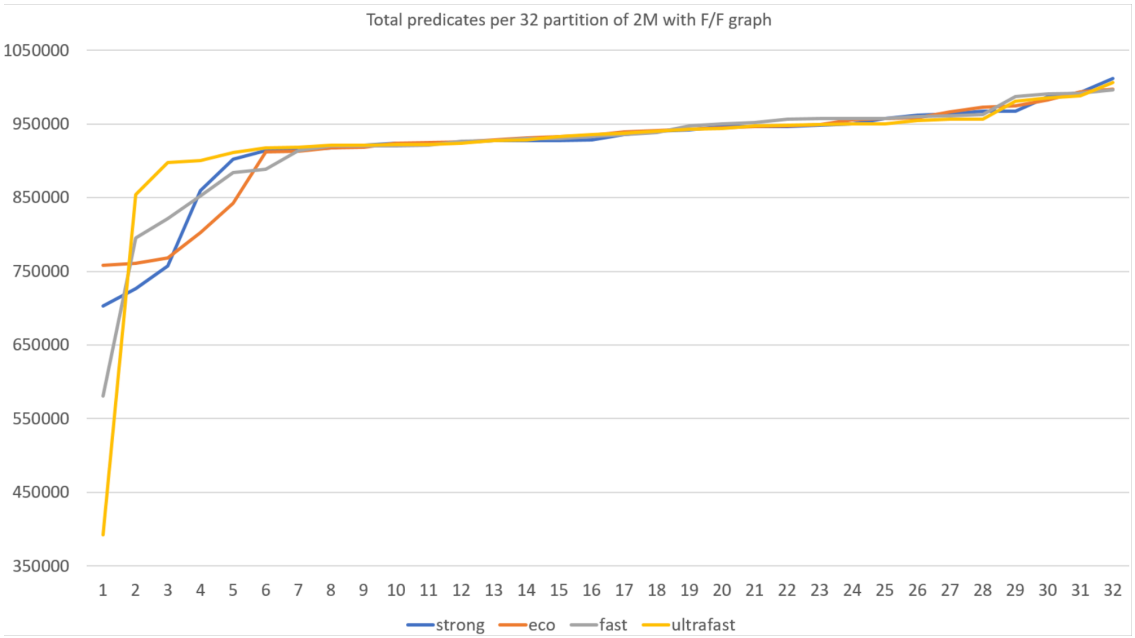



Figure 12: Distribution of ground predicates with varying preconfiguration on 32 partitions (2M tweets)

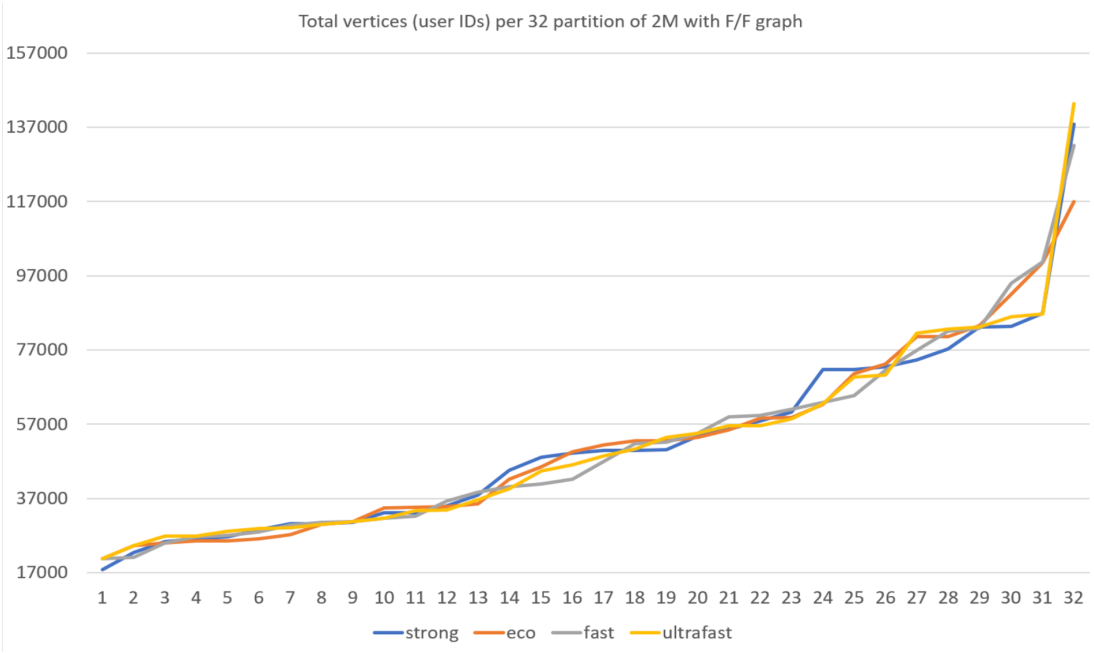



Figure 13: Distribution of vertices (user IDs) with varying preconfiguration on 32 partitions (2M tweets)

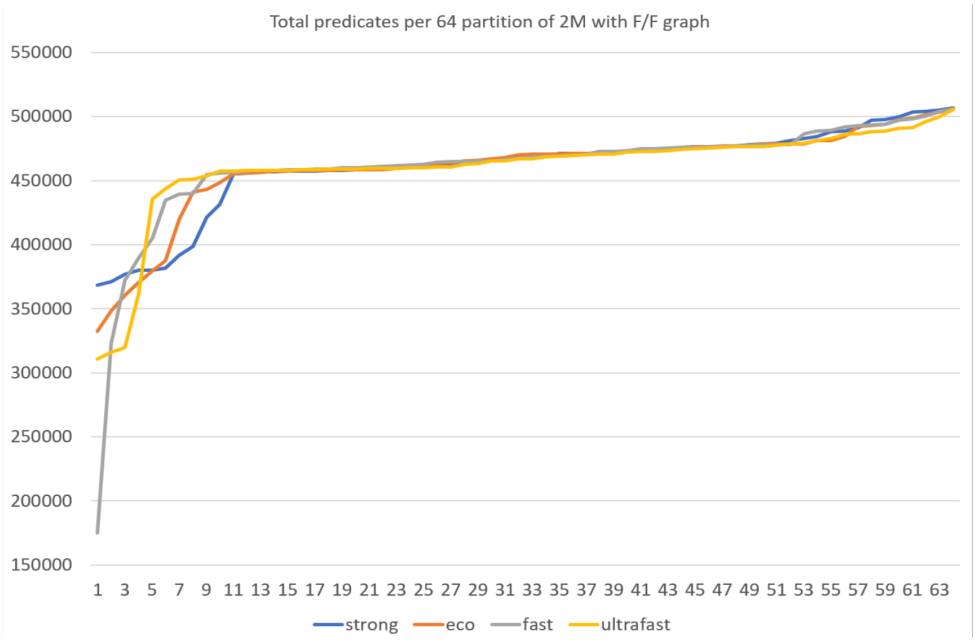



Figure 14: Distribution of ground predicates with varying preconfiguration on 64 partitions (2M tweets)

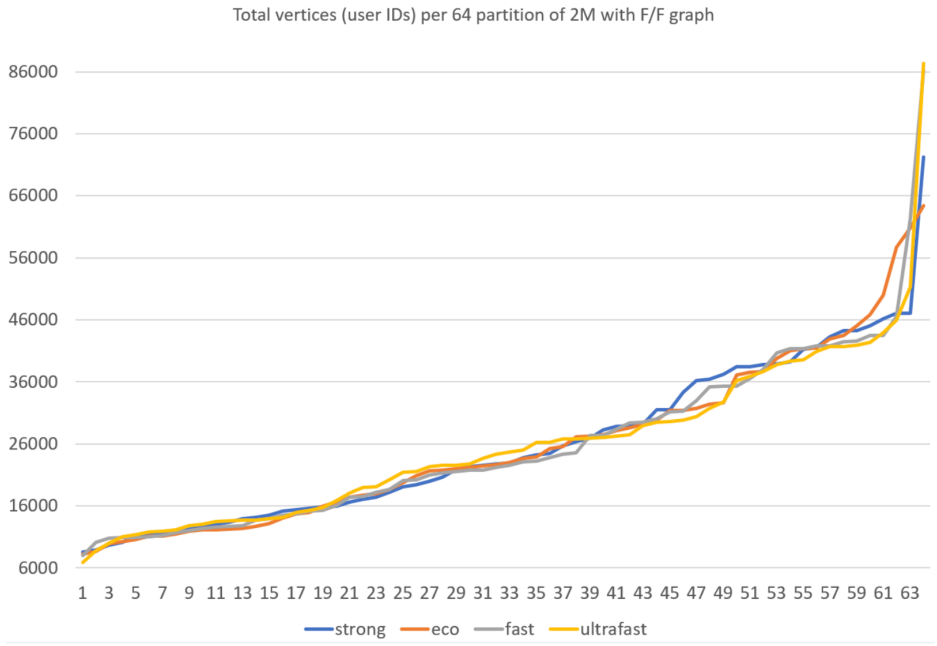



Figure 15: Distribution of vertices (user IDs) with varying preconfiguration on 64 partitions (2M tweets)

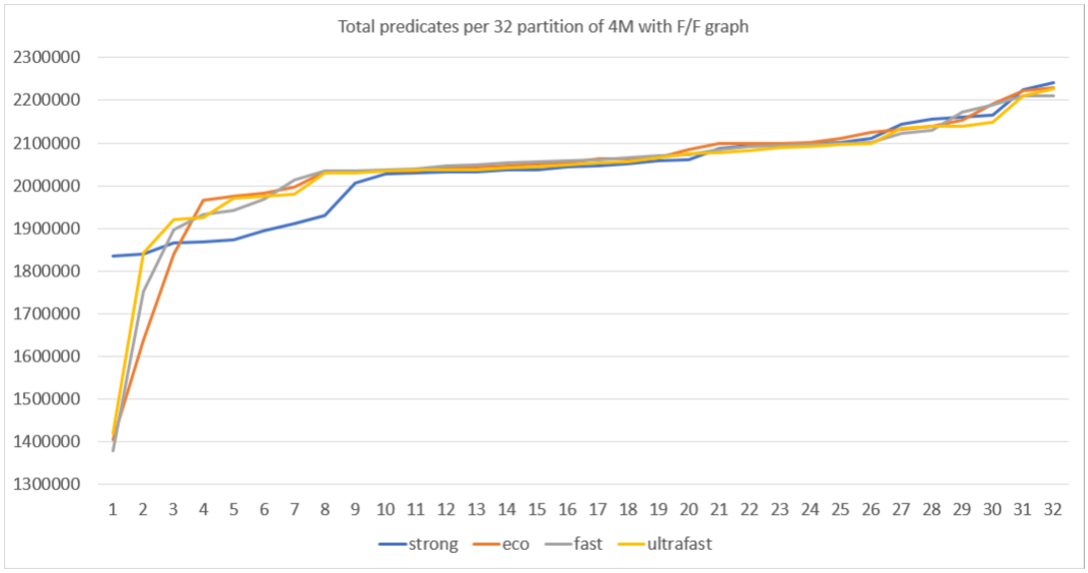



Figure 16: Distribution of ground predicates with varying preconfiguration on 32 partitions (4M tweets)

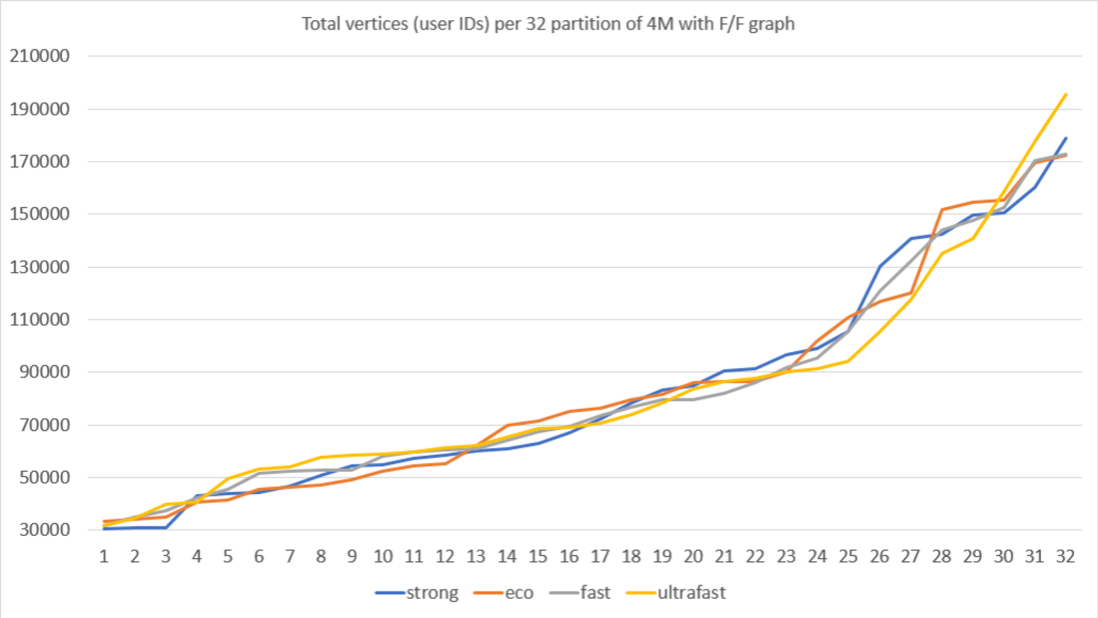



Figure 17: Distribution of vertices (user IDs) with varying preconfiguration on 32 partitions (4M tweets)

balanced set of partitions for all the graphs like *strong* preconfiguration but at a much faster rate (Table 4). This significantly reduces our overall runtime. Table 7 shows the time taken for the initial first stages of SRLearn.

#### 5.3.1.2 Alchemy evaluation

We implemented Alchemy's structure learning [30] for our preliminary phase of experimentation. We began with a small sample of tweets (20,000 tweets) collected during 2016-2017 and created an input file with 22.4 million ground predicates containing 17 unique predicates. The file size was 741 MB. We first ran Alchemy's structure learning approach on a single machine. Unfortunately, it was very slow even on a small number of ground predicates. It ran for 138 hours on just 15,417 ground predicates and eventually

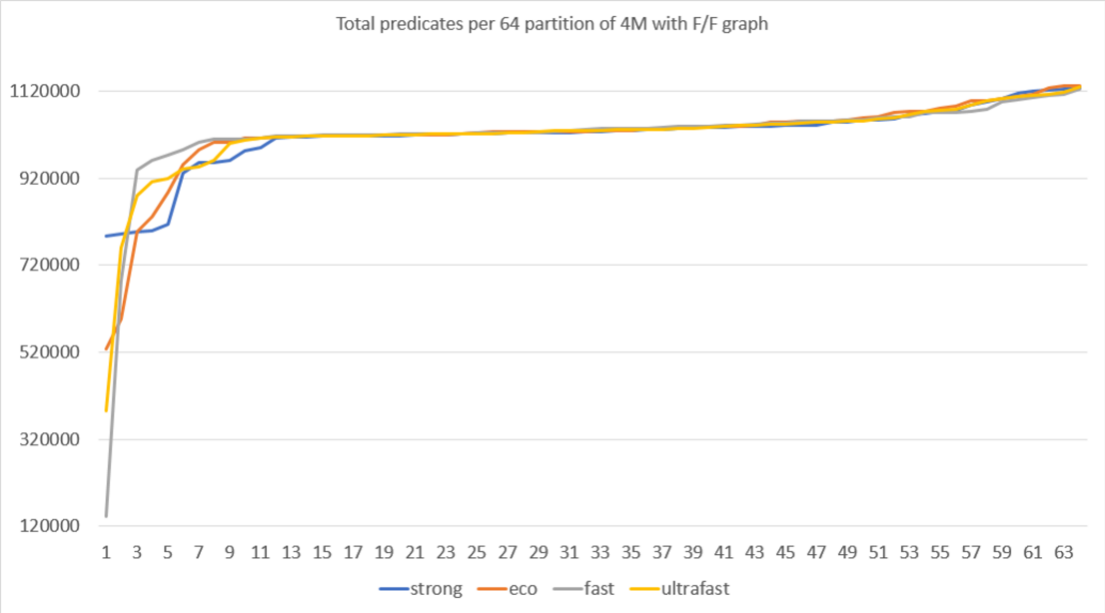



Figure 18: Distribution of ground predicates with varying preconfiguration on 64 partitions (4M tweets)

failed due to a memory allocation error. This is a clear indication of the problem of MLN structure learning in a centralized setting.

We then experimented SRLearn's performance on 22.4 million ground predicates. We first ran SRLearn on a 16-node cluster to produce 128 partitions of the ground predicates. This required a total of 2 hours and 30 minutes. For graph partitioning, we used KaHIP [33] and tested different strategies that provide a tradeoff between partition quality and time for different types of graphs. Fig. 20 shows the quality of partitioning in terms of the number of vertices (i.e., users) of the user-centric graph per partition in sorted order. Compared to the Eco, EcoStrong, and Strong settings in KaHIP, StrongSocial produced the most balanced partitions.

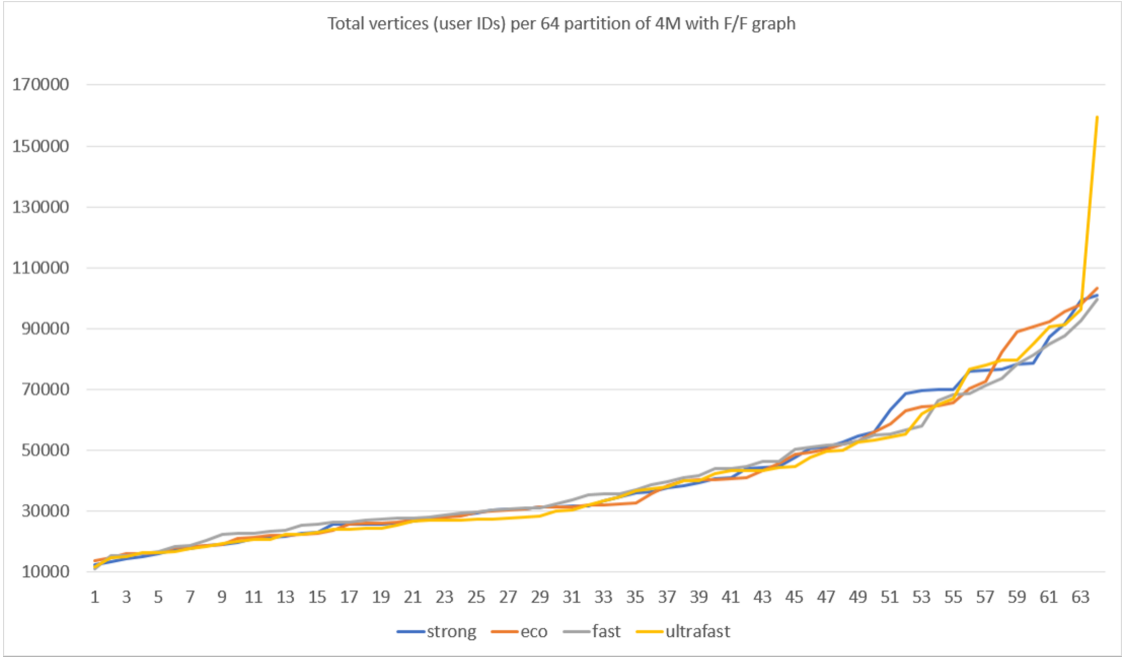



Figure 19: Distribution of vertices (user IDs) with varying preconfiguration on 64 partitions (4M tweets)

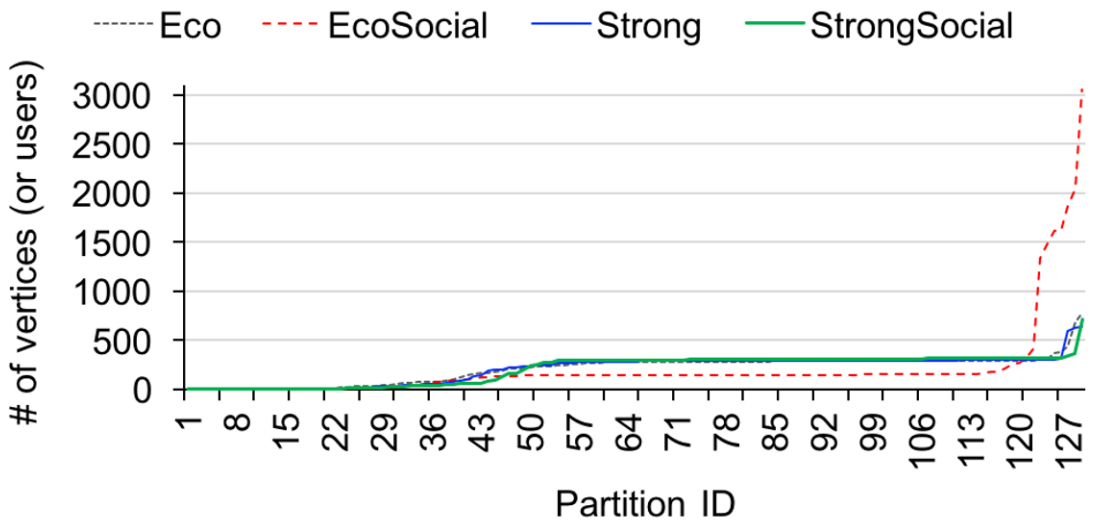



Figure 20: Quality of graph partitioning by KaHIP

Table 7: Runtime for different stages of SRLearn prior to structure learning

| Dataset | User-centric graph creation | Graph partitioning | Regroup predicates |
|---|---|---|---|
| 2M 32 partitions | 4m 43s | 3m 20s | 1m 27s |
| 2M 64 partitions | 4m 55s | 6m 23s | 2m 22s |
| 3M 32 partitions | 3m 58s | 3m 9s | 2m 20s |
| 3M 64 partitions | 3m 52s | 6m 1s | 2m 6s |
| 4M 32 partitions | 8m 18s | 4m 20s | 2m 53s |
| 4M 64 partitions | 8m 36s | 4m 11s | 3m 7s |

Table 8: Sample of rules learnt by SRLearn coupled with Alchemy

| First-order rules (in clausal form) |
|---|
| `friend(v0,v1) \V !verified(v0)` |
| `!mentions(v0,v1) \V !retweeted(v2,v3) \V !tweeted(v1,v3) \V !tweeted(v2,v0)` |
| `!containsHashtag(v0,v1) \V containsHashtag(v2,v1) \V !mentions(v0,v3) \V !mentions(v2,v3)` |

Next, we ran Alchemy's structure learning on each partition to generate rules using a 32-node cluster. The 128 partitions were distributed across 32 nodes. It took a total of 66 hours and 33 minutes to run the structure learning on the partitions. Although a node had many cores, we ran Alchemy on the partitions serially on the node. Some of the learned rules learned are displayed in Table 8.

Alchemy did provide a promising solution to rule learning and was a good candidate for our system. However, considering 66 and 1/2 hours on merely 20,000 tweets, it was not a lucrative option for large-scale data going further.

Table 9: Class imbalance, i.e. ratio of negatives to positives in 2M, 3M and 4M datasets

| Dataset | Number of negatives: Number of positives |
|---|---|
| 2M | 296:3 |
| 3M | 96:1 |
| 4M | 362:3 |

#### 5.3.1.3 BoostSRL evaluation

Once the predicates are grouped into the respective partitions and the input files are set up on all nodes for 5-fold cross-validation, we began the learning and inferencing using BoostSRL. We learnt the models on the target ground predicate *isPossiblySensitive*. Table 9 highlights the class imbalance present in each of the datasets for this target predicate.

For this stage, we experimented with both the RDN-Boost as well as CSSRL algorithms of BoostSRL. Table 10 describes the runtime for both algorithms. Note that we ran CSSRL using a library of BoostSRL, i.e SRLBoost[6], that is both lighter and hence runs faster than the former jar.

Table 11 shows the AUC ROC values for both algorithms. Since the AUC ROC values are almost similar for both approaches, while the CSSRL also considers the class imbalance, we choose to use the CSSRL algorithm through the SRLBoost library for structure learning.

Table 18 shows the global AUC PR for the target ground predicate *isPossiblySensitive* for SRLearn. Table 19 shows the global AUC ROC for the target ground predicate *isPossiblySensitive* for SRLearn.

[6]https://github.com/srlearn/SRLBoost

Table 10: Time taken to train models by BoostSRL for target predicate *isPossiblySensitive*

| Dataset | Alpha/Beta | Algorithm | Train time |
|---|---|---|---|
| 2M 32 partitions | - | RDN-Boost | 6h 16m 4s±41m 1s |
| | 0/40 | CSSRL | 17m 47s±5m 34s |
| | 40/40 | CSSRL | **12m 27s±1m 21s** |
| 2M 64 partitions | - | RDN-Boost | 6h 57m 19s±1h 53m 6s |
| | 0/40 | CSSRL | 11m 33s±3m 13s |
| | 40/40 | CSSRL | **2m 50s±20s** |
| 3M 32 partitions | - | RDN-Boost | **2h 33m 02s±34m 07s** |
| | 0/40 | CSSRL | 2h 41m 17s±52m 58s |
| | 40/40 | CSSRL | 6h 27m 15s± 10m 04s |
| 3M 64 partitions | - | RDN-Boost | 3h 17m 22s±1h 14m 11s |
| | 0/40 | CSSRL | **2h 31m 44s±16m 07s** |
| | 40/40 | CSSRL | 4h 27m 46s±20m 27s |
| 4M 32 partitions | - | RDN-Boost | 7h 35m 30s±2h 24m 44s |
| | 0/40 | CSSRL | **19m 13s±03m 42s** |
| | 40/40 | CSSRL | 19m 40s±8m 48s |
| 4M 64 partitions | - | RDN-Boost | 5h 50m 35s±1h 14m 19s |
| | 0/40 | CSSRL | **1h 02m 15s±10m 22s** |
| | 40/40 | CSSRL | 1h 35m 20s±23m 25s |

### 5.3.2 Vanilla SRLearn

Vanilla SRLearn is to show the effectiveness of rule mining in the absence of SRLearn. This version of the experimentation is intended to study the performance of BoostSRL on the group of ground predicates that are ”randomly” distributed, without the intricacies of graph building, partitioning, and regrouping predicates into the respective partitions.

We distribute the ground predicates randomly across the machines on the basis of tweet IDs. We group the predicates *tweeted(userID, tweetID)*, *containsLink(tweetID,*

Table 11: AUC ROC for both algorithms of BoostSRL for target predicate *isPossiblySensitive*

| Dataset | Alpha/Beta | Algorithm | AUC ROC |
|---|---|---|---|
| 2M 32 partitions | - | RDN-Boost | **0.823±0.004** |
| | 0/40 | CSSRL | 0.814±0.002 |
| | 40/40 | CSSRL | 0.783±0.005 |
| 2M 64 partitions | - | RDN-Boost | **0.827±0.005** |
| | 0/40 | CSSRL | 0.818±0.003 |
| | 40/40 | CSSRL | 0.793±0.003 |
| 3M 32 partitions | - | RDN-Boost | **0.787±0.004** |
| | 0/40 | CSSRL | 0.7781±0.004 |
| | 40/40 | CSSRL | 0.7485±0.003 |
| 3M 64 partitions | - | RDN-Boost | **0.802±0.002** |
| | 0/40 | CSSRL | 0.7917±0.004 |
| | 40/40 | CSSRL | 0.7696±0.004 |
| 4M 32 partitions | - | RDN-Boost | **0.826±0.002** |
| | 0/40 | CSSRL | 0.817±0.002 |
| | 40/40 | CSSRL | 0.786±0.002 |
| 4M 64 partitions | - | RDN-Boost | **0.836±0.004** |
| | 0/40 | CSSRL | 0.828±0.002 |
| | 40/40 | CSSRL | 0.804±0.004 |

*link)*, *mentions(tweetID, userID)*, *containsHashtag(tweetID, hashtag)*, *isPossiblySensitive(tweetID)* and *retweetCount(tweetID, count)* by user ID first. We group the remaining predicates into partitions with respect to the tweet ID. For example, consider there are two predicates: *tweeted(userID1, tweetID1)*, and *tweeted(userID1, tweetID2)*, which are placed in partition 1 and partitition 2 respectively. We place the predicate *friend(userID1, userID2)* only in either partition 1 or partition 2, not both. Thus, we divide the predicates for both 2 million and 4 million tweets, into 32 and 64 partitions. Once the predicates are divided into the partitions, we distributed these partitions across the nodes in the same

manner as before (with SRLearn) with 5-fold cross validation folders. We then trigger BoostSRL to learn and infer on these randomly distributed predicates on the target ground predicate *isPossiblySensitive*. As we can see in Table 19, Vanilla SRLearn fails to build an effective classification model with random distribution of tweets and their predicates.

### 5.3.3 ProbKB

In this section, we describe our experimental setup and evaluation criteria for ProbKB. We first list out the data preprocessing steps to create the knowledge base and the set of rules to be learned, followed by the experimental setup used and the evaluation technique followed.

#### 5.3.3.1 Data Preprocessing

A knowledge base is first constructed using the evidence files used to learn rules with SRLearn. With tweets originating from different countries and tweeted in diverse languages, many of the characters are not UTF-8 encoded. Hence the first step involved in creating the knowledge base for ProbKB was converting these characters into a hexadecimal format.

The knowledge base used by ProbKB is represented as a collection of subjects, predicates, and objects represented as *predicate subject object*. We converted the evidence into the following format. The predicates *verified*, *isPossiblySensitive* and *malicious* operate on a subject and do not have an object tag. To transform them to a form that is easily parsed by ProbKB, we added a constant object to these predicates. In addition to this, we introduced a new predicate, which relates the constant object to a tweet

Table 12: Graph dimensions for evidence datasets considered

| |
|---|
| p(x, y) ← q(x, y) |
| p(x, y) ← q(y, x) |
| p(x, y) ← q(z, x), r(z, y) |
| p(x, y) ← q(x, z), r(z, y) |
| p(x, y) ← q(z, x), r(y, z) |
| p(x, y) ← q(x, z), r(y, z) |

ID or user ID from which the evidence originated initially. For example, for a tweet ID *X* that contains a malicious link *Y*, we represent it as *malicious(Y)*, we use *malicious Y _malConst_* and *existsMalTweet _malConst_ X* as the facts. Similarly, for a tweet id *X* that is tweeted by a user, with the id *X*, verified by Twitter, is represented in the evidence as *verified(Y)*. We generate corresponding facts from the evidence as *verified Y _isVerConst_* and *existsVerTweet _isVerConst_ X*.

#### 5.3.3.2 Rule Description

With the aim to mine implied knowledge from known facts, ProbKB is designed to mine just first-order Horn clauses. It supports 6 rule types listed in Table 12, where $p$, $q$ and $r$ represent the predicates. The variables $x$, $y$ and $z$ represent the subjects and objects.

In order to learn all possible implied facts, we use all permutations of the predicates to create the rules. In total, there were 168 type 1 and 2 rules and 2196 rules of type 3, 4, 5 and 6. The Schema closure graph, represented as $G$ = (*Predicate*, *Edge 1* $\cup$ *Edge 1*), used for parallel rule mining is shown in Table 13.

Table 13: The schema closure graph used for rule mining

| Predicate | Edge 1 | Edge 2 |
|---|---|---|
| containsHashtag | tweetID | hashtag |
| containsLink | tweetID | link |
| followersCount | userID | flCount |
| friendsCount | userID | frCount |
| isPossiblySensitive | tweetID | isPSConst |
| malicious | link | malConst |
| mentions | tweetID | userID |
| retweetCount | tweetID | rtCount |
| retweeted | userID | tweetID |
| retweetedBy | tweetID | userID |
| statusesCount | userID | stCount |
| tweeted | userID | tweetID |
| verified | userID | verConst |
| existsMalTweet | malConst | tweetID |
| existsPSTweet | isPSConst | tweetID |
| existsVerTweet | verConst | tweetID |
| friend | userID | secUserID |
| isFollowedBy | userID | secUserID |

#### 5.3.3.3 ProbKB evaluation

We also used AUC ROC and AUC PR to evaluate ProbKB's performance on the datasets. As we can see in Table 19, Table 18 Table 22 and Table 21, ProbKB, even though is scalable, is unable learn a reliable model, and hence its results are not as accurate as the contemporaries.

### 5.3.4 XGBoost

We also compare our results to that with XGBoost, a state-of-the-art gradient tree boosting algorithm used to address machine learning problems. To make it comparable

with the distributed nature of SRLearn, we implement XGBoost through the Spark API on a 16-node cluster of CloudLab.

#### 5.3.4.1 Data Preprocessing

XGBoost, although scalable, ingests data in the relational form. Therefore, the ground predicates are converted into a tabular format, where each row represents a tweet. The columns translate to each of the ground predicate corresponding to the tweet. This is either a direct translation of the predicate or is one-hot-encoded manually.

With the direct translation of ground predicates, which have a one-to-one relationship with a tweet ID, we obtain the following columns.

- *tweetID*: Tweet ID of the tweet represented in the row;
- *userID*: User ID which tweeted the corresponding tweet ID;
- *isVerified*: 1 if user ID is verified on Twitter, 0 otherwise;
- *friendsCount*: Number of friends the user ID has;
- *followersCount*: Number of followers the user ID has;
- *statusesCount*: Number of statuses/posts made by the user;
- *retweetCount*: Number of times the tweet has been retweeted;
- *isPossiblySensitive*: 1 if the tweet ID is marked as *isPossiblySensitive* by Twitter, 0 otherwise.

For remaining ground predicates that have a many-to-one relationship with tweet ID, we employ a one-hot encoding mechanism. An example of a ground predicate demonstrating this behavior is *containsHashtag*. Multiple hashtags can be used in one tweet. Therefore, we have multiple *containsHashtag* predicates associated with the tweet ID. The predicates *containsLink* and *mentions* also exhibit a similar behaviour. To one-hot encode, these predicates, we create a column with the suffix ${}_o nehotencoded$, which contains the one-hot encoding in the form of a binary string. This binary string is of length $n$, which is the total number of unique hashtags or URLs or mentioned user IDs present in the aggregated input. The $i^{th}$ character in the string corresponds to the $i^{th}$ hashtag or URL or mentioned user ID. If the $i^{th}$ character in the string is 0, then the $i^{th}$ hashtag or URL or mentioned user ID is not associated with the tweet ID. If the $i^{th}$ character in the string is 1, then the $i^{th}$ hashtag or URL or mentioned user ID is associated with the tweet ID.

For ground predicates, *retweeted*, *friend* and *isFollowedBy*, which do not have the tweet ID in their variables, we use the user ID corresponding to the tweet ID as a deciding factor in the one-hot encoding process. We get the following columns after one-hot encoding.

- $containsHashtag\_onehotencoded$: One-hot encoding of ground predicate *containsHashtag*;
- $containsLink\_onehotencoded$: One-hot encoding of ground predicate *containsLink*;
- $mentions\_onehotencoded$: One-hot encoding of ground predicate *mentions*;

Table 14: Spark-submit configuration

| Parameter | Value |
|---|---|
| driver-memory | 100G |
| num-executors | 15 |
| executor-core | 1 |
| executor-memory | 150G |

- $retweeted\_onehotencoded$: One-hot encoding of ground predicate *retweeted*;
- $friend\_onehotencoded$: One-hot encoding of ground predicate *friend*;
- $isFollowedBy\_onehotencoded$: One-hot encoding of ground predicate *isFollowedBy*.

We use these one-hot encoded files as input for XGBoost to build a classification model for *isPossiblySensitive*.

#### 5.3.4.2 Training and testing with XGBoost

Table 14 denotes the spark-submit configuration used to run XGBoost on the 2M and 4M dataset. Table 15 denotes the XGBoost parameters that were finalized after iterating over a varied set of parameter configuration. In order to train the model, we used the rows corresponding to the tweet IDs that were also used in the positive and negative examples of training dataset for SRLearn. Similarly, we determined the sample for testing the classification model efficiency. We created similar sets for all 5 folds of cross-validation.

Table 15: XGBoost Classifier parameters

| Parameter | Value |
|---|---|
| missing | 0.0 |
| eta | 0.1f |
| max_depth | 2 |
| objective | binary:logistic |
| scale_pos_weight | $\frac{Numberofnegativeexamples}{Numberofpositiveexamples}$ |
| num_round | 10 |
| num_early_stopping_rounds | 15 |
| num_workers | 15 |
| tree_method | hist |

Table 16: Time taken to learn models for XGBoost and SRLearn

| Dataset | SRLearn train time | XGboost train time |
|---|---|---|
| 2M 32 partitions | **12m 27s±1m 21s** | 14h 37m 22s±32m 35s |
| 2M 64 partitions | **2m 50s±20**s | 14h 37m 22s±32m 35s |
| 3M 32 partitions | **2h 41m 17s±52m 58s** | 2d 10h 35m 30s±3h 46m 10s |
| 3M 64 partitions | **2h 31m 44s±16m 07s** | 2d 10h 35m 30s±3h 46m 10s |
| 4M 32 partitions | **19m 13s±3m 42s** | 2d 0h 17m 11s±33m 26s |
| 4M 64 partitions | **1h 2m 15s±10m 22s** | 2d 0h 17m 11s±33m 26s |

#### 5.3.4.3 AUC calculation for XGBoost

To ensure fair evaluation, we use the results from XGBoost (i.e. the predicted class, and the predicted class probabilities) to calculate the AUC ROC and AUC PR values using the same compiled jar that was used for the AUC calculation in the case of SRLearn. The outcomes are shown in Table 18 and Table 19.

### 5.3.5 Performance evaluation of SRLearn with its contemporaries

From Table 18 and Table 19 we see that SRLearn outperforms it's competitive methodologies in terms of AUC ROC. ProbKB, on the other hand, does not provide a very

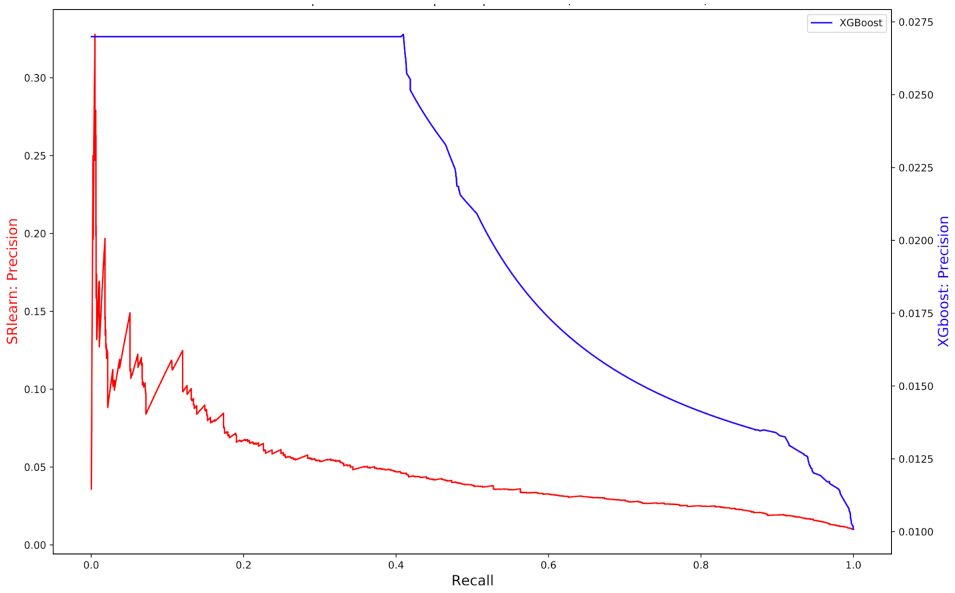



Figure 21: AUC PR curve for 2M 32 partitions

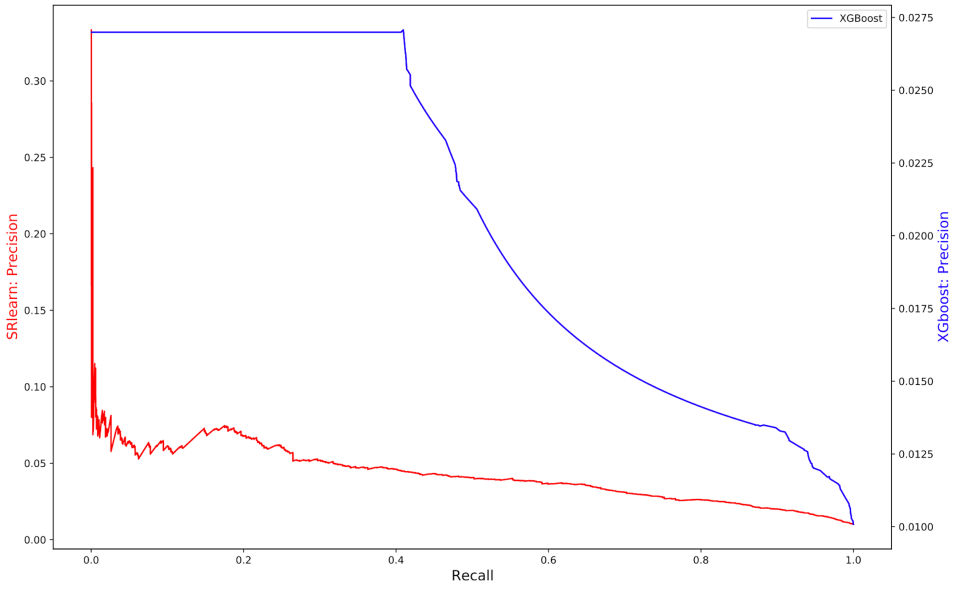



Figure 22: AUC PR curve for 2M 64 partitions

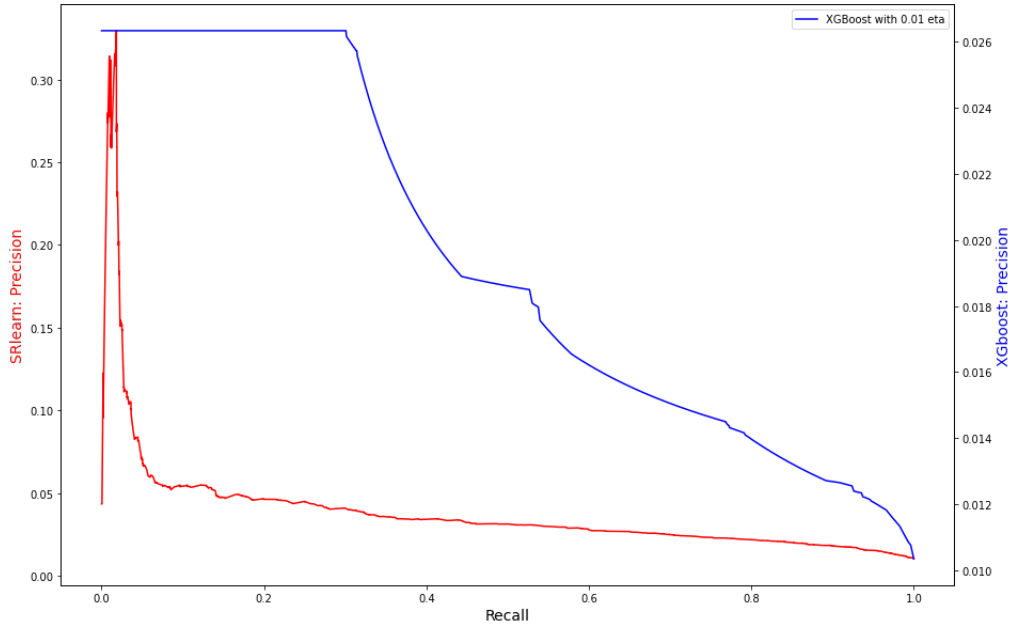



Figure 23: AUC PR curve for 3M 32 partitions

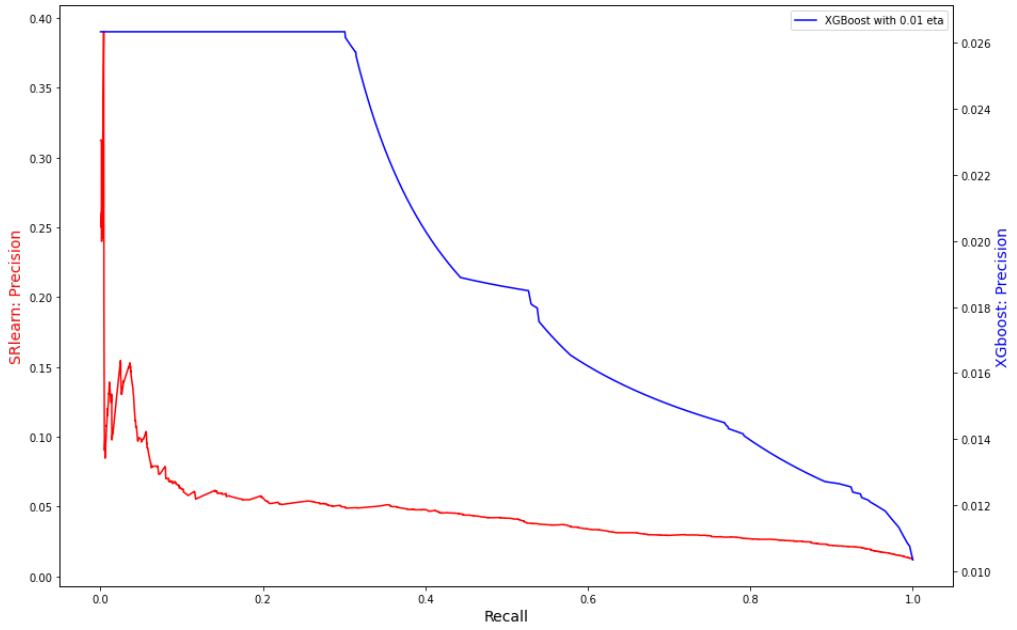



Figure 24: AUC PR curve for 3M 64 partitions

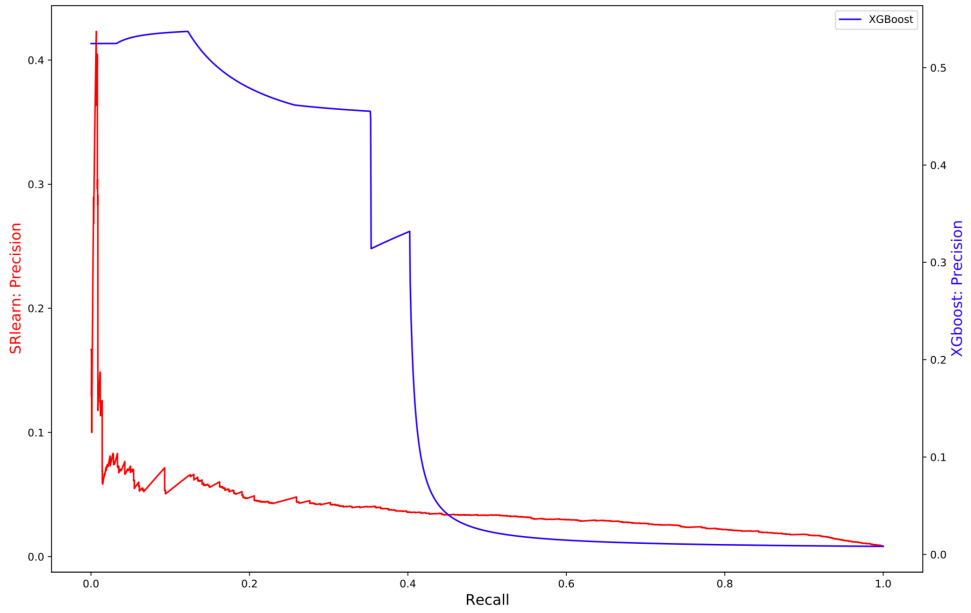



Figure 25: AUC PR curve for 4M 32 partitions

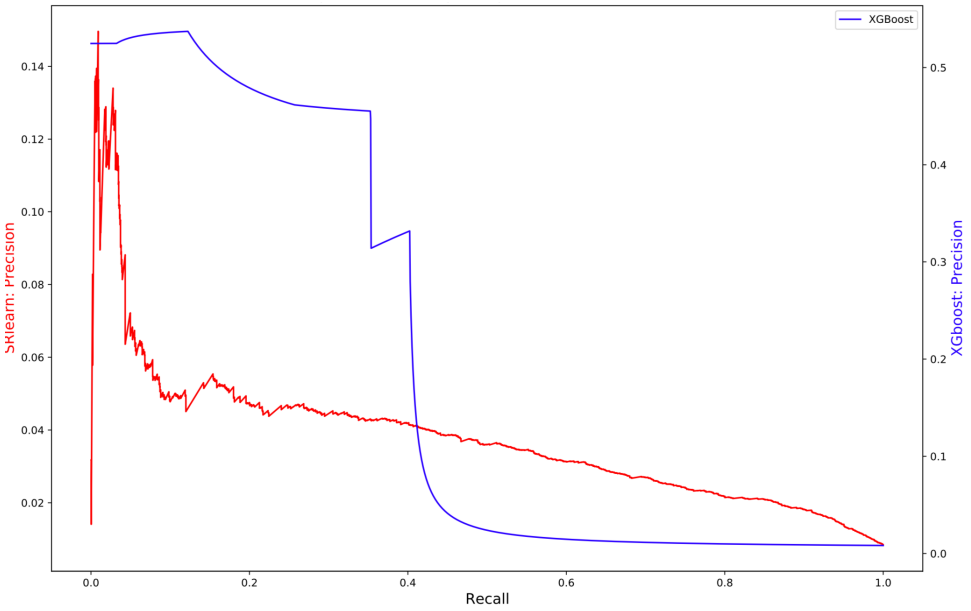



Figure 26: AUC PR curve for 4M 64 partitions

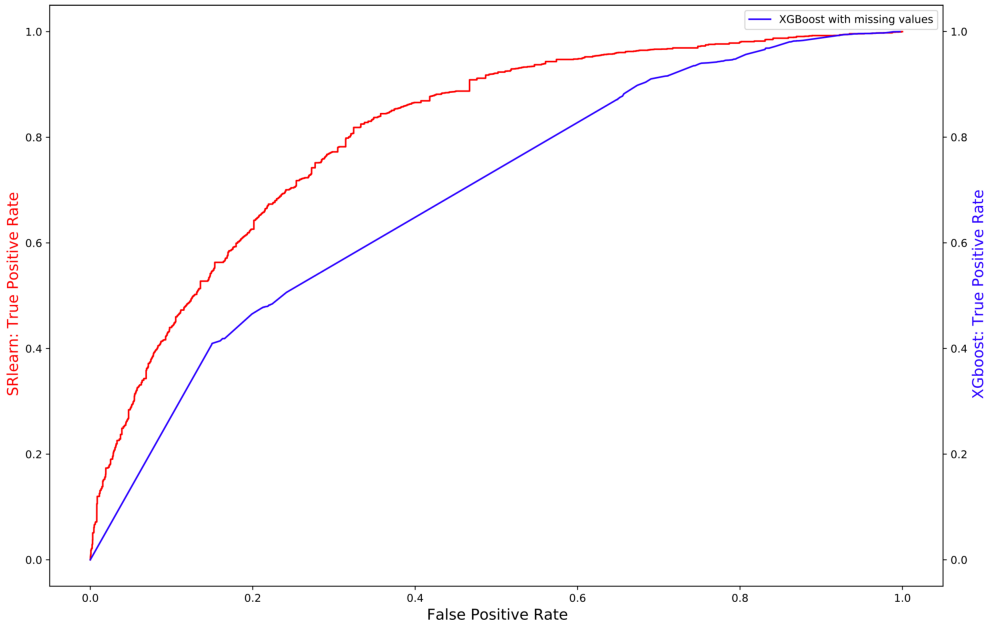



Figure 27: AUC ROC curve for 2M 32 partitions

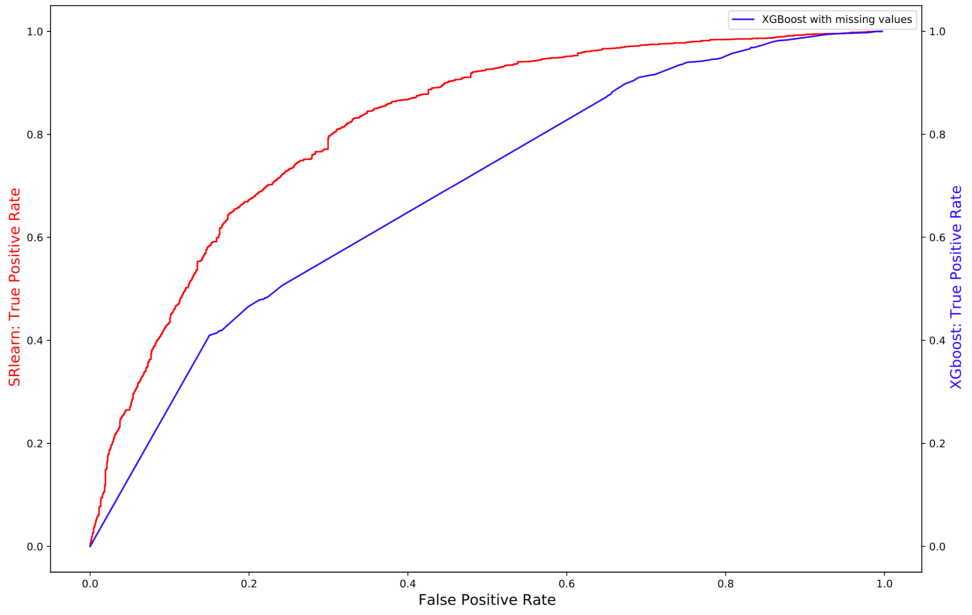



Figure 28: AUC ROC curve for 2M 64 partitions

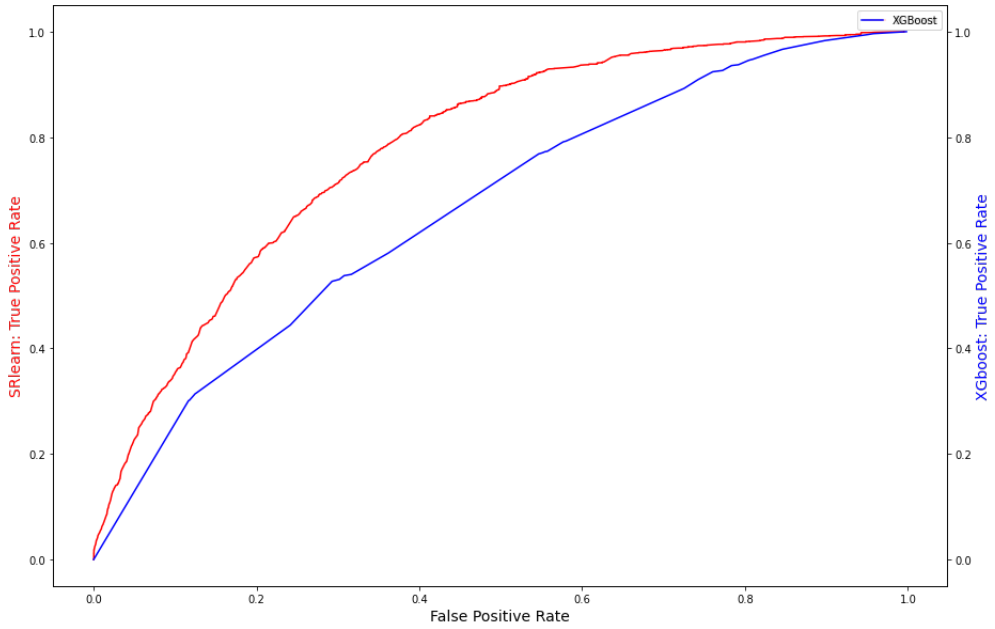



Figure 29: AUC ROC curve for 3M 32 partitions

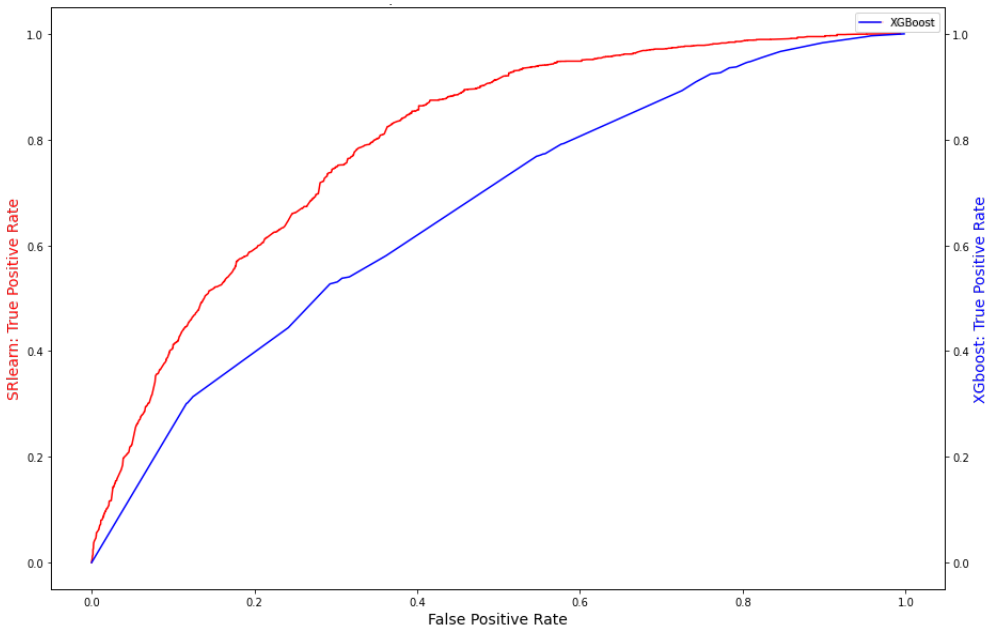



Figure 30: AUC ROC curve for 3M 64 partitions

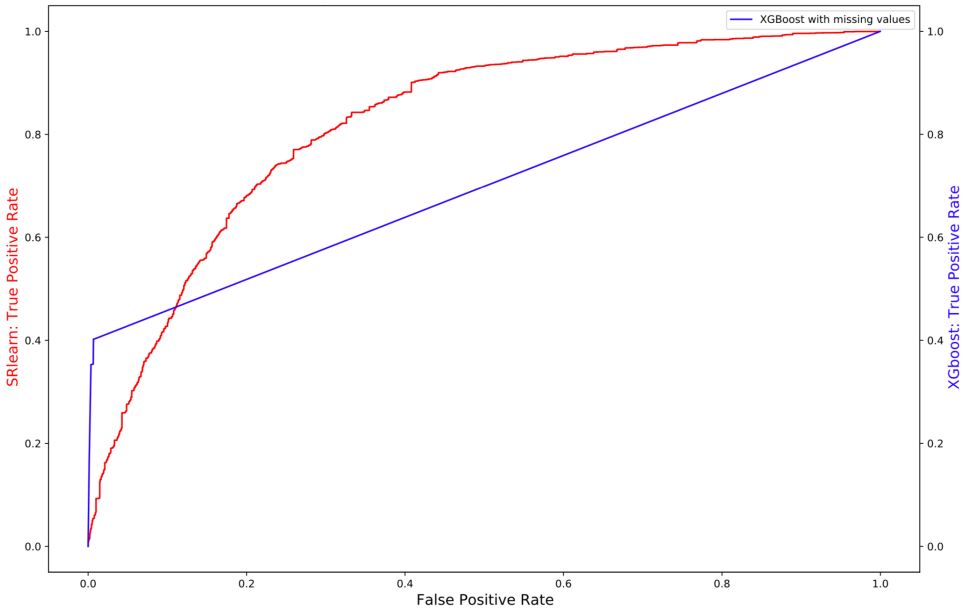



Figure 31: AUC ROC curve for 4M 32 partitions

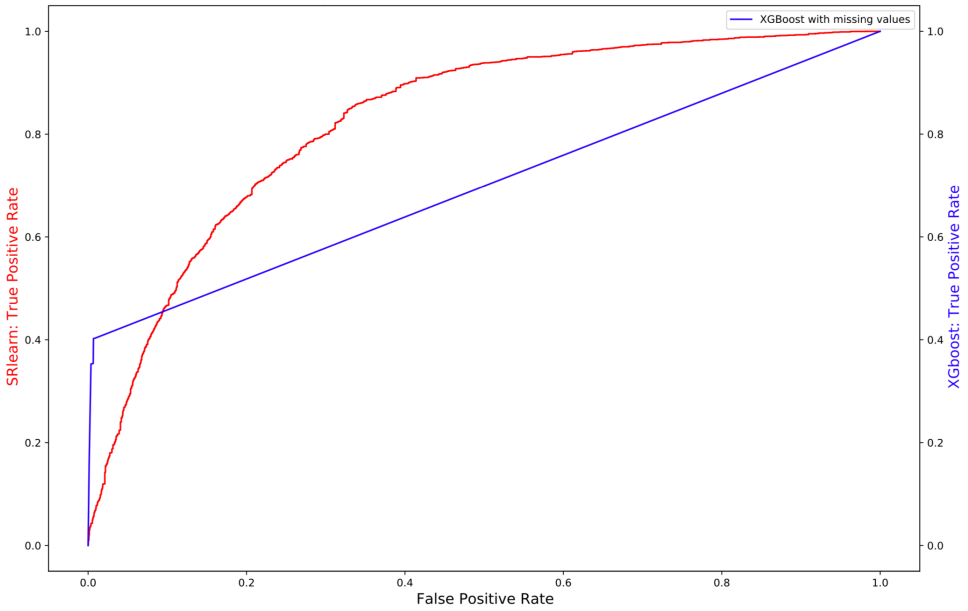



Figure 32: AUC ROC curve for 4M 64 partitions

Table 17: Comparision of total runtime for SRLearn and XGBoost

| Dataset | SRLearn | XGboost |
|---|---|---|
| 2M 32 partitions | **21m 57s±1m 21s** | 14h 37m 22s±32m 35s |
| 2M 64 partitions | **16m 30s±20**s | 14h 37m 22s±32m 35s |
| 3M 32 partitions | **2h 50m 44s±52m 58s** | 2d 10h 35m 30s±3h 46m 10s |
| 3M 64 partitions | **2h 44m 3s±16m 07s** | 2d 10h 35m 30s±3h 46m 10s |
| 4M 32 partitions | **34m 44s±3m 42s** | 2d 0h 17m 11s±33m 26s |
| 4M 64 partitions | **1h 18m 9s±10m 22s** | 2d 0h 17m 11s±33m 26s |

Table 18: AUC PR for target *isPossiblySensitive*: SRLearn vs Competitive methodologies

| Dataset | SRLearn | Vanilla SRLearn | ProbKB | XGBoost |
|---|---|---|---|---|
| 2M 32 partitions | **0.048±0.002** | 0.020±0.000 | 0.009±0.000 | 0.0205±0.000 |
| 2M 64 partitions | **0.043±0.001** | 0.019±0.000 | 0.009±0.000 | 0.0205±0.000 |
| 3M 32 partitions | **0.0408±0.003** | 0.022±0.000 | 0.006±0.001 | 0.019±0.0004 |
| 3M 64 partitions | **0.0448±0.003** | 0.022±0.001 | 0.006±0.001 | 0.019±0.0004 |
| 4M 32 partitions | 0.035±0.002 | 0.020±0.000 | 0.007±0.000 | **0.198±0.005** |
| 4M 64 partitions | 0.039±0.001 | 0.02±0.000 | 0.007±0.000 | **0.198±0.005** |

reliable rule learning methodology. We also show through AUC curves how XGBoost and SRLearn perform. Fig. 21, Fig. 22, Fig. 23, Fig. 24, Fig. 25 and Fig. 26 represent the area under the PR curves for the models learnt by SRLearn and XGBoost. Fig. 27, Fig. 28, Fig. 29, Fig. 30, Fig. 31 and Fig. 32 represent the area under the ROC curves for the models learnt by SRLearn and XGBoost. Davis and Goadrich in [6] have observed that algorithms that optimize the area under the ROC curve are not guaranteed to optimize the area under the PR curve. In our case, for the area under the ROC curve for SRLearn for 2M tweets (Fig. 27 and Fig. 28) and 3M tweets (Fig. 29 and Fig. 30), remains constantly higher than that for XGBoost. For the 4M tweets, the area under the ROC curve for SRLearn starts below that of XGBoost. However, it gradually picks up after a threshold

Table 19: AUC ROC for target *isPossiblySensitive*: SRLearn vs Competitive methodologies

| Dataset | SRLearn | Vanilla SRLearn | ProbKB | XGBoost |
|---|---|---|---|---|
| 2M 32 partitions | **0.814±0.002** | 0.704±0.002 | 0.5±0.000 | 0.687±0.004 |
| 2M 64 partitions | **0.818±0.003** | 0.736±0.001 | 0.5±0.000 | 0.687±0.004 |
| 3M 32 partitions | **0.778±0.002** | 0.703±0.004 | 0.5±0.0 | 0.670±0.004 |
| 3M 64 partitions | **0.792±0.004** | 0.708±0.003 | 0.5±0.0 | 0.670±0.004 |
| 4M 32 partitions | **0.817±0.002** | 0.737±0.003 | 0.5±0.000 | 0.696±0.003 |
| 4M 64 partitions | **0.828±0.002** | 0.696±0.006 | 0.5±0.000 | 0.696±0.003 |

of around 0.6 (Fig. 31 and Fig. 32) indicating a better performance.

Furthermore, considering the time taken to learn a model on these datasets, we also see the SRLearn is significantly faster than XGBoost (Table 16). We finally calculate the end-to-end time taken by SRLearn and XGBoost. The results are in Table 17. We can see that SRLearn fares well compared to XGBoost in terms of runtime, as well as scalability. On one hand, we can see XGBoost drastically increase in runtime with respect to the increase in the input data. On the other hand, the runtime increase in SRLearn is comparatively smooth.

### 5.3.6 Evaluation on additional target predicate

We also evaluated SRLearn along with it's competitive approaches on the ground predicate *malicious*. Table 20 shows the time taken by SRLearn's underlying rule learning tool, BoostSRL to train models for target predicate *malicious*.

Similarly, we also calculated the AUC ROC and AUC PR with SRLearn, Vanilla SRLearn and ProbKB on target ground predicate *malicious* as shown in Table 22 and Table 21. As we can see, the SRLearn performs better than other methodologies in terms

Table 20: Time taken to train models by SRLearn for target predicate *malicious*

| Dataset | Train time |
|---|---|
| 2M 32 partitions | 56m 43s±19m 30s |
| 2M 64 partitions | 40m 15s±14m 20s |
| 4M 32 partitions | 2h 11m 33s±32m 18s |
| 4M 64 partitions | 1h 54m 58s±21m 43s |

Table 21: AUC PR for target *malicious*: SRLearn vs Competitive methodologies

| Dataset | SRLearn | Vanilla SRLearn | ProbKB |
|---|---|---|---|
| 2M 32 partitions | **0.202±0.010** | 0.095±0.002 | 0.026±0.000 |
| 2M 64 partitions | **0.239±0.021** | 0.078±0.005 | 0.027±0.000 |
| 4M 32 partitions | **0.094±0.007** | 0.06±0.002 | 0.018±0.000 |
| 4M 64 partitions | **0.097±0.004** | 0.052±0.002 | 0.018±0.000 |

of AUC ROC. The evaluation with XGBoost remains open for further consideration. In the case of the target ground predicate *malicious*, the one-hot encoding centered around tweet IDs, similar to that of XGBoost for target ground predicate *isPossiblySensitive*, becomes overly complicated, leading to lack of disk space. Unlike tweet IDs, which have a 1:1 relationship with the tweets, a tweet can contain multiple tweets and therefore the URLs can have a many-to-one relationship with a tweet. Hence, the implementation of XGBoost on the target predicate *malicious*, is open to later evaluation and is beyond the scope of this dissertation.

Table 22: AUC ROC for target *malicious*: SRLearn vs Competitive methodologies

| **Dataset** | **SRLearn** | **Vanilla SRLearn** | **ProbKB** |
|---|---|---|---|
| 2M 32 partitions | **0.735±0.009** | 0.701±0.003 | 0.5±0.000 |
| 2M 64 partitions | **0.737±0.015** | 0.652±0.002 | 0.5±0.000 |
| 4M 32 partitions | **0.702±0.006** | 0.661±0.007 | 0.5±0.000 |
| 4M 64 partitions | **0.711±0.006** | 0.688±0.004 | 0.5±0.000 |

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

During the course of this dissertation, we have seen how crucial the problem to finding a scalable system for structure learning is. The capability to learn first order rules, not only helps build a more accurate system that can predict/classify the target variables, but also is able to identify interesting patterns from an ever evolving and growing dataset. This helps address an existing problem in today's practical scenario. Despite the existing state-of-the art structure learning algorithms for MLNs, the problem to use these algorithms with a large scale data persisted. Through this dissertation, we were successfully able to build a fast and scalable rule mining system on social media data. Through a series of data modelling techniques as well as taking advantage of the distributed nature of the tools like Spark and Hadoop, we are able to transform the tweets originally centered around the tweet IDs into a social graph, representing these attributes centered around user IDs instead. The social graph captures all the common social interactions users may have on social media. This is an effective mechanism to make sure the information remains intact and that we do not loose out on these signals once we partition the input data set into smaller sizes for further rule learning.

We deploy the state-of-the-art graph partitioning tool, ParHIP to partition the large social user-centric graph into a balanced set of partitions. This not only scales up smoothly, but also, makes an optimal use of the distributed node setup on CloudLab by

using multiple cores.

We also take advantage of the efficient state-of-the-art rule mining tools on these partitioned data sets. Thus, instead of having one instance consume the largely scaled social media data, we are now able to run multiple instances of the same tool on smaller meaningful partitions of the evidence data, resulting in a smaller throughput and a higher accuracy. Moreover, with the evaluation done on two different ground predicates, shows that SRLearn can be extended to other ground predicates in the social media domain.

One can expect several challenges while trying to solve a research problem. Not surprisingly, we too faced our fair share of roadblocks. A few deserve to be mentioned in this manuscript. With Alchemy, we were able to learn first order rules as well as get the rules in a clear and concise manner. But determining the accuracy and efficiency became a question. Also, not to ignore the fact that Alchemy's structure learning was not fast enough. We continued to believe that we could ultimately build a fast and scalable rule learning system. That is when we switched our rule learning component to that of BoostSRL, which finally gave us the speed and efficiency we were aiming for.

The next roadblock was to find any existing rule learning systems that can be implemented in a distributed environment. But except ProbKB, there was no scalable tool available then. Finally we incorporated a machine learning algorithm, called XGBoost, to gauge whether we are still comparable to any available scalable and/or distributed tools. XGBoost's Spark implementation is a significantly popular tool used to address classification problems in large scale data.

Modeling the data for XGBoost again posed further challenges in terms of storage.

We have gone through multiple iterations of data modeling to transition the relational first order predicates into a tabular structure that can potentially encompass the rich details of a social network. This transformed data is not only memory-greedy, but also is extremely sparse. By the time we reached 4 million tweets, we started to face storage issues for the one hot encoded files in a cluster. We believe that this approach of one hot encoding can possess a serious storage problem as the data scales up. Moreover, modeling the tweets for a different target predicate has its own challenges in terms of representation.

Challenges aside, keeping in mind the rate at which a social media platform like Twitter, produces data every minute, SRLearn is a novel mechanism that is able to consume millions of tweets at a given time, in order to make classification as accurate as possible. This is a stepping stone into making existing MLN-based rule mining systems, to consume data at a practical scale and infer meaningful relations from the data.

We believe that this system can be extended into a variety of usecases as well as datasets from other domains. Theoretically, one usecase can be to identify the users that could potentially pose a threat to the social media platform, or could help address the issue of fake news. The idea behind SRLearn is not only limited to social media, but also can be engineered to tackle big data problems in other domains. The challenges in the domain of big data are endless. We believe, SRLearn can serve well to address at least a few of them.

REFERENCE LIST

[1] Besag, J. "Statistical Analysis of Non-Lattice Data". *Journal of the Royal Statistical Society. Series D (The Statistician) 24* (1965), pp. 179–195.

[2] Casella, G., and George, E. "Explaning the Gibbs Sampler". *The American Statistician 46* (08 1992), pp. 167–174.

[3] Chen, T., and Guestrin, C. "XGBoost: A Scalable Tree Boosting System". *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Aug 2016), pp. 785–794.

[4] Chen, Y., Goldberg, S., Wang, D. Z., and Johri, S. S. "Ontological Pathfinding". In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, pp. 835–846.

[5] Chen, Y., and Wang, D. Z. "Knowledge Expansion over Probabilistic Knowledge Bases". In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, Association for Computing Machinery, pp. 649–660.

[6] Davis, J., and Goadrich, M. "The Relationship between Precision-Recall and ROC Curves". In *Proceedings of the 23rd International Conference on Machine Learning*

(New York, NY, USA, 2006), ICML '06, Association for Computing Machinery, pp. 233–240.

[7] Dean, J., and Ghemawat, S. "MapReduce: Simplified Data Processing on Large Clusters". *Communications of the ACM 51*, 1 (Jan. 2008), pp. 107–113.

[8] Dhami, D. S., Yan, S., Kunapuli, G., and Natarajan, S. "Non-Parametric Learning of Gaifman Models". *Computing Research Repository abs/2001.00528* (2020).

[9] Domingos, P., and Lowd, D. *"Markov Logic: An Interface Layer for Artificial Intelligence"*. Morgan and Claypool Publishers, 2009.

[10] Dumancic, S., Guns, T., Meert, W., and Blockeel, H. "Learning Relational Representations with Auto-encoding Logic Programs". *Computing Research Repository abs/1903.12577* (2019).

[11] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. "The Design and Operation of CloudLab". In *Proceedings of the USENIX Annual Technical Conference (ATC)* (July 2019), pp. 1–14.

[12] Eiter, T., and Gottlob, G. "Propositional Circumscription and Extended Closed World Reasoning are $\Pi_2^P$-complete". *Theoretical Computer Science 114*, 2 (1993), 231–245.

[13] Ewald, W. "The Emergence of First-Order Logic". In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Spring 2019 ed. Metaphysics Research Lab, Stanford University, 2019.

[14] Farabi, K. M. A., Sarkhel, S., and Venugopal, D. "Efficient Weight Learning in High-Dimensional Untied MLNs". In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics* (Apr 2018), A. Storkey and F. Perez-Cruz, Eds., vol. 84 of *Proceedings of Machine Learning Research*, PMLR, pp. 1637–1645.

[15] Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. "Learning probabilistic relational models". *IJCAI'99: Proceedings of the 16th international joint conference on Artificial intelligence 2* (July 1999), pp. 1300–1307.

[16] Getoor, L., Friedman, N., Koller, D., and Pfeffer, A. *"Learning Probabilistic Relational Models"*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 307–335.

[17] Hadoop. "Apache Hadoop: an open-source software for reliable, scalable, distributed computing". *Available: https://hadoop.apache.org/* (2018).

[18] Harsha Vardhan, L. V., Jia, G., and Kok, S. "Probabilistic Logic Graph Attention Networks for Reasoning". *Companion Proceedings of the Web Conference 2020* (2020), pp. 669–673.

[19] Heckerman, D., Chickering, D. M., Meek, C., Rounthwaite, R., and Kadie, C. "Dependency Networks for Inference, Collaborative Filtering, and Data Visualization". *Journal of Machine Learning Research 1*, Oct (2000), pp. 49–75.

[20] Huynh, T. N., and Mooney, R. J. "Online Structure Learning for Markov Logic Networks". In *Machine Learning and Knowledge Discovery in Databases* ("Berlin, Heidelberg", 2011), D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds., Springer Berlin Heidelberg, pp. 81–96.

[21] Karypis, G., and Kumar, V. "Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs". In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* (USA, 1996), Supercomputing '96, IEEE Computer Society, pp. 35–35.

[22] Karypis, G., and Kumar, V. "A fast and high quality multilevel scheme for partitioning irregular graphs". *SIAM Journal on Scientific Computing 20*, 1 (1998), pp. 359–392.

[23] Kersting, K., and Raedt, L. D. *"Bayesian Logic Programs"*. Introduction to Statistical Relational Learning, 2001.

[24] Kersting, K., and Raedt, L. D. "Towards Combining Inductive Logic Programming with Bayesian Networks". In *Lecture Notes in Computer Science* (2001), Springer, pp. 118–131.

[25] Khosravi, H., and Bina, B. "A Survey on Statistical Relational Learning". *Advances in Artificial Intelligence* (2010), pp. 256–268.

[26] Khosravi, H., Schulte, O., Man, T., Xu, X., and Bina, B. "Structure Learning for Markov Logic Networks with Many Descriptive Attributes". *Proceedings of the AAAI Conference on Artificial Intelligence 24*, 1 (Jul. 2010), pp. 487–493.

[27] Khot, T., Natarajan, S., Kersting, K., and Shavlik, J. "Learning Markov Logic Networks via Functional Gradient Boosting". In *2011 IEEE 11th International Conference on Data Mining* (2011), pp. 320–329.

[28] Kok, S., and Domingos, P. "Learning the Structure of Markov Logic Networks". In *Proceedings of the 22nd International Conference on Machine Learning* (New York, NY, USA, 2005), Association for Computing Machinery, pp. 441–448.

[29] Kok, S., and Domingos, P. "Learning Markov Logic Network Structure via Hypergraph Lifting". In *Proceedings of the 26th Annual International Conference on Machine Learning* (New York, NY, USA, 2009), Association for Computing Machinery, pp. 505–512.

[30] Kok, S., and Domingos, P. "Learning Markov Logic Networks Using Structural Motifs". In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (Madison, WI, USA, 2010), Omnipress, pp. 551–558.

[31] Li, L., Li, W., Zhu, L., Li, C., and Zhang, Z. "Automatic Data Repairs with Statistical Relational Learning". In *2021 International Symposium on Networks, Computers and Communications (ISNCC)* (2021), pp. 1–6.

[32] McAfee. "How Cybercriminals Target Social Media Accounts". *Available: https://www.mcafee.com/enterprise/en-us/security-awareness/cybersecurity/cybercriminal-social-media.html* (August 2021).

[33] Meyerhenke, H., Sanders, P., and Schulz, C. "Parallel Graph Partitioning for Complex Networks". *IEEE Transactions on Parallel and Distributed Systems 28*, 9 (2017), pp. 2625–2638.

[34] Michel, M. M., and Jacobs, R. A. "Parameter Learning but not Structure Learning: A Bayesian Network Model of Constraints on Early Perceptual Learning". *Journal of Vision 7*, 1 (01 2007), pp. 4–4.

[35] Muggleton, S. "Inductive Logic Programming". *New Generation Computing 8*, 4 (1991), pp. 295–318.

[36] Natarajan, S., Kersting, K., Khot, T., and Shavlik, J. *"Boosted Statistical Relational Learners: From Benchmarks to Data-Driven Medicine"*. Springer Publishing Company, Incorporated, 2015.

[37] Natarajan, S., Khot, T., Kersting, K., Gutmann, B., and Shavlik, J. "Gradient-based boosting for statistical relational learning: The relational dependency network case". *Special issue of Machine Learning Journal 86*, 1 (2012), pp. 25–56.

[38] Neville, J., and Jensen, D. "Relational Dependency Networks". *Journal of Machine Learning Research 8* (May 2007), pp. 653–692.

[39] Rao, P., Kamhoua, C., Kwiat, K., and Njilla, L. ”System and Article of Manufacture to Analyze Twitter Data to Discover Suspicious Users and Malicious Content”, US Patent, Sr. No. 10,348,752, July 9, 2019.

[40] Rao, P., Kamhoua, C., Kwiat, K., and Njilla, L. ”Method of Cyberthreat Detection by Learning First-Order Rules on Large-Scale Social Media”, US Patent, Sr. No. 10,812,500, Oct 20, 2019.

[41] Rao, P., Katib, A., Kamhoua, C., Kwiat, K., and Njilla, L. ”Probabilistic Inference on Twitter Data to Discover Suspicious Users and Malicious Content”. In *2016 IEEE International Conference on Computer and Information Technology (CIT)* (2016), pp. 407–414.

[42] Richardson, M., and Domingos, P. ”Markov logic networks”. *Machine Learning 62* (Jan. 2006), pp. 107–136.

[43] Sabek, I. ”Towards Scalable Spatial Probabilistic Graphical Modeling”. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2019), pp. 606–607.

[44] Sanders, P., and Schulz, C. ”Think Locally, Act Globally: Highly Balanced Graph Partitioning”. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings* (2013), vol. 7933, Springer, pp. 164–175.

[45] Senapati, M., Njilla, L., and Rao, P. "A Method for Scalable First-Order Rule Learning on Twitter Data". In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)* (2019), pp. 274–277.

[46] Shao, C., Ciampaglia, G. L., Varol, O., Flammini, A., and Menczer, F. "The Spread of Fake News by Social Bots". *Computing Research Repository* (2017).

[47] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. "The Hadoop Distributed File System". In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), pp. 1–10.

[48] Sun, Z., Zhao, Y., Wei, Z., Zhang, W., and Wang, J. "Scalable Learning and Inference in Markov Logic Networks". *International Journal of Approximate Reasoning 82* (Mar. 2017), pp. 39–55.

[49] Versprite. "Top 3 Motives Why Cybercriminals Attack Social Media According to 2020 Threat Trends". *Available: https://versprite.com/blog/top-motives-hackers-attack-social-media-2020/* (October 2020).

[50] Versprite. "US-Iran Conflict: Increased Geopolitical Risk From Cyber Attacks Based Out of Iran". *Available: https://versprite.com/geopolitical-risk/iran-tensions-increase-cyber-threat/* (February 2020).

[51] Yang, S., Khot, T., Kersting, K., Kunapuli, G., Hauser, K., and Natarajan, S. "Learning from Imbalanced Data in Relational Domains: A Soft Margin Approach". In *2014 IEEE International Conference on Data Mining* (2014), pp. 1085–1090.

[52] Yang Chen, D. Z. W. ”Web-Scale Knowledge Inference Using Markov Logic Networks”. *ICML Workshop on Structured Learning: Inferring Graphs from Structured and Unstructured Inputs* (June 2013).

[53] Zachariah, A., Rao, P., Katib, A., Senapati, M., and Barnard, K. ”A Gossip-Based System for Fast Approximate Score Computation in Multinomial Bayesian Networks”. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), pp. 1968–1971.

[54] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. ”Apache Spark: A Unified Engine for Big Data Processing”. *Communications of the ACM 59*, 11 (Oct. 2016), pp. 56–65.

[55] Zhong, P., Li, Z., Chen, Q., Hou, B., and Ahmed, M. ”Numerical Markov Logic Network: A Scalable Probabilistic Framework for Hybrid Knowledge Inference”. *Information 12*, 3 (2021).

# VITA

Monica Senapati entered the field of Computer Science and Engineering with a Bachelors degree from College of Engineering and Technology, Bhubaneswar, India in 2014. She worked as a system engineer at Tata Consultancy Services in Pune, India for 2 years, after which she joined University of Missouri-Kansas City in Fall 2016 to pursue Ph.D. in Computer Science under the guidance of Dr. Praveen Rao.

Her research interests include Big Data systems, Data Science and Machine Learning. She has two publications [45,53] to her name. During her time at UMKC, she worked as a research assistant, teaching assistant for a number of courses and a primary instructor for Introduction to Operating Systems. She has been the recipient of the Graduate Assistance Fund awards from the UMKC Women's Council in 2018 and 2019 and the Mahatma Gandhi Scholarship (2020-2021) by UMKC School of Computing and Engineering. She also worked as a Software Intern at Oracle, Bedford, MA in 2017 and Machine Learning intern at LelexPrime, Kansas in 2020-21.

Post completion of Ph.D. she joined Etsy as Senior Software Engineer in 2021 in Brooklyn, NY, where she continues to work with distributed systems and big data technologies, to solve the problems that come with scalable data.