

# Machine Learning for Classifying Anomalies and Intrusions in Communication Networks

by

**Zhida Li**

M.A.Sc., University College Cork, 2016

M.Eng.Sc., University College Cork, 2013

B.Eng., University College Cork, 2011

Thesis Submitted in Partial Fulfillment of the  
Requirements for the Degree of  
Doctor of Philosophy

in the  
School of Engineering Science  
Faculty of Applied Sciences

© **Zhida Li 2022**

**SIMON FRASER UNIVERSITY**

**Spring 2022**

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

# Declaration of Committee

**Name:** Zhida Li

**Degree:** Doctor of Philosophy (Engineering Science)

**Title:** Machine Learning for Classifying Anomalies and Intrusions in Communication Networks

**Committee:** **Chair: Ivan V. Bajić**  
Professor, Engineering Science

**Ljiljana Trajković**  
Supervisor  
Professor, Engineering Science

**Uwe Glässer**  
Committee Member  
Professor, Computing Science

**Qianping Gu**  
Committee Member  
Professor, Computing Science

**Bernhard Rabus**  
Examiner  
Professor, Engineering Science

**Francesco Sorrentino**  
External Examiner  
Professor, Mechanical Engineering  
University of New Mexico

# Abstract

Cyber attacks are becoming more sophisticated and, hence, more difficult to detect. Using efficient and effective machine learning techniques to detect network anomalies and intrusions is an important aspect of cyber security. A variety of machine learning models have been employed to help detect malicious intentions of network users. In this dissertation, we have applied various machine learning algorithms to classify known network anomalies such as Internet worms, denial of service attacks, power outages, and ransomware attacks. We have proposed novel Broad Learning System-based algorithms with and without incremental learning. Generalized models have been developed by using subsets of input data based on selected features and by expanding the network structure. Furthermore, a Border Gateway Protocol anomaly detection tool *BGPGuard* has been developed to integrate various stages of the anomaly detection process.

**Keywords:** Communication networks, intrusion detection, traffic and routing anomalies, machine learning, feature selection, deep learning, broad learning system, gradient boosting algorithms.

# Dedication

*To my parents, wife, and son*

# Acknowledgements

I would like to express my gratitude to all the individuals who offered assistance during the preparation of this dissertation.

I give my sincere thanks and appreciation to my advisor Prof. Ljiljana Trajković for giving me the opportunity to join her research team and work in the field of communication networks and machine learning. She dedicated her time to guide my research and encouraged me to explore ideas and research directions that will significantly influence my career.

I would also like to thank the Chair Prof. Ivan V. Bajić, Committee Members Prof. Uwe Glässer and Prof. Qianping Gu, the Internal Examiner Prof. Bernhard Rabus, and the External Examiner Prof. Francesco Sorrentino for reviewing this dissertation and providing constructive feedback.

I would like to thank my current colleagues Ana Laura Gonzalez Rios and Hardeep Kaur Takhar, my former colleagues Prerna Batta, Kamila Bekshentayeva, Qingye Ding, Soroush Haeri, and Guangyu Xu, and my friends Jiawei He and Xiaoyu Liu for their assistance during my Ph.D. journey.

Last but not least, I would like to thank my family for their support and care throughout my life.

# Table of Contents

<b>Declaration of Committee</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Dedication</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>v</b>
<b>Table of Contents</b>	<b>vi</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xiv</b>
<b>List of Abbreviations</b>	<b>xix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Summary of Research Contributions . . . . .	1
1.2.1 Applications of Machine Learning Techniques for Classifying Network Anomalies . . . . .	2
1.2.2 Variable Features Broad Learning Systems . . . . .	3
1.2.3 BGP Anomaly Detection Tool for Real-Time and Off-Line Detection . . . . .	3
1.3 Research Publications . . . . .	4
1.4 Roadmap . . . . .	8
<b>2 Network Anomalies and Intrusions</b>	<b>9</b>
2.1 Literature Review . . . . .	9
2.1.1 Detection of Network Anomalies . . . . .	9
2.1.2 Conventional Techniques used for Intrusion Detection . . . . .	10
2.1.3 Machine Learning Techniques used for Intrusion Detection . . . . .	10
2.1.4 Intrusion Detection Systems . . . . .	11
2.1.5 Description of Datasets . . . . .	12
2.2 Border Gateway Protocol Datasets . . . . .	13

2.2.1	Overview of BGP . . . . .	13
2.2.2	BGP Data Collections . . . . .	15
2.2.3	BGP Anomalies . . . . .	16
2.2.4	Extraction of Features from BGP Update Messages . . . . .	20
2.2.5	Generation of BGP Training and Test Datasets . . . . .	22
2.3	NSL-KDD Dataset . . . . .	25
2.4	Canadian Institute for Cybersecurity Datasets . . . . .	28
<b>3</b>	<b>Feature Selection and Dimension Reduction</b>	<b>33</b>
3.1	Feature Selection Algorithms . . . . .	33
3.1.1	Fisher . . . . .	34
3.1.2	Minimum Redundancy Maximum Relevance . . . . .	35
3.1.3	Odds Ratio . . . . .	36
3.1.4	Decision Tree . . . . .	38
3.1.5	Extra-Trees . . . . .	39
3.2	Dimension Reduction Algorithms . . . . .	40
3.2.1	Autoencoders . . . . .	40
3.3	Discussion . . . . .	45
<b>4</b>	<b>Machine Learning Algorithms</b>	<b>46</b>
4.1	Overview of Machine Learning Algorithms . . . . .	46
4.1.1	Performance Metrics . . . . .	48
4.2	Support Vector Machine . . . . .	49
4.2.1	Experiments and Performance Evaluation . . . . .	50
4.3	Hidden Markov Model . . . . .	51
4.3.1	Experiments and Performance Evaluation . . . . .	51
4.4	Naïve Bayes . . . . .	55
4.4.1	Experiments and Performance Evaluation . . . . .	56
4.5	Decision Tree . . . . .	57
4.5.1	Experiments and Performance Evaluation . . . . .	57
4.6	Extreme Learning Machine . . . . .	60
4.6.1	Experiments and Performance Evaluation . . . . .	61
4.7	Discussion . . . . .	61
<b>5</b>	<b>Deep Learning Networks</b>	<b>63</b>
5.1	Convolutional Neural Networks . . . . .	63
5.1.1	Experiments and Performance Evaluation . . . . .	64
5.2	Recurrent Neural Networks . . . . .	67
5.2.1	Long Short-Term Memory . . . . .	67
5.2.2	Gated Recurrent Unit . . . . .	69

5.2.3	Experiments and Performance Evaluation . . . . .	70
5.2.3.1	Discussion . . . . .	72
5.3	Performance Comparison: CNN, RNN, and Bi-RNN . . . . .	72
5.3.1	Discussion . . . . .	75
<b>6</b>	<b>Fast Machine Learning Algorithms</b>	<b>76</b>
6.1	Broad Learning System . . . . .	76
6.1.1	BLS and its Extensions: Performance Comparison . . . . .	80
6.1.1.1	Discussion . . . . .	81
6.1.2	BLS and RNNs (LSTM and GRU): Performance Comparison . . . .	83
6.1.2.1	Discussion . . . . .	84
6.2	Gradient Boosting Decision Tree Algorithms . . . . .	87
6.2.1	XGBoost . . . . .	88
6.2.2	LightGBM . . . . .	89
6.2.3	CatBoost . . . . .	90
6.3	BLS and GBDT: Performance Comparison . . . . .	90
6.3.1	Effect of Hyper-Parameters on Algorithm Performance . . . . .	92
6.3.2	Discussion . . . . .	94
<b>7</b>	<b>Variable Features Broad Learning Systems</b>	<b>95</b>
7.1	VFBL and VCFBL Algorithms . . . . .	95
7.2	Experiments and Performance Evaluation . . . . .	102
<b>8</b>	<b>BGPGuard: BGP Anomaly Detection Tool</b>	<b>111</b>
8.1	Architectures . . . . .	111
8.2	Terminal-Based Application . . . . .	113
8.2.1	Structure . . . . .	113
8.2.2	External Libraries . . . . .	114
8.2.3	Sample Settings . . . . .	115
8.3	Web-Based Application . . . . .	115
8.3.1	Structure . . . . .	115
8.3.2	External Libraries . . . . .	116
8.3.3	Web Pages . . . . .	117
<b>9</b>	<b>Conclusion and Future Work</b>	<b>122</b>
	<b>Bibliography</b>	<b>125</b>
	<b>Appendix A Anomaly Classification: Machine Learning Algorithms</b>	<b>138</b>
	<b>Appendix B BGPGuard: Main Modules</b>	<b>154</b>

<b>Appendix C BGPGuard: Developed Python Functions</b>	<b>169</b>
<b>Appendix D BGPGuard: Sample output</b>	<b>174</b>

# List of Tables

Table 2.1	Sample of a BGP <i>update</i> message. IGP: Interior Gateway Protocol. . . . .	15
Table 2.2	Examples of known BGP Internet worms. . . . .	18
Table 2.3	List of features extracted from BGP <i>update</i> messages. The <i>AS-path</i> is a BGP <i>update</i> message attribute that enables the protocol to select the best path for routing packets. The features are categorized as <i>AS-path</i> and <i>volume</i> features. . . . .	21
Table 2.4	Definition of <i>volume</i> and <i>AS-path</i> features extracted from BGP <i>update</i> messages. . . . .	22
Table 2.5	Training (concatenations of two datasets) and test datasets. . . . .	23
Table 2.6	BGP Internet worm datasets: Number of data points. Note that Route Views data collection began in 2003. . . . .	24
Table 2.7	BGP power outage datasets: Number of data points. . . . .	24
Table 2.8	BGP ransomware attack datasets: Number of data points. . . . .	24
Table 2.9	NSL-KDD dataset: Four types of intrusion attacks are listed: DoS, User to Root (U2R), Remote to Local (R2L), and Probe. . . . .	28
Table 2.10	NSL-KDD dataset: Number of data points. The NSL-KDD dataset contains one training (KDDTrain <sup>+</sup> ) and two test datasets (KDDTest <sup>+</sup> and KDDTest <sup>-21</sup> ). . . . .	28
Table 2.11	NSL-KDD features: Definitions, types, and descriptions [1]. Each network connection is represented by 41 features: 38 numerical and 3 categorical (“protocol_type”, “service”, and “flag”) features. . . . .	29
Table 2.12	Application-layer DoS and TCP/UDP DDoS attacks. . . . .	31
Table 3.1	The top ten features selected using the Fisher feature selection algorithm. Two-way classification includes two classes: anomaly and regular. The four-way classification detects the specific type of BGP anomaly (Slammer, Nimda, Code Red) or regular class. . . . .	35
Table 3.2	The top ten features selected using the mRMR feature selection algorithms for two-way classification. Two-way classification distinguishes two classes: anomaly and regular. . . . .	36

Table 3.3	The top ten features selected using the mRMR feature selection algorithms for four-way classification. The four-way classification detects the specific type of BGP anomaly (Slammer, Nimda, Code Red) or regular class. . . . .	37
Table 3.4	The top ten features selected using the OR feature selection algorithms for two-way classification. Two-way classification distinguishes two classes: anomaly and regular. . . . .	38
Table 3.5	The top ten features selected using the OR feature selection algorithms for four-way classification. The four-way classification detects the specific type of BGP anomaly (Slammer, Nimda, Code Red) or regular class. . . . .	38
Table 3.6	Selected features using the decision tree algorithm for two-way classification. Two-way classification distinguishes two classes: anomaly and regular. . . . .	39
Table 3.7	Sixteen most relevant features and their importance based on the extra-trees algorithm. The Gini importance is used to compute feature scores using CICIDS2017 and CSE-CIC-IDS2018 datasets. . . . .	40
Table 4.1	Confusion matrix. . . . .	48
Table 4.2	Performance of the two-way SVM classification. Two-way classification distinguishes two classes: anomaly and regular. SVM <sub>1</sub> : Training datasets (Slammer and Nimda) are used to identify the specific type of BGP data points (Code Red or regular (RIPE or BCNET)); SVM <sub>2</sub> : Training datasets (Slammer and Code Red) are used to identify the specific type of BGP data points (Nimda or regular (RIPE or BCNET)); SVM <sub>3</sub> : Training datasets (Nimda and Code Red) are used to identify the specific type of BGP data points (Slammer or regular (RIPE or BCNET)). . . . .	52
Table 4.3	Accuracy of the four-way SVM classification. Concatenation of training datasets (Slammer, Nimda, Code Red, and RIPE) is used to identify regular BGP data points (RIPE or BCNET). . . . .	54
Table 4.4	HMM models: Two-way classification. Two-way classification distinguishes two classes: anomaly and regular. Dataset 4: concatenation of Slammer, Nimda, and Code Red datasets. . . . .	55
Table 4.5	Accuracy of the two-way HMM classification. Two-way classification distinguishes two classes: anomaly and regular. Training datasets (Slammer, Nimda, and Code Red) are used to identify the specific type of BGP data points: anomaly or regular (RIPE or BCNET). . . . .	55

Table 4.6	Performance of the two-way naïve Bayes classification. Two-way classification distinguishes two classes: anomaly and regular. NB1: Training datasets (Slammer and Nimda) are used to identify the specific type of BGP data points (Code Red or regular (RIPE or BCNET)); NB2: Training datasets (Slammer and Code Red) are used to identify the specific type of BGP data points (Nimda or regular (RIPE or BCNET)); NB3: Training datasets (Nimda and Code Red) are used to identify the specific type of BGP data points (Slammer or regular (RIPE or BCNET)). . . . .	58
Table 4.7	Accuracy of the four-way naïve Bayes classification. Concatenation of training datasets (Slammer, Nimda, Code Red, and RIPE) is used to identify regular BGP data points (RIPE or BCNET). . . . .	60
Table 4.8	Performance of the two-way decision tree classification. Two-way classification distinguishes two classes: anomaly and regular. Dataset 1: concatenation of Slammer and Nimda datasets; Dataset 2: concatenation of Slammer and Code Red datasets; Dataset 3: concatenation of Nimda and Code Red datasets. . . . .	60
Table 4.9	Performance of the ELM algorithm using datasets with 37 and 17 features. Dataset 1: concatenation of Slammer and Nimda datasets; Dataset 2: concatenation of Slammer and Code Red datasets; Dataset 3: concatenation of Nimda and Code Red datasets. . . . .	62
Table 5.1	The best performance of CNN models: Pakistan power outage and WestRock ransomware. . . . .	68
Table 5.2	Parameters of RNN models used in cross-validation. . . . .	72
Table 5.3	Performance of LSTM and GRU RNN models: Slammer, Moscow blackout, and WannaCrypt datasets. Highlighted are the best models for Slammer (purple), Moscow blackout (brown), and WannaCrypt (blue). . . . .	73
Table 5.4	The best performance of LSTM, GRU, Bi-LSTM, and Bi-GRU models: Pakistan power outage and WestRock ransomware. . . . .	74
Table 6.1	Best parameters obtained when using CICIDS2017 and CSE-CIC-IDS2018 datasets; Incremental learning: <i>incremental learning steps</i> = 2, <i>enhancement nodes/step</i> = 20, and <i>data points/step</i> = 55,680 (CICIDS2017), 49,320 (CSE-CIC-IDS2018). . . . .	81
Table 6.2	Best performance for models tested with CICIDS2017 and CSE-CIC-IDS2018 datasets. . . . .	83
Table 6.3	Number of BLS training parameters. . . . .	84
Table 6.4	Training time for RNN and BLS models: BGP and NSL-KDD datasets. . . . .	85
Table 6.5	Performance of RNN (LSTM and GRU) models: BGP datasets (Python). . . . .	86

Table 6.6	Performance of BLS Model and its extensions: BGP datasets (Python).	86
Table 6.7	Performance of RNN (LSTM and GRU) and BLS models: NSL-KDD datasets (Python). . . . .	87
Table 6.8	Performance and training time of incremental BLS model: BGP and NSL-KDD datasets (MATLAB). . . . .	87
Table 6.9	BLS and incremental BLS hyper-parameters leading to the best performance: CIC datasets. . . . .	92
Table 6.10	XGBoost, LightGBM, and CatBoost hyper-parameters leading to the best performance: CIC datasets. . . . .	92
Table 6.11	The best performance of BLS and incremental BLS models: CIC datasets.	93
Table 6.12	The best performance of XGBoost, LightGBM, and CatBoost models: CIC datasets. . . . .	93
Table 7.1	LightGBM parameters: BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets. . . . .	104
Table 7.2	BLS and incremental BLS parameters: BGP RIPE datasets. . . . .	104
Table 7.3	BLS and incremental BLS parameters: NSL-KDD dataset. . . . .	105
Table 7.4	VFBLs and VCFBLs parameters: BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets. . . . .	105
Table 7.5	Incremental VFBLs and VCFBLs parameters: BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets. . . . .	106
Table 7.6	Training time: BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets. . . . .	110

# List of Figures

Figure 2.1	Route Views collector map: 36 collectors are located in 5 RIR regions [2]. . . . .	16
Figure 2.2	Slammer (top), Moscow blackout (middle), and WannaCrypt (bottom): Number of BGP announcements and announced NLRI prefixes. Examples of features that exhibit visible differences in patterns during regular and anomalous events for the Slammer, Moscow blackout, and WannaCrypt BGP datasets. . . . .	26
Figure 2.3	Slammer: Average AS-path length vs. number of BGP announcements vs. number of BGP withdrawals. These graphs indicate spatial separation for regular ( $\circ$ ) and anomalous ( $*$ ) classes. . . . .	27
Figure 2.4	Moscow blackout: Number of announced NLRI prefixes vs. number of BGP announcements vs. number of withdrawn NLRI prefixes. These graphs indicate spatial separation for regular ( $\circ$ ) and anomalous ( $*$ ) classes. . . . .	27
Figure 2.5	WannaCrypt: Number of announced NLRI prefixes vs. number of BGP announcements vs. average edit distance. These graphs indicate spatial separation for regular ( $\circ$ ) and anomalous ( $*$ ) classes. . . . .	28
Figure 2.6	<i>KDDTrain<sup>+</sup></i> dataset: traces of “Src_bytes” (top left), “Hot” (top right), “Dis_host_srv_diff_host_rate” (bottom left), and “Duration” (bottom right) features. Traces include both anomalous and regular data points. . . . .	30
Figure 2.7	Scattered plots of the CICIDS2017 (top), CSE-CIC-IDS2018 (middle), and CICDDoS2019 (bottom). Illustrated are spatial separations of regular (class 0) and anomalous (class 1) data points. . . . .	32
Figure 3.1	Scattered graph of Feature 9 vs. Feature 1 (left) and Feature 9 vs. Feature 6 (right) extracted from the BCNET traffic. Feature values are normalized to have zero mean and unit variance. Shown are two traffic classes: regular ( $\circ$ ) and anomaly ( $*$ ). . . . .	34

Figure 3.2	Sixteen most relevant features and their importance based on the extra-trees algorithm: CICIDS2017 (top) and CSE-CIC-IDS2018 (bottom) datasets. Note that not all features are equally important. The top two features (“Minimum forward segment size” and “initial forward window bytes”) in the CSE-CIC-IDS2018 dataset exhibit higher feature scores than those in the CICIDS2017 dataset. . . . .	41
Figure 3.3	Structure of an autoencoder with encoder and decoder modules. The encoder and decoder consist of the same type of networks and have the same number of layers and nodes. The central layer in the autoencoder is the code layer that belongs to both encoder and decoder and contains the smallest number of hidden nodes. The encoder is used to map the higher dimensional input data to lower dimensional data (latent vectors) in the code layer. The output data are reconstructed from the code layer through a decoder. . . . .	42
Figure 3.4	An autoencoder implementation using Slammer dataset: mapping input data to the code and decoding (reconstructing) input data. The dimension (size) of the input and output data is $7,200 \times 37$ . Shown are three BGP features before (left) and after (right) reconstruction.	43
Figure 3.5	Comparison between PCA and autoencoder used for dimension reduction of the Slammer dataset: The first three principal components generated by PCA (top) and the compressed data generated by the autoencoder model (bottom) described in Fig. 3.4. The silhouette coefficients for the clusters generated by PCA and the autoencoder are 0.36 and 0.67, respectively. The higher value of the silhouette coefficient indicates that the autoencoder generates better spatial separation thus leading to better classification performance. . . . .	44
Figure 4.1	Illustration of the soft margin SVM [3]. Shown are correctly and incorrectly classified data points. Regular and anomalous data points are denoted by circles and stars, respectively. The circled points are support vectors. Data points for which $\zeta = 0$ are correctly classified and are either on the margin or on the correct side of the margin. Data points for which $0 \leq \zeta < 1$ are also correctly classified because they lie inside the margin and on the correct side of the decision boundary. Data points for which $\zeta > 1$ lie on the wrong side of the decision boundary and are misclassified. The outputs 1 and -1 correspond to anomalous and regular data points, respectively. . .	50

Figure 4.2	SVM classifier applied to Slammer traffic collected from January 23 to 27, 2003: Shown in purple are correctly classified anomalies (true positive) while shown in green are incorrectly classified anomalies (false positive) (top). Shown in red are incorrectly classified anomalous (false positive) and regular (false negative) data points while shown in blue are correctly classified anomalous (true positive) and regular (true negative) data points (bottom). . . . .	53
Figure 4.3	Naïve Bayes classifier applied to Slammer traffic collected from January 23 to 27, 2003: Shown in purple are correctly classified anomalies (true positive) while shown in green are incorrectly classified anomalies (false positive) (top). Shown in red are incorrectly classified anomalies (false positive) and regular (false negative) data points while shown in blue are correctly classified anomalous (true positive) and regular (true negative) data points (bottom). . . . .	59
Figure 4.4	Neural network architecture of the ELM algorithm. $[\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_d]$ is the input vector; $d$ is the feature dimension; $f(\cdot)$ is the activation function; $\mathbf{W}_{in}$ is the vector of weights connecting the inputs to hidden units; $[\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_m]$ is the output vector; and $\mathbf{W}_{out}$ is the weight vector connecting the hidden and the output units. . . . .	61
Figure 5.1	The high-level structure of a CNN using 1-dimensional input data: Convolutional and max pooling operations are performed twice; The output of the second max pooling layers is then flattened and passed through fully-connected layers; The probabilities of the output nodes are given in the output layer using the softmax function. . . . .	64
Figure 5.2	Shown are examples of features extracted from regular and anomalous events collected during the Pakistan power outage (top) and the WestRock ransomware attack (bottom). Pakistan power outage (top): Number of announced NLRI prefixes vs. number of implicit withdrawals vs. date; WestRock ransomware attack (bottom): Number of duplicate announcements vs. number of implicit withdrawals vs. date. Regular data points (class 0) are represented with circles while the anomalous data points (class 1) are represented with stars. Labels of data points within each anomaly window are refined using the IF algorithm. . . . .	65

Figure 5.3	Pakistan power outage (top): Number of announced NLRI prefixes vs. average unique AS-path vs. number of implicit withdrawals; WestRock ransomware attack (bottom): Average unique AS-path vs. number of duplicate announcements vs. number of implicit withdrawals. In case of the Pakistan power outage, spatial separation of classes is not visible, which may explain lower accuracy and F-Score compared to the WestRock ransomware attack. . . . .	66
Figure 5.4	Repeating cell for the LSTM neural network. The current cell state $c_t$ and output $h_t$ are calculated based on input $x_t$ , previous cell state $c_{t-1}$ , and output $h_{t-1}$ . . . . .	69
Figure 5.5	Repeating cell for the GRU neural network. The current output $h_t$ is calculated based on input $x_t$ and output $h_{t-1}$ . . . . .	70
Figure 5.6	Deep learning neural network model. It consists of 37 RNNs, 80 (Slammer)/64 (Moscow blackout)/64 (WannaCrypt) FC <sub>1</sub> , 32 FC <sub>2</sub> , and 16 FC <sub>3</sub> fully connected (FC) hidden nodes. . . . .	71
Figure 6.1	Module of the original BLS algorithm with mapped features and enhancement nodes [4]. . . . .	77
Figure 6.2	Module of the BLS algorithm with increments of mapped features, enhancement nodes, and new input data [4]. . . . .	78
Figure 6.3	Modules of the CFBLs (top) and CEBLS (bottom) algorithms. Shown are cascades of mapped features (top) and enhancement nodes (bottom) without incremental learning. . . . .	79
Figure 6.4	Module of the CFEBLS algorithm with increments of mapped features $\mathbf{Z}_{n+1}$ , enhancement nodes $\mathbf{H}_{m+1}$ , and new input data $\mathbf{X}_a$ . . .	80
Figure 6.5	Performance of BLS (blue) and incremental BLS (red). Shown are accuracy, F-Score, and training time for 78 features and subsets of $2^n$ ( $n = 3, 4, 5$ , and $6$ ) most relevant features using CICIDS2017 (top) and CSE-CIC-IDS2018 (bottom) datasets. . . . .	82
Figure 6.6	Time series split for the 10-fold cross-validation of the CICIDS2017 training dataset. Illustrated are the generations of training (orange) and validation (purple) datasets. . . . .	91
Figure 7.1	VFBLs algorithm with input data $\mathbf{X}$ , sets of groups of mapped features ( $\mathbf{Z}^{n_1}, \dots, \mathbf{Z}^{n_f}$ ), and enhancement nodes ( $\mathbf{H}_1, \dots, \mathbf{H}_m$ ). . .	96
Figure 7.2	VCFBLs algorithm with input data $\mathbf{X}$ , sets of groups of mapped features with cascades ( $\mathbf{Z}^{n_1}, \dots, \mathbf{Z}^{n_f}$ ), and enhancement nodes ( $\mathbf{H}_1, \dots, \mathbf{H}_m$ ). . . . .	97

Figure 7.3	Incremental VFBL algorithm with input data $\mathbf{X}$ , sets of groups of mapped features without cascades ( $\mathbf{Z}^{n_1}, \dots, \mathbf{Z}^{n_f}$ ), and enhancement nodes ( $\mathbf{H}_1, \dots, \mathbf{H}_m$ ). Increments include mapped features $\mathbf{Z}_{n+1}$ , enhancement nodes $\mathbf{H}_{m+1}$ , and new input data $\mathbf{X}_a$ . . . . .	98
Figure 7.4	Incremental VCFBL algorithm with input data $\mathbf{X}$ , sets of groups of mapped features with cascades ( $\mathbf{Z}^{n_1}, \dots, \mathbf{Z}^{n_f}$ ), and enhancement nodes ( $\mathbf{H}_1, \dots, \mathbf{H}_m$ ). Increments include mapped features $\mathbf{Z}_{n+1}$ , enhancement nodes $\mathbf{H}_{m+1}$ , and new input data $\mathbf{X}_a$ . . . . .	99
Figure 7.5	Best performance results for LightGBM, RNN and Bi-RNN (LSTM and GRU), BLS, Incremental BLS, VFBL, and VCFBL models: Slammer (top), Nimda (middle), and Code Red (bottom) datasets. . . . .	107
Figure 7.6	Best performance results for LightGBM, RNN and Bi-RNN (LSTM and GRU), BLS, Incremental BLS, VFBL, and VCFBL models: KDDTest <sup>+</sup> (top) and KDDTest <sup>-21</sup> (bottom) datasets. . . . .	108
Figure 7.7	Best performance results for LightGBM, RNN and Bi-RNN (LSTM and GRU), BLS, Incremental BLS, VFBL, and VCFBL models: CICIDS2017 (top) and CSE-CIC-IDS2018 (bottom) datasets. . . . .	109
Figure 8.1	Architecture of <i>BGPGuard</i> for real-time detection and its modules: data download, real-time data retrieval, feature extraction, data processing, pre-trained models, and classification. . . . .	111
Figure 8.2	Architecture of <i>BGPGuard</i> for off-line classification and its modules: data download, feature extraction, data partition, data processing, machine learning (ML) algorithms, parameter selection, ML models, and classification. . . . .	112
Figure 8.3	The web-based <i>BGPGuard</i> : The index page. . . . .	118
Figure 8.4	The web-based <i>BGPGuard</i> : The real-time detection view. The front-end displayed prediction results and data statistics are generated by the back-end modules: data download, real-time data retrieval, feature extraction, data processing, pre-trained models, and classification. . . . .	119
Figure 8.5	The web-based <i>BGPGuard</i> : The off-line classification view. The classification results are generated by the back-end modules: data download, feature extraction, data partition, data processing, ML algorithms, parameter selection, ML models, and classification. . . . .	120
Figure 8.6	The web-based <i>BGPGuard</i> : Introduction of the author. . . . .	121

# List of Abbreviations

ASes	Autonomous Systems
BCI	Brain-Computer Interface
BGP	Border Gateway Protocol
Bi-RNN	Bidirectional Recurrent Neural Network
BLS	Broad Learning System
CAIDA	Center for Applied Internet Data Analysis
CatBoost	Categorical Boosting
CDM	Class Discriminating Measure
CIC	Canadian Institute for Cybersecurity
CICIDS	Canadian Institute for Cybersecurity Intrusion Detection System
CIDR	Classless Inter-Domain Routing
CNNs	Convolutional Neural Networks
DARPA	Defense Advanced Research Projects Agency
DDoS	Distributed Denial of Service
DL	Deep Learning
DoS	Denial of Service
EFB	Exclusive Feature Bundling
EGP	Exterior Gateway Protocol
ELM	Extreme Learning Machine
EOR/WOR/MOR	Extended/Weighted/Multi-Class Odds Ratio
ESNs	Echo State Networks
Extra-Trees	Extremely Randomized Trees
FN	False Negative
FP	False Positive

FTP	File Transfer Protocol
GBDT	Gradient Boosting Decision Tree
GBMs	Gradient Boosting Machines
GOSS	Gradient-Based One-Side Sampling
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HMM	Hidden Markov Model
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDSs	Intrusion Detection Systems
IIS	Internet Information Service
IP	Internet Protocol
ISPs	Internet Service Providers
KDD	Knowledge Discovery in Databases
LightGBM	Light Gradient Boosting Machine
LSTM	Long Short-Term Memory
MIBASE	Mutual Information Base
MID	Mutual Information Difference
MIQ	Mutual Information Quotient
mRMR	Minimum Redundancy Maximum Relevance
NCC	Network Coordination Centre
NLRI	Network Layer Reachability Information
OR	Odds Ratio
PCA	Principal Component Analysis
R2L	Remote to Local
RBF	Radial Basis Function
RIB	Routing Information Base
RIPE	Réseaux IP Européens
RIRs	Regional Internet Registries

RIS	Routing Information Service
RNNs	Recurrent Neural Networks
RRCs	Remote Route Collectors
SQL	Structured Query Language
SSH	Secure Shell Protocol
SVM	Support Vector Machine
TCP	Transport Control Protocol
TN	True Negative
TP	True Positive
U2R	User to Root
UDP	User Datagram Protocol
VFBLs and VCFBLs	Variable Features Broad Learning Systems
XGBoost	eXtreme Gradient Boosting

# Chapter 1

## Introduction

In this Chapter, I give the motivation for the research topic, provide summary of my research contributions, and list and describe the publications emanating from this work. We also outline the roadmap of the dissertation.

### 1.1 Motivation

The Internet has been highly susceptible to failures and attacks that may greatly degrade its performance. Over the past years, frequent cases of complex and challenging threats have been encountered. Hence, various machine learning algorithms have been considered to enhance cybersecurity [5–7]. Machine learning algorithms [3] have been used to address a variety of engineering and scientific problems. They classify data using a feature matrix where its rows and columns correspond to data points and feature values, respectively. By providing a sufficient number of relevant features, machine learning approaches help build generalized classification models and improve their performance. Supervised machine learning models used to classify network anomalies and intrusions include Logistic Regression [3], Naïve Bayes [8], Support Vector Machine (SVM) [9], Decision Tree [10], Boosting algorithms [11, 12], Deep Learning (DL) Networks [13, 14], and the Broad Learning System (BLS) [4, 15]. In this study, we use training time as one of the measures affecting the generation of a model. Note that a model’s testing time, essential for detecting impending anomalies, is usually rather short. Selecting algorithms that have a short training time is important if they are to be implemented in network intrusion detection systems in order to adequately prevent the onset of malicious attacks. It will enable system administrators to effectively and timely remove affected network elements.

### 1.2 Summary of Research Contributions

This dissertation includes three main contributions: (1) implementation and comparison of various machine learning algorithms; (2) development of new machine learning algorithms; (3) development of an anomaly detection tool named *BGPGuard*.

### 1.2.1 Applications of Machine Learning Techniques for Classifying Network Anomalies

In this dissertation, I have investigated various machine learning algorithms, intrusion detection systems, and benchmark datasets related to network traffic and routing anomalies.

***Intrusion detection datasets:*** The following datasets are used for training and testing models: Border Gateway Protocol (BGP) datasets that contain routing records from Réseaux IP Européens (RIPE) [16], Route Views [17] collection sites, and BCNET; NLS-KDD [18] dataset containing network connection records; and Canadian Institute for Cybersecurity Intrusion Detection System (CICIDS) datasets with collections of Transport Control Protocol (TCP) and User Datagram Protocol (UDP) protocol flows: CICIDS2017 [19], CSE-CIC-IDS2018 [20], and CICDDoS2019 [21].

***Data processing, feature selection, and dimension reduction:*** We have generated BGP datasets by extracting relevant features from the BGP *update* messages. We have only performed data cleaning (removing redundancy) for the benchmark datasets (NSL-KDD and CIC) because they already included various features/attributes and labels. These experimental datasets are then processed by converting categorical to numerical features. We have also performed data analysis by visualizing features during regular and anomalous events and examining the importance of features. Their spatial separations were visualized using scattered plots. Feature selection and dimension reduction algorithms are employed to remove redundant information and transform data to lower dimension, respectively.

***Machine learning algorithms:*** We have implemented traditional machine learning, deep learning, and fast machine learning algorithms to evaluate their performance: SVM, Naïve Bayes, Decision Tree, Hidden Markov Model (HMM), Extreme Learning Machine (ELM), Recurrent Neural Networks (RNNs), BLS and its extensions, eXtreme Gradient Boosting (XGBoost), Light Gradient Boosting Machine (LightGBM), and Categorical Boosting (CatBoost) Gradient Boosting Decision Tree (GBDT) algorithms. Extensions to the BLS algorithm include: RBF-BLS and BLS with cascades of mapped features (CF-BLS), enhancement nodes (CEBLS), and both mapped features and enhancement nodes (CFEBLS) with and without incremental learning.

Algorithms are compared based on performance metrics including: training time, accuracy, F-Score, precision, sensitivity, and confusion matrix: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN).

Experiments are performed using a Dell Alienware Aurora with 32 GB memory and Intel Core i7 7700K processor. Python 3.6 and *NumPy* (a scientific computing library), *pandas* (a data manipulation library) *scikit-learn* (a machine learning library), *XGBoost*, *LightGBM*, *CatBoost* (gradient boosting frameworks), and *PyTorch* (a deep learning framework) libraries [22] are used to create input matrices and to train and test the machine learning models. MATLAB and *Matplotlib* library are employed to perform data visualizations.

### 1.2.2 Variable Features Broad Learning Systems

I have developed two new BLS-based algorithms with and without incremental learning. The algorithms are used to develop generalized models by using various subsets of input data and expanding the network structure. We evaluate accuracy, F-Score, and training time of LightGBM, RNN, Bidirectional Recurrent Neural Network (Bi-RNN), and BLS algorithms using BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets.

Experimental results indicated that the best BLS models were sometimes derived by including all features in analyzed datasets. Using a subset of relevant features may enhance the model performance [11, 23–28]. Reported BLS models [29–32] that achieved the best performance were trained using a single subset of features extracted from the input data. Existing BLS-based algorithms [4, 15, 33–36] include a single set of groups of mapped features where each group has a constant number of mapped features.

The two new introduced BLS-based algorithms are: variable features BLS without (VF-BLS) and with cascades (VCFBLS), which are implemented with and without incremental learning. VF-BLS and VCFBLS algorithms consist of a variable number of mapped features and groups of mapped features as well as an explicit feature selection algorithm to create subsets of input data thus making the model more generalized compared to BLS algorithms. Models generated using the developed algorithms with integrated feature selection enable utilizing a variable number of selected features. Each mapped feature in VF-BLS is created based on input data while in case of VCFBLS, consequent mapped features are generated based on the previous mapped feature. The advantage of VF-BLS and VCFBLS algorithms is their ability to derive generalized models by using various subsets of input data to generate mapped features thus providing an easy process for creating models. Furthermore, VF-BLS and VCFBLS models are developed using a single experiment with integrated stages for selecting features and generating models. Allocating a smaller number of features generated from the input data may improve the model training time.

### 1.2.3 BGP Anomaly Detection Tool for Real-Time and Off-Line Detection

A BGP anomaly detection tool called *BGPGuard* has been developed to integrate various stages of the anomaly detection process. The tool consists of the following modules: data download, feature extraction, data partition, data processing, machine learning algorithms, parameter selection, machine learning models, and classification. *BGPGuard* is based on Python and JavaScript and has been developed with both terminal-based and web-based applications for Linux platforms. The developed tool is used for retrieving real-time data using the *update* messages collected by RIPE or Route Views. It may be used to detect (in real-time mode) anomalies using VF-BLS pre-trained models. The off-line classification mode may be customized by specifying the collection site, start and end dates as well as times of

the anomalous event, by partitioning the training and test datasets, and by choosing the RNN or VFBL algorithm.

## 1.3 Research Publications

I have co-authored 2 book chapters, 1 journal paper, and 10 conference publications. An additional paper is in preparation for publication in a magazine. Listed are the publications with brief descriptions of the main contributions. (Note: Conference publications [12] and [14] emanated from an unrelated research project.)

### Book Chapters

[1] Q. Ding, **Z. Li**, S. Haeri, and Lj. Trajković, “Application of machine learning techniques to detecting anomalies in communication networks: datasets and feature selection algorithms,” in *Cyber Threat Intelligence*, A. Dehghantanha, M. Conti, and T. Dargahi, Eds., Berlin: Springer, pp. 47–70, 2018.

Border Gateway Protocol enables the Internet data routing. BGP anomalies may affect the Internet connectivity and cause routing disconnections, route flaps, and oscillations. Hence, detection of anomalous BGP routing dynamics is a topic of great interest in cybersecurity. In this Book Chapter, we describe main properties of the protocol and datasets that contain BGP records collected from various public and private domain data collection sites such as RIPE, Route Views, and BCNET. We employ various feature selection algorithms to extract the most relevant features that are later used to classify BGP anomalies.

[2] **Z. Li**, Q. Ding, S. Haeri, and Lj. Trajković, “Application of machine learning techniques to detecting anomalies in communication networks: classification algorithms,” in *Cyber Threat Intelligence*, A. Dehghantanha, M. Conti, and T. Dargahi, Eds., Berlin: Springer, pp. 71–92, 2018.

In this Book Chapter, we apply various machine learning techniques for classification of known network anomalies. With the advent of fast computing platforms, many neural network-based algorithms have proved useful in detecting BGP anomalies. Performance of classification algorithms depends on the selected features and their combinations.

### Journal Publications

[3] **Z. Li**, A. L. Gonzalez Rios, and Lj. Trajković, “Machine learning for detecting power outage and ransomware using BGP routing records,” *IEEE Commun. Mag.*, to be submitted.

Various anomaly and intrusion detection approaches based on machine learning have been employed to analyze BGP *update* messages collected from RIPE and Route Views. In

this paper, we survey machine learning algorithms for detecting BGP anomalies and intrusions. GBDT and deep learning algorithms are evaluated by creating models using collected datasets that contain power outage and ransomware events.

[4] **Z. Li**, A. L. Gonzalez Rios, and Lj. Trajković, “Machine learning for detecting anomalies and intrusions in communication networks,” *IEEE J. Sel. Areas Commun.*, vol. 39, no. 7, pp. 2254–2264, July 2021.

In this paper, we evaluate performance of RNNs including Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), Bi-RNNs (Bi-LSTM and Bi-GRU), LightGBM, and BLS with its extensions to classify known network intrusions. We propose two BLS-based algorithms with and without incremental learning. The algorithms may be used to develop generalized models by using various subsets of input data and expanding the network structure. The models are trained and tested using BGP routing records as well as network connection records from the NSL-KDD and Canadian Institute for Cybersecurity (CIC) datasets. Performance of the models is evaluated based on selected features, accuracy, F-Score, and training time.

## Conference Publications

[5] **Z. Li**, A. L. Gonzalez Rios, and Lj. Trajković, “Classifying Denial of Service Attacks using Fast Machine Learning Algorithms,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Melbourne, Australia, Oct. 2021, pp. 1221–1226 (virtual).

Denial of Service (DoS) attacks are harmful cyberattacks that diminish Internet resources and services. Hence, detecting these cyberattacks is a topic of great interest in cybersecurity. Using traditional machine learning approaches in intrusion detection systems requires long training time and has high computational complexity. Thus, we evaluate performance of fast machine learning algorithms for training and generating models to detect denial of service attacks in communication networks. We use synthetically generated datasets that captured TCP and UDP network flows in a controlled testbed laboratory environment. Evaluated algorithms include BLS and its extensions as well as XGBoost, LightGBM, and CatBoost algorithms. Experiments indicate that boosting algorithms often require shorter training time and have better performance.

[6] **Z. Li**, A. L. Gonzalez Rios, and Lj. Trajković, “Detecting Internet worms, ransomware, and blackouts using recurrent neural networks,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Toronto, Canada, Oct. 2020, pp. 2165–2172 (virtual).

Analyzing and detecting BGP anomalies are topics of great interest in cybersecurity. Various anomaly detection approaches such as time series and historical-based analysis, statistical validation, reachability checks, and machine learning have been applied to BGP datasets. In this paper, we use BGP *update* messages collected from RIPE and Route Views

to detect BGP anomalies caused by Slammer worm, WannaCrypt ransomware, and Moscow blackout by employing RNN machine learning algorithms.

[7] A. L. Gonzalez Rios, **Z. Li**, K. Bekshentayeva, and Lj. Trajković, “Detection of denial of service attacks in communication networks,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Seville, Spain, Oct. 2020.

We propose detection of DoS cyber attacks in communication networks by employing the BLS that requires shorter training time while achieving comparable performance. Because designing effective detection systems relies on training and test datasets that contain anomalous network traffic data, in this paper we evaluate the performance of various BLS models by using recently generated network intrusion datasets: CICIDS2017 and CSECIC-IDS2018 that contained Slowloris, GoldenEye, HULK, and SlowHTTP DoS attacks. The best accuracy and F-Score were often achieved using BLS with cascades while BLS with incremental learning usually required shorter training time. The Extremely Randomized Trees (Extra-Trees) algorithm is used to calculate feature importance.

[8] **Z. Li**, A. L. Gonzalez Rios, G. Xu, and Lj. Trajković, “Machine learning techniques for classifying network anomalies and intrusions,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Sapporo, Japan, May 2019, pp. 1–4.

We employ two deep learning RNNs with a variable number of hidden layers: LSTM and GRU. We also evaluate the BLS and its extensions. The models are trained and tested using BGP datasets that contain routing records collected from RIPE and BCNET as well as the NLS-KDD dataset containing network connection records. The algorithms are compared based on accuracy and F-Score.

[9] A. L. Gonzalez Rios, **Z. Li**, G. Xu, A. Diaz Alonso, and Lj. Trajković, “Detecting network anomalies and intrusions in communication networks,” in *Proc. 23<sup>rd</sup> IEEE Int. Conf. Intell. Eng. Syst.*, Godollo, Hungary, April 2019, pp. 29–34.

In this paper, we compare the performance of SVM and BLS supervised machine learning approaches using data from the BGP routing records, a simulated air force base network, and an experimental testbed.

[10] **Z. Li**, P. Batta, and Lj. Trajković, “Comparison of machine learning algorithms for detection of network intrusions,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Miyazaki, Japan, Oct. 2018, pp. 4238–4243.

We apply RNNs and BLS machine learning algorithms to classify known network intrusions. The developed models are trained and tested using the NSL-KDD dataset containing information about intrusion and regular network connections. The algorithms are used to classify various types of intrusion classes and regular data and are compared based on accu-

racy and F-Score. Performance results indicate that the BLS algorithm shows comparable performance with shorter training time.

[11] P. Batta, M. Singh, **Z. Li**, Q. Ding, and Lj. Trajković, “Evaluation of support vector machine kernels for detecting network anomalies,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Florence, Italy, May 2018, pp. 1–4.

In this paper, we evaluate performance of SVM with linear, quadratic, and cubic kernels. The SVM kernels are compared based on accuracy and the F-Score when detecting BGP anomalies in Internet traffic traces. The performance heavily depends on the selected features and their combinations.

[12] H. Ben Yedder, Q. Ding, U. Zakia, **Z. Li**, S. Haeri, and Lj. Trajković, “Comparison of virtualization algorithms and topologies for data center networks,” in *Proc. The 26th Int. Conf. Compt. Comm. Netw., 2nd Workshop Netw. Secur. Analytics Automat.*, Vancouver, Canada, Aug. 2017.

Data centers are core infrastructure of cloud computing. Network virtualization in these centers is a promising solution that enables coexistence of multiple virtual networks on a shared infrastructure. It offers flexible management, lower implementation cost, higher network scalability, increased resource utilization, and improved energy efficiency. In this paper, we consider switch-centric data center network topologies and evaluate their use for network virtualization by comparing Deterministic (D-ViNE) and Randomized (R-ViNE) Virtual Network Embedding, Global Resource Capacity (GRC), and Global Resource Capacity-Multicommodity (GRC-M) Flow algorithms.

[13] Q. Ding, **Z. Li**, P. Batta, and Lj. Trajković, “Detecting BGP anomalies using machine learning techniques,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Budapest, Hungary, Oct. 2016, pp. 3352–3355.

In this paper, we first employ the Minimum Redundancy Maximum Relevance (mRMR) feature selection algorithms to extract 10 most relevant features used for classifying BGP anomalies and then apply the SVM and LSTM algorithms for data classification. The SVM and LSTM algorithms are compared based on accuracy and F-score. Their performance was improved by choosing balanced data for model training.

[14] S. Haeri, Q. Ding, **Z. Li**, and Lj. Trajković, “Global resource capacity algorithm with path splitting for virtual network embedding,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Montreal, Canada, May 2016, pp. 666–669.

Virtual Network Embedding problem that addresses efficient mapping of virtual network elements onto a physical infrastructure (substrate network) is one of the main challenges in network virtualization. The GRC is a VNE algorithm that utilizes for virtual link mapping

a modified version of Dijkstra’s shortest path algorithm. In this paper, we propose the GRC-M algorithm that utilizes the Multicommodity Flow (MCF) algorithm. MCF enables path splitting and yields to higher substrate resource utilizations. Simulation results show that MCF significantly enhances performance of the GRC algorithm.

## 1.4 Roadmap

The dissertation is organized as follows:

In Chapter 1, we describe the motivation and the importance of using machine learning algorithms to enhance cybersecurity. We then present the summary of my research contributions and the list of publications.

In Chapter 2, given literature review includes the investigation of algorithms used for intrusion detection, collected network datasets, and intrusion detection systems. We then describe the datasets (BGP, NSL-KDD, CICIDS2017, CSE-CIC-IDS2018, CICDDoS2019) used for comparison of various intrusion detection approaches. Described are detailed steps for extracting, cleaning, and transforming the raw network traffic data into the matrices used for model training and testing. We also use data visualization to observe properties of the analyzed data.

In Chapter 3, we introduce supervised feature selection and unsupervised dimension reduction approaches and present examples of most relevant features and their feature scores using BGP datasets.

In Chapter 4, we describe and evaluate the performance of traditional supervised machine learning algorithms: SVM, naïve Bayes, HMM, decision tree, ELM. Their strengths and weaknesses are also discussed.

In Chapter 5, CNN, RNNs (LSTM and GRU), Bi-RNNs (Bi-LSTM and Bi-GRU) are explained. We compare their performance using small (BGP) and large size (CIC) datasets.

In Chapter 6, two families of fast machine learning algorithms (BLS, BLS extensions, and GBDT) are described and compared. Experimental results indicate that they may be viable candidates for intrusion detection systems because of their fast training speed. We conclude this Chapter with discussion section addressing advantages of GBDT algorithms.

In Chapter 7, we provide details and pseudocode of the proposed VFBLs and VCFBLs algorithms as well as their extensions. Performance of the VFBLs and VCFBLs is compared to the deep learning and fast machine learning algorithms (BLS and GBDT) using BGP, NSL-KDD, and CIC datasets.

In Chapter 8, we introduce a BGP anomaly detection tool called *BGPGuard*. Its high-level architecture consists of eight modules that are based on the process for anomaly detection. Linux terminal-based and web-based applications are developed. Sample settings and operations are also given in the user guide.

We conclude and describe the future work in Chapter 9.

## Chapter 2

# Network Anomalies and Intrusions

In this Chapter, we first conduct a literature review with attention focused on algorithms, datasets, and systems for intrusion detection. We also describe the experimental datasets used in this dissertation.

### 2.1 Literature Review

In this Section, we review relevant literature dealing with conventional and machine learning techniques used for network intrusion detection.

#### 2.1.1 Detection of Network Anomalies

Anomalies refer to the data patterns that exhibit unexpected behavior. Anomalies are categorized into three types: point anomalies, contextual anomalies, and collective anomalies [6]. Point anomaly is the simplest type when the data point is outside the boundary of regions consisting of regular data points. Contextual anomaly refers to the case when a data point is defined as anomalous in a specific context. Collective anomaly is a collection of anomalous data points in the entire dataset. Detecting anomalies is important for system security and has been applied to a variety of research and application domains [37]. These domains include detection of network intrusions (BGP traffic), fraud (credit cards, mobile phones), medical anomalies (magnetic resonance imaging data, electroencephalograms), industrial damage (mechanical units), and maritime suspicious activities (automated identification system data) [38, 39].

Detailed comparison of various network intrusion techniques has been reported in the literature [37]. Demands for Internet services have been steadily increasing and anomalous events and their effects have dire economic consequences. Determining the anomalous events and their causes is an important step in assessing loss of data by anomalous routing. Hence, it is important to classify these anomalous events and prevent their effects on the network.

Anomaly detection techniques have been applied in communication networks. These techniques are employed to detect network anomalies such as intrusion attacks, worms, and

Distributed Denial of Service (DDoS) attacks [40, 41], that frequently affect the Internet and its applications. Network anomalies are detected by analyzing collected traffic data and generating various classification models. A variety of techniques have been proposed to detect network anomalies.

### **2.1.2 Conventional Techniques used for Intrusion Detection**

Early approaches include developing traffic models using statistical signal processing techniques where a baseline profile of network regular operation is developed based on a parametric model of traffic behavior and a large collection of traffic samples to account for regular (anomaly-free) cases [42]. Anomalies may then be detected as sudden changes in the mean values of variables describing the baseline model. However, it is infeasible to acquire datasets that include all possible cases. In a network with quasi-stationary traffic, statistical signal processing methods have been employed to detect anomalies as correlated abrupt changes in network traffic [43].

The main focus of approaches also proposed in the past is developing models for classification of anomalies. The accuracy of a classifier depends on the extracted features, combination of selected features, and underlying models. Recent research reports describe a number of applicable classification techniques. One of the most common approaches is based on a statistical pattern recognition model implemented as an anomaly classifier and detector [44]. Its main disadvantage is the difficulty in estimating distributions of higher dimensions. For example, a Bayesian detection algorithm was designed to identify unexpected route misconfigurations as statistical anomalies [45]. An instance-learning framework also employed wavelets to systematically identify anomalous BGP route advertisements [46]. Other proposed techniques are rule-based methods that have been employed for detecting anomalous BGP events. An example is the Internet Routing Forensics (IRF) that was applied to classify anomalous events [47]. However, rule-based techniques are not adaptable learning mechanisms. They are slow, have high degree of computational complexity, and require a priority knowledge of network conditions.

### **2.1.3 Machine Learning Techniques used for Intrusion Detection**

Detection of evolving cyber attacks is a challenging task for conventional network intrusion detection techniques. Machine learning algorithms have been used to successfully classify network anomalies and intrusions [48, 49]. A survey of various methods, systems, and tools used for detecting network anomalies reviews a variety of existing approaches [37]. The authors have examined recent techniques to detect network anomalies and discussed detection strategy and employed datasets. They included performance metrics for evaluating detection methods and description of various datasets and their taxonomy. They also identified issues and challenges in developing new anomaly detection methods and systems.

Various machine learning techniques to detect cyber threats have been reported in the literature. One-class SVM classifier with a modified kernel function was employed [50] to detect anomalies in Internet Protocol (IP) records. However, unlike the approach in our studies, the classifier is unable to indicate the specific type of anomalies. Stacked LSTM networks with several fully connected LSTM layers have been used for anomaly detection in time series [51]. The method was applied to medical electrocardiograms, a space shuttle, power demand, and multi-sensor engine data. The analyzed data contain both long-term and short-term temporal dependencies. Another example is the multi-scale LSTM that was used to detect BGP anomalies [52] using accuracy as a performance measure. In the preprocessing phase, data were compressed using various time scales. An optimal size of the sliding window was then used to determine the time scale to achieve the best performance of the classifier. Multiple HMM classifiers were employed to detect Hypertext Transfer Protocol (HTTP) payload-based anomalies for various web applications [53]. Authors first treated payload as a sequence of bytes and then extracted features using a sliding window to reduce the computational complexity. HMM classifiers were then combined to classify network intrusions. It was shown [8] that the naïve Bayes classifier performs well for categorizing the Internet traffic emanating from various applications. Weighted ELM [54] deals with unbalanced data by assigning relatively larger weights to the input data arising from a minority class. Signature-based and statistics-based detection methods have also been proposed in [55]. Echo State Networks (ESNs) were used to compare the performance of Bi-LSTM deep neural networks using datasets collected by the Canadian Institute for Cybersecurity and the RIPE and Route Views data collection sites: CICIDS2017, CSE-CIC-IDS2018, CICDDoS2019, Slammer, Nimda, Code Red, DDoS2019, and DDoS2020 [56]. ESNs models achieved comparable performance with shorter training time.

#### 2.1.4 Intrusion Detection Systems

Various Intrusion Detection Systems (IDSs) [6, 46, 57, 58] are used to identify and classify malicious activities such as anomalies and intrusions in communication networks. They may be host-based or network-based. Host-based systems protect the host (endpoint) by monitoring operating system files and processes. Network-based systems (NIDSs) monitor incoming network traffic (IP addresses, service ports, and protocols) by analyzing flows of packets or by inspecting packet headers. Their role is to enhance security by identifying suspicious events in the observed network traffic. Detecting malicious network intrusions may be signature-based or anomaly-based. Signature-based techniques [49] rely on known events that follow certain rules and patterns while anomaly-based intrusion detection techniques [5, 59] rely on detecting deviations from an expected behavior.

An early hybrid IDS was proposed to identify misuse and anomaly intrusions using random forests [60]. Intrusion detection systems [6, 37, 61] have been designed using machine learning and deep neural networks such as stacked non-symmetric deep auto-encoder [62]

and RNNs [63]. Training time is often of concern when employing deep learning algorithms. Hence, a stacked non-symmetric deep auto-encoder (NDAE) that combines deep and shallow learning offered by the random forest classifier has been proposed and implemented using a Graphics Processing Unit (GPU) [62]. Network (NIDS) [64] and recurrent neural network (RNN-IDS) [63] intrusion detection systems were proposed and compared [48,65] with various machine learning algorithms such as J48, naïve Bayes, naïve Bayes tree, random forests, random tree, multilayer perceptron, and SVM. A hybrid framework was proposed [66] to include binary classifier modules based on the C4.5 algorithm [67] and an aggregation module generating certain and uncertain output classes. The data points from both classes are imported to a  $k$ -NN module that refines the class of uncertain data points. SwiftIDS employs LightGBM for generating models. Real-time data acquisition and data processing were used prior to the classification of anomalies using the proposed parallel intrusion detection mechanism [68]. An unsupervised deep learning approach (AP2Vec) [69] has been implemented in a detection system for classifying BGP hijacking by converting autonomous system numbers and IP address prefixes to vectors in training process.

Industrial tools for detecting anomalies and intrusions include BGProtect [70], Cisco secure intrusion prevention system [71], FortiGuard intrusion prevention system [72], and Palo Alto Networks advanced threat prevention [73].

### 2.1.5 Description of Datasets

Machine learning algorithms have been evaluated for robustness, high accuracy, and training time when classifying various datasets collected from communication networks. Reliable testing and validation of anomaly and intrusion detection algorithms depend on the quality of datasets such as traffic collected from deployed networks or experimental testbeds. The most widely used benchmark datasets in the literature [37,48,74–76] are Knowledge Discovery in Databases (KDD) Cup 1999 (KDD’99) [77] and NSL-KDD [18]. KDD’99 intrusion dataset is based on the Defense Advanced Research Projects Agency (DARPA) 1998 dataset that contains nine weeks of collected traffic when various intrusions were introduced in a simulated US Air Force base network [78,79]. NSL-KDD [18] dataset is an improved version of the KDD’99.

Other benchmark datasets that are adopted by researchers include DARPA 2000 [80], Center for Applied Internet Data Analysis (CAIDA) [81], BGP [16,17,82], UNIBS-2009 [83], UNSW-NB15 [84], and CIC datasets [85]. The DARPA 2000 dataset is generated based on two attack simulation scenarios (LLDOS 1.0 and LLDOS 2.0.2) executed by Lincoln Laboratory. Several types of attacks such as probing, break-in, and DDoS were simulated. Internet measurement and research including network data collection, topology analysis, and cybersecurity have been conducted by CAIDA. Their DDoS Attack 2007 dataset is widely used and also contains records of DDoS attacks. However, the traffic traces of the attack only lasted limited time (one hour). The BGP datasets are generated by using routing

records collected by RIPE and Route Views sites. We have created several BGP datasets [82] based on well-known BGP anomalies: Slammer [86, 87], Nimda [88, 89], and Code Red [90], which occurred in January 2003, September 2001, and July 2001, respectively. Several recent BGP anomalous events are also discussed in this dissertation. An example of regular network traffic is the UNIBS-2009 dataset that contains a large number of flow-based network packet traces without malicious behavior. This dataset is collected from the edge router of the University of Brescia, Italy campus network. Traffic traces are collected from web, mail, peer-to-peer, and Skype applications. UNSW-NB15 dataset is created in a synthetic environment at the University of New South Wales, Australia cyber security lab and contains both regular and malicious traffic. This dataset includes nine major types of attacks: Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode, and Worms. CIC datasets are generated by the Canadian Institute for Cybersecurity at University of New Brunswick. The CIC provides various datasets for intrusion detection including ISCXIDS2012 [91], CICIDS2017 [19], CSE-CIC-IDS2018 [20], CICDDoS2019 [21], and CIC-Darknet2020 [92]. In the following section we discuss datasets (BGP, NSL-KDD, CIC) used in our experiments.

## 2.2 Border Gateway Protocol Datasets

In this Section, we describe BGP, BGP data collection sites, various BGP anomalous events, extraction of features, and training and test datasets for BGP Internet worms, power outages, and ransomware attacks.

### 2.2.1 Overview of BGP

BGP [93] is a routing protocol that plays an essential role in forwarding IP traffic between the source and the destination Autonomous Systems (ASes). An AS is a collection of BGP peers managed by a single administrative domain [94]. It consists of one or more networks that possess uniform routing policies while operating independently. Internet operations such as connectivity and data packet delivery are facilitated by various ASes.

The main function of BGP is to select the best routes between ASes based on network policies enforced by network administrators. Routing algorithms determine the route that a data packet takes while traversing the Internet. They exchange reachability information about possible destinations. BGP is an upgrade of the Exterior Gateway Protocol (EGP) [95]. It is an interdomain routing protocol used for routing packets in networks consisting of a large number of ASes. The current version of BGP (BGP-4) allows Classless Inter-Domain Routing (CIDR), aggregation of routes, incremental additions, better filtering options, and it has the ability to set routing policies. BGP employs the path vector protocol, which is a modified version of the distance vector protocol [96]. BGP is a standard for the exchange of information among the Internet Service Providers (ISPs).

BGP relies on the TCP to establish a connection (port 179) between the routers. A BGP router establishes a TCP connection with its peers that reside in different ASes. Because of their size, BGP routing tables are exchanged once between the peering routers when they first connect. BGP allows ASes to exchange reachability information with peering ASes to transmit information about the availability of routes within an AS. Based on the exchanged information and routing policies, it determines the most appropriate path to destination. BGP allows each subnet to announce its existence to the Internet and to publish its reachability information. Hence, all sub-networks are inter-connected and are known to the Internet.

BGP is an incremental protocol that sends updates only if there are reachability or topology changes within the network. Afterwards, only updates regarding new prefixes or withdrawals of the existing prefixes are exchanged. BGP routers exchange four types of messages: *open*, *keepalive*, *update*, and *notification* [95]. The *open* message that contains basic information such as router identifier, BGP version, and the AS number is used to open a peering session. An *open* message is sent after the TCP connection is established, followed by a *keepalive* message upon confirmation of its receipt. Routing information (advertisements and withdrawals) are exchanged between BGP peers using *update* messages. When an error condition is detected, a *notification* message is sent and the BGP connection is closed. BGP data are used to analyse the Internet topology and hierarchy, infer AS relationships [97], and evaluate various intrusion and anomaly detection mechanisms [57, 98]. Data are collected using BGP trace collectors (RIPE [16] and Route Views [17]), route servers, looking glasses, and the Internet routing registries. Various BGP data collections may be combined to provide a more complete Internet topology [99].

A sample of a BGP *update* message is shown in Table 2.1. It contains two Network Layer Reachability Information (NLRI) announcements, which share attributes such as the *AS-path*. The *AS-path* attribute in the BGP *update* message indicates the path that a BGP packet traverses among AS peers. The *AS-path* attribute enables BGP to route packets via the best path.

While BGP is a simple and flexible routing protocol, implementing routing policies is a complex and error-prone task. It also lacks security mechanisms to verify legitimate route updates. Therefore, BGP dynamics is prone to anomalies that may impede successful exchange of reachability messages and generate a large volume of anomalous *update* messages. BGP anomalies include worms, ransomware attacks, routing misconfigurations, IP prefix hijacks, and link failures. BGP anomalies are caused by changes in network topologies, updated AS policies, or router misconfigurations. They affect the Internet servers and hosts and are manifested by anomalous traffic behavior. Anomalous events in communication networks cause traffic behavior to deviate from its usual profile. These events may spread false routing information throughout the Internet by either dropping packets or directing traffic through unauthorized ASes and, hence, risking eavesdropping. Large-scale power outages

Table 2.1: Sample of a BGP *update* message. IGP: Interior Gateway Protocol.

Field	Value
TIME	2022-1-16 01:40:06
TYPE	BGP4MP/BGP4MP_MESSAGE_AS4 AFI_IP
FROM	192.65.185.195
TO	192.65.185.40
BGP PACKET TYPE	UPDATE
ORIGIN	IGP
AS-PATH	15547 6939 4788 45259 10094 10030
NEXT-HOP	192.65.185.195
ANNOUNCED	183.171.121.0/24
ANNOUNCED	183.171.120.0/24

may affect ISPs due to unreliable power backup. They could also cause network equipment failures leaving affected networks isolated and their service disrupted. Configuration errors in BGP routers induce anomalous routing behavior. Routing table leak and prefix hijack [100] events are examples of BGP configuration errors that may lead to large-scale disconnections in the Internet. A routing table leak occurs when an AS such as an ISP announces a prefix from its Routing Information Base (RIB) that violates previously agreed upon routing policy. A prefix hijack is the consequence of an AS originating a prefix that it does not own. IP BGP hijacks may be detected and analyzed by using algorithms and tools developed by BGProtect [70].

### 2.2.2 BGP Data Collections

BGP routing information is shared by ISPs located in various geographical locations [99]. BGP trace collectors receive BGP messages from their peers and periodically store the routing updates and tables into publicly available archives. Routing tables contain numerous entries from each peering AS that indicate the preferred paths to destination prefixes at a given time. Routing messages indicate alternative paths and backup links. BGP routing *update* messages are available from global BGP monitoring systems such as RIPE [16] and Route Views [17]. They may be collected using Quagga [101], a suite derived from the multi-server routing software Zebra [102]. BGP *update* messages are stored in multi-threaded routing toolkit (MRT) format.

**RIPE:** The RIPE Routing Information Service (RIS) [103] is a RIPE Network Coordination Centre (NCC) project established in 2001 to collect and store routing data from several ASes worldwide. The main collector is located at NCC and consists of a route collector, database, and user interface. Remote Route Collectors (RRCs) installed at major topologically interesting Internet points use the Quagga routing software to collect BGP data. Routes are collected directly from the AS border routers at the rrc or via multi-hop BGP peering from nearby routers. The raw data are collected using state dumps while

batches of updates for each rrc are periodically made available. The Zebra tool is used to collect BGP *update* messages every 15 min before July 23, 2003 and every 5 min afterwards while the BGP routing tables are stored every 8 h. RIS currently consists of 25 rrcs located at Europe (16), North America (4), Asia (2), South America (2), and Africa (1).

**Route Views:** Route Views [17] is the University of Oregon project to collect real-time BGP routing data from various backbone routers and locations worldwide. The publicly available data have been used for routing analysis, AS path visualization, analysis of IPv4 address space utilization, topological studies, and generation of geographic host locations. Backbone routers (Cisco, Juniper), configured as IPv4 or IPv6 Route-Views-like route servers, connect as peers via multi-hop BGP sessions.

Route Views project employs three types of collectors: FRRouting, Quagga, and Cisco. FRRouting and Quagga collectors are based on Zebra. BGP *update* messages and routing tables are stored in MRT format and collected every 15 min and 2 h, respectively. Data from Cisco collectors are generated every 2 h starting at 00:00 by using the Cisco command line interface to extract routes and their attributes. There are 36 Route Views collectors (29 FRRouting, 6 Quagga, and 1 Cisco) distributed across Regional Internet Registries (RIRs): ARIN (14), LACNIC (6), APNIC (7), AFRINIC (4), and RIPE NCC (5) as shown in Fig. 2.1.

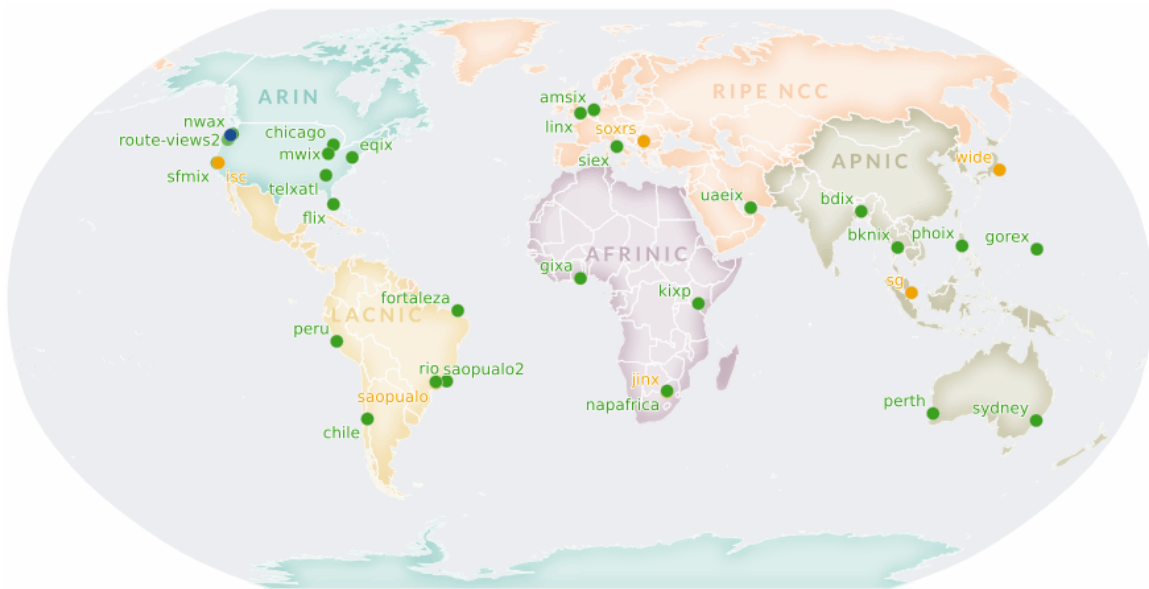


Figure 2.1: Route Views collector map: 36 collectors are located in 5 RIR regions [2].

### 2.2.3 BGP Anomalies

BGP anomalies are harmful changes in the protocol’s behaviour and may consist of single updates (invalid AS numbers, invalid or reserved IP prefixes, a prefix announced by an illegitimate AS, AS-PATH without a physical equivalent) or a set of updates (longest and

shortest paths, changes in the behaviour of BGP traffic over time) [57]. These anomalies are classified as: direct intended anomalies, direct unintended anomalies, indirect anomalies, and link failures [6]. BGP hijackings are direct intended anomalies where the attacker redirects routes from a valid AS by claiming the ownership of a prefix or sub-prefix. DoS, DDoS, man-in-the-middle, and phishing attacks employ BGP hijackings. BGP misconfigurations are direct unintended anomalies that may cause announcements of used (hijacking) or unused (leaked routers) prefixes. The origin misconfigurations occur when non-owned prefixes are accidentally announced or private ASes are not filtered while export misconfigurations appear when BGP policies are accidentally configured. BGP indirect anomalies occur when Internet web servers are attacked, which generates BGP instabilities such as routing overload. For example, during the Slammer worm attack a critical BGP instability was caused by a significant increase in the number of announcements of BGP updates. A BGP link failure causes reachability or connectivity loss between private (dedicated connection) or public (service provider) BGP peers. The Moscow power system blackout (2005) and Mediterranean cable break (2008) resulted in BGP link failures that affected cities in more than 20 countries.

Worms are self-replicating codes that exploit systems vulnerabilities and propagate via networks [104, 105]. They employ email applications or scan engines to spread to various hosts and may carry other malware as their payload. While antivirus systems may require several hours to identify worms, an IDS is capable of detecting worms faster because they take large portion of a network bandwidth. Slammer [86], Nimda [88], and Code Red [90] are well-known worms that exploited vulnerabilities of Microsoft Structured Query Language (SQL) and Internet Information Service (IIS).

Power system blackouts are the loss of electrical power to end users and are caused by failures or overload of transmission lines, failures of automatic emergency control systems, malfunctions of protection devices, or human errors. Blackouts are critical to environment and public safety and, hence, investigating their cascading process is important to determine triggering events, evaluate the consequences, and develop preventive solutions and automatic protection systems [106].

Ransomware attacks rely on advanced cryptography to lock the victim's data until a ransom is paid. They may be classified as: cryptoworm, ransomware-as-a-service (RaaS), and automated active adversary [107]. Cryptoworms replicate themselves to other hosts for maximum reach and impact. RaaS attacks, sold on the dark web as distribution kits, are typically deployed via malicious spam e-mails or exploit kits. In case of automated active adversary ransomware, the attackers scan the Internet for systems with weak protection, enter the system, and plan the attack for maximum damage. Ransomware attacks rely on tools and processes such as runtime packers and exploits. Runtime packers are compressed executable-files used to avoid detection of attacks until they have completed their core task while exploits (EternalBlue, Windows Event Viewer process, CVE-2018-8453) are

tools that ensure that the attacks gain administrative privileges by taking advantage of the vulnerabilities in an operating system. A ransomware may store the encrypted data on the same (overwrite) or available (copy) disk sectors. During the encryption, data are partially or fully renamed. Well-known ransomware attacks include WannaCrypt [107], Petya, and Locky [108].

The following is the chronological list of BGP anomalies considered to generate the datasets used for evaluating the machine learning algorithms.

**Code Red:** Although the Code Red [90] worm attacked Microsoft IIS web servers earlier, the peak of infected computers was observed on July 19, 2001. The worm replicated itself by exploiting weakness of the IIS servers and, unlike the Slammer worm, Code Red searched for vulnerable servers to infect. Rate of infection was doubling every 37 minutes.

**Nimda:** Nimda [88, 89] occurred in September 18, 2001 and exploited vulnerabilities in the Microsoft IIS web servers for the Internet Explorer 5. It used three methods for propagation: email, network shares, and the web. The worm propagated by sending an infected attachment that was automatically downloaded after viewing email. A user could also download it from the website or access an infected file through the network.

**Slammer:** Slammer [86, 87] is the fastest worm that self-propagated by using the UDP. It commenced on January 23, 2003 at 05:31 (GMT) and lasted 14.5 h. This worm does not store itself in the memory of affected hosts. It only exists as a network packet and acts by running processes in the victim’s host. During the Slammer attack, SQL servers and PCs with Microsoft SQL server data engine (MSDE) were infected by exploiting the buffer overflow vulnerability in Microsoft SQL server 2000 resolution service. The code replicated itself by infecting new vulnerable machines through scanning randomly generated IP addresses and sending packets to UDP port 1434. The number of infected machines doubled approximately every 9 s. This infection speed caused a DoS attack in affected networks.

The dates and duration of the three BGP well-known anomalous events are shown in Table 2.2.

Table 2.2: Examples of known BGP Internet worms.

Dataset	Beginning of event GMT	End of event GMT	Duration (min)
Slammer	25.01.2003, 05:31	25.01.2003, 19:59	869
Nimda	18.09.2001, 13:19	19.09.2001, 10:59	1,301
Code Red	19.07.2001, 13:20	19.07.2001, 23:19	600

**Moscow Blackout:** The Chagino substation of the Moscow energy ring experienced a transformer failure on May 24, 2005 at 20:57 (MSK) [109, 110]. The event caused a complete shutdown of the substation and a blackout that affected all customer until 16:00 (MSK) of May 26, 2005. At 11:00 (MSK) on May 25, 2005, Unified Energy System of Russia set up an

Emergency Response Center to eliminate the blackout. The dispatching control staff and automatic protective devices stopped the cascading failure in 2 h and 20 min and power was restored to all socially important facilities and vital infrastructure in Moscow by 18:00 (MSK). The Moscow city transportation system regained power at 21:00 (MSK). Power was fully restored to all customers on May 26, 2005.

During the blackout, the Internet traffic exchange point MSK-IX was disconnected from 11:00 to 17:00 (MSK) [111]. Routing instabilities were observed because several MSK-IX ISP peers lost connectivity for an extended period [44] and primarily affected ASes in Russia and the APNIC RIR [112]. Based on reports [109], the peak power outage occurred during the morning peak load and its duration was 4 h [113]. Hence, in this study, we consider as anomalies data collected from 7:00 to 10:59 (MSK) on May 25, 2005.

**Pakistan Power Outage:** The Pakistan power outage occurred on January 09, 2021 between 18:40 and 23:59 UTC. It was caused by a cascading effect after an abrupt frequency drop in the power transmission system of the Guddu power plant. As a result, the network connectivity levels in Pakistan decreased to 62 % within the first hour and to 52 % after six hours. Critical infrastructure and telecommunication providers were the least affected because they relied on power backup systems. Power to most cities was restored by 12:00 UTC on January 10, 2021.

**WannaCrypt Ransomware Attack:** WannaCrypt (WannaCry) is a cryptoworm ransomware that works by gaining administrative privileges and employs the EternalBlue exploit and DoublePulsar backdoor in systems running Microsoft Windows 7 [107, 114]. It lasted from May 12, 2017 to May 15, 2017 and infected over 230,000 computers in 150 countries. A victim’s data files are encrypted using 128-bit advanced encryption standard (AES) in cipher block chaining (CBC) mode. After the encryption is completed, data files are renamed by adding the extension “.wncry” while the string “WannaCrypt!” is added to the combination of encrypted AES key and data. WannaCrypt may copy or overwrite the data after encryption. It uses the Volume Shadow Service (vssadmin.exe) Windows utility to delete previous versions of the locked data. By manipulating the Windows Boot Configuration Data (bcdedit.exe), the attack: (1) prevents Windows diagnostics-and-repair feature to automatically run after a third unsuccessful boot or (2) attempts a normal boot even in case of a failed boot, shutdown, or checkpoint. WannaCrypt flushes buffers to ensure that all encrypted data are only located in the storage drive. It replaces the Windows desktop wallpaper with a message to inform the victim that data have been locked and to demand a ransom. After the ransom is paid, the risk remains that decryption of data fails.

WannaCrypt spreads throughout a network by attempting to connect to port 445. After the connection is established, the ransomware scans for the Windows server message block (SMB) EternalBlue vulnerability and checks if it is infected with the DoublePulsar backdoor. EternalBlue exploits the wrong casting, wrong parsing, and non-paged pool allocation defects of the SMB protocol as well as an address space layout randomization

(ASLR) bypass. Exploiting the wrong casting and parsing defects causes buffer overflow and overwrite while the non-paged pool allocation and ASLR allow placing the shellcode at a predefined executable address [115]. EternalBlue then implants the DoublePulsar backdoor in the victim’s host to send the cryptoworm payload using dynamic link library (DLL) injection [114, 115]. WannaCrypt replicates by querying for the non-existing domains:

- `www[.]iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com`
- `www[.]ifferfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com`.

Its replication may be prevented if the victims receive a response indicating that these domains are registered.

**WestRock Ransomware Attack:** The WestRock Company experienced a ransomware attack in late January 2021. This is the second largest packaging company in USA and owns over 320 manufacturing facilities worldwide. The ransomware attack was detected on January 23, 2021. It impacted the company’s information (IT) and operational (OT) technology systems for over six days. While IT systems store, process, maintain, and operate data, OT systems monitor and control industrial processes, events, and devices. Hence, the attack caused delays in shipments and production levels. The company implemented a controlled remediation plan that was executed in phases including systems shutdown and enhancement of existing security measures. We assume that the ransomware attack lasted between 1:12 UTC on 23.01.2021 and 23:59 UTC on 29.01.2021.

## 2.2.4 Extraction of Features from BGP Update Messages

Feature extraction and selection are the first steps in the classification process. We developed a tool (written in C#) [116] to parse the ASCII files and extract statistics of the desired features. The *AS-path* is a BGP *update* message attribute that enables the protocol to select the best path for routing packets. It indicates a path that a packet may traverse to reach its destination. If a feature is derived from the *AS-path* attribute, it is categorized as an *AS-path* feature. Otherwise, it is categorized as a *volume* feature. There are three types of features: continuous, categorical, and binary. We extracted *AS-path* and *volume* features shown in Table 2.3 [116].

Definitions of the extracted features are listed in Table 2.4. BGP *update* messages are either announcement or withdrawal messages for the NLRI prefixes. The NLRI prefixes that have identical BGP attributes are encapsulated and sent in one BGP packet [117]. Hence, a BGP packet may contain more than one announced or withdrawn NLRI prefix. The average and the maximum number of AS peers are used for calculating *AS-path* lengths. Duplicate announcements are the BGP update packets that have identical NLRI prefixes and the *AS-path* attributes. Implicit withdrawals are the BGP announcements with different *AS-paths* for already announced NLRI prefixes [118]. The edit distance between two *AS-path*

Table 2.3: List of features extracted from BGP *update* messages. The *AS-path* is a BGP *update* message attribute that enables the protocol to select the best path for routing packets. The features are categorized as *AS-path* and *volume* features.

Feature	Name	Category
1	Number of announcements	<i>volume</i>
2	Number of withdrawals	<i>volume</i>
3	Number of announced NLRI prefixes	<i>volume</i>
4	Number of withdrawn NLRI prefixes	<i>volume</i>
5	Average <i>AS-path</i> length	<i>AS-path</i>
6	Maximum <i>AS-path</i> length	<i>AS-path</i>
7	Average unique <i>AS-path</i> length	<i>AS-path</i>
8	Number of duplicate announcements	<i>volume</i>
9	Number of implicit withdrawals	<i>volume</i>
10	Number of duplicate withdrawals	<i>volume</i>
11	Maximum edit distance	<i>AS-path</i>
12	Arrival rate	<i>volume</i>
13	Average edit distance	<i>AS-path</i>
14-23	Maximum <i>AS-path</i> = $n$ , $n = (11, \dots, 20)$	<i>AS-path</i>
24-33	Maximum edit distance = $n$ , $n = (7, \dots, 16)$	<i>AS-path</i>
34	Number of Interior Gateway Protocol (IGP) packets	<i>volume</i>
35	Number of Exterior Gateway Protocol (EGP) packets	<i>volume</i>
36	Number of incomplete packets	<i>volume</i>
37	Packet size ( $B$ )	<i>volume</i>

attributes is the minimum number of deletions, insertions, or substitutions that need to be executed to match the two attributes [44]. For example, the edit distance between *AS-path* 513 940 and *AS-path* 513 4567 1318 is two because one insertion and one substitution are sufficient to match the two *AS-paths*. The maximum *AS-path* length and the maximum edit distance are used to count Features 14 to 33. We also consider Features 34, 35, and 36 based on distinct values of the origin attribute that specifies the origin of a BGP update packet and may assume three values: IGP, EGP, and incomplete. Even though the EGP protocol is the predecessor of BGP, EGP packets still appear in traffic traces containing BGP updates messages. Under a worm attack, BGP traces contain large volume of EGP packets. Furthermore, incomplete *update* messages imply that the announced NLRI prefixes are generated from unknown sources. They usually originate from BGP redistribution configurations [117].

Performance of the BGP protocol is based on trust among BGP peers because they assume that the interchanged announcements are accurate and reliable. This trust relationship is vulnerable during BGP anomalies. For example, during BGP hijacks, a BGP peer may announce unauthorized prefixes that indicate to other peers that it is the originating peer. These false announcements propagate across the Internet to other BGP peers and, hence, affect the number of BGP announcements (updates and withdrawals) worldwide.

Table 2.4: Definition of *volume* and *AS-path* features extracted from BGP *update* messages.

Feature	Name	Definition
1	Number of announcements	Routes available for delivery of data
2	Number of withdrawals	Routes no longer reachable
3/4	Number of announced/withdrawn NLRI prefixes	BGP <i>update</i> messages that have type field set to announcement/withdrawal
5/6/7	Average/maximum/average unique <i>AS-path</i> length	Various <i>AS-path</i> lengths
8/10	Number of duplicate announcements/withdrawals	Duplicate BGP <i>update</i> messages with type field set to announcement/withdrawal
9	Number of implicit withdrawals	BGP <i>update</i> messages with type field set to announcement and different <i>AS-path</i> attribute for already announced NLRI prefixes
11/13	Maximum/average edit distance	Maximum/average of edit distances of messages
12	Arrival rate	Average number of messages arrived during one-minute time interval
14–23/24–33	Maximum <i>AS-path</i> length/edit distance	Histograms with the most frequent values of maximum <i>AS-path</i> length/edit distance
34/35/36	Number of IGP, EGP or, incomplete packets	BGP <i>update</i> messages generated by IGP, EGP, or unknown sources
37	Packet size	Average size of received BGP <i>update</i> messages in bytes

This storm of BGP announcements affects the quantity of *volume* features. For example, we have observed that 65 % of the influential features are *volume* features. They proved to be more relevant to the anomaly class than the *AS-path* features, which confirms the known consequence of BGP anomalies on the volume of announcements. Hence, using BGP *volume* features is a feasible approach for detecting BGP anomalies and possible worm attacks in communication networks.

The top selected *AS-path* features appear on the boundaries of the distributions. This indicates that during BGP anomalies, the edit distance and *AS-path* length of the BGP announcements tend to have a very high or a very low value and, hence, large variance. This implies that during an anomaly attack, *AS-path* features are the distribution outliers. For example, approximately 58 % of the *AS-path* features are larger than the distribution mean. Large length of the *AS-path* BGP attribute implies that the packet is routed via a longer path to its destination, which causes large routing delays during BGP anomalies. In a similar case, very short lengths of *AS-path* attributes occur during BGP hijacks when the new (false) originator usually gains a preferred or shorter path to the destination [100].

### 2.2.5 Generation of BGP Training and Test Datasets

BGP datasets containing anomalies and regular data may be extracted from BGP *update* messages collected during periods of Internet anomalies. We download the *update* messages from RIPE [16] and Route Views [17] collection sites. We consider both anomalous (the

days of the attack) and regular (two days prior and two days after the attack) data [27, 30, 31, 119]. The BGP C# tool [120] was used to create various BGP datasets collected during Internet worms, power outages, and ransomware attacks. A previously generated BCNET [121] dataset was collected at the BCNET location in Vancouver, British Columbia, Canada [122, 123]. The dataset contains only regular data points.

**BGP Internet Worms:** We extract 37 numerical features [82] from BGP *update* messages [16] that originated from AS 513 (route collector rrc04). Slammer and Code Red datasets contain 7,200 data points consisting of five days of anomalous and regular data while Nimda contains 8,609 data points. Hence, each data point represents one minute of routing records. Training and test datasets for Slammer, Nimda, and Code Red from RIPE and Route Views are shown in Table 2.6. Note that the BGP Internet Worm datasets collected by Route Views do not contain record of all considered anomalous events because the earliest *update* messages collected by Route Views were from October 2001.

For each collected dataset (Slammer, Nimda, Code Red), we have experimented with a number of training and test data points that are selected from periods of anomalies [31]. We partitioned the datasets by selecting 80 %, 70 %, or 60 % of anomalous data for training and the remaining 20 %, 30 %, or 40 % for testing. Using 60 % of anomalous data for training and 40 % for testing generated the best models. Note that the smaller percentage of anomalous data points for training may be closer to the realistic scenario. Another approach is to use concatenations of two datasets for training while the third dataset is used for testing as shown in Table 2.5. For example, Slammer and Nimda are concatenated in the training phase and used to test Code Red dataset. Experiments indicated that the two approaches did not greatly affect performance of the employed algorithms.

Table 2.5: Training (concatenations of two datasets) and test datasets.

Training dataset	Concatenations of Anomalous Events	Test dataset
1	Slammer and Nimda	Code Red
2	Slammer and Code Red	Nimda
3	Nimda and Code Red	Slammer
4	Slammer, Nimda, and Code Red	RIPE or BCNET

**BGP Power Outages:** The data collected during periods of BGP power outages include five days of Pakistan power outage and five days of Moscow blackout events. Granularity of collected routing records is also 1 min of routing records. The created matrices for Pakistan power outage and Moscow blackout both consist of  $7,200 \times 37$  elements (RIPE). However, there are missing data in Route Views for the Moscow blackout. Training and test datasets for Pakistan power outage and Moscow blackout from RIPE and Route Views are shown in Table 2.7.

**BGP Ransomware Attacks:** We consider BGP *update* messages over eleven days of the WestRock and eight days of the WannaCrypt ransomware attacks. Generated datasets

Table 2.6: BGP Internet worm datasets: Number of data points. Note that Route Views data collection began in 2003.

Collection site	Dataset	Regular (min)	Anomaly (min)	Regular (training)	Anomaly (training)	Regular (test)	Anomaly (test)	Collection date Start	Collection date End
<b>RIPE</b>	Slammer	6,331	869	3,210	530	3,121	339	23.01.2003 00:00:00	27.01.2003 23:59:59
	Nimda	7,308	1,301	3,673	827	3,635	474	16.09.2001 00:00:00	21.09.2001 23:59:59
	Code Red	6,600	600	3,679	361	2,921	239	17.07.2001 00:00:00	21.07.2001 23:59:59
<b>Route Views</b>	Slammer	6,319	869	3,198	530	3,121	339	23.01.2003 00:00:00	27.01.2003 23:59:59

Table 2.7: BGP power outage datasets: Number of data points.

Collection site	Dataset	Regular (min)	Anomaly (min)	Regular (training)	Anomaly (training)	Regular (test)	Anomaly (test)	Collection date Start	Collection date End
<b>RIPE</b>	Pakistan power outage	6,880	320	4,000	200	2,880	120	07.01.2021 00:00:00	11.01.2021 23:59:59
	Moscow blackout	6,960	240	3,120	180	3,840	60	23.05.2005 00:00:00	27.05.2005 23:59:59
<b>Route Views</b>	Pakistan power outage	6,880	320	4,000	200	2,880	120	07.01.2021 00:00:00	11.01.2021 23:59:59
	Moscow blackout	6,865	130	3,075	85	3,790	45	23.05.2005 00:00:00	27.05.2005 23:59:59

Table 2.8: BGP ransomware attack datasets: Number of data points.

Collection site	Dataset	Regular (min)	Anomaly (min)	Regular (training)	Anomaly (training)	Regular (test)	Anomaly (test)	Collection date Start	Collection date End
<b>RIPE/Route Views</b>	WestRock ransomware	5,832	10,008	2,952	6,008	2,880	4,000	21.01.2021 00:00:00	31.01.2021 23:59:59
	WannaCrypt	5,760	5,760	2,880	3,420	2,880	2,340	10.05.2017 00:00:00	17.05.2017 23:59:59

consist of 15,840 (WestRock ransomware) and 11,520 (WannaCrypt) data points as shown in Table 2.8. The duration of the WestRock and WannaCrypt ransomware attacks are 10,008 minutes and 5,760 minutes, respectively.

The patterns in a dataset may be viewed by using 2-dimensional (feature vs. date) or 3-dimensional scattered (considering three BGP features) figures. We use Slammer, Moscow blackout, and WannaCrypt datasets as examples for the data visualization. Examples of features that exhibit visible differences in patterns during regular and anomalous events for the Slammer, Moscow blackout, and WannaCrypt BGP datasets are shown in Fig. 2.2. For both collection sites, the number of BGP announcements and the number of announced NLRI prefixes for Slammer, Moscow blackout, and WannaCrypt increase. However, some features better illustrate the anomalies in RIPE dataset due to missing data in Route Views for Moscow blackout. The selected collectors *rrc04* (RIPE) contains 8 peer ASes and 8 routers while *route-views2* (Route Views) contains 37 peer ASes and 45 peer routers [99]. This larger number of the *update* messages collected by *route-views2* better illustrates the presence of anomalies.

Several features extracted from RIPE and Route Views datasets are visualized in scattered plots shown in Fig. 2.3, Fig. 2.4, and Fig. 2.5. These graphs indicate spatial separation for regular and anomalous classes. Separation into two distinct classes is more visible for Slammer and WannaCrypt in Route Views data while separation of the Moscow blackout data is more prominent in RIPE.

## 2.3 NSL-KDD Dataset

The NSL-KDD [18] dataset is an improved version of the KDD’99 intrusion dataset. Data were captured from an evaluation testbed and included large numbers of virtual hosts and user automata. The KDD’99 dataset [77, 124, 125] was used in various IDSs [1, 78, 126, 127]. NSL-KDD is a randomly selected subset of KDD’99 after redundant data were removed [128] and is a widely used benchmark for evaluating anomaly detection techniques. NSL-KDD dataset captures TCP, UDP, and Internet Control Message Protocol (ICMP) traffic collected using the *tcpdump* utility. It contains four types of intrusion attacks: DoS, U2R, R2L, and Probe described in Table 2.9 [64].

The NSL-KDD dataset contains one training ( $\text{KDDTrain}^+$ ) and two test datasets ( $\text{KDDTest}^+$  and  $\text{KDDTest}^{-21}$ ).  $\text{KDDTest}^{-21}$  is a subset of the  $\text{KDDTest}^+$  dataset that includes records that could not be correctly classified by 21 models [128]. The number of data points is shown in Table 2.10 [18]. Each network connection is represented by 41 features: 38 numerical and 3 categorical (“protocol\_type”, “service”, and “flag”) features. Categorical features are converted to numerical features using the dummy coding method to generate 71 additional features. The features with continuous and discrete types are described in Table 2.11

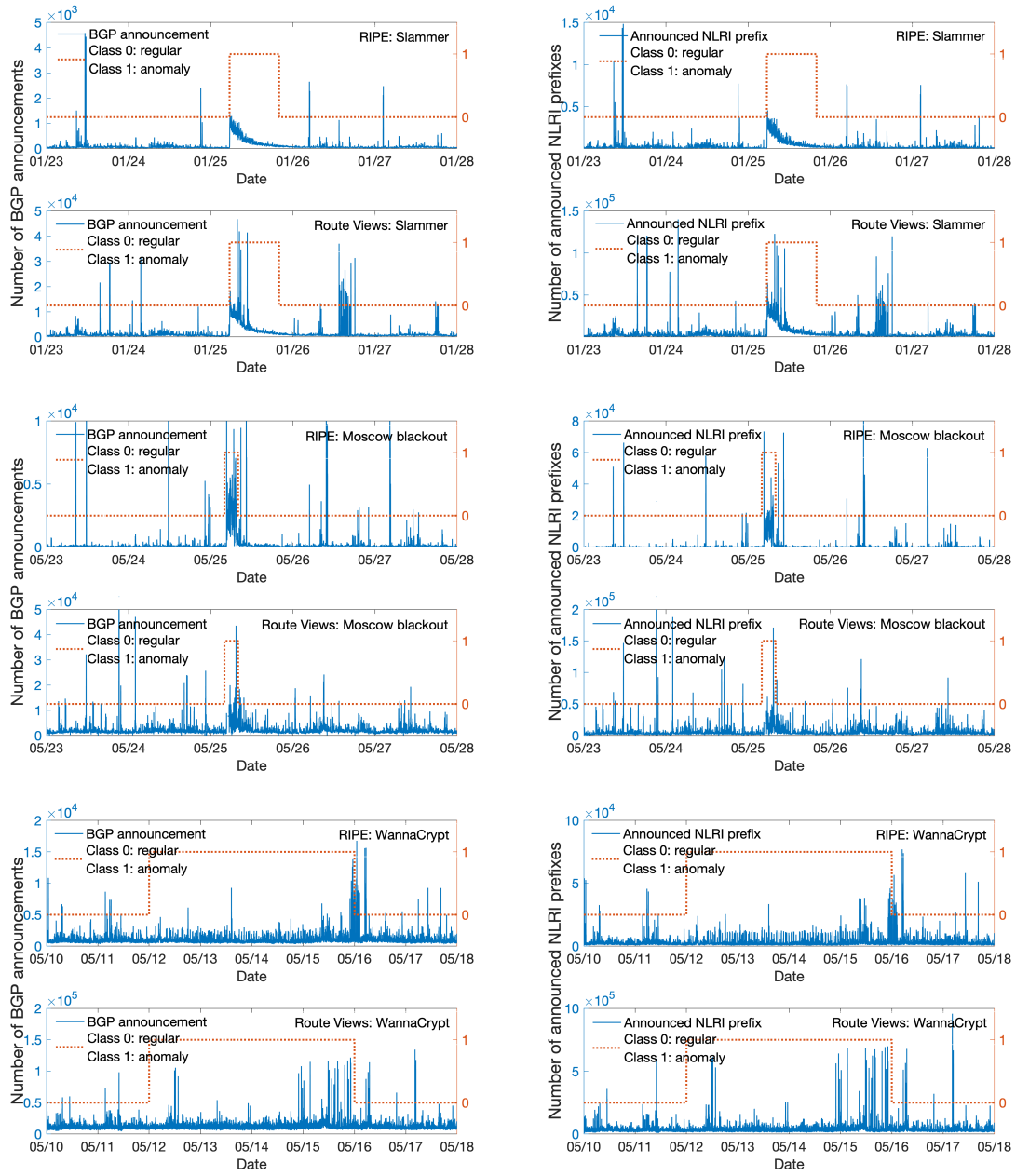


Figure 2.2: Slammer (top), Moscow blackout (middle), and WannaCrypt (bottom): Number of BGP announcements and announced NLRI prefixes. Examples of features that exhibit visible differences in patterns during regular and anomalous events for the Slammer, Moscow blackout, and WannaCrypt BGP datasets.

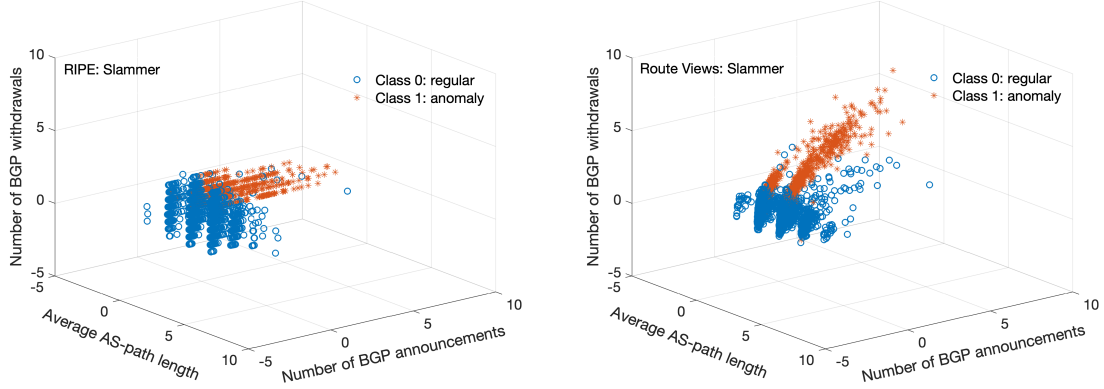


Figure 2.3: Slammer: Average AS-path length vs. number of BGP announcements vs. number of BGP withdrawals. These graphs indicate spatial separation for regular ( $\circ$ ) and anomalous ( $*$ ) classes.

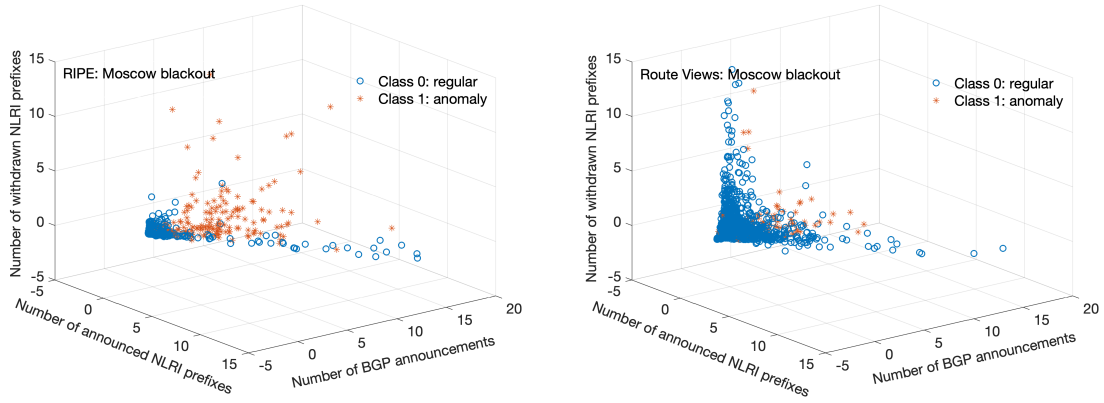


Figure 2.4: Moscow blackout: Number of announced NLRI prefixes vs. number of BGP announcements vs. number of withdrawn NLRI prefixes. These graphs indicate spatial separation for regular ( $\circ$ ) and anomalous ( $*$ ) classes.

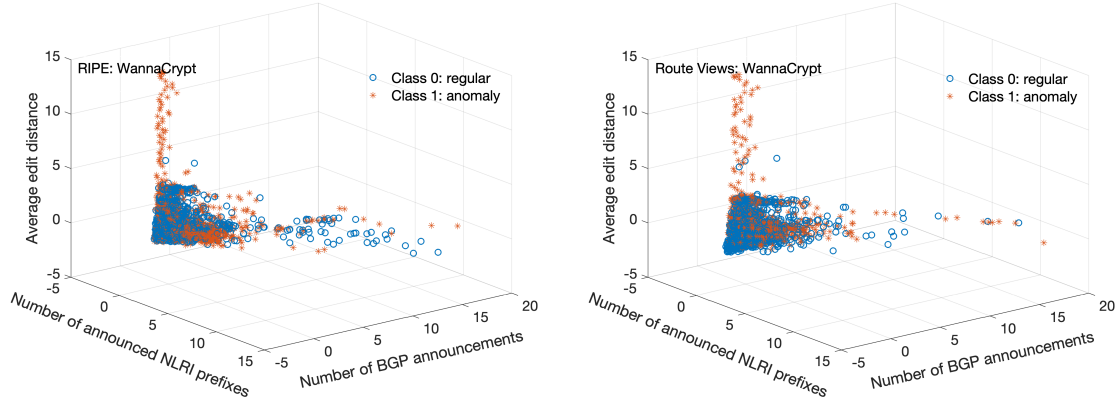


Figure 2.5: WannaCrypt: Number of announced NLRI prefixes vs. number of BGP announcements vs. average edit distance. These graphs indicate spatial separation for regular ( $\circ$ ) and anomalous ( $*$ ) classes.

Table 2.9: NSL-KDD dataset: Four types of intrusion attacks are listed: DoS, U2R, R2L, and Probe.

Type	Intrusion attacks
DoS	back, land, neptune, pod, smurf, teardrop, mailbomb, processtable, udpstorm, apache2, worm
U2R	buffer-overflow, loadmodule, perl, rootkit, sqlattack, xterm, ps
R2L	fpt-write, guess-passwd, imap, multihop, phf, spy, warezmaster, xlock, xsnoop, snmpguess, snmpgetattack, httptunnel, sendmail, named
Probe	ipsweep, nmap, portsweep, satan, mscan, saint

while example of traces are shown in Fig. 2.6. Traces include both anomalous and regular data points.

Table 2.10: NSL-KDD dataset: Number of data points. The NSL-KDD dataset contains one training ( $KDDTrain^+$ ) and two test datasets ( $KDDTest^+$  and  $KDDTest^{-21}$ ).

	Regular	DoS	U2R	R2L	Probe	Total
$KDDTrain^+$	67,343	45,927	52	995	11,656	125,973
$KDDTest^+$	9,711	7,458	200	2,754	2,421	22,544
$KDDTest^{-21}$	2,152	4,342	200	2,754	2,402	11,850

## 2.4 Canadian Institute for Cybersecurity Datasets

The CIC datasets [85] were collected using testbeds that consist of victim and attacker networks. Regular (benign) traffic was generated by implementing B-profiles that replicated the behavior of regular users. M-profiles were used to generate malicious traffic based on common techniques that execute various attacks: botnet, brute force File Transfer Proto-

Table 2.11: NSL-KDD features: Definitions, types, and descriptions [1]. Each network connection is represented by 41 features: 38 numerical and 3 categorical (“protocol\_type”, “service”, and “flag”) features.

Feature	Definition	Type	Description
1	duration	continuous	length (seconds) of the connection
2	protocol_type	discrete	type of the protocol (TCP, UDP)
3	service	discrete	network service on the destination, (HTTP, telnet)
4	flag	discrete	normal or error status of the connection
5	src_bytes	continuous	no. of data bytes from source to destination
6	dst_bytes	continuous	no. of data bytes from destination to source
7	land	discrete	1 if connection is from/to the same host/port; 0 otherwise
8	wrong_fragment	continuous	no. of “wrong” fragments
9	urgent	continuous	no. of urgent packets
10	hot	continuous	no. of “hot” indicators
11	num_failed_logins	continuous	no. of failed login attempts
12	logged_in	discrete	1 if successfully logged in; 0 otherwise
13	num_compromised	continuous	no. of “compromised” conditions
14	root_shell	discrete	1 if root shell is obtained; 0 otherwise
15	su_attempted	discrete	1 if “su root” command attempted; 0 otherwise
16	num_root	continuous	no. of “root” accesses
17	num_file_creations	continuous	number of file creation operations
18	num_shells	continuous	no. of shell prompts
19	num_access_files	continuous	no. of operations on access control files
20	num_outbound_cmds	continuous	no. of outbound commands in an ftp session
21	is_host_login	discrete	1 if the login belongs to the “hot” list; 0 otherwise
22	is_guest_login	discrete	1 if the login is a “guest”login; 0 otherwise
23	count	continuous	no. of connections to the same host as the current connection in the past 2 s
24	srv_count	continuous	no. of connections to the same service as the current connection in the past 2 s
25-26	error_rate and srv_error_rate	continuous	no. of connections that have “SYN” errors
27-28	error_rate and srv_error_rate	continuous	no. of connections that have “REJ” errors
29	same_srv_rate	continuous	no. of connections to the same service
30	diff_srv_rate	continuous	no. of connections to different services
31	srv_diff_host_rate	continuous	no. of connections to different hosts
32	dst_host_count	continuous	no. of connections to the same service as the current connection in the past 2 s
33	dst_host_srv_count	continuous	no. of connections to the same service as the current connection in the past 2 s
34	dst_host_same_srv_rate	continuous	no. of connections to the same service
35	dst_host_diff_srv_rate	continuous	no. of connections to different services
36	dst_host_same_src_port_rate	continuous	no. of connections to the same source port
37	dst_host_srv_diff_host_rate	continuous	no. of connections to different hosts
38-39	dst_host_error_rate and dst_host_srv_error_rate	continuous	no. of connections that have “SYN” errors
40-41	dst_host_error_rate and dst_host_srv_error_rate	continuous	no. of connections that have “REJ” errors

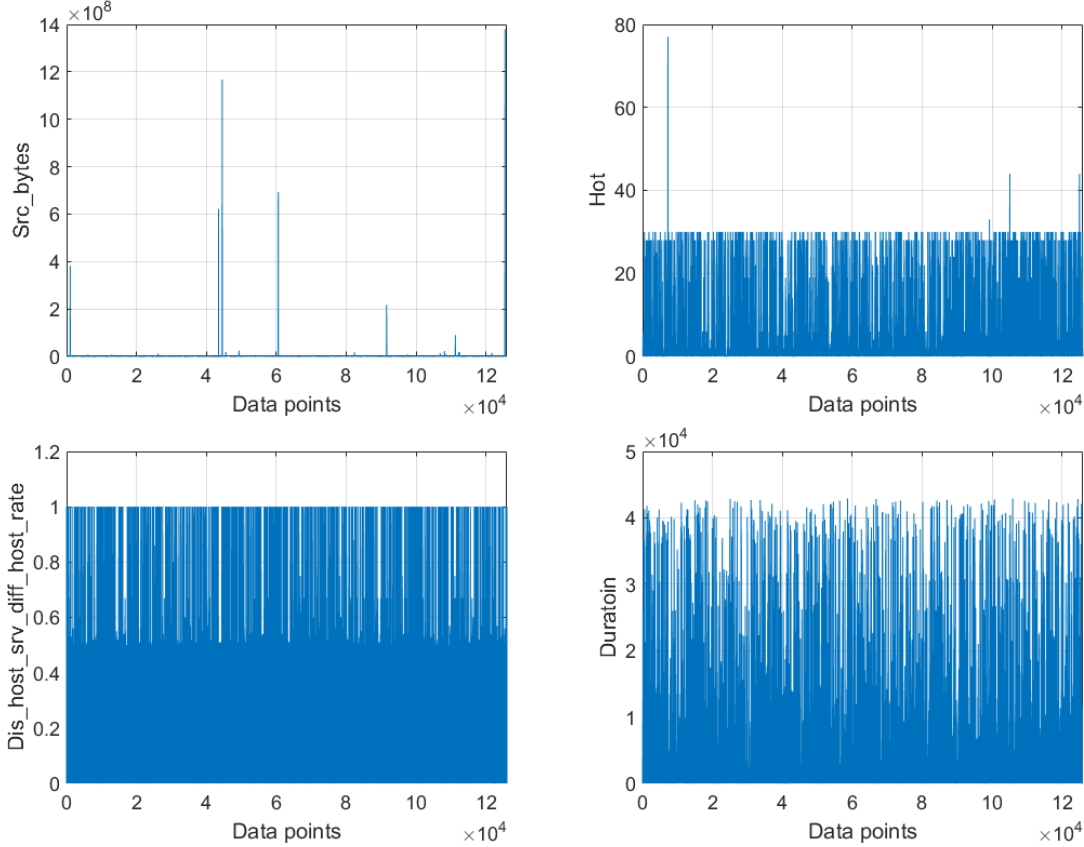


Figure 2.6:  $KDDTrain^+$  dataset: traces of “Src\_bytes” (top left), “Hot” (top right), “Dis\_host\_srv\_diff\_host\_rate” (bottom left), and “Duration” (bottom right) features. Traces include both anomalous and regular data points.

col (FTP) and Secure Shell Protocol (SSH), DoS, DDoS, heartbleed, infiltration, and web attack. Dataset features were extracted from collected TCP and UDP network flows with a network traffic flow analyzer. Each dataset has over 80 features including destination IP and port, protocol type, flow duration, and maximum/minimum packet size. Attacks considered in our experiments are listed in Table 2.12.

The CICIDS2017 [19, 129] dataset was generated using a testbed where the victim network consisted of three servers, one firewall, two switches, and ten terminals while the attacker network had one router, one switch, and four terminals. Network traffic flows were collected over five business days and 84 features were extracted. The testbed used to generate the CSE-CIC-IDS2018 [20, 130] dataset consisted of an attacker-network with 50 terminals and a victim-network with five subnets that included 420 terminals and 30 servers. Network traffic flows were collected over ten business days and 83 features were extracted. The CICDDoS2019 dataset [21, 131] was collected on November 03, 2018 and December 01, 2018. (CICDDoS2019 dataset was actually collected in 2018. The reported capturing dates are January 12 and March 11, 2018 while the PCAP files show the dates are November 03,

Table 2.12: Application-layer DoS and TCP/UDP DDoS attacks.

<b>Dataset</b>	<b>Attack</b>	<b>No. of data points</b>
<b>CICIDS2017</b> July 05, 2017	GoldenEye	10,293
	Hulk	230,124
	SlowHTTPTest	5,499
	Slowloris	5,796
<b>CSE-CIC-IDS2018</b> February 15, 2018	GoldenEye	41,508
	Slowloris	10,990
<b>CICDDoS2019</b> December 01, 2018	Domain Name System	5,071,011
	Lightweight Directory	2,179,930
	Access Protocol	
	Network Time Protocol	1,202,642

2018 and December 01, 2018.) The testbed designed to generate this dataset consisted of a victim network with one server, one firewall, two switches, and four terminals while the attacker network was an external (third-party) enterprise. Captured network traffic flows were analyzed to extract 87 features.

Spatial separations of features selected from the CICIDS2017, CSE-CIC-IDS2018, and CICDDoS2019 datasets are visualized in scattered plots shown in Fig. 2.7. The datasets show visible separation between regular and anomalous classes for various combinations of features. In the case of the CICDDoS2019 dataset, better spatial separation of the two classes captures distinct data patterns leading to better classification performance.

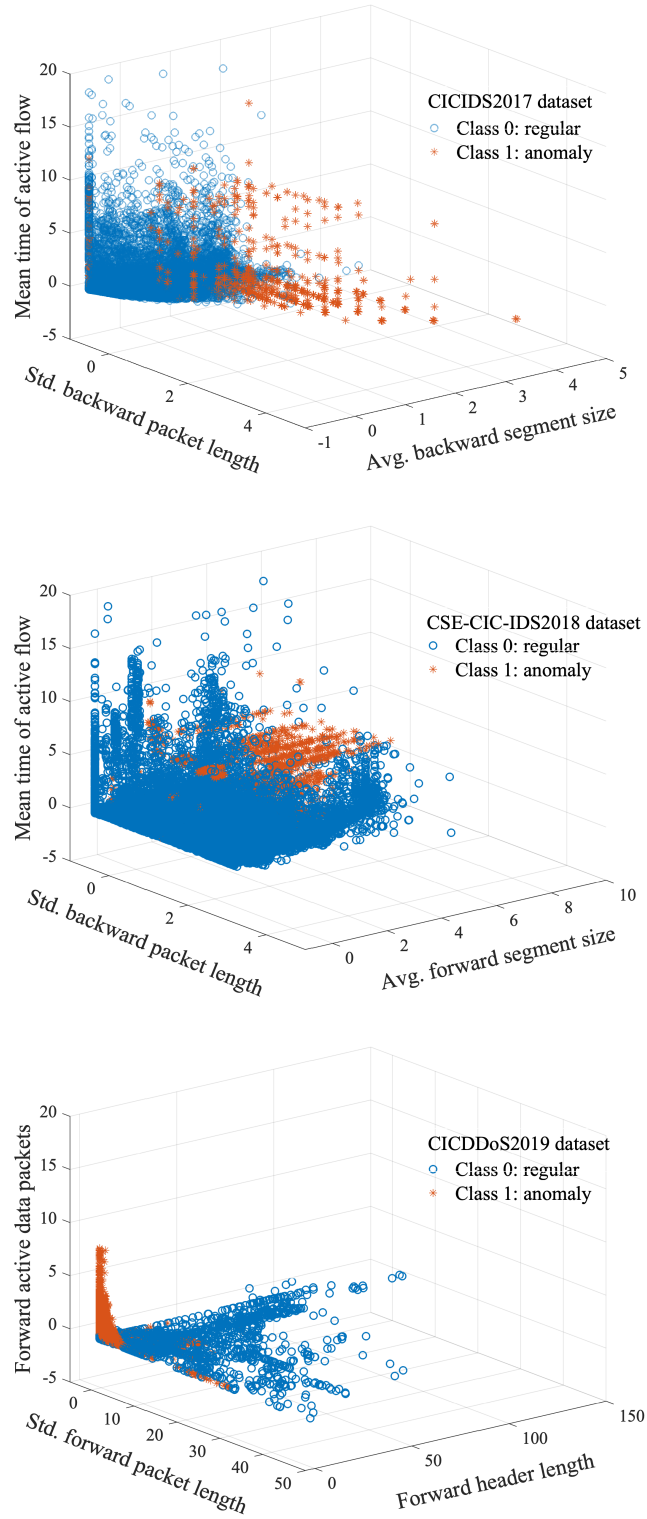


Figure 2.7: Scattered plots of the CICIDS2017 (top), CSE-CIC-IDS2018 (middle), and CICDDoS2019 (bottom). Illustrated are spatial separations of regular (class 0) and anomalous (class 1) data points.

## Chapter 3

# Feature Selection and Dimension Reduction

Using feature selection algorithms to select the most relevant features in the original dataset often improves classification performance. Various feature selection algorithms are used to reduce redundancies by ranking and thus identifying the most relevant features based on their importance. In this Chapter, we describe feature selection and dimension reduction algorithms such as Fisher, mRMR, Mutual Information Base (MIBASE), Odds Ratio (OR), Decision Trees, Extra-Trees, and Autoencoder. Examples illustrating the most relevant features and their scores are presented.

### 3.1 Feature Selection Algorithms

Machine learning models classify data points using a feature matrix. The rows and columns of the matrix correspond to data points and feature values, respectively. Even though machine learning provides general models to classify anomalies, it may easily misclassify test data points due to the redundancy or noise contained in datasets. By providing a sufficient number of relevant features, machine learning approaches help build a generalized model to classify data and lead to smaller number of classification errors [132, 133]. Performance of anomaly classifiers is closely related to feature selection algorithms [134].

Feature selection is used to pre-process data prior to applying machine learning algorithms for classification. Selecting appropriate combination of features is essential for an accurate classification. For example, the 2D scatterings of anomalous and regular classes for Feature 9 (*volume*) vs. Feature 1 (*volume*) and Feature 9 (*volume*) vs. Feature 6 (*AS-path*) are shown in Fig. 3.1 (left) and (right), respectively. The graphs indicate spatial separation of features. While selecting Feature 9 and Feature 1 may lead to a feasible classification based on visible clusters, using only Feature 9 and Feature 6 would lead to poor classification.

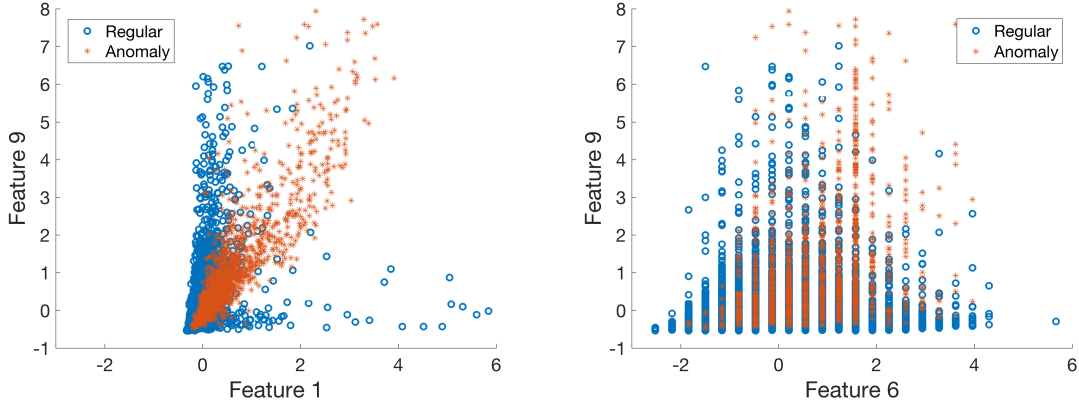


Figure 3.1: Scattered graph of Feature 9 vs. Feature 1 (left) and Feature 9 vs. Feature 6 (right) extracted from the BCNET traffic. Feature values are normalized to have zero mean and unit variance. Shown are two traffic classes: regular ( $\circ$ ) and anomaly (\*).

Feature selection follows a feature extraction process and it is used to decrease dimension of the dataset matrix by selecting a subset of original features to create a new matrix according to certain criteria. The number of features is reduced by removing irrelevant, redundant, and noisy features [23]. Feature selection reduces overfitting by minimizing the redundancies in data, improves modeling accuracy, and decreases training time. It also reduces computational complexity and memory usage. Performance of classification algorithms may also be improved by using pre-selection of features that are most relevant to the classification task. We select the relevant features while dismissing weak and distorted features.

In this Section, we employ Fisher [135–137], mRMR [138] including Mutual Information Difference (MID), Mutual Information Quotient (MIQ), and MIBASE, OR including Extended/Weighted/Multi-Class Odds Ratio (EOR/WOR/MOR), and Class Discriminating Measure (CDM) [139], and Decision Tree [10] algorithms to select relevant features from BGP datasets created by concatenating (a) Slammer and Code Red or (b) Slammer, Nimda, and Code Red. The first dataset consists two classes (anomaly and regular) while the second dataset consists of four classes (Slammer, Nimda, and Code Red, and regular) as shown in Table 2.5. We then give examples of feature importance by using the extra-trees algorithm.

### 3.1.1 Fisher

The Fisher feature selection algorithm [135], [136], [137] computes the score  $\Phi_k$  for the  $k^{th}$  feature as a ratio of inter-class separation and intra-class variance. Features with higher inter-class separation and lower intra-class variance have higher Fisher scores. If there are  $N_a^k$  anomalous samples and  $N_r^k$  regular samples of the  $k^{th}$  feature, the mean values  $m_a^k$  of

anomalous samples and  $m_r^k$  of regular samples are calculated as:

$$m_a^k = \frac{1}{N_a^k} \sum_{i \in \mathbf{a}_k} x_{ik}$$

$$m_r^k = \frac{1}{N_r^k} \sum_{i \in \mathbf{r}_k} x_{ik}, \quad (3.1)$$

where  $\mathbf{a}_k$  and  $\mathbf{r}_k$  are the sets of anomalous and regular samples for the  $k^{th}$  feature, respectively. The Fisher score for the  $k^{th}$  feature is calculated as:

$$\Phi_k = \frac{|(m_a^k)^2 - (m_r^k)^2|}{\frac{1}{N_a^k} \sum_{i \in \mathbf{a}_k} (x_{ik} - m_a^k)^2 + \frac{1}{N_r^k} \sum_{i \in \mathbf{r}_k} (x_{ik} - m_r^k)^2}. \quad (3.2)$$

Examples of features selected using the Fisher algorithm applied to various training datasets are shown in Table 3.1.

Table 3.1: The top ten features selected using the Fisher feature selection algorithm. Two-way classification includes two classes: anomaly and regular. The four-way classification detects the specific type of BGP anomaly (Slammer, Nimda, Code Red) or regular class.

Two-way										
Feature	9	10	8	3	6	11	1	34	36	2
Fisher score	0.2280	0.1665	0.0794	0.0656	0.0614	0.0610	0.0528	0.0526	0.0499	0.0336
Four-way										
Feature	9	10	6	11	8	36	3	37	1	34
Fisher score	0.1259	0.0502	0.0414	0.0409	0.0281	0.0271	0.0240	0.0239	0.0210	0.0203

### 3.1.2 Minimum Redundancy Maximum Relevance

The mRMR algorithm [140], [138] relies on information theory for feature selection. It selects a subset of features that contains more information about the target class while having less pairwise mutual information. A subset of features  $S = \{\mathbf{X}_1, \dots, \mathbf{X}_k, \dots\}$  with  $|S|$  elements has the minimum redundancy if it minimizes:

$$\mathcal{W} = \frac{1}{|S|^2} \sum_{\mathbf{X}_k, \mathbf{X}_l \in S} \mathcal{I}(\mathbf{X}_k, \mathbf{X}_l). \quad (3.3)$$

It has maximum relevance to the classification task if it maximizes:

$$\mathcal{V} = \frac{1}{|S|} \sum_{\mathbf{X}_k \in S} \mathcal{I}(\mathbf{X}_k, \mathbf{C}), \quad (3.4)$$

where  $\mathbf{C}$  is a class vector and  $\mathcal{I}$  denotes the mutual information function calculated as:

$$\mathcal{I}(\mathbf{X}_k, \mathbf{X}_l) = \sum_{k,l} p(\mathbf{X}_k, \mathbf{X}_l) \log \frac{p(\mathbf{X}_k, \mathbf{X}_l)}{p(\mathbf{X}_k)p(\mathbf{X}_l)}. \quad (3.5)$$

The mRMR algorithm offers three variants for feature selection: MID, MIQ, and MIBASE. MID and MIQ select the best features based on  $\max_{S \subset \Omega} [\mathcal{V} - \mathcal{W}]$  and  $\max_{S \subset \Omega} [\mathcal{V}/\mathcal{W}]$ , respectively, where  $\Omega$  is the set of all features.

The top 10 features selected by mRMR from various datasets are shown in Table 3.2 and Table 3.3. They are used for two-way and four-way classifications. Selected features shown in Table 3.2 are generated by using Dataset 2 (Table 2.5) and are intended for two-way classification. Features shown in Table 3.3 are generated by using Dataset 4 (Table 2.5) and are used for four-way classification.

Table 3.2: The top ten features selected using the mRMR feature selection algorithms for two-way classification. Two-way classification distinguishes two classes: anomaly and regular.

mRMR					
MID		MIQ		MIBASE	
Feature	Score	Feature	Score	Feature	Score
34	0.0554	34	0.0554	34	0.0554
10	0.0117	10	0.7527	1	0.0545
20	0.0047	8	0.6583	8	0.0469
25	0.0014	20	0.5014	10	0.0469
24	0.0012	4	0.4937	3	0.0421
23	0.0008	36	0.4095	9	0.0411
4	0.0007	1	0.3720	36	0.0377
8	0.0007	9	0.3260	4	0.0367
22	0.0006	3	0.2824	6	0.0205
21	0.0005	6	0.2809	11	0.0201

### 3.1.3 Odds Ratio

The OR algorithm and its variants perform well when selecting features to be used in binary classification using naive Bayes models. In case of a binary classification with two target classes  $c$  (anomaly) and  $\bar{c}$  (regular), the odds ratio for a feature  $\mathbf{X}_k$  is calculated as:

$$OR(\mathbf{X}_k) = \log \frac{\Pr(\mathbf{X}_k|c)(1 - \Pr(\mathbf{X}_k|\bar{c}))}{\Pr(\mathbf{X}_k|\bar{c})(1 - \Pr(\mathbf{X}_k|c))}, \quad (3.6)$$

where  $\Pr(\mathbf{X}_k|c)$  and  $\Pr(\mathbf{X}_k|\bar{c})$  are the probabilities of feature  $\mathbf{X}_k$  being in classes  $c$  and  $\bar{c}$ , respectively.

Table 3.3: The top ten features selected using the mRMR feature selection algorithms for four-way classification. The four-way classification detects the specific type of BGP anomaly (Slammer, Nimda, Code Red) or regular class.

mRMR					
MID		MIQ		MIBASE	
Feature	Score	Feature	Score	Feature	Score
9	0.0407	9	0.0407	9	0.0407
20	0.0030	34	0.4797	1	0.0308
36	0.0024	36	0.3790	34	0.0305
34	0.0024	10	0.3730	36	0.0305
22	0.0017	5	0.3333	3	0.0234
21	0.0017	8	0.3322	8	0.0225
5	0.0003	1	0.3156	10	0.0200
10	0.0003	6	0.2920	6	0.0179
29	0.0002	37	0.2387	11	0.0177
23	0.0000	3	0.2299	37	0.0175

The extended odds ratio (EOR), weighted odds ratio (WOR), multi-class odds ratio (MOR), and class discriminating measure (CDM) are variants that enable multi-class feature selections in case of  $\gamma = \{c_1, c_2, \dots, c_J\}$  classes:

$$\begin{aligned}
EOR(\mathbf{X}_k) &= \sum_{j=1}^J \log \frac{\Pr(\mathbf{X}_k|c_j)(1 - \Pr(\mathbf{X}_k|\bar{c}_j))}{\Pr(\mathbf{X}_k|\bar{c}_j)(1 - \Pr(\mathbf{X}_k|c_j))} \\
WOR(\mathbf{X}_k) &= \sum_{j=1}^J \Pr(c_j) \times \log \frac{\Pr(\mathbf{X}_k|c_j)(1 - \Pr(\mathbf{X}_k|\bar{c}_j))}{\Pr(\mathbf{X}_k|\bar{c}_j)(1 - \Pr(\mathbf{X}_k|c_j))} \\
MOR(\mathbf{X}_k) &= \sum_{j=1}^J \left| \log \frac{\Pr(\mathbf{X}_k|c_j)(1 - \Pr(\mathbf{X}_k|\bar{c}_j))}{\Pr(\mathbf{X}_k|\bar{c}_j)(1 - \Pr(\mathbf{X}_k|c_j))} \right| \\
CDM(\mathbf{X}_k) &= \sum_{j=1}^J \left| \log \frac{\Pr(\mathbf{X}_k|c_j)}{\Pr(\mathbf{X}_k|\bar{c}_j)} \right|, \tag{3.7}
\end{aligned}$$

where  $\Pr(\mathbf{X}_k|c_j)$  is the conditional probability of  $\mathbf{X}_k$  given the class  $c_j$  and  $\Pr(c_j)$  is the probability of occurrence of the class  $j$ . The OR algorithm is extended by calculating  $\Pr(\mathbf{X}_k|c_j)$  for continuous features. If the sample points are independent and identically distributed, (3.6) is written as:

$$OR(\mathbf{X}_k) = \sum_{i=1}^{|\mathbf{X}_k|} \log \frac{\Pr(X_{ik} = x_{ik}|c)(1 - \Pr(X_{ik} = x_{ik}|\bar{c}))}{\Pr(X_{ik} = x_{ik}|\bar{c})(1 - \Pr(X_{ik} = x_{ik}|c))}, \tag{3.8}$$

where  $|\mathbf{X}_k|$  denote the size of the  $k^{th}$  feature vector,  $X_{ik}$  is the  $i^{th}$  element of the  $k^{th}$  feature vector, and  $x_{ik}$  is the realization of the random variable  $X_{ik}$ . Other variants of the OR feature selection algorithm are extended to continuous cases in a similar manner. The top

ten selected features used for two-way and four-way classifications are shown in Table 3.4 and Table 3.5, respectively. Note that negative feature scores are obtained using OR and EOR due to the fact that the probability of having the anomalous data points is smaller than the probability of regular data points. Therefore,  $\Pr(\mathbf{X}_k|c_j)$  and  $\Pr(X_{ik} = x_{ik}|c)$  are smaller than  $\Pr(\mathbf{X}_k|\bar{c}_j)$  (3.7) and  $\Pr(X_{ik} = x_{ik}|\bar{c})$  (3.8), respectively.

Table 3.4: The top ten features selected using the OR feature selection algorithms for two-way classification. Two-way classification distinguishes two classes: anomaly and regular.

Odds Ratio variants							
OR		WOR		MOR		CDM	
Feature	Score $\times 10^4$	Feature	Score $\times 10^4$	Feature	Score $\times 10^5$	Feature	Score $\times 10^5$
13	-2.7046	12	3.9676	12	1.0789	12	1.0713
7	-2.8051	1	3.4121	34	0.9214	34	0.9199
5	-2.8064	34	3.4095	1	0.9213	1	0.9198
29	-2.8774	3	3.3482	3	0.8908	3	0.8885
15	-2.8777	4	3.3468	4	0.8775	4	0.8702
28	-2.9136	23	2.9348	9	0.7406	9	0.7224
14	-2.9137	24	2.7628	36	0.7264	36	0.7201
6	-2.9190	22	2.7051	37	0.7229	37	0.7192
11	-2.9248	21	2.6662	23	0.7208	2	0.7145
30	-2.9288	20	2.5821	2	0.6782	8	0.6624

Table 3.5: The top ten features selected using the OR feature selection algorithms for four-way classification. The four-way classification detects the specific type of BGP anomaly (Slammer, Nimda, Code Red) or regular class.

Odds Ratio variants							
EOR		WOR		MOR		CDM	
Feature	Score $\times 10^5$	Feature	Score $\times 10^4$	Feature	Score $\times 10^5$	Feature	Score $\times 10^5$
3	-1.5496	12	1.7791	12	3.0894	12	3.0700
13	-1.5681	1	1.3293	34	2.5964	34	2.5924
9	-1.6063	34	1.3273	1	2.5723	1	2.5688
11	-1.6184	4	1.1140	4	2.5252	4	2.5190
6	-1.6184	36	1.0763	36	2.4617	36	2.4422
37	-1.6191	2	1.0140	2	2.3024	2	2.2300
5	-1.6499	23	0.8669	23	2.2832	8	2.2068
7	-1.6522	24	0.8529	24	2.2733	9	2.1357
29	-1.6783	21	0.8508	21	2.2696	10	2.1168
15	-1.6784	20	0.8504	22	2.2696	3	2.0848

### 3.1.4 Decision Tree

The decision tree approach is commonly used in data mining to predict the class label based on several input variables. A classification tree is a directed tree where the root is the source sample set and each internal (non-leaf) node is labeled with an input feature. The tree branches are prediction outcomes that are labeled with possible feature values while

each leaf node is labeled with a class or a class probability distribution [141]. A top-down approach is commonly used for constructing decision trees. At each step, an appropriate variable is chosen to best split the set of items. A quality measure is the homogeneity of the target variable within subsets and it is applied to each candidate subset. The combined results measure the split quality [142], [143].

The C5 [144] software tool is used to generate decision tree for both feature selections and anomaly classifications. The C5 decision tree algorithm relies on the information gain measure. The continuous attribute values are discretized and the most important features are iteratively used to split the sample space until a certain portion of samples associated with the leaf node has the same value as the target attribute. For each training dataset, a set of rules used for classification is extracted from the constructed decision tree.

The selected features are shown in Table 3.6. Based on the outcome of the decision tree algorithm, some features are removed in the constructed trees. Fewer features are selected either based on the number of leaf nodes with the largest correct classified samples or based on the number of rules with maximum sample coverage. The features that appear in the selected rules are considered to be important and, therefore, are preserved.

Table 3.6: Selected features using the decision tree algorithm for two-way classification. Two-way classification distinguishes two classes: anomaly and regular.

Training dataset	Selected features
Slammer and Nimda	1–21, 23–29, 34–37
Slammer and Code Red	1–22, 24–29, 34–37
Nimda and Code Red	1–29, 34–37

### 3.1.5 Extra-Trees

We use extra-trees algorithm [145] for feature selection in various experiments. It is a variant of random forests and is used to rank features based on *Gini importance*. Additional randomness is introduced by randomly splitting nodes instead of finding the best split using random forests. The Gini importance is used to compute feature scores in a given dataset [146]:

$$Importance(\mathbf{X}_c) = \frac{1}{N_T} \sum_T \sum_{t \in T: v(s_t) = \mathbf{X}_c} p(t) \Delta i(s_t, t), \quad (3.9)$$

where  $\mathbf{X}_c$  is the subset of input data  $\mathbf{X}$  corresponding to one feature,  $N_T$  is the number of trees,  $t$  is the index of a node in a tree,  $s_t$  is the direction of the split,  $v(s_t)$  is a randomly generated threshold,  $p(t)$  is the weight, and  $\Delta i(s_t, t)$  is the decrease of the node impurity equivalent to its importance.

The examples of feature importance that are calculated using CICIDS2017 and CSE-CIC-IDS2018 datasets are presented. The sixteen most relevant features and their importance are shown in Table 3.7 and Fig. 3.2.

Table 3.7: Sixteen most relevant features and their importance based on the extra-trees algorithm. The Gini importance is used to compute feature scores using CICIDS2017 and CSE-CIC-IDS2018 datasets.

<b>CICIDS2017</b>	
1. Dst Port (0.0595)	2. Pkt Size Avg (0.0464)
3. Bwd Pkt Len Mean (0.0462)	4. Flow IAT Max (0.0440)
5. Protocol (0.0437)	6. Pkt Len Std (0.0423)
7. Bwd Seg Size Avg (0.0418)	8. ACK Flag Cnt (0.0416)
9. Fwd IAT Max (0.0403)	10. Fwd IAT Std (0.0345)
11. Fwd Seg Size Min (0.0329)	12. Idle Max (0.0319)
13. Fwd Pkts/s (0.0296)	14. Bwd Pkt Len Max (0.0287)
15. Idle Mean (0.0264)	16. Bwd Pkt Len Std (0.0256)
<b>CSE-CIC-IDS2018</b>	
1. Fwd Seg Size Min (0.2904)	2. Init Fwd Win Byts (0.1082)
3. Bwd Pkt Len Std (0.0474)	4. Bwd Pkt Len Max (0.0306)
5. Pkt Len Max (0.0264)	6. Flow IAT Min (0.0254)
7. Flow Duration (0.0248)	8. Fwd IAT Tot (0.0218)
9. Fwd IAT Min (0.0206)	10. Bwd Pkt Len Mean (0.0197)
11. ACK Flag Cnt (0.0190)	12. Init Bwd Win Byts (0.0190)
13. Fwd IAT Max (0.0185)	14. PSH Flag Cnt (0.0167)
15. Dst Port (0.0164)	16. Flow IAT Max (0.0162)

## 3.2 Dimension Reduction Algorithms

Unsupervised dimension reduction belongs to unsupervised learning that trains a model with unlabelled input data. Its goal is to transform original data into the lower dimensional data that preserve characteristics of the original data. The transformed data is used as input to the supervised machine learning algorithms [14]. Dimension reduction may also facilitate visualization of the high dimensional data. Principal Component Analysis (PCA) is commonly used for data compression as one of the dimension reduction techniques. However, PCA may not perform well when the features of the input data have nonlinear relationship. Hence, using autoencoders may be preferable because it can be used for both linear and nonlinear transformations of high to low dimensional data [11]. An autoencoder with a single fully-connected hidden layer, the linear activation function, and the mean squared error (error function) is equivalent to PCA. The description of the autoencoder and the comparison between input and the reconstructed data follows.

### 3.2.1 Autoencoders

Autoencoders [147] are unsupervised neural networks used to learn a representation from a given dataset [14]. They are commonly used for dimension reduction. They consist of an encoder and a decoder for data compression and reconstruction, receptively. The encoder and decoder consist of the same type of networks and have the same number of layers and

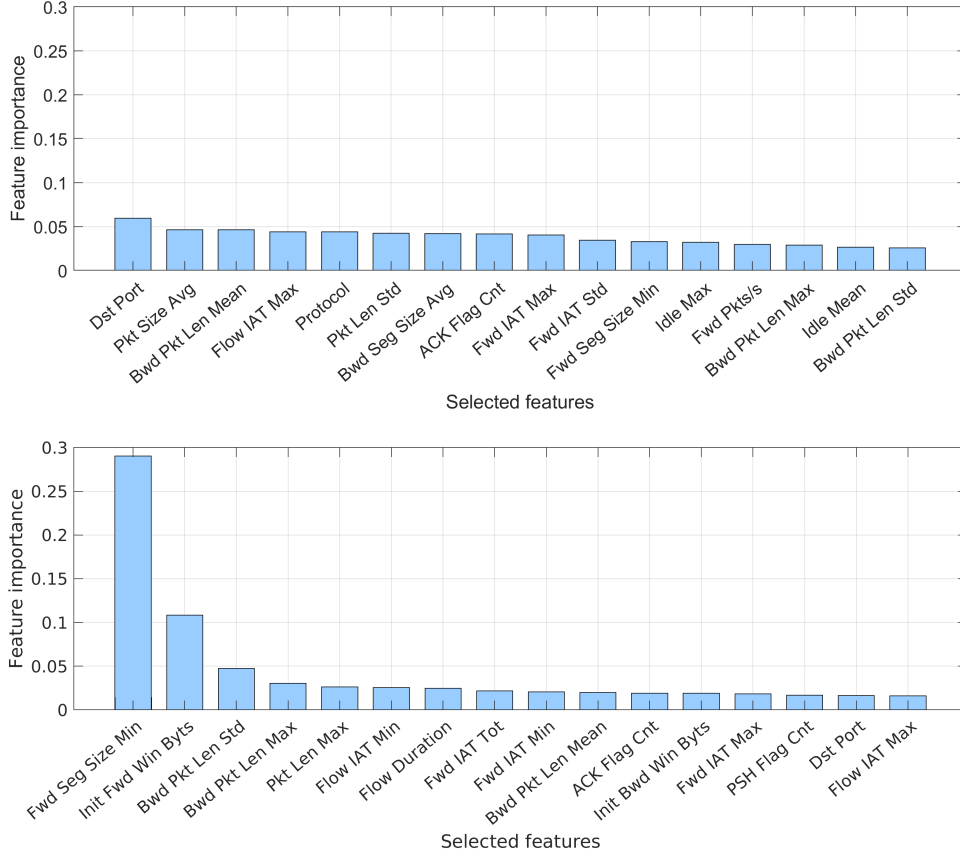


Figure 3.2: Sixteen most relevant features and their importance based on the extra-trees algorithm: CICIDS2017 (top) and CSE-CIC-IDS2018 (bottom) datasets. Note that not all features are equally important. The top two features (“Minimum forward segment size” and “initial forward window bytes”) in the CSE-CIC-IDS2018 dataset exhibit higher feature scores than those in the CICIDS2017 dataset.

nodes. The central layer in the autoencoder is the code layer that belongs to both encoder and decoder and contains the smallest number of hidden nodes. The encoder is used to map the higher dimensional input data to lower dimensional data (latent vectors) in the code layer. The output data are reconstructed from the code layer through a decoder. The reconstruction error is introduced during the training to minimize the discrepancy between the input and output data. An autoencoder structure is shown in Fig. 3.3.

The encoder and decoder are represented as [14]:

$$\mathbf{h}_i = f(\mathbf{x}_i \mathbf{W}_i + \beta_i), \quad i = 1, 2, \dots, n, \quad (3.10)$$

$$\mathbf{x}'_i = g(\mathbf{h}_i \mathbf{W}_r + \beta_r), \quad r = 1, 2, \dots, n, \quad (3.11)$$

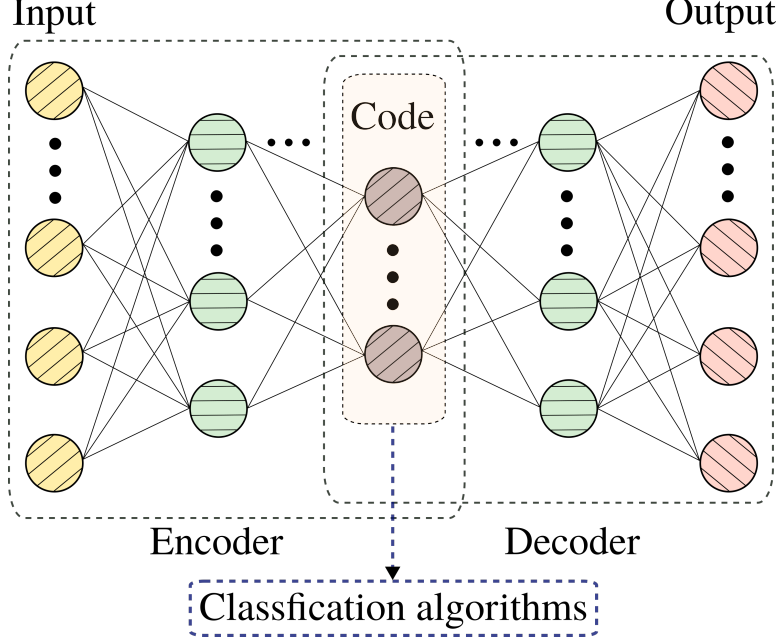


Figure 3.3: Structure of an autoencoder with encoder and decoder modules. The encoder and decoder consist of the same type of networks and have the same number of layers and nodes. The central layer in the autoencoder is the code layer that belongs to both encoder and decoder and contains the smallest number of hidden nodes. The encoder is used to map the higher dimensional input data to lower dimensional data (latent vectors) in the code layer. The output data are reconstructed from the code layer through a decoder.

where  $f(\cdot)$  and  $g(\cdot)$  denote encoder and decoder functions, respectively,  $\mathbf{W}_i$  and  $\mathbf{W}_r$  are weights,  $\beta_i$  and  $\beta_r$  are bias parameters, and  $n$  denotes the number of data points. For each data point  $i$ ,  $\mathbf{x}_i$ ,  $\mathbf{h}_i$ , and  $\mathbf{x}'_i$  represent the input data, latent vector, and reconstructed data, respectively.

The loss function  $\mathcal{L}(\cdot)$  is used to minimize the reconstruction error during training:

$$\mathcal{L}(\mathbf{x}_i, \mathbf{x}'_i), \quad i = 1, 2, \dots, n. \quad (3.12)$$

A high-level training process using an autoencoder is shown in Fig. 3.4. Each element of the grid represents a numerical value corresponding to an instantaneous feature value. We use autoencoder with the symmetric structure consisting of the encoder and decoder. The input layer, encoder's hidden layer, code layer, decoder's hidden layer, and output layer consist of 37, 16, 3, 16, and 37 nodes, respectively. ReLU is used as the activation function in encoder's and decoder's hidden layers. Three features (number of announcements, maximum *AS-path* length, and packet size) are used to illustrate the input data and reconstructed input data. The similarity between input data and reconstructed input data reflects the quality of the compressed representation generated by the code layer. This representation is the abstraction of the original data. It is generated by extracting the most

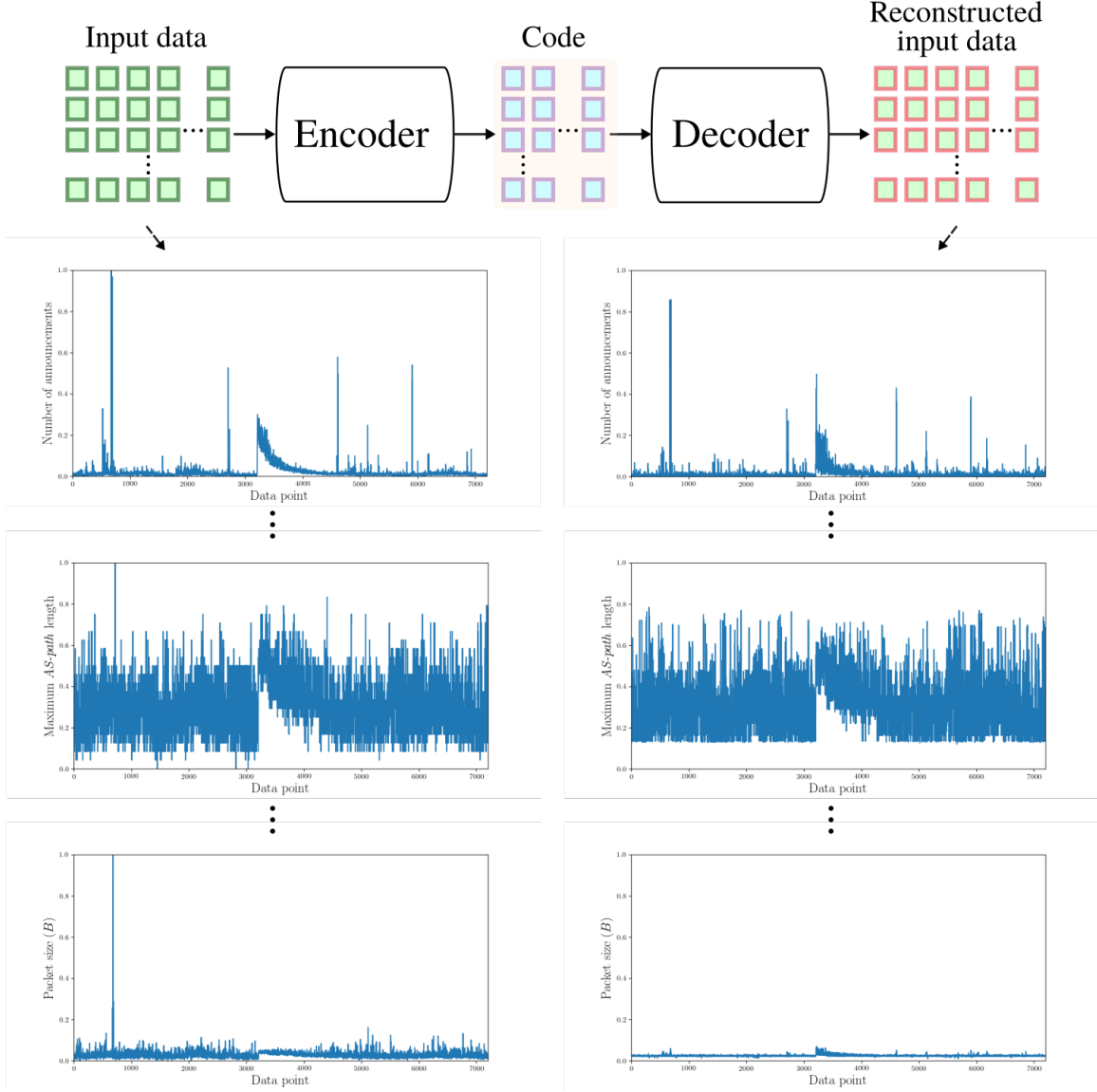


Figure 3.4: An autoencoder implementation using Slammer dataset: mapping input data to the code and decoding (reconstructing) input data. The dimension (size) of the input and output data is  $7,200 \times 37$ . Shown are three BGP features before (left) and after (right) reconstruction.

relevant features from the autoencoder. By employing the activation functions, the autoencoder often generates a better quality of the compressed representation than PCA when applied to data having nonlinearly separable features [147, 148]. The compressed representation is of a lower dimension and as such also reduces the training time when used as input to train a classifier. The compressed data generated by PCA and an autoencoder are visualized in scattered plots shown in Fig. 3.5. Both plots exhibit visible spatial separation between regular ( $\circ$ ) and anomalous ( $*$ ) classes. The quality of the separations is evaluated using silhouette coefficients. The silhouette coefficients for the clusters generated by PCA

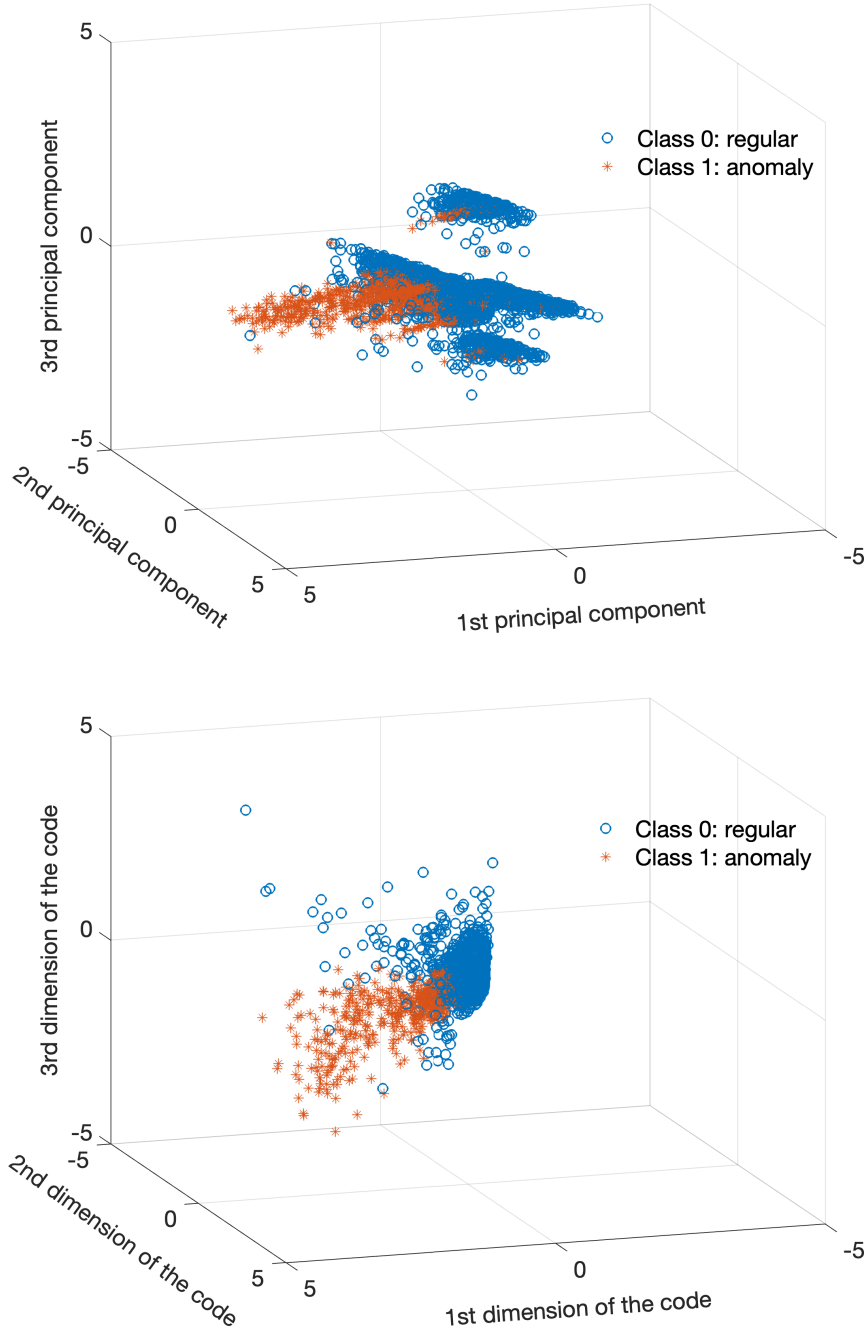


Figure 3.5: Comparison between PCA and autoencoder used for dimension reduction of the Slammer dataset: The first three principal components generated by PCA (top) and the compressed data generated by the autoencoder model (bottom) described in Fig. 3.4. The silhouette coefficients for the clusters generated by PCA and the autoencoder are 0.36 and 0.67, respectively. The higher value of the silhouette coefficient indicates that the autoencoder generates better spatial separation thus leading to better classification performance.

and the autoencoder are 0.36 and 0.67, respectively. The higher value of the silhouette coefficient indicates that the autoencoder generates better spatial separation thus leading to better classification performance. CNN and RNN networks may be used in the autoencoder to effectively reduce dimensions of image and time series data, respectively.

Variants of the autoencoder such as deep, sparse, denoising, and variational autoencoders have been used for feature learning and for generative modeling. We use the deep autoencoder with various LSTM/GRU hidden layers for dimension reduction. After training the autoencoder, we remove the decoder and only retain the encoder. The output of the encoder is directly used as the input to a subsequent gradient boosting model. Experiments with using autoencoder for dimension reduction are described in Chapter 6.

### 3.3 Discussion

Various feature selection and dimension reduction algorithms are used to select a subset of features important for classification. After the feature selection process, extracted features are used as input to machine learning classification algorithms. Feature selection and dimension reduction are related to supervised and unsupervised learning, respectively. Feature selection algorithms are useful in the case of large labeled training data. Dimension reduction algorithms are often used in the case of large unlabeled training data. Labeling data, usually performed by human experts, is time-consuming. In our study, we often use labeled datasets and employ feature selection algorithms to select the most relevant features based on feature scores. The role of feature selection algorithms is to remove redundant features and create subsets from the original data. In contrast, dimension reduction algorithms are used to transform the original data into compressed data. Therefore, the features of the compressed data are not necessarily related to features from the original data.

## Chapter 4

# Machine Learning Algorithms

Various machine learning algorithms have been used to perform classification. Classification aims to identify various classes in a dataset. Each element in the classification domain is called a class. A classifier labels the data points as either an anomalous or a regular event. We consider datasets of known network anomalies and test the classifiers' ability to reliably identify anomalous class, which usually contains fewer samples than the regular class in training and test datasets. Classifier models are usually trained using datasets containing limited number of anomalies and are then applied on a test dataset. Performance of a classification model depends on a model's ability to correctly predict classes. Classifiers are evaluated based on various metrics such as accuracy, F-Score, precision, and sensitivity.

Most classification algorithms minimize the number of incorrectly predicted class labels while ignoring the difference between types of misclassified labels by assuming that all misclassifications have equal costs. The assumption that all misclassification types are equally costly is inaccurate in many application domains. In the case of network anomaly detection, incorrectly classifying an anomalous sample may be more costly than the incorrect classification of a regular sample.

In this Chapter, we present the overview of machine learning algorithms including traditional, deep learning, and fast machine learning algorithms used for classification tasks. We then give evaluations of SVM, naïve Bayes, hidden Markov model, decision tree, and ELM algorithms using BGP datasets. These early experimental results reflect the limitations of employing traditional machine learning algorithms.

### 4.1 Overview of Machine Learning Algorithms

Supervised, unsupervised, and semi-supervised machine learning techniques have been heavily used to detect network anomalies. Most approaches rely on robust supervised learning algorithms, which train models using the given labels. Traditional supervised machine learning algorithms such as SVM and naïve Bayes may achieve desired performance using smaller datasets but requiring longer training time. Deep learning algorithms such as CNNs, RNNs,

and Bi-RNNs are suitable to design more generalized models for larger datasets due to the number of hidden layers and the back-propagation algorithm. Fast machine learning algorithms such as BLS and Gradient Boosting Machines (GBMs) have also been successful in analyzing large datasets albeit in shorter training time.

Among various machine learning approaches, SVM is often used due to its robust nature [3]. It provides better generalization of features by defining the decision boundary to geometrically lie midway between the support vectors. Support vectors are the data points that lie closest to the decision surface. SVM deals with both linearly and nonlinearly separable features of the input dataset by using kernels. Polynomial, Gaussian Radial Basis Function (RBF), and sigmoid kernels may be employed.

Deep neural networks such as RNNs, including Bi-RNNs [149,150], LSTM [151,152], and GRU [153] are trained to identify important features in the input data by adjusting weights in each iteration. Their significant advantage over logistic regression, naïve Bayes, SVM, and decision tree algorithms is in using back-propagation to calculate gradients and update the weights. Furthermore, deep neural networks may achieve desired classification results by adjusting the number of hidden nodes and layers, activation functions, and optimization algorithms. They employ linear or nonlinear activation functions such as rectified linear unit (ReLU), logistic sigmoid, or hyperbolic tangent (*tanh*). The numbers of hidden nodes and layers are chosen depending on the size of the dataset. An important advantage of RNNs is their ability to use contextual information between input and output sequences. The Bi-RNNs [149,150] were proposed to enhance handwriting and speech recognition. Bi-RNNs utilize both forward and backward information for prediction: an RNN layer calculates the output by using the sequential input data stream starting from its beginning while another RNN layer uses the sequential input backward starting from its end. The outputs of the two RNN layers are then combined. The LSTM neural network was proposed to address certain deficiencies in RNNs. The GRU structure is a special case of LSTM with a simpler structure. RNNs, CNNs, deep belief networks, and autoencoders offer promising results for anomaly detection [154].

BLS [4] and its extensions [15,33–36,155] are alternatives to deep learning networks. They achieve comparable classification accuracy and require a considerably shorter time for training than the conventional deep learning networks because of a small number of hidden layers. They also use pseudo-inverse or ridge regression approaches rather than back-propagation during the training process. BLS offers desired classification when used for function approximation, time series forecast, and image recognition. Generating a set of mapped features is an important step in BLS and its extensions. Various modifications for creating the set of mapped features have been proposed to improve the generalization of the BLS-based models such as CFBLs, CCFBLs [15], and multi-kernel BLS [36].

GBDT is a variant of the GBM and employs decision trees as estimators. The goal of the GBDT models is to minimize an objective loss function. Optimized GBDT algorithms include XGBoost [156], LightGBM [12], and CatBoost [157].

#### 4.1.1 Performance Metrics

The confusion matrix shown in Table 4.1 is used to evaluate performance of classification algorithms. TP and FN are the number of anomalous test data points that are classified as anomaly and regular, respectively. FP and TN are the number of regular test data points that are classified as anomaly and regular, respectively.

Variety of performance measures are calculated to evaluate classification algorithms, such as accuracy and F-Score:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (4.1)$$

$$\text{F-Score} = 2 \times \frac{\text{precision} \times \text{sensitivity}}{\text{precision} + \text{sensitivity}}, \quad (4.2)$$

where

$$\text{precision} = \frac{TP}{TP + FP} \quad (4.3)$$

$$\text{sensitivity (recall)} = \frac{TP}{TP + FN}. \quad (4.4)$$

Table 4.1: Confusion matrix.

<b>Actual class</b>	<b>Predicted class</b>	
	Anomaly (positive)	Regular (negative)
Anomaly (positive)	TP	FN
Regular (negative)	FP	TN

As a performance measure, accuracy reflects the true prediction over the entire dataset. It is commonly used in evaluating the classification performance. Accuracy assumes equal cost for misclassification and relatively uniform distributions of classes. It treats the regular data points to be as important as the anomalous points. Hence, it may be an inadequate measure when comparing performance of classifiers [158] and misleading in the case of unbalanced datasets. The F-Score, which considers the false predictions, is important for anomaly detection because it is a harmonic mean of the precision and sensitivity, which measure the discriminating ability of the classifier to identify classified and misclassified anomalies. Precision identifies true anomalies among all data points that are correctly classified as anomalies. Sensitivity measures the ability of the model to identify correctly predicted anomalies.

As an example, consider a dataset that contains 900 regular and 100 anomalous data points. If a classifier identifies these 1,000 data points as regular, its accuracy is 90%, which seems high at first glance. However, no anomalous data point is correctly classified and, hence, the F-Score is zero. Therefore, the F-Score is often used to compare performance of classification models. It reflects the success of detecting anomalies rather than detecting either anomalies or regular data points. In our study, we use both Accuracy and F-Score to compare various classification algorithms.

## 4.2 Support Vector Machine

SVM is a supervised learning algorithm used for classification and regression tasks. Given a set of labeled training samples, the SVM algorithm learns a classification hyperplane (decision boundary) by maximizing the minimum distance between data points belonging to various classes. There are two types of SVM models: hard-margin and soft-margin [159]. The hard-margin SVMs require that each data point is correctly classified while the soft-margin SVMs allow some data points to be misclassified. The hyperplane is acquired by minimizing the loss function [3]:

$$C \sum_{n=1}^N \zeta_n + \frac{1}{2} \|w\|^2, \quad (4.5)$$

with constraints:  $t_n y(x_n) \geq 1 - \zeta_n$ ,  $n = 1, \dots, N$ . The regularization parameter  $C$  controls the trade-off between the slack variable  $\zeta_n$ ,  $N$  is the number of data points, and  $\frac{1}{2} \|w\|^2$  is the margin.

The regularization parameter  $C > 0$  is used to avoid over-fitting. The target value is denoted by  $t_n$  while  $y(x_n)$  and  $x_n$  are the training model and data points, respectively. The SVM solves a loss function as an optimization problem (4.5).

An illustration of the soft margin is shown in Fig. 4.1 [3]. The solid line indicates the decision boundary while dashed lines indicate the margins. Encircled data points are support vectors. The maximum margin is the perpendicular distance between the decision boundary and the closest support vectors. Data points for which  $\zeta = 0$  are correctly classified and are either on the margin or on the correct side of the margin. Data points for which  $0 \leq \zeta < 1$  are also correctly classified because they lie inside the margin and on the correct side of the decision boundary. Data points for which  $\zeta > 1$  lie on the wrong side of the decision boundary and are misclassified. The outputs 1 and -1 correspond to anomalous and regular data points, respectively. The SVM solution maximizes the margin between the data points and the decision boundary. Data points that have the minimum distance to the decision boundary are called support vectors.

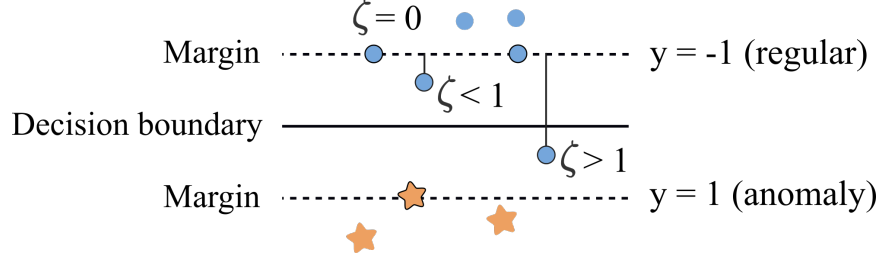


Figure 4.1: Illustration of the soft margin SVM [3]. Shown are correctly and incorrectly classified data points. Regular and anomalous data points are denoted by circles and stars, respectively. The circled points are support vectors. Data points for which  $\zeta = 0$  are correctly classified and are either on the margin or on the correct side of the margin. Data points for which  $0 \leq \zeta < 1$  are also correctly classified because they lie inside the margin and on the correct side of the decision boundary. Data points for which  $\zeta > 1$  lie on the wrong side of the decision boundary and are misclassified. The outputs 1 and -1 correspond to anomalous and regular data points, respectively.

The SVM employs a kernel function to compute a nonlinear separable function to map the feature space into a linear space. The RBF was chosen because it creates a large function space and outperforms other types of SVM kernels [3]. The RBF kernel  $k$  is used to avoid the high dimension of the feature matrix:

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2 / 2\sigma^2). \quad (4.6)$$

It depends on the Euclidean distance between  $\mathbf{x}$  and  $\mathbf{x}'$  feature vectors. The datasets are trained using 10-fold cross validation to select parameters  $(C, 1/2\sigma^2)$  leading to the best accuracy.

#### 4.2.1 Experiments and Performance Evaluation

The SVM algorithm is applied to datasets listed in Table 2.5. The parameter  $C$  (4.5) controls the trade-off between the training error and the margin as well as the cost factor [160].

Feature selection algorithms were implemented in MATLAB and were used to minimize the dimension of the dataset matrix by selecting the 10 most relevant features. We used: Fisher, mRMR (MID, MIQ, and MIBASE), OR, EOR/WOR/MOR, and CDM. Hence, the dimension of feature matrices that correspond to a five-day period of collected data is  $7,200 \times 10$ . Each matrix row corresponds to the top ten selected features within the one-minute interval. In two-way classifications, we target two classes: anomaly (positive) and regular (negative) for each training dataset. While the two-way classification only identifies whether or not a data point is anomalous, the four-way classification detects the specific type of BGP anomaly: Slammer, Nimda, Code Red, or regular (RIPE and BCNET). SVM classifies each data point  $x_n$ ,  $n = 1, \dots, 7,200$ , with a training target class  $t_n$  either as anomaly  $y = 1$  or regular  $y = -1$ .

In a two-way classification, all anomalies are treated as one class. Validity of the proposed models is tested by applying two-way SVM classification on BGP traffic traces collected from RIPE and BCNET on December 20, 2011. All data points in the regular RIPE and BCNET datasets contain no anomalies and are, hence, labeled as regular traffic, as shown in Table 4.2. The results are generated using the MATLAB *fitcsvm* support vector classifier. The index of models reflects the dataset used for training and testing. These datasets contain no anomalies (both TP and FN values are zero) and, hence, precision is equal to zero while sensitivity is not defined. Consequently, the F-Score is also not defined and the accuracy reduces to:

$$\text{accuracy} = \frac{TN}{TN + FP}. \quad (4.7)$$

The best accuracy (93.40 %) and the best F-Score (76.52 %) for two-way classification is achieved by using SVM<sub>3</sub> with the features selected by the MIBASE algorithm for the Slammer test dataset. The Slammer test data points that are correctly classified as anomalies (true positive) in the two-way classification are shown in Fig. 4.2 (top) while incorrectly classified anomalous and regular data points are shown in Fig. 4.2 (bottom).

The SVM classifier may be also used to identify multiple classes [161]. For four-way classification, we use four training datasets (Slammer, Nimda, Code Red, and RIPE) to identify the specific type of BGP data point: Slammer, Nimda, Code Red, or regular (RIPE or BCNET). Classification of RIPE dataset achieves 91.85 % accuracy, as shown in Table 4.3.

## 4.3 Hidden Markov Model

HMMs are statistical tools used to model stochastic processes that consist of two embedded processes: the observable process that maps features and the unobserved hidden Markov process. We assume that the observations are independent and identically distributed (iid). In this dissertation, HMMs are used for non-parametric supervised classification.

### 4.3.1 Experiments and Performance Evaluation

We implement the first order HMMs using the MATLAB statistical toolbox. Each HMM model is specified by a tuple  $\lambda = (N, M, \alpha, \beta, \pi)$ , where:

$N$ : number of hidden states (cross-validated)

$M$ : number of observations

$\alpha$ : transition probability distribution  $N \times N$  matrix

$\beta$ : emission probability distribution  $N \times M$  matrix

$\pi$ : initial state probability distribution matrix.

Table 4.2: Performance of the two-way SVM classification. Two-way classification distinguishes two classes: anomaly and regular. SVM<sub>1</sub>: Training datasets (Slammer and Nimda) are used to identify the specific type of BGP data points (Code Red or regular (RIPE or BCNET)); SVM<sub>2</sub>: Training datasets (Slammer and Code Red) are used to identify the specific type of BGP data points (Nimda or regular (RIPE or BCNET)); SVM<sub>3</sub>: Training datasets (Nimda and Code Red) are used to identify the specific type of BGP data points (Slammer or regular (RIPE or BCNET)).

No.	SVM	Feature	Accuracy (%)		F-Score (%)	
			Test dataset	RIPE	BCNET	Test dataset
1	SVM <sub>1</sub>	37 features	78.65	69.17	57.22	39.51
2	SVM <sub>1</sub>	Fisher	81.93	85.67	80.49	41.16
3	SVM <sub>1</sub>	MID	85.38	92.63	83.68	45.18
4	SVM <sub>1</sub>	MIQ	80.86	86.89	83.75	39.56
5	SVM <sub>1</sub>	MIBASE	80.86	87.10	88.47	39.51
6	SVM <sub>1</sub>	OR	78.57	70.15	66.74	38.01
7	<b>SVM<sub>1</sub></b>	WOR	<b>88.03</b>	89.88	70.90	<b>47.18</b>
8	SVM <sub>1</sub>	MOR	83.88	83.40	83.75	44.53
9	SVM <sub>1</sub>	CDM	84.40	81.36	90.56	44.05
10	<b>SVM<sub>2</sub></b>	37 features	<b>55.50</b>	89.89	82.08	<b>24.29</b>
11	<b>SVM<sub>2</sub></b>	Fisher	54.22	96.28	<b>98.33</b>	16.43
12	SVM <sub>2</sub>	MID	53.89	95.88	95.76	11.89
13	SVM <sub>2</sub>	MIQ	55.10	96.10	97.57	20.74
14	SVM <sub>2</sub>	MIBASE	55.11	92.78	95.83	19.32
15	SVM <sub>2</sub>	OR	54.93	93.90	97.64	15.87
16	<b>SVM<sub>2</sub></b>	WOR	54.53	<b>97.39</b>	93.26	14.56
17	SVM <sub>2</sub>	MOR	55.36	96.74	97.85	20.21
18	SVM <sub>2</sub>	CDM	54.67	96.60	97.64	16.65
19	SVM <sub>3</sub>	37 features	93.04	73.92	59.24	75.93
20	SVM <sub>3</sub>	Fisher	93.31	80.63	57.71	76.51
21	SVM <sub>3</sub>	MID	93.35	75.33	59.65	76.30
22	SVM <sub>3</sub>	MIQ	93.28	78.99	57.50	76.27
23	<b>SVM<sub>3</sub></b>	MIBASE	<b>93.40</b>	79.14	57.85	<b>76.52</b>
24	SVM <sub>3</sub>	OR	90.44	80.53	79.03	70.32
25	SVM <sub>3</sub>	WOR	93.08	77.51	58.19	75.61
26	SVM <sub>3</sub>	MOR	92.92	76.79	68.68	75.48
27	SVM <sub>3</sub>	CDM	92.93	76.97	68.13	75.54

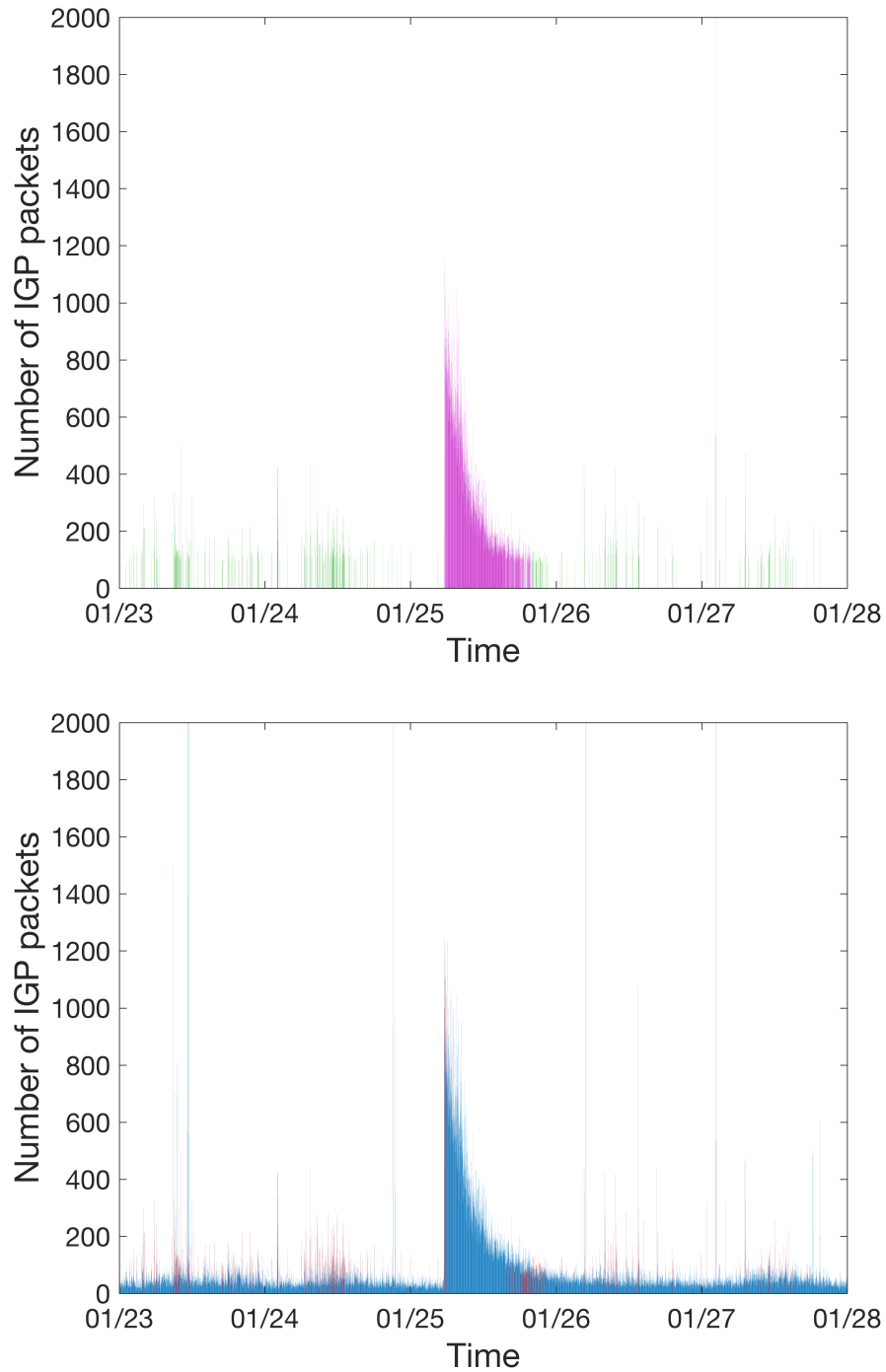


Figure 4.2: SVM classifier applied to Slammer traffic collected from January 23 to 27, 2003: Shown in purple are correctly classified anomalies (true positive) while shown in green are incorrectly classified anomalies (false positive) (top). Shown in red are incorrectly classified anomalous (false positive) and regular (false negative) data points while shown in blue are correctly classified anomalous (true positive) and regular (true negative) data points (bottom).

Table 4.3: Accuracy of the four-way SVM classification. Concatenation of training datasets (Slammer, Nimda, Code Red, and RIPE) is used to identify regular BGP data points (RIPE or BCNET).

No.	Feature	Accuracy (%)	
		RIPE	BCNET
1	37 features	84.47	76.11
2	<b>Fisher</b>	87.00	<b>83.13</b>
3	MID	91.11	80.07
4	MIQ	87.26	73.06
5	MIBASE	87.14	82.64
6	EOR	87.69	74.38
7	WOR	88.57	65.35
8	MOR	88.72	66.39
9	<b>CDM</b>	<b>91.85</b>	72.15

The proposed HMM classification models consist of three stages:

- *Sequence extracting and mapping*: All features are mapped to an observation vector.
- *Training*: Two HMMs for a two-way classification and four HMMs for a four-way classification are trained to identify the best probability distributions  $\alpha$  and  $\beta$  for each class. The HMMs are trained and validated for various numbers of hidden states  $N$ .
- *Classification*: Maximum likelihood probability  $p(x|\lambda)$  is used to classify the test observation sequences.

In the sequence extraction stage, the BGP feature matrix is mapped to a sequence of observations by selecting feature sets: FS1 (2, 5, 6, 7, 9, 10, 11, 13) and FS2 (2, 5, 6, 7, 10, 11, 13). In both cases, the number of observations is chosen to be the maximum number of selected features.

HMMs are trained and validated for various numbers of hidden states. For each HMM, a 10-fold cross-validation with the Baum-Welch algorithm [3] is used for training to find the best  $\alpha$  (transition) and  $\beta$  (emission) matrices by calculating the largest maximum likelihood probability  $p(x|\lambda_{\text{HMM}_x})$ . We construct seven two-way HMM models, as shown in Table 4.4. The name of each HMM model reflects the training dataset and the number of hidden states. Each test observation sequence is classified based on  $p(x|\lambda_{\text{HMM}_x})$ .

We use regular RIPE and BCNET datasets to evaluate performance of various classification models. The FS1 and FS2 are used to create an observation sequence for each HMM. The accuracy (4.1) is calculated using the highest  $p(x|\lambda_{\text{HMM}_x})$  in the numerator while sequences in the denominator share the same number of hidden states. As shown in Table 4.5, HMMs have higher accuracy using feature set FS1. The regular RIPE and BCNET datasets

Table 4.4: HMM models: Two-way classification. Two-way classification distinguishes two classes: anomaly and regular. Dataset 4: concatenation of Slammer, Nimda, and Code Red datasets.

Training dataset	Number of hidden states						
	2	3	4	5	6	7	8
Dataset 4	HMM <sub>1</sub>	HMM <sub>2</sub>	HMM <sub>3</sub>	HMM <sub>4</sub>	HMM <sub>5</sub>	HMM <sub>6</sub>	HMM <sub>7</sub>

have the highest accuracy when classified using HMMs with seven and three hidden states, respectively. Similar HMM models are developed for four-way classification and tested on regular RIPE and BCNET datasets.

Table 4.5: Accuracy of the two-way HMM classification. Two-way classification distinguishes two classes: anomaly and regular. Training datasets (Slammer, Nimda, and Code Red) are used to identify the specific type of BGP data points: anomaly or regular (RIPE or BCNET).

$N$ No. of hidden states	Feature set	Accuracy (%)	
		RIPE	BCNET
2	(2,5,6,7,9, 10,11,13)	42.15	50.62
<b>3</b>		45.21	<b>62.99</b>
4		16.11	36.53
5		19.31	27.15
6		16.81	21.11
<b>7</b>		<b>83.26</b>	52.01
8		67.50	41.04
$N$ No. of hidden states	Feature set	Accuracy (%)	
		RIPE	BCNET
2	(2,5,6,7, 10,11,13)	41.46	50.56
<b>3</b>		26.60	<b>59.17</b>
4		10.14	25.76
5		4.03	28.06
6		16.67	21.11
<b>7</b>		<b>82.99</b>	51.94
8		66.87	40.97

## 4.4 Naïve Bayes

The naïve Bayes (NB) classifiers are among the most efficient machine learning classification techniques. The generative Bayesian models are used as classifiers using labeled datasets. They assume conditional independence among features. Hence,

$$\Pr(\mathbf{X}_k = \mathbf{x}_k, \mathbf{X}_l = \mathbf{x}_l | c_j) = \Pr(\mathbf{X}_k = \mathbf{x}_k | c_j) \Pr(\mathbf{X}_l = \mathbf{x}_l | c_j), \quad (4.8)$$

where  $\mathbf{x}_k$  and  $\mathbf{x}_l$  are realizations of feature vectors  $\mathbf{X}_k$  and  $\mathbf{X}_l$ , respectively. In a two-way classification, classes labeled  $c_1 = 1$  and  $c_2 = -1$  denote anomalous and regular data points, respectively. For a four-way classification, four classes labeled  $c_1 = 1$ ,  $c_2 = 2$ ,  $c_3 = 3$ , and  $c_4 = 4$  refer to Slammer, Nimda, Code Red, and regular data points, respectively. Even though it is naive to assume that features are independent for a given class (4.8), for certain applications naïve Bayes classifiers perform better compared to other classifiers. They have low complexity, may be trained effectively with smaller datasets, and may be used for online real time detection of anomalies.

The probability distributions of the priors  $\Pr(c_j)$  and the likelihoods  $\Pr(\mathbf{X}_i = \mathbf{x}_i | c_j)$  are estimated using the training datasets. Posterior of a data point represented as a row vector  $\mathbf{x}_i$  is calculated using the Bayes rule:

$$\begin{aligned} \Pr(c_j | \mathbf{X}_i = \mathbf{x}_i) &= \frac{\Pr(\mathbf{X}_i = \mathbf{x}_i | c_j) \Pr(c_j)}{\Pr(\mathbf{X}_i = \mathbf{x}_i)} \\ &\approx \Pr(\mathbf{X}_i = \mathbf{x}_i | c_j) \Pr(c_j). \end{aligned} \quad (4.9)$$

The naive assumption of independence among features helps calculate the likelihood of a data point as:

$$\Pr(\mathbf{X}_i = \mathbf{x}_i | c_j) = \prod_{k=1}^K \Pr(X_{ik} = x_{ik} | c_j), \quad (4.10)$$

where  $K$  denotes the number of features. The probabilities on the right-hand side (4.10) are calculated using the Gaussian distribution  $\mathcal{N}$ :

$$\Pr(\mathbf{X}_{ik} = \mathbf{x}_{ik} | c_j, \mu_k, \sigma_k) = \mathcal{N}(X_{ik} = x_{ik} | c_j, \mu_k, \sigma_k), \quad (4.11)$$

where  $\mu_k$  and  $\sigma_k$  are the mean and standard deviation of the  $k^{th}$  feature, respectively. We assume that priors are equal to the relative frequencies of the training data points for each class  $c_j$ . Hence,

$$\Pr(c_j) = \frac{N_j}{N}, \quad (4.12)$$

where  $N_j$  is the number of training data points that belong to the  $j^{th}$  class and  $N$  is the total number of data points.

The parameters of two-way and four-way classifiers are estimated and validated by 10-fold cross-validation. In a two-way classification, an arbitrary training data point  $\mathbf{x}_i$  is classified as anomalous if the posterior  $\Pr(c_1 | \mathbf{X}_i = \mathbf{x}_i)$  is larger than  $\Pr(c_2 | \mathbf{X}_i = \mathbf{x}_i)$ .

#### 4.4.1 Experiments and Performance Evaluation

We use the MATLAB statistical toolbox to implement naïve Bayes classifiers and identify anomalous or regular data points. Datasets listed in Table 2.5 are used to train the two-way classifiers. The test datasets are Code Red, Nimda, and Slammer. The combination

of Code Red and Nimda training data points (NB3) achieves the accuracy (92.79 %) and F-Score (66.49 %), as shown in Table 4.6. The NB models classify the data points of regular RIPE and regular BCNET datasets with 90.28% and 88.40% accuracies, respectively. The OR and EOR algorithms generate identical results for the two-way classification and thus performance of the EOR algorithm is omitted.

The test data points from Slammer that are correctly classified as anomalies (true positives) during the 16 hours time interval are shown in Fig. 4.3 (top). Incorrectly classified (false positives and false negatives) data points using the NB3 classifier trained based on the features selected by Fisher in the two-way classification are shown in Fig. 4.3 (bottom). Most anomalous data points with large number of IGP packets (*volume* feature) are correctly classified.

The four-way naïve Bayes model classifies data points as Slammer, Nimda, Code Red, or Regular. Both regular RIPE and BCNET datasets are tested. Classification results for regular datasets are shown in Table 4.7. Although it is more difficult to classify four distinct anomalies, the classifier trained based on the features selected by the CDM algorithm achieves 90.14 % accuracy. Variants of the OR feature selection algorithm perform well when combined with the naïve Bayes classifiers because feature scores are calculated using the probability distribution that the naïve Bayes classifiers use for posterior calculations. Hence, the features selected by the OR variants are expected to have stronger influence on the posteriors calculated by the naïve Bayes classifiers [162]. Performance of the naïve Bayes classifiers is often inferior to the SVM and HMM classifiers.

## 4.5 Decision Tree

The decision tree algorithm is one of the most successful supervised learning techniques [10]. A tree is “learned” by splitting the source set into subsets based on an attribute value. This process is repeated on each derived subset using recursive partitioning. The recursion is completed when the subset at a node contains all values of the target variable or when the splitting no longer adds value to the predictions. After a decision tree is learned, each path from the root node (source) to a leaf node is transformed into a decision rule. Therefore, a set of rules is obtained by a trained decision tree that is used for classifying unseen samples.

### 4.5.1 Experiments and Performance Evaluation

Test accuracies and F-Scores are shown in Table 4.8. The results are generated using MATLAB *fitctree* from the statistics and machine learning toolbox. The lower accuracy of the training dataset 2 is due to the distributions of anomalous and regular data points in training and test datasets.

Table 4.6: Performance of the two-way naïve Bayes classification. Two-way classification distinguishes two classes: anomaly and regular. NB1: Training datasets (Slammer and Nimda) are used to identify the specific type of BGP data points (Code Red or regular (RIPE or BCNET)); NB2: Training datasets (Slammer and Code Red) are used to identify the specific type of BGP data points (Nimda or regular (RIPE or BCNET)); NB3: Training datasets (Nimda and Code Red) are used to identify the specific type of BGP data points (Slammer or regular (RIPE or BCNET)).

No.	NB	Feature	Accuracy (%)			F-Score (%)
			Test dataset	RIPE	BCNET	Test dataset
1	NB1	37 features	82.03	82.99	79.03	29.52
2	<b>NB1</b>	Fisher	<b>90.94</b>	88.13	76.46	36.08
3	NB1	MID	86.75	86.04	83.61	24.64
4	NB1	MIQ	88.86	87.78	75.21	30.38
5	NB1	MIBASE	90.92	87.64	77.92	35.12
6	<b>NB1</b>	OR	90.35	83.82	72.57	<b>37.33</b>
7	NB1	WOR	86.64	86.88	78.06	18.34
8	NB1	MOR	89.28	86.04	75.56	26.05
9	NB1	CDM	89.53	87.78	75.14	27.50
10	<b>NB2</b>	37 features	<b>62.56</b>	82.85	86.25	<b>48.78</b>
11	NB2	Fisher	57.51	87.01	83.26	27.97
12	<b>NB2</b>	MID	57.64	79.58	<b>88.40</b>	31.31
13	NB2	MIQ	56.68	84.38	82.15	26.35
14	NB2	MIBASE	57.60	86.88	82.99	28.55
15	NB2	OR	60.38	84.31	84.93	37.31
16	NB2	WOR	53.76	80.69	87.36	18.23
17	NB2	MOR	56.81	88.06	84.10	26.34
18	<b>NB2</b>	CDM	56.50	<b>90.28</b>	83.26	25.50
19	NB3	37 features	83.58	84.79	81.18	51.12
20	<b>NB3</b>	Fisher	<b>92.79</b>	87.36	75.97	<b>66.49</b>
21	NB3	MID	86.71	87.08	86.32	52.08
22	NB3	MIQ	92.47	88.68	77.15	65.03
23	NB3	MIBASE	92.49	89.31	80.42	62.97
24	NB3	OR	80.63	67.29	59.79	52.47
25	NB3	WOR	88.22	87.22	81.81	50.29
26	NB3	MOR	89.89	88.06	81.39	51.01
27	NB3	CDM	90.89	88.54	77.92	55.43

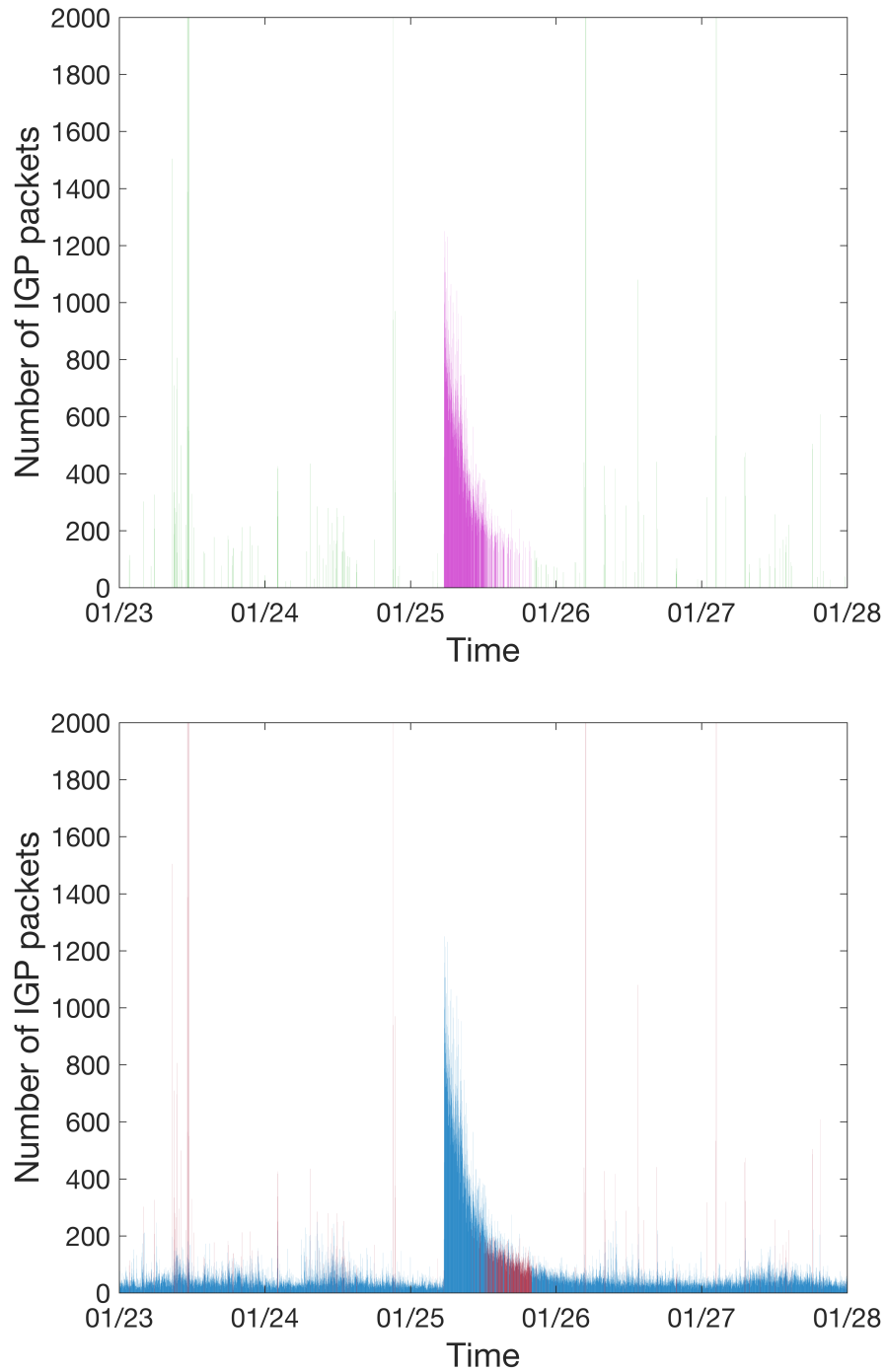


Figure 4.3: Naïve Bayes classifier applied to Slammer traffic collected from January 23 to 27, 2003: Shown in purple are correctly classified anomalies (true positive) while shown in green are incorrectly classified anomalies (false positive) (top). Shown in red are incorrectly classified anomalies (false positive) and regular (false negative) data points while shown in blue are correctly classified anomalous (true positive) and regular (true negative) data points (bottom).

Table 4.7: Accuracy of the four-way naïve Bayes classification. Concatenation of training datasets (Slammer, Nimda, Code Red, and RIPE) is used to identify regular BGP data points (RIPE or BCNET).

No.	Feature	Accuracy (%)	
		RIPE regular	BCNET
1	37 features	85.90	85.07
2	Fisher	88.75	84.24
3	MID	86.18	82.85
4	MIQ	89.38	84.51
5	MIBASE	88.75	84.24
6	EOR	89.03	90.07
7	WOR	88.40	87.36
8	MOR	88.75	87.71
9	<b>CDM</b>	<b>90.14</b>	83.54

Table 4.8: Performance of the two-way decision tree classification. Two-way classification distinguishes two classes: anomaly and regular. Dataset 1: concatenation of Slammer and Nimda datasets; Dataset 2: concatenation of Slammer and Code Red datasets; Dataset 3: concatenation of Nimda and Code Red datasets.

Training dataset	Test dataset	Accuracy (%) (test)	Accuracy (%)		F-Score (%) (test)
			RIPE	BCNET	
Dataset 1	Code Red	85.36	89.00	77.22	47.82
Dataset 2	Nimda	58.13	94.19	81.18	26.16
<b>Dataset 3</b>	Slammer	<b>95.89</b>	89.42	77.78	<b>84.34</b>

## 4.6 Extreme Learning Machine

The ELM [163] is an efficient learning algorithm used with a single hidden layer feed-forward neural network. It randomly selects the weights of the hidden layer and analytically determines the output weights. ELM avoids the iterative tuning of the weights used in traditional neural networks and, hence, it is fast and may be used as an online algorithm.

ELM employs weights connecting the input and hidden layers with the bias terms randomly initialized while the weights connecting the hidden and output layers are analytically determined. Its learning speed is higher than the traditional gradient descent-based method. Reported research results indicate that ELM may learn much faster than SVMs. Incremental and weighted extreme learning machines are variants of ELM.

A neural network architecture of the ELM algorithm is shown in Fig. 4.4, where  $[\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_d]$  is the input vector;  $d$  is the feature dimension;  $f(\cdot)$  is the activation function;  $\mathbf{W}_{in}$  is the vector of weights connecting the inputs to hidden units;  $[\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_m]$  is the output vector; and  $\mathbf{W}_{out}$  is the weight vector connecting the hidden and the output units.

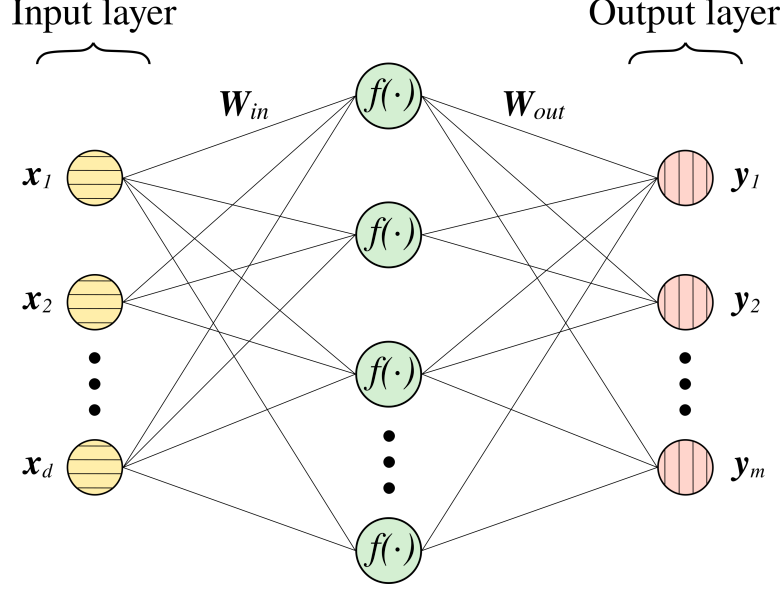


Figure 4.4: Neural network architecture of the ELM algorithm.  $[x_1, x_2, \dots, x_d]$  is the input vector;  $d$  is the feature dimension;  $f(\cdot)$  is the activation function;  $\mathbf{W}_{in}$  is the vector of weights connecting the inputs to hidden units;  $[y_1, y_2, \dots, y_m]$  is the output vector; and  $\mathbf{W}_{out}$  is the weight vector connecting the hidden and the output units.

#### 4.6.1 Experiments and Performance Evaluation

The three datasets used to verify ELM's performance are shown in Table 2.5. The number of hidden units is selected by the 5-fold cross validation for each training dataset. The best testing accuracy was achieved by choosing 315 hidden units for each dataset. The input vectors of the training datasets are mapped onto  $[-1, 1]$  as:

$$x_i^{(p)} = 2 \frac{x_i^{(p)} - x_{i_{min}}}{x_{i_{max}} - x_{i_{min}}} - 1, \quad (4.13)$$

where  $x_i^{(p)}$  is the  $i^{th}$  feature of the  $p^{th}$  sample while  $x_{i_{min}}$  and  $x_{i_{max}}$  are the minimum and maximum values of the  $i^{th}$  feature of the training sample, respectively.

The accuracies and F-Scores for the three ELM test datasets with 37 or 17 features are shown in Table 4.9. Accuracies for RIPE and BCNET datasets are also included.

## 4.7 Discussion

We have introduced and examined various traditional machine learning techniques for detecting network anomalies. Each approach has its unique advantages and limitations. Soft-margin SVMs perform well in classification tasks. However, they require relatively long time for training models when dealing with large datasets. HMM and naïve Bayes algorithms compute probabilities that events occur and are suitable for detecting multiple classes of

Table 4.9: Performance of the ELM algorithm using datasets with 37 and 17 features. Dataset 1: concatenation of Slammer and Nimda datasets; Dataset 2: concatenation of Slammer and Code Red datasets; Dataset 3: concatenation of Nimda and Code Red datasets.

No. of features	Training dataset	Test dataset	Accuracy (%) (test)	Accuracy (%)		F-Score (%) (test)
				RIPE	BCNET	
37	Dataset 1	Code Red	80.92	75.81	69.03	36.27
	Dataset 2	Nimda	54.42	96.15	91.88	13.72
	<b>Dataset 3</b>	Slammer	<b>86.96</b>	78.57	73.47	<b>55.31</b>
17	Dataset 1	Code Red	80.75	73.43	62.43	39.90
	Dataset 2	Nimda	55.13	94.11	83.75	15.97
	<b>Dataset 3</b>	Slammer	<b>92.57</b>	80.57	72.71	<b>72.52</b>

anomalies. Decision tree is commonly used in data mining due to its explicit and efficient decision making. ELM is an efficient classifier while its performance is limited due to its simple structure. When the testing accuracy of the classifiers is low, feature selection is used to improve their performance. Performance of the classifiers is greatly influenced by the employed datasets. While no single classifier that we have employed performs the best across all used datasets, traditional machine learning proved to be a feasible approach to successfully classify BGP anomalies.

## Chapter 5

# Deep Learning Networks

Deep learning networks are trained to identify important features in the input data by adjusting weights in each iteration. Their notable advantage is the back-propagation method that calculates gradients and updates the weights [150,164]. Furthermore, they may achieve desired results by adjusting the number of hidden nodes, hidden layers, optimization algorithms, and activation functions such as *rectified linear unit (ReLU)*, *sigmoid*, and *tanh*. The numbers of hidden nodes and layers are chosen depending on the size of the dataset. Note that additional hidden layers may not lead to higher accuracy because of over-fitting.

Neural networks have been applied in computer vision, natural language processing, and time-series prediction including detection of network anomalies. Deep learning neural networks exhibit more robust behaviour when dealing with large datasets compared to the conventional machine learning algorithms. Convolutional and RNNs learn the characteristics from data by extracting features and memorizing information, respectively. In this Chapter, we describe these two deep learning networks, experiments, and performance evaluation using BGP datasets.

### 5.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are used to process data containing one dimensional or multidimensional arrays by employing convolutional operations [13,14]. Convolutional and pooling layers are the main layers in a CNN model. Predefined kernels (weight matrices) are used to slide across the input matrix and create multiple feature maps by performing convolutions in the convolutional layer. The activation function is then applied to the output of the convolutional layer. Max or average pooling operations are used to reduce the dimensions of the feature maps by selecting their maximum elements or averaging regions using predefined filters. Fully-connected layers are employed after pooling layers. A CNN model may consist of multiple convolutional and pooling layers. CNNs have been successfully applied to object detection and recognition as well as image classification. CNNs may also be applied to classification of sequential data emanating from collections of

the Internet traffic. Fig. 5.1 illustrates an example of employ CNN network using network traffic data. We train CNN models using 1-dimensional input data due to the nature of our datasets: Each data point (a row in the BGP dataset) represents 1 minute of routing records.

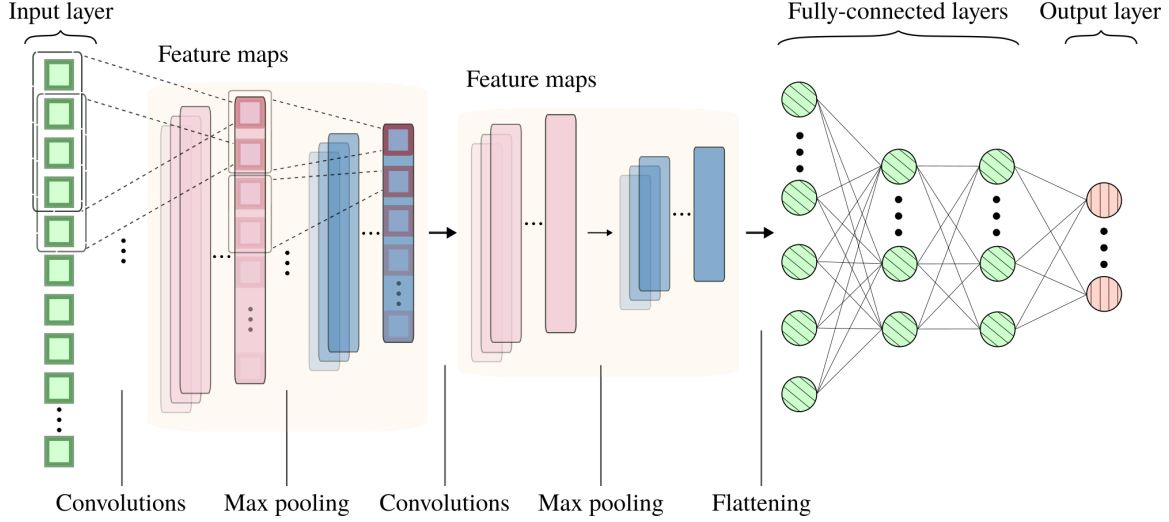


Figure 5.1: The high-level structure of a CNN using 1-dimensional input data: Convolutional and max pooling operations are performed twice; The output of the second max pooling layers is then flattened and passed through fully-connected layers; The probabilities of the output nodes are given in the output layer using the softmax function.

### 5.1.1 Experiments and Performance Evaluation

To evaluate the performance of CNN, we employ BGP datasets from the Pakistan power outage and the WestRock ransomware attack described in Chapter 2, Section 2.2.5.

**Label Refinement for the Recent Power Outage and Ransomware Attack:** A number of regular events may appear within the considered window of anomalous events. Hence, we refine labeling of anomalous data points by applying k-means and IF clustering algorithms. Label refinement may help to better identify anomalous data points and, thus, improves model performance. We assume that regular data points (class 0) correspond to days before and after the anomalous events. Data within the window of anomalous events are initially labeled as anomalies (class 1). The window with anomalous data may also contain data from regular events. Hence, we first use k-means and isolation forest (IF) unsupervised learning algorithms to identify and label regular data points within the anomalous periods. Majority of data points within each window retains the same original label (anomalies) after the label refinement using k-means and IF clustering. Note that regular data points are easier to identify using IF because they are detected as outliers.

We download BGP *update* messages from RIPE (rrc06 in Japan, rrc14 in Palo Alto, CA) and Route Views (WIDE in Japan, TELXATL in Atlanta, GA) data collection sites located

near the considered anomalous events. Examples of features extracted from regular and anomalous events collected during the Pakistan power outage and the WestRock ransomware attack are shown in Fig. 5.2. The regular data points within the windows of anomalous events are identified by using the IF algorithm. The patterns exhibit visible difference between regular and anomalous events for the WestRock ransomware dataset. In contrast, patterns are hardly separable in case of the Pakistan power outage dataset. Note that the power outage did not affect major telecommunication providers. Several features extracted from RIPE and Route Views datasets are visualized in scattered plots shown in Fig. 5.3.

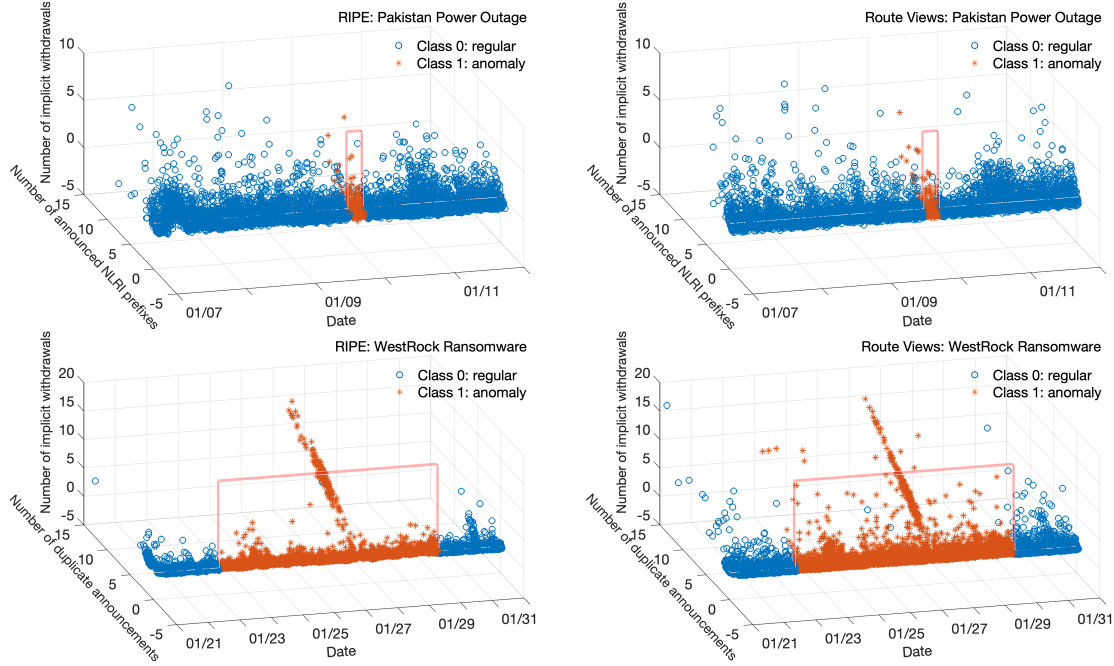


Figure 5.2: Shown are examples of features extracted from regular and anomalous events collected during the Pakistan power outage (top) and the WestRock ransomware attack (bottom). Pakistan power outage (top): Number of announced NLRI prefixes vs. number of implicit withdrawals vs. date; WestRock ransomware attack (bottom): Number of duplicate announcements vs. number of implicit withdrawals vs. date. Regular data points (class 0) are represented with circles while the anomalous data points (class 1) are represented with stars. Labels of data points within each anomaly window are refined using the IF algorithm.

Four datasets are generated for the Pakistan power outage and WestRock ransomware attack from data collected by RIPE and Route Views. The labels are generated with and without employing unsupervised learning algorithms for label refinement. Hence, we evaluate twelve datasets that are created based on the combinations of data points and corresponding labels. Each dataset is split into training and test datasets consisting of 60 % and 40 % anomalous data points, receptively.

The clusters are generated by using the k-means random initialization option to choose the initial centroids of the dataset.

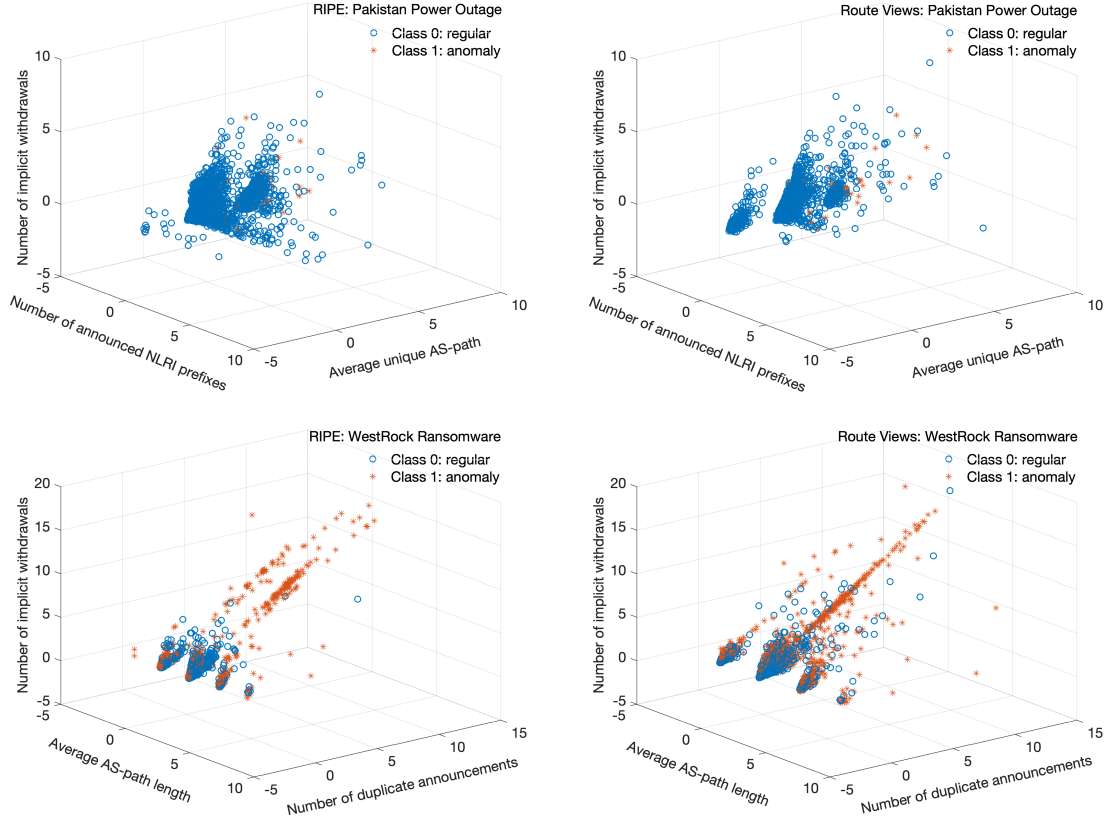


Figure 5.3: Pakistan power outage (top): Number of announced NLRI prefixes vs. average unique AS-path vs. number of implicit withdrawals; WestRock ransomware attack (bottom): Average unique AS-path vs. number of duplicate announcements vs. number of implicit withdrawals. In case of the Pakistan power outage, spatial separation of classes is not visible, which may explain lower accuracy and F-Score compared to the WestRock ransomware attack.

Random seeds and number of iterations are varied to select the initial centroids that lead to the highest silhouette coefficient. In the case of isolation forest, varied are parameters such as random seeds (to control pseudo-randomness for feature and split values), the number of trees, and the contamination parameter. We select the best separation based on the highest silhouette coefficient.

**CNN Model:** We develop the CNN model consisting of 2 convolutional, 2 maximum pooling, and 3 fully-connected layers. One-dimensional convolutional kernels are applied to each input data point (one-dimensional vector). Various feature maps are generated using 24 and 12 kernels in the first and second convolutional layers, respectively. These kernels are of sizes 5 and 7 and use stride of 1 to slide across the input data and the output of the first maximum pooling layer, respectively. The generated feature maps retain properties of the original input data. The maximum pooling kernels of size 2 are applied to all feature maps thus discarding the redundancy. The 3 fully-connected layers consist of 72,

32, and 16 hidden nodes. The ReLU activation function is applied after the convolutional and fully-connected layers. The output layer consists of 2 output nodes (regular and anomaly) indicating probabilities of the outcomes. In our experiments, decreasing the number of hidden nodes between the fully-connected layers often generates a robust model and, thus, enhance performance.

The best performance of CNN models using the Pakistan power outage and WestRock ransomware datasets is shown in Table 5.1. A large number of false negatives (FNs) is obtained from the testing phase, indicating that most anomalies are classified as regular data points (TNs). Performance of CNN models is compared to performance of LightGBM (Chapter 6) and RNNs and Bi-RNNs models (Section 5.3).

## 5.2 Recurrent Neural Networks

RNNs are deep learning networks used to process sequential data  $x^{(t)} = x(1), \dots, x(\tau)$  having  $\tau$  elements of fixed or variable lengths where  $t$  represents the index in the sequence [14]. These recurrent networks share parameters during the learning process: an output is a function of the previous time step (computation) and the same update rule applies to each element of the sequence. Recurrent connections may be present between hidden units or between the output and hidden units at consecutive time steps. RNNs are trained using mini-batches of the input data (sequence) where elements of the mini-batch may be of different lengths. They may be designed to produce an output at each time step or after processing an entire sequence. Bidirectional RNNs (Bi-RNNs) may be employed when the prediction depends on interpretation of past and future time steps. They consist of forward and backward layers that process data starting at the beginning and at the end of the sequence, respectively.

### 5.2.1 Long Short-Term Memory

LSTM networks are RNNs that are capable of learning long-term dependencies by connecting time intervals to form a continuous memory [151, 152]. Traditional RNN networks perform poorly when they need to bridge segments of information with long time gaps. Hence, LSTM networks were introduced to overcome long-term dependency and vanishing gradient problems.

The LSTM cell shown in Fig. 5.4, is composed of: (a) *forget gate*  $f_t$ , (b) *input gate*  $i_t$ , and (c) *output gate*  $o_t$ . The *forget gate* discards irrelevant memories based on the cell state, the *input gate* controls the information that will be updated in the LSTM cell, the *output gate* functions as a filter that controls the output, while the logistic sigmoid  $\sigma$  and  $\tanh$  are used as cell functions. The output of the LSTM cell is connected to the output layer and the next cell.

Table 5.1: The best performance of CNN models: Pakistan power outage and WestRock ransomware.

Model	Collection site	Training time (s)	Accuracy (%)	F-Score (%)	Precision (%)	Sensitivity (%)	TP	FP	TN	FN
Pakistan Power Outage										
CNN	No refinement	RIPE	51.00	84.93	7.00	4.64	14.17	17	349	2,531
		Route Views	52.01	95.00	3.82	8.11	2.50	3	34	2,846
	k-means	RIPE	50.99	93.50	4.88	5.62	4.31	5	84	2,800
		Route Views	52.00	95.87	1.59	12.50	0.85	1	7	2,875
	IF	RIPE	50.81	86.53	8.18	5.63	15.00	18	302	2,578
		Route Views	57.16	83.37	6.03	3.89	13.33	16	395	2,485
WestRock Ransomware										
CNN	No refinement	RIPE	18.79	55.33	70.96	57.04	93.85	3,754	2,827	53
		Route Views	18.66	57.67	72.96	58.04	98.23	3,929	2,841	39
	k-means	RIPE	19.31	55.31	71.00	56.99	94.25	3,770	2,845	35
		Route Views	18.88	57.06	72.32	57.82	96.52	3,859	2,815	67
	IF	RIPE	19.20	55.29	70.96	57.00	94.00	3,759	2,836	45
		Route Views	18.80	57.12	72.44	57.83	96.93	3,877	2,827	53

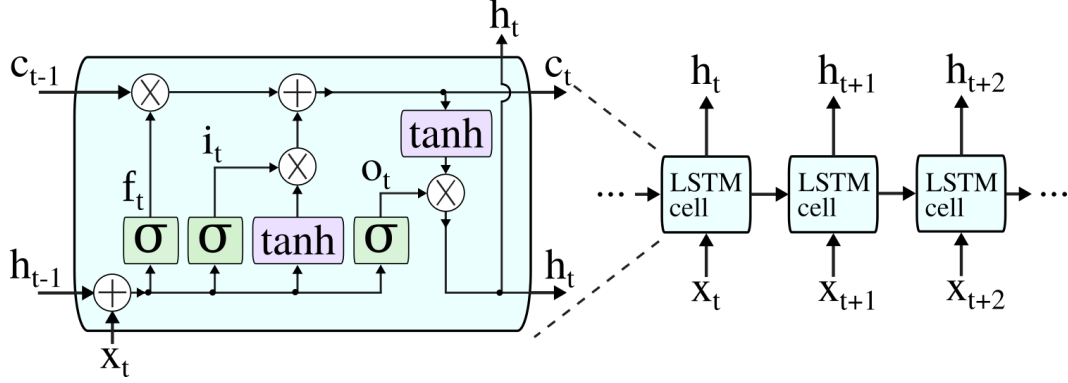


Figure 5.4: Repeating cell for the LSTM neural network. The current cell state  $c_t$  and output  $h_t$  are calculated based on input  $x_t$ , previous cell state  $c_{t-1}$ , and output  $h_{t-1}$ .

The outputs of the *forget gate*  $f_t$ , *input gate*  $i_t$ , and *output gate*  $o_t$  at time  $t$  are [165]:

$$\begin{aligned} f_t &= \sigma(W_{if}x_t + b_{if} + U_{hf}h_{t-1} + b_{hf}) \\ i_t &= \sigma(W_{ii}x_t + b_{ii} + U_{hi}h_{t-1} + b_{hi}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + U_{ho}h_{t-1} + b_{ho}), \end{aligned} \quad (5.1)$$

where  $\sigma(\cdot)$  is the logistic sigmoid function,  $x_t$  is the current input,  $h_{t-1}$  is the previous output,  $W_{if}$ ,  $U_{hf}$ ,  $W_{ii}$ ,  $U_{hi}$ ,  $W_{io}$ , and  $U_{ho}$  are weight matrices, and  $b_{if}$ ,  $b_{hf}$ ,  $b_{ii}$ ,  $b_{hi}$ ,  $b_{io}$ , and  $b_{ho}$  are bias vectors. The information is stored in the cell state depending on the output  $i_t$  of the *input gate*. The sigmoid function is used to update the cell state  $c_t$  calculated as:

$$c_t = f_t * c_{t-1} + i_t * \tanh(W_{ic}x_t + b_{ic} + U_{hc}h_{t-1} + b_{hc}), \quad (5.2)$$

where  $*$  denotes element-wise multiplications and the  $\tanh$  function is used to calculate the input to the next cell state. The output of the LSTM cell is:

$$h_t = o_t * \tanh(c_t). \quad (5.3)$$

### 5.2.2 Gated Recurrent Unit

The GRU cell, shown in Fig. 5.5, is derived from LSTM and has a simpler structure. To make predictions, it employs gated mechanisms to control input and memory at the current timestep. While an LSTM cell consists of three gates, a GRU cell contains only *reset*  $r_t$  and *update*  $z_t$  gates [153]. The *reset gate* determines the combination of new input information and previous memory content while the *update gate* defines the content stored at the current timestep.

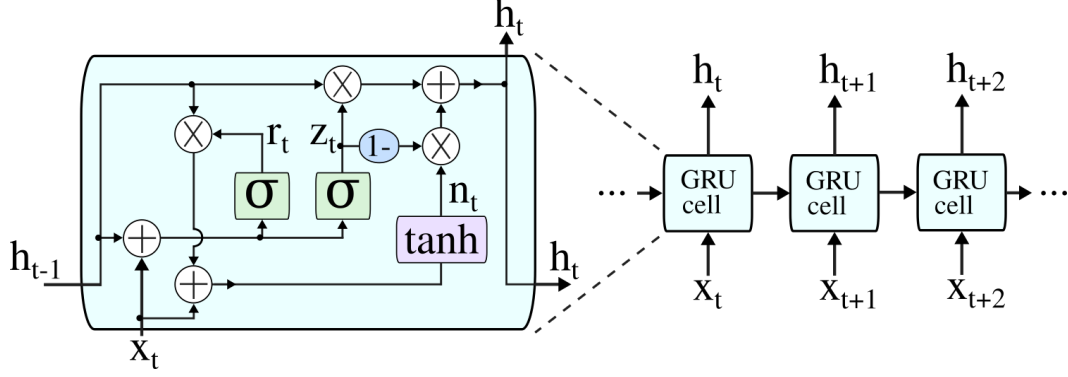


Figure 5.5: Repeating cell for the GRU neural network. The current output  $h_t$  is calculated based on input  $x_t$  and output  $h_{t-1}$ .

The outputs of the *reset gate*  $r_t$  and the *update gate*  $z_t$  at time  $t$  are [165]:

$$\begin{aligned} r_t &= \sigma(W_{ir}x_t + b_{ir} + U_{hr}h_{t-1} + b_{hr}) \\ z_t &= \sigma(W_{iz}x_t + b_{iz} + U_{hz}h_{t-1} + b_{hz}), \end{aligned} \quad (5.4)$$

where  $\sigma$  is the sigmoid function,  $x_t$  is the input,  $h_{t-1}$  is the previous cell output,  $W_{ir}$ ,  $U_{hr}$ ,  $W_{iz}$ , and  $U_{hz}$  are the weight matrices, and  $b_{ir}$ ,  $b_{hr}$ ,  $b_{iz}$ , and  $b_{hz}$  are the bias vectors. The output of the GRU cell is:

$$h_t = (1 - z_t) * n_t + z_t * h_{t-1}, \quad (5.5)$$

where  $n_t$  is:

$$n_t = \tanh(W_{in}x_t + b_{in} + r_t * (U_{hn}h_{t-1} + b_{hn})), \quad (5.6)$$

$W_{in}$  and  $U_{hn}$  are the weight matrices, and  $b_{in}$  and  $b_{hn}$  are the bias vectors.

### 5.2.3 Experiments and Performance Evaluation

We perform two-way classification to identify regular (0) and anomalous (1) data based on Slammer worm, Moscow blackout, and WannaCrypt ransomware events described in Chapter 2, Section 2.2.5. Deep learning RNN (LSTM and GRU) models are utilized due to their unique structure and capability to classify time series data.

We label data points corresponding to Slammer, Moscow blackout, or WannaCrypt as anomalous data and employ supervised machine learning to classify anomalies. BGP anomalies caused by Slammer and WannaCrypt resulted in visible changes in volume (number of BGP announcements and BGP withdrawals) and AS-path (average AS-path length and average edit distance) features. In case of the BGP link failures experienced during the Moscow blackout, some affected ASes found alternative routes, which resulted in an inconclusive period of the Internet anomaly and a narrower window compared to the power

system downtime [109]. BGP changes are primarily observed in volume features (number of BGP announcements, number of announced NLRI prefixes, and number of interior gateway protocol packets).

The Slammer and WannaCrypt training and test datasets consist of 60 % and 40 % of anomalous data, respectively while Moscow blackout training and test datasets consist of 75 % and 25 % (RIPE) and 65% and 35% (Route Views) of anomalous data, respectively. We keep the number of data points in the training and test datasets to be divisible by pre-defined input sequence length by moving the remaining data points from the training to the test dataset.

**Multi-Layer Networks:** We consider models with only up to four hidden layers and a small number of hidden nodes near the output layer in order to prevent over-fitting. We evaluate performance of LSTM and GRU models with 2 (LSTM<sub>2</sub> and GRU<sub>2</sub>), 3 (LSTM<sub>3</sub> and GRU<sub>3</sub>), and 4 (LSTM<sub>4</sub> and GRU<sub>4</sub>) hidden layers. A model with 4 hidden layers is shown in Fig. 5.6. We implement deep learning RNN models using one GPU (NVIDIA GeForce GTX

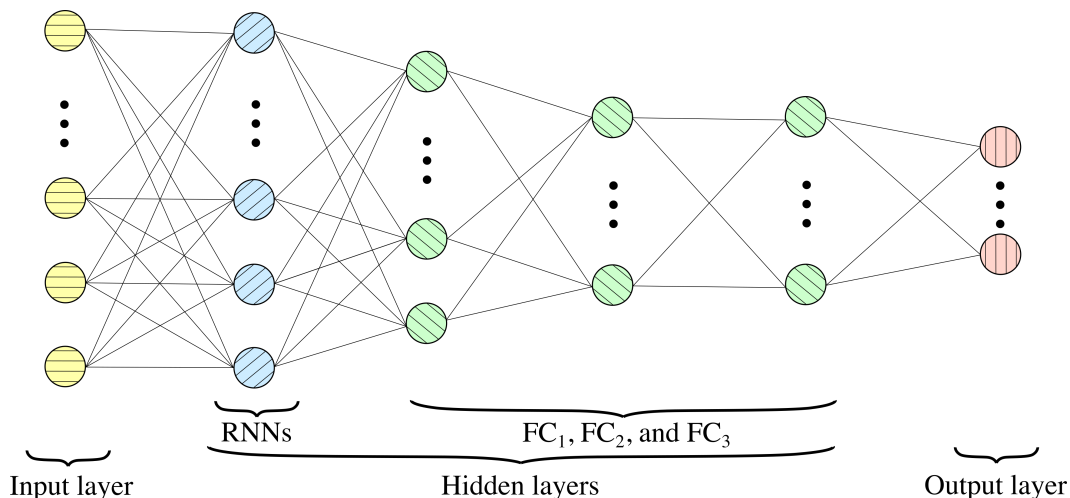


Figure 5.6: Deep learning neural network model. It consists of 37 RNNs, 80 (Slammer)/64 (Moscow blackout)/64 (WannaCrypt) FC<sub>1</sub>, 32 FC<sub>2</sub>, and 16 FC<sub>3</sub> fully connected (FC) hidden nodes.

1080 GPU). Python 3.6 running on Ubuntu 16.04 was used to generate simulation results. Cross-validation is performed for various parameters. The best parameters for training models are shown in Table 5.2.

**Performance Comparison:** We evaluate performance of deep learning RNN classification models based on accuracy and F-Score. Performance of LSTM and GRU models with various hidden layers using Slammer, Moscow blackout, and WannaCrypt datasets is shown in Table 5.3. The best classification results for RIPE datasets are achieved using LSTM<sub>2</sub> (Slammer and Moscow blackout) and LSTM<sub>3</sub> (WannaCrypt) models. For the Route Views datasets, the best classification results are obtained using LSTM<sub>3</sub> (Slammer), GRU<sub>2</sub> and

Table 5.2: Parameters of RNN models used in cross-validation.

Parameter	Value	Best selection
Length of input sequence	5, 10, 20, 50, 100	Slammer: 10 Moscow blackout: 100 (RIPE), 20 (Route Views) WannaCrypt: 100
Number of epochs	30, 50, 100	30
Number of hidden nodes	80, 64, 32, 16	Slammer: $FC_1 = 80, FC_2 = 32, FC_3 = 16$ Moscow blackout/WannaCrypt: $FC_1 = 64, FC_2 = 32, FC_3 = 16$
Dropout rate	0.2, 0.4, 0.6	0.4
Learning rate	0.001, 0.01, 0.1	0.001

GRU<sub>4</sub> (Moscow blackout), and GRU<sub>3</sub> and GRU<sub>2</sub> (WannaCrypt) models. It has been observed that increasing the number of hidden layers may result in over-fitting. As expected, the best accuracy and F-Score generated by RNN models using Route Views datasets are higher than RIPE. Moscow blackout data collected by RIPE during the time of anomaly are more reliable than Route Views data, hence better classification results. Note that a much smaller number of anomalous data points (130) is collected during the four-hour interval.

#### 5.2.3.1 Discussion

Models with two and three hidden layers often exhibited the best performance. The best accuracy and F-Score for Slammer and WannaCrypt were generated using BGP *update* messages collected by Route Views. In contrast, the best performance for classifying Moscow blackout was obtained using RIPE datasets due to missing data points in Route Views. Classification models for Slammer datasets offered better results because the data had better spatial separation between regular and anomalous classes.

### 5.3 Performance Comparison: CNN, RNN, and Bi-RNN

In this Section, we compare the performance of CNN, RNN, and Bi-RNN models. CNN results shown in Table 5.1 are used for the comparison.

RNNs and Bi-RNNs models consist of one RNN and one Bi-RNN layer, respectively, and up to three fully-connected (FC) layers. The generated models have 2 (LSTM<sub>2</sub>/Bi-LSTM<sub>2</sub>, GRU<sub>2</sub>/Bi-GRU<sub>2</sub>), 3 (LSTM<sub>3</sub>/Bi-LSTM<sub>3</sub>, GRU<sub>3</sub>/Bi-GRU<sub>3</sub>), and 4 (Bi-LSTM<sub>4</sub>, Bi-GRU<sub>4</sub>) hidden layers. The first layer in the LSTM and GRU deep learning neural network models consist of 37 RNNs or Bi-RNNs. The second, third, and fourth hidden layers consist of 64 (FC<sub>1</sub>), 32 (FC<sub>2</sub>), and 16 (FC<sub>3</sub>) nodes, respectively, depending on the the number of fully connected layers. Layers with 0.5 dropout probability are inserted after the FC<sub>2</sub> and FC<sub>3</sub>

Table 5.3: Performance of LSTM and GRU RNN models: Slammer, Moscow blackout, and WannaCrypt datasets. Highlighted are the best models for Slammer (purple), Moscow blackout (brown), and WannaCrypt (blue).

Model	Dataset	Accuracy (%)		F-Score (%)	
		RIPE	Route Views	RIPE	Route Views
LSTM <sub>2</sub>	Slammer	92.98	91.24	72.42	69.11
	Moscow blackout	99.21	96.23	75.20	5.26
	WannaCrypt	58.08	67.23	61.48	70.14
LSTM <sub>3</sub>	Slammer	90.90	95.72	67.29	81.77
	Moscow blackout	98.38	97.77	55.94	32.00
	WannaCrypt	65.48	64.35	63.22	67.16
LSTM <sub>4</sub>	Slammer	92.49	91.39	70.72	69.34
	Moscow blackout	97.46	95.81	36.94	18.37
	WannaCrypt	57.94	72.29	62.42	73.86
GRU <sub>2</sub>	Slammer	91.88	92.60	69.42	72.59
	Moscow blackout	97.64	98.30	41.77	32.99
	WannaCrypt	57.27	72.58	60.56	74.21
GRU <sub>3</sub>	Slammer	91.76	93.24	68.72	74.34
	Moscow blackout	98.38	97.51	57.14	28.57
	WannaCrypt	52.85	72.63	53.96	74.14
GRU <sub>4</sub>	Slammer	92.14	93.15	70.11	74.04
	Moscow blackout	97.92	97.20	49.06	35.15
	WannaCrypt	52.15	68.71	52.70	71.61

layers. The “Adam” algorithm is used to optimize RNN/Bi-RNN models using 30 (Pakistan power outage) and 50 (WestRock ransomware attack) epochs.

The best performance for LSTM and GRU algorithms using the Pakistan power outage and WestRock ransomware datasets are shown in Table 5.4. The highest accuracy for the Pakistan power outage is achieved when using the Bi-GRU<sub>3</sub> model with k-means label refinement. The LSTM<sub>4</sub> model with IF label refinement leads to the highest F-Score and precision. Telecommunication providers were the least affected during the power outage and, hence, BGP records do not capture characteristics of the event leading to low F-Score values. The highest accuracy and F-Score for the WestRock ransomware attack are generated using the Bi-GRU<sub>4</sub> model with IF label refinement. Because the WestRock ransomware attack and its remediation required several days, LSTM, GRU, and Bi-GRU models lead to higher performance due to their ability to capture long-term dependencies. The BGP data collected by RIPE and Route Views sites during the Pakistan power outage and WestRock ransomware attack have similar patterns and spatial separations. Note that BGP data collected during the period of known anomalous events often contain regular traffic. Unlike collection of TCP/UDP network flows, BGP data may not capture anomalies that directly affect host computers and, hence, may not provide sufficient information for the machine

Table 5.4: The best performance of LSTM, GRU, Bi-LSTM, and Bi-GRU models:  
Pakistan power outage and WestRock ransomware.

Model	Collection site	Training time (s)	Accuracy (%)	F-Score (%)	Precision (%)	Sensitivity (%)	TP	FP	TN	FN
Pakistan Power Outage										
LSTM <sub>4</sub>	No refinement	45.05	92.83	4.44	4.76	4.17	5	100	2,780	115
	Route Views	42.29	95.77	14.77	37.93	9.17	11	18	2,862	109
LSTM <sub>2</sub>	k-means	32.42	93.93	7.14	8.75	6.03	7	73	2,811	109
	Route Views	32.15	95.70	12.24	31.03	7.63	9	20	2,862	109
GRU <sub>3</sub>	IF	66.47	93.03	3.69	4.12	3.33	4	93	2,787	116
LSTM <sub>4</sub>	Route Views	41.93	95.83	14.97	40.74	9.17	11	16	2,864	109
Bi-LSTM <sub>2</sub>	RIPE	25.83	95.57	9.52	25.93	5.83	7	20	2,860	113
Bi-GRU <sub>2</sub>	Route Views	41.92	95.60	2.94	12.50	1.67	2	14	2,866	118
Bi-LSTM <sub>3</sub>	RIPE	29.94	95.57	11.92	25.71	7.76	9	26	2,858	107
Bi-LSTM <sub>2</sub>	Route Views	43.37	95.73	3.03	14.29	1.69	2	12	2,870	116
Bi-GRU <sub>3</sub>	RIPE	27.71	95.90	8.89	40.00	5.00	6	9	2,871	114
Bi-LSTM <sub>2</sub>	Route Views	43.40	95.77	3.05	18.18	1.67	2	9	2,871	118
WestRock Ransomware										
GRU <sub>4</sub>	RIPE	13.99	75.23	80.24	74.84	86.48	3,459	1,163	1,717	541
LSTM <sub>4</sub>	Route Views	18.95	55.42	70.72	57.20	92.60	3,704	2,771	109	296
GRU <sub>4</sub>	RIPE	14.44	75.44	79.73	76.63	83.10	3,324	1,014	1,866	676
GRU <sub>2</sub>	Route Views	13.44	62.30	69.61	65.47	74.31	2,971	1,567	1,315	1,027
LSTM <sub>2</sub>	RIPE	12.63	75.36	79.73	76.41	83.35	3,333	666	1,029	1,852
LSTM <sub>3</sub>	Route Views	13.77	60.00	69.06	62.75	76.80	3,072	1,824	1,056	928
Bi-GRU <sub>4</sub>	RIPE	20.59	78.49	81.92	80.10	83.83	3,353	833	2,047	647
Bi-GRU <sub>3</sub>	Route Views	21.89	62.50	69.70	65.73	74.18	2,967	1,547	1,333	1,033
Bi-GRU <sub>3</sub>	RIPE	20.27	77.76	82.05	77.30	87.43	3,497	1,027	1,853	503
	Route Views	20.14	63.36	72.15	64.61	81.69	3,266	1,789	1,093	732
Bi-GRU <sub>4</sub>	RIPE	23.73	84.27	86.90	84.23	89.75	3,589	672	2,209	410
Bi-GRU <sub>3</sub>	Route Views	20.23	64.74	72.19	66.67	78.70	3,148	1,574	1,306	852

learning models to correctly perform classification. However, synthetic datasets containing TCP/UDP network flows are generated in controlled testbed environments.

### **5.3.1 Discussion**

Performance results indicated that in most cases RNNs and Bi-RNNs outperformed CNN models. Label refinement improved performance as illustrated by the best performance results obtained after applying k-means and IF clustering algorithms. CNN, RNNs, and Bi-RNNs had comparable training time. Models based on datasets generated from RIPE and Route Views routing records exhibited comparable performance. In the case of WestRock ransomware dataset, distinct patterns and spatial separations between regular and anomalous data points as well as large number of anomalous data points led to higher F-Score values. Employing label refinement and Bi-RNN models with more than two hidden layers improved the classification performance for longer-lasting anomalous events such as the ransomware attack.

## Chapter 6

# Fast Machine Learning Algorithms

Training time is important for the decision-making process at the onset of anomalies when preventing cyberattacks on servers and avoiding denial of service to legitimate users. In this Chapter, we evaluate fast machine learning algorithms including: broad learning system [4] and its extensions [15, 33, 166] as well as the gradient boosting decision tree algorithms: XGBoost [156], LightGBM [12], and CatBoost [157]. BLS is a fast machine learning algorithm that relies on pseudo-inverse during the training process and has a single layer feed-forward neural network. Training time of GBDT algorithms is optimized by iteratively constructing an ensemble of decision trees and using functional gradient descent.

### 6.1 Broad Learning System

Deep learning neural networks may require long training time because of their high computational complexity and a large number of hidden layers. In contrast, boosting algorithms such as LightGBM [12] and BLS [155] offer comparable performance with shorter training time. BLS is based on a single layer feedforward neural network and pseudo-inverse or ridge regression to calculate outputs. Several BLS extensions exploit the algorithm's flexible structure to include: incremental learning [4], radial basis function network (RBF-BLS) [33] as well as cascades of mapped features (CFBLS), enhancement nodes (CEBLS), and both mapped features and enhancement nodes (CFEBLS) [15].

BLS improves the random vector functional-link neural network [167] by mapping the input data  $\mathbf{X}$  to a set of groups of mapped features  $\mathbf{Z}^n \triangleq [\mathbf{Z}_1, \dots, \mathbf{Z}_n]$  that generates enhancement nodes  $\mathbf{H}^m \triangleq [\mathbf{H}_1, \dots, \mathbf{H}_m]$  using random weights as shown in Fig. 6.1. Groups of mapped features and enhancement nodes are defined as:

$$\mathbf{Z}_i = \phi(\mathbf{X}\mathbf{W}_{e_i} + \beta_{e_i}), \quad i = 1, 2, \dots, n \quad (6.1)$$

$$\mathbf{H}_j = \xi(\mathbf{Z}_x^n \mathbf{W}_{h_j} + \beta_{h_j}), \quad j = 1, 2, \dots, m, \quad (6.2)$$

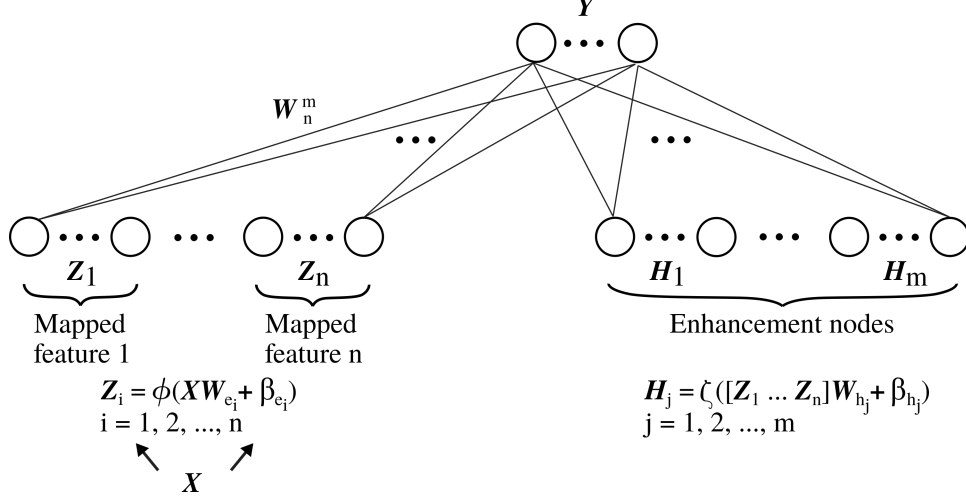


Figure 6.1: Module of the original BLS algorithm with mapped features and enhancement nodes [4].

where  $\phi$  (linear) and  $\xi$  (tanh) are the feature and enhancement mappings, respectively,  $W_{e_i}$  and  $W_{h_j}$  are weights, and  $\beta_{e_i}$  and  $\beta_{h_j}$  are bias parameters. A state matrix  $A_n^m$  is constructed by concatenating matrices  $Z^n$  and  $H^m$  associated with  $n$  groups of mapped features and  $m$  groups of enhancement nodes, respectively.

In the training phase, the Moore-Penrose pseudo-inverse or ridge regression is used to invert the state matrix and calculate the output weights  $W_n^m$  for the given labels  $Y$ :

$$W_n^m = (\lambda I + (A_n^m)^T A_n^m)^{-1} (A_n^m)^T Y, \quad (6.3)$$

where  $\lambda$  is the regularization coefficient and  $I$  is the identity matrix. During testing, data labels are predicted using the calculated weights, mapped features, and enhancement nodes.

Limitations of the original BLS are due to using a predefined (fixed) number of mapped features and enhancement nodes. The BLS structure may be dynamically expanded by using incremental learning to include additional input data  $X_a$ , mapped features  $Z_{n+1}$ , and enhancement nodes  $H_{m+1}$  as shown in Fig. 6.2, thus, leading to improved model performance. It requires a shorter training time because the weights are updated using only the incremental input data instead of retraining the entire model. RBF-BLS extension employs the Gaussian rather than *tanh* function as the enhancement mapping  $\xi$ . The structure of BLS with cascades is defined by the connections within and between the mapped features and enhancement nodes. The CFBLs and CEBLs architectures are shown in Fig. 6.3. In the case of CFBLs, the first group of mapped features is based on input data and weights (6.1) while subsequent groups ( $k$ ) of mapped features are created by using the previous

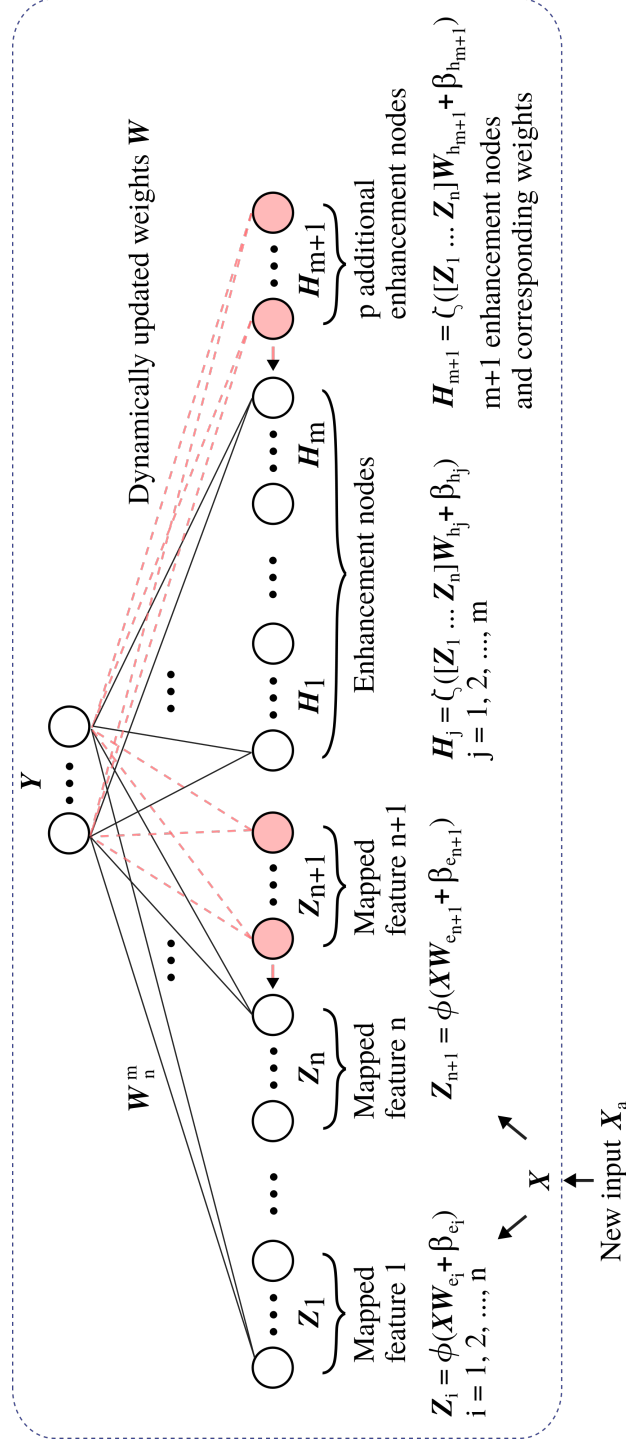


Figure 6.2: Module of the BLS algorithm with increments of mapped features, enhancement nodes, and new input data [4].

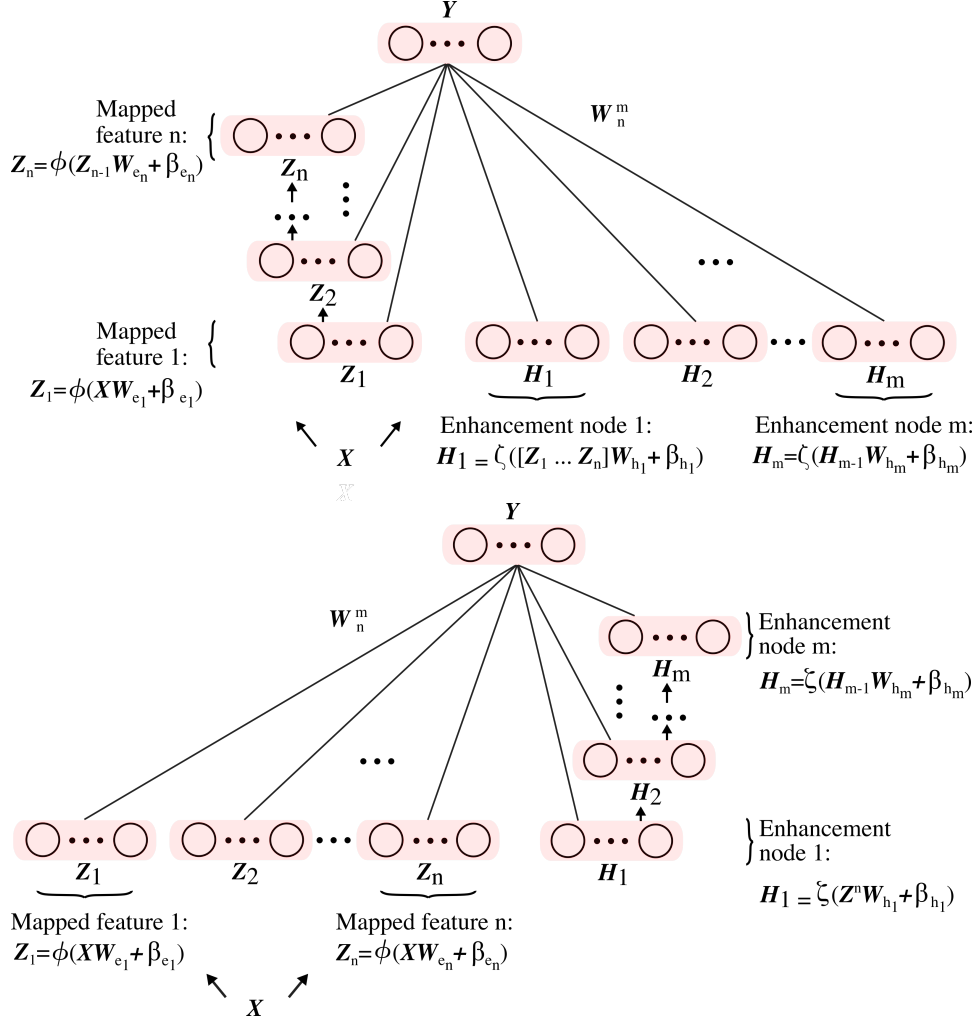


Figure 6.3: Modules of the CFBLs (top) and CEBLs (bottom) algorithms. Shown are cascades of mapped features (top) and enhancement nodes (bottom) without incremental learning.

group  $(k - 1)$ . The groups of mapped features are formulated as:

$$\begin{aligned} \mathbf{Z}_k &= \phi(\mathbf{Z}_{k-1} \mathbf{W}_{e_k} + \beta_{e_k}) \\ &\triangleq \phi^k(\mathbf{X}; \{\mathbf{W}_{e_i}, \beta_{e_i}\}_{i=1}^k), \text{ for } k = 1, \dots, n. \end{aligned} \quad (6.4)$$

The cascades of these groups  $\mathbf{Z}^n \triangleq [\mathbf{Z}_1, \dots, \mathbf{Z}_n]$  are used to generate the enhancement nodes  $\{\mathbf{H}_j\}_{j=1}^m$ . The first CEBLS enhancement node is generated from mapped features while subsequent nodes are generated from previous nodes creating a cascade:

$$\begin{aligned} \mathbf{H}_u &= \xi(\mathbf{H}_{u-1} \mathbf{W}_{e_u} + \beta_{e_u}) \\ &\triangleq \xi^u(\mathbf{Z}^n; \{\mathbf{W}_{h_i}, \beta_{h_i}\}_{i=1}^u), \text{ for } u = 1, \dots, m, \end{aligned} \quad (6.5)$$

where  $\mathbf{W}_{h_i}$  and  $\beta_{h_i}$  are randomly generated. The CFEBLS architecture is a combination of the two cascading approaches as shown in Fig. 6.4.

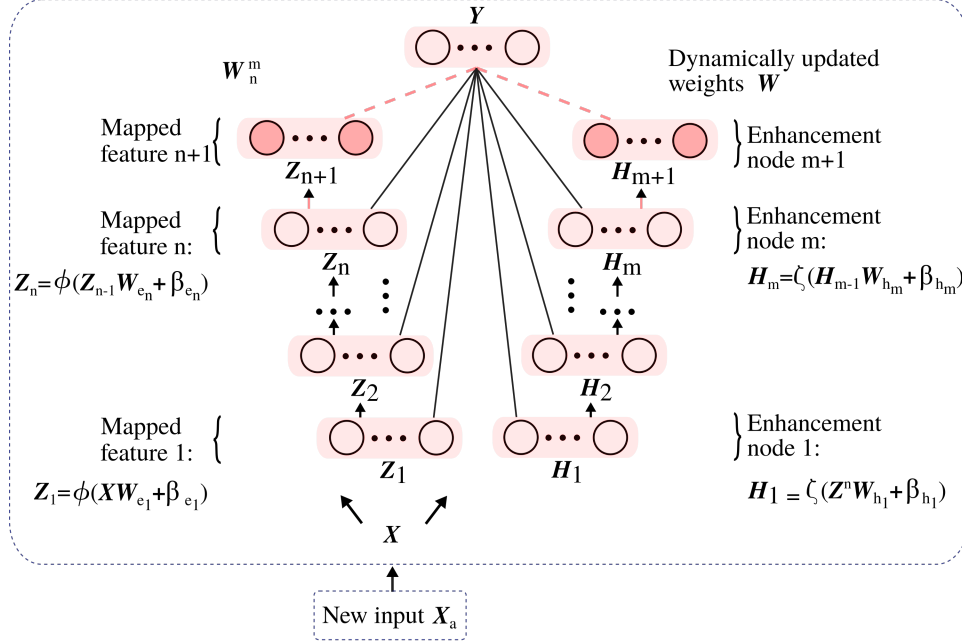


Figure 6.4: Module of the CFEBLS algorithm with increments of mapped features  $Z_{n+1}$ , enhancement nodes  $H_{m+1}$ , and new input data  $X_a$ .

### 6.1.1 BLS and its Extensions: Performance Comparison

We implement and evaluate performance of RBF-BLS and BLS with cascades of mapped features (CFBLS), enhancement nodes (CEBLS), and both mapped features and enhancement nodes (CFEBLS) with and without incremental learning. We perform two-way classification to identify regular (0) and anomalous (1) data using subsets of CICIDS2017 and CSE-CICIDS2018 datasets (Chapter 2, Section 2.4) that include application-layer DoS attacks.

Note that the entire set of features is used when training BLS models. Because using all features may include redundancies and the most relevant features cannot be identified when using BLS, its performance may be enhanced by applying feature selection before training BLS models. We experiment with 78 features and subsets of top-ranked  $2^n$  ( $n = 3, 4, 5$ , and  $6$ ) features to evaluate their effect on BLS performance. Experiments with decision tree, extra-trees, and autoencoder were performed for feature selection and dimension reduction. The best models were generated using the features selected by the extra-trees algorithm. Features are ranked using the extra-trees [145]. We use parameters  $n\_estimators = 100$ ,  $random\_state = 1$ , and default values for the remaining settings.

Performance of BLS models based on the number of selected features is shown in Fig. 6.5. As expected, shorter training time is required for models using fewer number of features

and models based on the incremental BLS. Selecting only 8 features generated accuracy and F-Score over 90 % for CICIDS2017 dataset. No such advantage is observed for the F-Score using the CSE-CIC-IDS2018 dataset. Using 32 features shows comparable results for both datasets while taking shorter training time compared to using 64 or 78 features. Non-incremental and incremental model parameters leading to the best performance are shown in Table 6.1 while accuracy and F-Score are shown in Table 6.2. Most generated models achieve accuracy and F-Score above 96 % and 90 %, respectively. Our past results when applying non-incremental and incremental BLS to BGP anomaly data were comparable [30]. Experiments are performed using a Dell Alienware Aurora with 32 GB memory and Intel Core i7 7700K processor. Results are generated using Python 3.6.

Table 6.1: Best parameters obtained when using CICIDS2017 and CSE-CIC-IDS2018 datasets; Incremental learning: *incremental learning steps* = 2, *enhancement nodes/step* = 20, and *data points/step* = 55,680 (CICIDS2017), 49,320 (CSE-CIC-IDS2018).

Number of features	Dataset	Model	Mapped features	Groups of mapped features	Enhancement nodes
<b>Non-incremental BLS</b>					
<b>32</b>	CICIDS2017	CEBLS	10	10	40
	CSE-CIC-IDS2018	CEBLS	15	20	80
<b>64</b>	CICIDS2017	BLS	10	30	20
	CSE-CIC-IDS2018	RBF-BLS	20	10	80
<b>78</b>	CICIDS2017	RBF-BLS	20	30	40
	CSE-CIC-IDS2018	CFBLS	20	10	80
<b>Incremental BLS</b>					
<b>32</b>	CICIDS2017	CFBLS	10	20	40
	CSE-CIC-IDS2018	BLS	10	20	20
<b>64</b>	CICIDS2017	CFEBLS	20	20	20
	CSE-CIC-IDS2018	CEBLS	20	10	40
<b>78</b>	CICIDS2017	CFBLS	10	20	40
	CSE-CIC-IDS2018	BLS	15	30	20

#### 6.1.1.1 Discussion

We considered malicious intrusions and anomalies in communication networks and evaluated performance of machine learning algorithm such as BLS, RBF-BLS as well as BLS with cascades of mapped features and cascades of enhancement nodes with and without incremental learning. Experiments were conducted using subsets of CICIDS2017 and CSE-CIC-IDS2018 datasets that contained DoS attacks captured from experimental testbeds. The developed models generated comparable performance even when selecting a smaller number of relevant features. The BLS learning models were compared based on accuracy, F-Score, and training time. Larger number of mapped features, groups of mapped features,

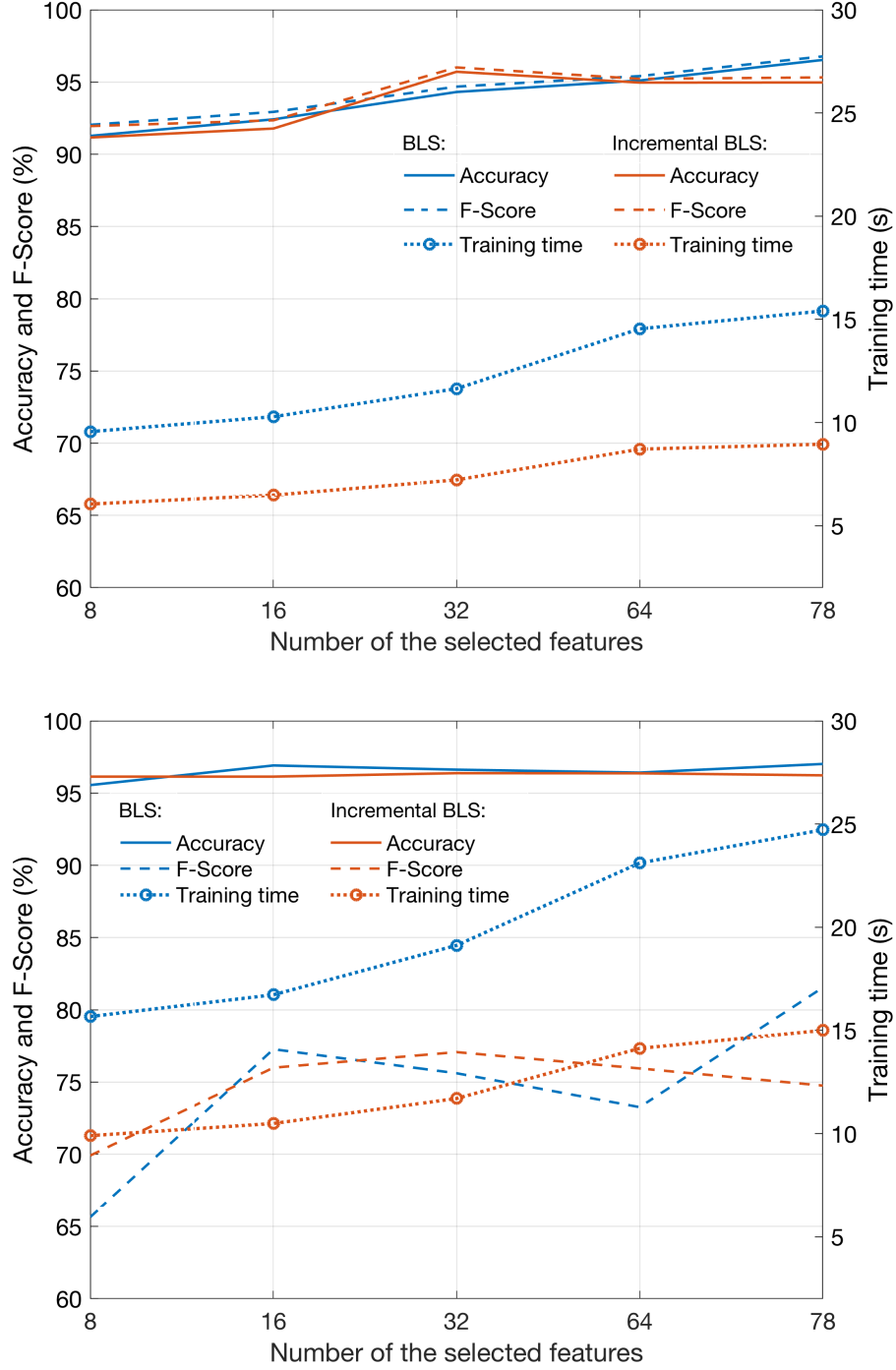


Figure 6.5: Performance of BLS (blue) and incremental BLS (red). Shown are accuracy, F-Score, and training time for 78 features and subsets of  $2^n$  ( $n = 3, 4, 5$ , and  $6$ ) most relevant features using CICIDS2017 (top) and CSE-CIC-IDS2018 (bottom) datasets.

Table 6.2: Best performance for models tested with CICIDS2017 and CSE-CIC-IDS2018 datasets.

Features	Dataset (%)	Accuracy (%)	F-Score	Model (s)	Training time
<b>Non-incremental BLS</b>					
<b>32</b>	CICIDS2017	96.34	96.62	CEBLS	39.25
	CSE-CIC-IDS2018	98.83	92.26	CEBLS	33.46
<b>64</b>	CICIDS2017	96.10	96.35	BLS	8.97
	CSE-CIC-IDS2018	98.60	90.49	RBF-BLS	4.65
<b>78</b>	CICIDS2017	96.63	96.87	RBF-BLS	15.60
	CSE-CIC-IDS2018	97.46	81.46	CFBLS	4.13
<b>Incremental BLS</b>					
<b>32</b>	CICIDS2017	95.39	95.75	CFBLS	6.39
	CSE-CIC-IDS2018	97.08	77.89	BLS	5.65
<b>64</b>	CICIDS2017	94.88	95.38	CFEBLS	7.39
	CSE-CIC-IDS2018	96.70	74.64	CEBLS	11.59
<b>78</b>	CICIDS2017	95.12	95.44	CFBLS	3.69
	CSE-CIC-IDS2018	97.47	81.35	BLS	6.78

and enhancement nodes required additional memory and longer training time. Experimental results indicated that most generated models achieved accuracy and F-Score above 90 %. While CEBLS and CFEBLS models required longer training time, the training time for incremental BLS was significantly shorter than for non-incremental BLS because models did not need to be retrained.

### 6.1.2 BLS and RNNs (LSTM and GRU): Performance Comparison

The performance of BLS and RNN is compared using BGP (Chapter 2, Section 2.2.5) and NSL-KDD (Chapter 2, Section 2.3) datasets. We implement deep learning models using CPU (Intel Core i7 7700K) and GPU (NVIDIA GeForce GTX 1080). For LSTM, GRU, and BLS and its extensions, we generate results using Python. MATLAB implementation [155] of BLS and its extensions is converted to Python code in order to have consistent comparison. The deep learning models are trained using 50 epochs with learning rate  $lr = 0.001$ . We use 10 and 50 data points as input sequences for BGP and NSL-KDD datasets, respectively. In the training process, the model is optimized by using Adam algorithm [168] while “over-fitting” is avoided by employing dropout rate [169] of 50 %. The best results for LSTM<sub>2</sub> and GRU<sub>2</sub> are obtained with 16 fully connected hidden nodes (FC<sub>3</sub>). LSTM<sub>3</sub> and GRU<sub>3</sub> offer the best performance with 32 FC<sub>2</sub> and 16 FC<sub>3</sub> fully connected hidden nodes. The sigmoid function is replaced by the RBF function for enhancement nodes in the RBF-BLS model. We implement the CFBLS, CEBLS, and CFEBLS models by modifying the original BLS functions. The BLS training parameters that generate the best performance results are listed in Table 6.3.

Table 6.3: Number of BLS training parameters.

Parameters	Slammer	Nimda	Code Red	NSL-KDD
Mapped features	100	500	100	100
Groups of mapped features	25	1	1	5
Enhancement nodes	300	700	500	100
Incremental learning steps	2	9	10	3
Data points/step	100	200	100	3000
Enhancement nodes/step	50	10	10	60

We evaluate the performance of the deep learning and broad learning classification models based on accuracy and F-Score. The training times for RNN and BLS models using the BGP and NSL-KDD datasets are shown in Table 6.4. The GRU algorithm has shorter training time than the LSTM algorithm due to its simpler structure. Note that MATLAB employs the optimized function *mapminmax()*, which results in shorter training time for BLS and its extensions. Performance of RNN and BLS models using the BGP datasets is shown in Table 6.5 and Table 6.6, respectively. RNN and BLS models offer variable performance with the best accuracy and F-Score in the range of 90 %-95 %.

The RNNs and BLS performance using NSL-KDD dataset is shown in Table 6.7. The best performance is achieved using the LSTM<sub>4</sub> and GRU<sub>3</sub> RNNs while the CFBLs architecture offers the best results among BLS models. Performance and training times of incremental BLS are shown in Table 6.8. The incremental BLS algorithm requires shorter training time because the weights are updated based on only new data.

#### 6.1.2.1 Discussion

We employed LSTM and GRU deep neural networks with a variable number of hidden layers. We also evaluated performance of BLS models that employ radial basis function (RBF), cascades of mapped features and enhancement nodes, and incremental learning. Compared to deep neural networks, BLS and cascade combinations of mapped features and enhancement nodes achieved comparable performance and shorter training time because of their wide and deep structure. BLS models consist of a small number of hidden layers and adjust weights using pseudo-inverse instead of back-propagation. They also dynamically update weights in case of incremental learning. They better optimized weights due to additional data points for large datasets such as NSL-KDD. While increasing the number of mapped features and enhancement nodes as well as mapped groups led to better performance, it required additional memory and training time.

Table 6.4: Training time for RNN and BLS models: BGP and NSL-KDD datasets.

Model	Datasets	LSTM <sub>2</sub>	LSTM <sub>3</sub>	LSTM <sub>4</sub>	GRU <sub>2</sub>	GRU <sub>3</sub>	GRU <sub>4</sub>	BLS	RBF-BLS	CFBLS	CEBLS	CFEBLS
Time (s)	BGP (Slammer)	224.52	259.91	819.78	54.12	60.76	759.82	21.53	18.68	18.89	32.36	32.13
	NSL-KDD	4481.73	4614.66	11478.62	1108.31	1161.80	11581.30	99.47	98.27	98.13	108.23	108.14
		Python (GPU)				MATLAB (CPU)						
Time (s)	BGP (Slammer)	30.74	34.94	38.82	31.03	35.46	40.22	1.36	1.20	1.03	5.49	5.98
	NSL-KDD	344.93	355.86	394.55	317.53	345.04	369.86	6.91	6.24	6.55	8.88	8.95

Table 6.5: Performance of RNN (LSTM and GRU) models: BGP datasets (Python).

Model	Training Dataset	Test	Accuracy (%)		F-Score (%)
			RIPE (regular)	BCNET (regular)	Test
LSTM <sub>2</sub>	Slammer	92.98	92.99	85.97	72.42
	Nimda	78.36	47.15	48.61	87.87
	Code Red	94.08	83.75	60.49	68.89
LSTM <sub>3</sub>	Slammer	90.90	92.01	84.38	67.29
	Nimda	85.57	39.10	40.28	92.22
	Code Red	88.54	79.38	58.82	55.96
LSTM <sub>4</sub>	Slammer	92.49	92.22	86.18	70.72
	Nimda	92.00	26.94	35.21	95.83
	Code Red	86.96	75.00	57.01	51.53
GRU <sub>2</sub>	Slammer	91.88	93.33	90.90	69.42
	Nimda	70.71	48.96	58.26	82.83
	Code Red	87.47	80.07	60.21	52.97
GRU <sub>3</sub>	Slammer	91.76	95.21	90.83	68.72
	Nimda	80.21	38.40	44.24	89.02
	Code Red	88.07	79.44	60.56	53.51
GRU <sub>4</sub>	Slammer	92.14	92.15	90.35	70.11
	Nimda	87.36	35.00	39.38	93.25
	Code Red	91.84	77.50	60.07	63.87

Table 6.6: Performance of BLS Model and its extensions: BGP datasets (Python).

Model	Training Dataset	Test	Accuracy (%)		F-Score (%)
			RIPE (regular)	BCNET (regular)	Test
BLS	Slammer	87.65	75.62	68.40	57.68
	Nimda	76.57	70.69	54.93	86.73
	Code Red	94.97	69.79	65.21	66.38
RBF-BLS	Slammer	91.21	90.55	70.76	64.57
	Nimda	57.92	70.63	57.22	73.36
	Code Red	95.92	90.69	73.96	70.07
CFBLS	Slammer	89.28	71.25	61.81	60.99
	Nimda	55.71	68.06	58.26	71.56
	Code Red	95.16	69.38	61.74	71.08
CEBLS	Slammer	91.01	87.71	82.43	66.38
	Nimda	66.43	74.10	54.51	79.83
	Code Red	94.94	70.69	60.35	65.22
CFEBLS	Slammer	86.36	71.11	57.71	55.30
	Nimda	64.29	70.83	57.43	78.24
	Code Red	95.66	70.07	59.51	71.75

Table 6.7: Performance of RNN (LSTM and GRU) and BLS models:  
NSL-KDD datasets (Python).

<b>Model</b>	<b>Accuracy (%)</b>		<b>F-Score (%)</b>	
	KDDTest <sup>+</sup>	KDDTest <sup>-21</sup>	KDDTest <sup>+</sup>	KDDTest <sup>-21</sup>
LSTM <sub>4</sub>	82.78	66.74	83.34	76.21
GRU <sub>3</sub>	82.87	65.42	83.05	74.06
CFBLS	82.20	67.47	82.23	76.29

Table 6.8: Performance and training time of incremental BLS model:  
BGP and NSL-KDD datasets (MATLAB).

<b>Test</b>	<b>Accuracy (%)</b>	<b>F-Score (%)</b>	<b>Time (s)</b>
Slammer	89.31	63.07	2.805
Nimda	91.64	95.64	2.757
Code Red	94.37	65.10	0.926
KDDTest <sup>+</sup>	81.34	81.99	32.99
KDDTest <sup>-21</sup>	78.70	88.06	29.71

## 6.2 Gradient Boosting Decision Tree Algorithms

Boosting algorithms, a class of ensemble learning, are greedy algorithms that sequentially include estimators (base learners) to enhance the model performance [11]. The goal is to minimize the loss function by including estimators that are trained based on residuals. The forward stage-wise additive modeling is used to generate boosting models. The number of training iterations is equivalent to the number of estimators because a new estimator is added to the boosting model in each iteration. Boosting models employ loss functions such as squared error, absolute error, exponential loss, or log-loss.

The GBMs [170] are boosting algorithms that employ functional gradient descent to minimize the loss function. GBDT is a GBM variant that employs decision trees as estimators. Optimized GBDT algorithms include XGBoost [156], LightGBM [12], and CatBoost [157].

When training a GBDT model [11, 156] with  $K$  estimators using  $N$  data points, the predicted output is:

$$\hat{y}_i = \sum_{k=1}^K f_k(\mathbf{x}_i), \quad (6.6)$$

where  $f_k$  is the  $k^{th}$  decision tree and  $\mathbf{x}_i$  is the  $i^{th}$  data point. Note that in the experiments,  $\mathbf{x}_i$  is a row vector of matrix  $\mathbf{X}$  containing input data and represents one collection sample. In the  $k^{th}$  iteration, predicted output is evaluated using the  $k^{th}$  decision tree (estimator):

$$\hat{y}_i^{(k)} = \hat{y}_i^{(k-1)} + f_k(\mathbf{x}_i), \quad (6.7)$$

where  $\hat{y}_i^{(k)}$  is the predicted output of the  $i^{th}$  data point and  $\hat{y}_i^{(k-1)}$  is the previously predicted output. The goal of the GBDT models is to minimize the objective function:

$$\begin{aligned}\mathcal{L}^{(k)} &= \sum_{i=1}^N l(y_i, \hat{y}_i^{(k)}) + \Omega(f_k) \\ &= \sum_{i=1}^N l(y_i, \hat{y}_i^{(k-1)} + f_k(\mathbf{x}_i)) + \Omega(f_k),\end{aligned}\tag{6.8}$$

where  $l(\cdot)$  is the loss function,  $y_i$  is the label of the  $i^{th}$  input data point, and  $\Omega(f_k)$  (optional) is the regularization term.

### 6.2.1 XGBoost

GBDT may be improved by adding an  $L^2$  norm regularization term to avoid over-fitting. XGBoost [156] employs the second-order Taylor series to approximate its objective function and a sparsity-aware algorithm to deal with the sparse data. A cache-aware block structure is used to generate the XGBoost model on parallel and distributed computing and to increase training speed.

The regularization function is:

$$\Omega(f_k) = \gamma T + \frac{1}{2} \lambda \|\omega\|^2, \tag{6.9}$$

where  $\gamma$  and  $\lambda$  are the regularization coefficients,  $T$  is the number of leaves in the tree, and  $\omega$  are the leaf weights.

The second-order Taylor series is used to approximate (6.8):

$$\mathcal{L}^{(k)} \simeq \sum_{i=1}^N \left[ l(y_i, \hat{y}_i^{(k-1)}) + g_i f_k(\mathbf{x}_i) + \frac{1}{2} h_i f_k^2(\mathbf{x}_i) \right] + \Omega(f_k), \tag{6.10}$$

where  $g_i = \frac{\partial l(y_i, \hat{y}_i^{(k-1)})}{\partial \hat{y}_i^{(k-1)}}$  and  $h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(k-1)})}{\partial (\hat{y}_i^{(k-1)})^2}$  are known and  $l(y_i, \hat{y}_i^{(k-1)})$  is a constant.

By removing the constant term  $l(y_i, \hat{y}_i^{(k-1)})$  from (6.10), and substituting (6.9) into (6.10), the objective function is expressed as:

$$\mathcal{L}^{(k)} = \sum_{t=1}^T \left[ \left( \sum_{i \in I_t} g_i \right) \omega_t + \frac{1}{2} \left( \sum_{i \in I_t} h_i + \lambda \right) \omega_t^2 \right] + \gamma T, \tag{6.11}$$

where  $\omega_t$  is the weight of each leaf  $t$  and  $q(\mathbf{x}_i)$  is the tree structure for  $i^{th}$  data point. The terms  $\sum_{i \in I_t} g_i$  and  $(\sum_{i \in I_t} h_i + \lambda)$  are known. For a known tree structure  $q(\mathbf{X})$ ,  $I_t$  is a set containing the indices of data points in leaf  $t$ . Setting the derivative of (6.11) to zero gives

the optimal weight  $\omega_t^*$  for leaf  $t$ :

$$\omega_t^* = -\frac{\sum_{i \in I_t} g_i}{\sum_{i \in I_t} h_i + \lambda}. \quad (6.12)$$

The optimal solution of the objective function is:

$$\mathcal{L}^{*(k)} = -\frac{1}{2} \sum_{t=1}^T \frac{(\sum_{i \in I_t} g_i)^2}{\sum_{i \in I_t} h_i + \lambda} + \gamma T. \quad (6.13)$$

This optimal value is used to evaluate the quality of a tree structure  $q(\mathbf{X})$ . The tree structure with the lowest optimal value is selected for each iteration.

## 6.2.2 LightGBM

Gradient-Based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB) techniques are employed to significantly accelerate the training speed. LightGBM [12] achieves performance comparable to XGBoost albeit with lower memory usage.

LightGBM employs the histogram-based algorithm that accelerates the process to locate the best splitting point for each feature. Using the training data points, mutually exclusive features are bundled to create feature histograms. GOSS involves sorting the training data points in a descending order based on the absolute value of their gradients. Top  $N_t$  data points (subset  $\mathbf{A}$ ) with the largest gradients are selected and random sampling of the remaining input data points is performed to create a subset  $\mathbf{B}$ . The dimensions of  $\mathbf{A}$  and  $\mathbf{B}$  depend on predefined sampling ratios  $a$  and  $b$ , respectively. When training a GBDT model with a given dataset, gradients are calculated in each iteration.

In a decision tree, nodes are split based on features with the largest information gain that depends on the variance gain  $\tilde{V}_j(d)$  for feature  $j$  computed after splitting as [12]:

$$\begin{aligned} \tilde{V}_j(d) = & \frac{1}{N \times N_l^j(d)} \left( \sum_{\mathbf{x}_i \in \mathbf{A}_l} g_i + \frac{1-a}{b} \sum_{\mathbf{x}_i \in \mathbf{B}_l} g_i \right)^2 \\ & + \frac{1}{N \times N_r^j(d)} \left( \sum_{\mathbf{x}_i \in \mathbf{A}_r} g_i + \frac{1-a}{b} \sum_{\mathbf{x}_i \in \mathbf{B}_r} g_i \right)^2, \end{aligned} \quad (6.14)$$

where  $d$  is the splitting point,  $N$  is the number of data points,  $N_l^j$  and  $N_r^j$  are number of input data points related to left and right child nodes, respectively, and  $g_i$  is the gradient for input data point  $\mathbf{x}_i$ . The sampling ratios  $a$  and  $b$  are used to calculate the normalization coefficient  $(1-a)/b$ .  $\mathbf{A}_l$  ( $\mathbf{B}_l$ ) and  $\mathbf{A}_r$  ( $\mathbf{B}_r$ ) are the subsets of  $\mathbf{A}$  ( $\mathbf{B}$ ) for the left and right child nodes, respectively.

LightGBM is based on the GOSS technique and utilizes leaf-wise growth approach to grow the decision trees instead of level-wise growth used in XGBoost. Level-wise growth splits the leaves of the same layer, which enables easy control of the model complexity.

However, it introduces unnecessary overhead because the leaves with low variance gains are also split. Compared to level-wise tree growth, leaf-wise is a more efficient approach because it splits the leaf that has the maximum variance gain thus reducing additional loss after a number of splits. Furthermore, it may lead to deeper decision trees resulting in over-fitting. Hence, the hyper-parameter “max depth” is introduced to limit their depth.

The EFB technique combines dataset features in order to reduce the dimension of the input data and the complexity of building the histogram from  $\mathcal{O}(n_d n_f)$  to  $\mathcal{O}(n_d n_b)$ , where  $n_d$ ,  $n_f$ , and  $n_b$  are the number of data points, features, and bundles, respectively.

### 6.2.3 CatBoost

The XGBoost algorithm only accepts numerical values and employs one-hot encoding to convert categorical features to numerical values while LightGBM converts these features to gradient statistics. Hence, CatBoost [157] is introduced to deal with categorical features. It employs the ordered boosting algorithm and offers an effective approach (ordered target statistic) when compared to XGBoost and LightGBM. Target statistic was used to convert categorical to numerical features by using the values that estimate the expected labels based on the categories while keeping the dimension of the dataset unchanged.

In the existing GBDT models, residuals are calculated in each iteration and are used as the target values in the next training iteration. This leads to bias increase and prediction shift in subsequent iterations and, thus, model over-fitting. Hence, ordered boosting was proposed to address the prediction shift when building the decision trees during the training process. It performs permutation and trains multiple decision trees in each iteration. Each residual is calculated based on the target and predicted values generated by the previous decision tree. Symmetric (oblivious) decision trees are used to avoid over-fitting and reduce the time required to grow the tree. CatBoost offers plain and ordered boosting modes with target statistics and ordered boosting, respectively. In each iteration, the two boosting modes have the same asymptotic complexity for calculating gradients  $\mathcal{O}(sN)$ , updating decision trees  $\mathcal{O}(sN)$ , and computing ordered target statistic  $\mathcal{O}(N_{TS}N)$ , where  $s$ ,  $N$ , and  $N_{TS}$  are the number of permutations, data points, and features using target statistics, respectively.

## 6.3 BLS and GBDT: Performance Comparison

Five BLS algorithms with and without incremental learning and three GBDT algorithms are implemented for two-way classification of regular and anomalous data points. The CIC datasets are used to compare performance of algorithms based on training time, accuracy, F-Score, precision, sensitivity, and confusion matrix: TP, FP, TN, and FN.

We use subsets of the CIC datasets to create training and test datasets with 78, 78, and 79 most relevant features for the CICIDS2017, CSE-CIC-IDS2018, and CICDDoS2019

datasets, respectively. (The CICDDoS2019 dataset includes an additional feature that indicates the traffic direction.) Training and test datasets are partitioned based on the 60 % and 40 % content of the anomalous data points, respectively. The experiments (cross-validation and testing) are conducted using a supercomputer managed by Compute Canada. The Cedar [171] cluster consists of 94,528 CPU cores. We requested 64 GB memory and an Intel E5-2683 v4 Broadwell (2.1 GHz) processor with 8 cores and used Python 3.6.10 and its libraries [22]: *NumPy*, *pandas*, *scikit-learn*, *XGBoost*, *LightGBM*, and *CatBoost*.

The partitioning process for training and validation datasets for the 10-fold cross-validation based on the time series split is illustrated in Fig. 6.6. In each fold, 25,000 data points are used as the validation dataset. In the first step (Fold 1), 25,000 data points are used for training. The training dataset in each subsequent fold is the concatenation of the previous training and validation datasets.

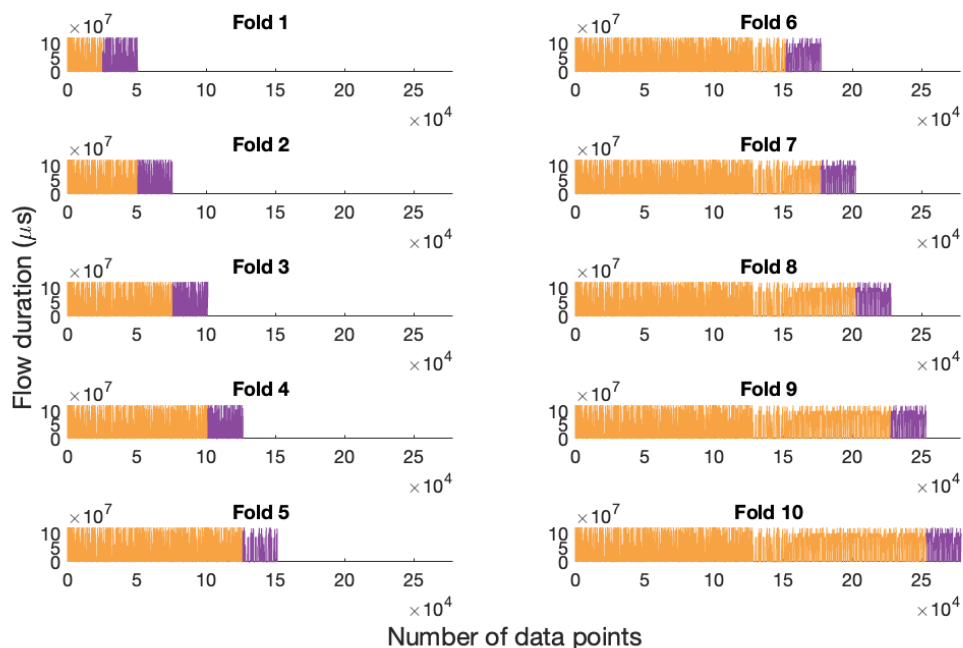


Figure 6.6: Time series split for the 10-fold cross-validation of the CICIDS2017 training dataset. Illustrated are the generations of training (orange) and validation (purple) datasets.

Training hyper-parameters of the BLS and GBDT models that generate the best performance results are listed in Table 6.9 and Table 6.10, respectively. In our experiments,  $\phi(\cdot)$  (6.1) and  $\xi(\cdot)$  (6.2) are *linear* and *tanh* mappings, respectively. Additional hyper-parameters for incremental BLS are: *incremental learning steps* = 2, *enhancement nodes/step* = 20 (CICIDS2017, CSE-CIC-IDS2018) and 10 (CICDDoS2019), and *data points/step* = 55,680 (CICIDS2017), 49,320 (CSE-CIC-IDS2018), and 382,929 (CICDDoS2019). Additional hyper-parameters for GBDT algorithms are: *maximum depth in a tree* = 6 (XGBoost,

CatBoost), *maximum number of leaves* = 31 (LightGBM, CatBoost), and *loss function* = log-loss. We implement *gbtree* (XGBoost), *gbdt* (LightGBM), and *Plain* (CatBoost) boosting modes. Classification results for the BLS and GBDT models are shown in Table 6.11 and Table 6.12, respectively.

Table 6.9: BLS and incremental BLS hyper-parameters leading to the best performance: CIC datasets.

Model	Dataset	Mapped features	Groups of mapped features	Enhancement nodes
<b>BLS</b>				
RBF-BLS	CICIDS2017	20	30	40
CFBLS	CSE-CIC-IDS2018	20	10	80
BLS	CICDDoS2019	15	5	20
<b>Incremental BLS</b>				
CFBLS	CICIDS2017	10	20	40
BLS	CSE-CIC-IDS2018	15	30	20
CFEBS	CICDDoS2019	20	5	10

Table 6.10: XGBoost, LightGBM, and CatBoost hyper-parameters leading to the best performance: CIC datasets.

Model	Dataset	Number of estimators	Learning rate
XGBoost	CICIDS2017	100	0.01
	CSE-CIC-IDS2018	100	0.01
	CICDDoS2019	20	0.01
LightGBM	CICIDS2017	200	0.10
	CSE-CIC-IDS2018	150	0.02
	CICDDoS2019	20	0.05
CatBoost	CICIDS2017	150	0.10
	CSE-CIC-IDS2018	150	0.01
	CICDDoS2019	20	0.01

### 6.3.1 Effect of Hyper-Parameters on Algorithm Performance

Performance of BLS models depends on the number of mapped features, groups of mapped features, and enhancement nodes. As the value of these hyper-parameters increases, the models require additional memory and longer training time. Selecting more than 30 mapped features, 30 groups of mapped features, and 100 enhancement nodes requires over 64 GB of the supercomputer memory. Calculation of ridge regression further exacerbates the memory requirements. Hence, the range of hyper-parameters has been selected based on memory consumption. Performance (accuracy and F-Score) of BLS models does not monotonically depend on the choice of hyper-parameters. Thus, we rely on the 10-fold cross-validation to

Table 6.11: The best performance of BLS and incremental BLS models: CIC datasets.

Model	Dataset	Training time (s)	Accuracy (%)	F-Score (%)	Precision (%)	Sensitivity (%)	TP	FP	TN	FN
<b>BLS</b>										
RBF-BLS	CICIDS2017	37.72	96.63	96.87	97.57	96.18	96,832	2,416	82,511	3,841
CFBLS	CSE-CIC-IDS2018	17.04	97.46	81.46	98.26	69.56	14,597	258	240,057	6,388
BLS	CICDDoS2019	46.64	99.98	99.99	99.99	99.99	2,541,553	204	954	220
<b>Incremental BLS</b>										
CFBLS	CICIDS2017	17.60	95.12	95.44	96.73	94.19	94,827	3,206	81,721	5,846
BLS	CSE-CIC-IDS2018	38.09	97.47	81.35	99.51	68.80	14,437	71	240,244	6,548
CFEBLS	CICDDoS2019	79.01	99.97	99.99	99.97	99.99	2,541,764	646	512	9

Table 6.12: The best performance of XGBoost, LightGBM, and CatBoost models: CIC datasets.

Model	Dataset	Training time (s)	Accuracy (%)	F-Score (%)	Precision (%)	Sensitivity (%)	TP	FP	TN	FN
XGBoost	CICIDS2017	24.49	98.62	98.72	99.43	98.02	98,684	568	84,359	1,989
	CSE-CIC-IDS2018	14.43	99.90	99.39	99.99	98.79	20,731	1	240,314	254
	CICDDoS2019	62.99	99.99	99.99	99.99	99.99	2,541,767	7	1,151	6
LightGBM	CICIDS2017	3.35	97.93	98.06	99.94	96.25	96,896	60	84,867	3,777
	CSE-CIC-IDS2018	1.73	98.73	91.44	99.99	84.23	17,675	1	240,314	3,310
	CICDDoS2019	8.12	99.99	99.99	99.99	99.99	2,541,767	8	1,150	6
CatBoost	CICIDS2017	20.27	98.01	98.13	99.91	96.41	97,056	83	84,844	3,617
	CSE-CIC-IDS2018	19.03	99.95	99.72	99.97	99.46	20,872	6	240,309	113
	CICDDoS2019	17.38	99.99	99.99	99.99	99.99	2,541,762	19	1,139	11

select hyper-parameters leading to the highest average accuracy of the validated models. The BLS models trained with the CICDDoS2019 dataset consist of fewer number of mapped features (5) and enhancement nodes (up to 20) compared to models for the CICIDS2017 and CSE-CIC-IDS2018 datasets, without affecting their performance.

The number of estimators and the learning rate are the main hyper-parameters for the GBDT algorithms. The log-loss function was monitored for various hyper-parameters in order to observe its behavior. A range of hyper-parameters was selected around small values of log-loss function to proceed with cross-validation. For each combination of parameters, the 10-fold cross-validation is used to select hyper-parameters leading to the highest average accuracy of the models. Generated GBDT models using the CICDDoS2019 dataset have fewer number of estimators compared to models used with the CICIDS2017 and CSE-CIC-IDS2018 datasets. The largest number of estimators is employed to generate the LightGBM model using the CICIDS2017 dataset.

LightGBM models offer the shortest training time for all considered datasets. Their training time is approximately 20 times shorter than the BLS, XGBoost, and CatBoost models that have comparable training times. The GBDT models outperform original and incremental BLS models using the CICIDS2017 and CSE-CIC-IDS2018 datasets. The best accuracy and F-Score for XGBoost and CatBoost models are obtained for the CICIDS2017 and CSE-CIC-IDS2018 datasets, respectively. The XGBoost and CatBoost models generate the lowest number of FNs for the CICIDS2017 and CSE-CIC-IDS2018 datasets, respectively. The BLS and GBDT models using the CICDDoS2019 dataset have similar (two decimal points) and very high accuracy, F-Score, precision, and sensitivity because the regular class has few data points. Hence, the confusion matrix is used for comparison. The XGBoost and LightGBM models generate the lowest number of misclassified anomalous (FN) and regular (FP) data points when using the CICDDoS2019 dataset.

### 6.3.2 Discussion

We compared performance of BLS and GBDT supervised machine learning algorithms using three CIC datasets. Performance metrics such as training time, accuracy, F-Score, precision, sensitivity, and confusion matrix were used. Training time for the BLS models depended on the number of mapped features, groups of mapped features, and enhancement nodes while time for GBDT models was affected by the number of estimators, learning rate as well as the maximum depth and number of leaves in the decision trees. XGBoost, LightGBM, and CatBoost offered better accuracy and F-Score than BLS models. The shortest training time was required for LightGBM models. In case of similar results (accuracy, F-Score, precision, and sensitivity), we used the confusion matrix to further compare the classification performance. The experiments illustrated advantages of GBDT algorithms when detecting DoS and DDoS attacks.

## Chapter 7

# Variable Features Broad Learning Systems

In this Chapter, we introduce two new algorithms named Variable Features Broad Learning Systems (VFBL and VCFBL). We compare their performance with LightGBM, RNNs, Bi-RNNs, BLS (including its extensions with and without incremental learning) using the BGP RIPE (Slammer, Nimda, and Code Red), NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets.

### 7.1 VFBL and VCFBL Algorithms

Mapping the input data to sets of mapped features is an essential step of BLS algorithms. We propose a variable features system (VFBL) and system with cascades (VCFBL) shown in Fig. 7.1 and Fig. 7.2. that consist of a variable number of mapped features and groups of mapped features. They employ feature selection algorithms that enable models to be trained based on a variable number of features extracted from the input data. The two algorithms also offer variants with incremental learning as shown in and Fig. 7.3 and Fig. 7.4.

The VFBL and VCFBL algorithms expand the BLS network by using both original and subsets of input data as well as sets of groups of mapped features. A set consists of predefined groups of mapped features. Each mapped feature contains a predefined number of nodes. Each mapped feature within a set contains equal number of nodes. The VFBL and VCFBL algorithms enable developing more generalized models and, hence, prevent overfitting and enhance the performance. Generating the best BLS and incremental BLS models is rather time-consuming because they rely on multiple two-stage experiments: selecting features and generating models. In contrast, VFBL and VCFBL models are developed using a single experiment with integrated stages. A variable number of mapped features is used to reduce training time because the proposed algorithms introduce additional complexity by using the entire input dataset and by incorporating a feature selection algorithm and additional sets of mapped features. In case of incremental learning, features are selected in

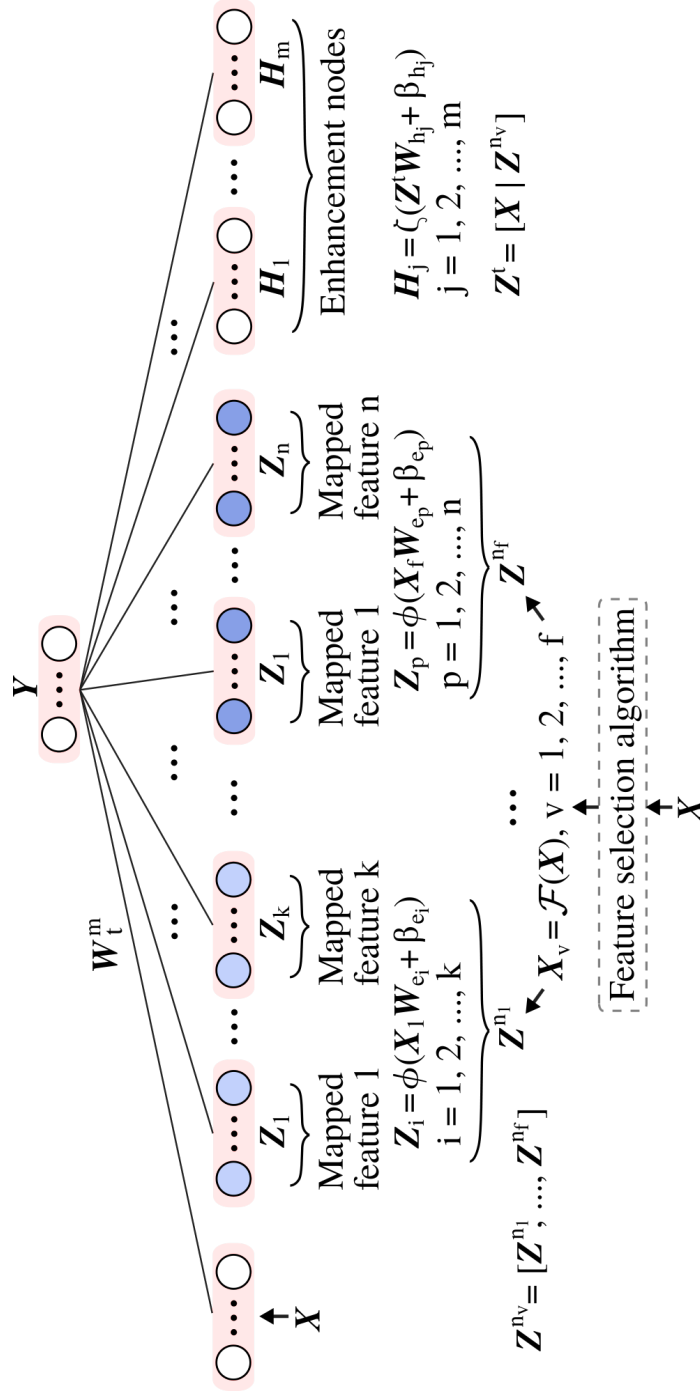


Figure 7.1: VFBL algorithm with input data  $\mathbf{X}$ , sets of groups of mapped features  $(\mathbf{Z}^{n_1}, \dots, \mathbf{Z}^{n_f})$ , and enhancement nodes  $(\mathbf{H}_1, \dots, \mathbf{H}_m)$ .

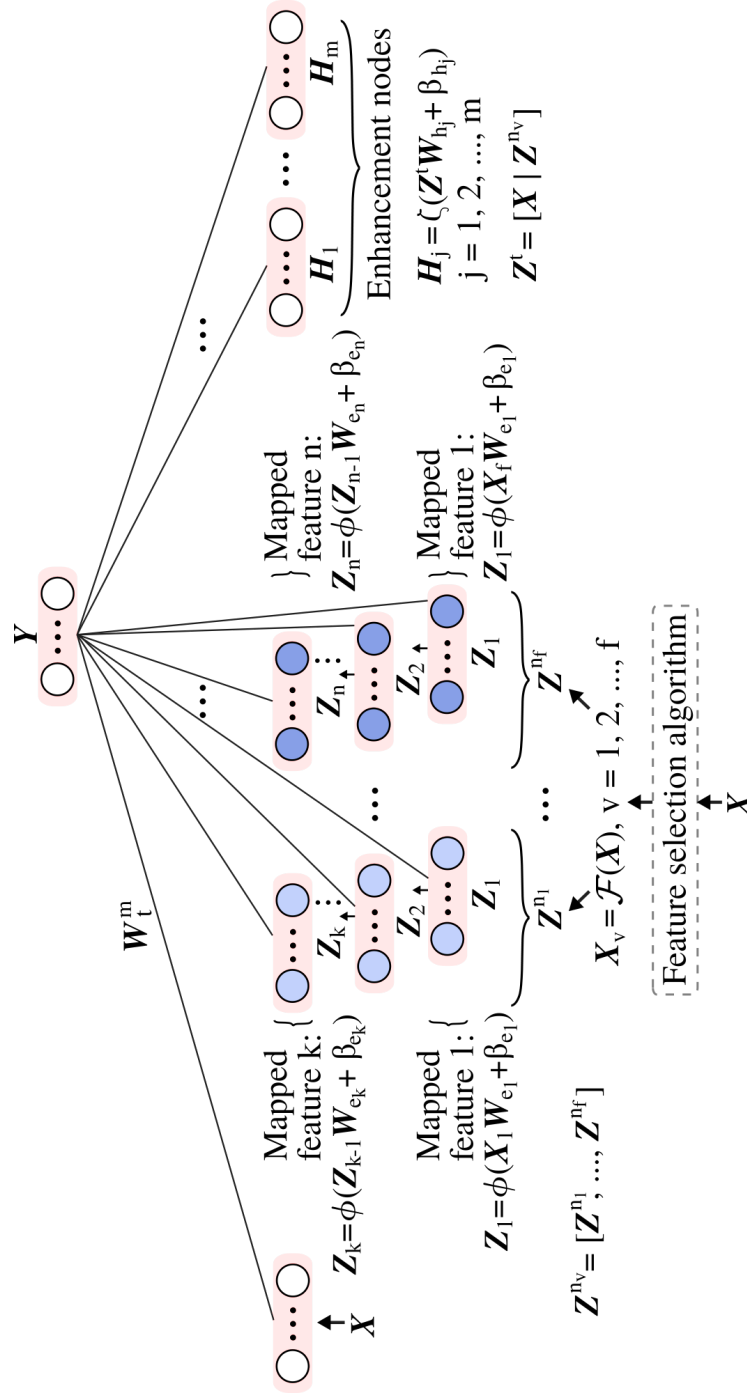


Figure 7.2: VCFBLS algorithm with input data  $\mathbf{X}$ , sets of groups of mapped features with cascades  $(\mathbf{Z}^{n_1}, \dots, \mathbf{Z}^{n_f})$ , and enhancement nodes  $(\mathbf{H}_1, \dots, \mathbf{H}_m)$ .

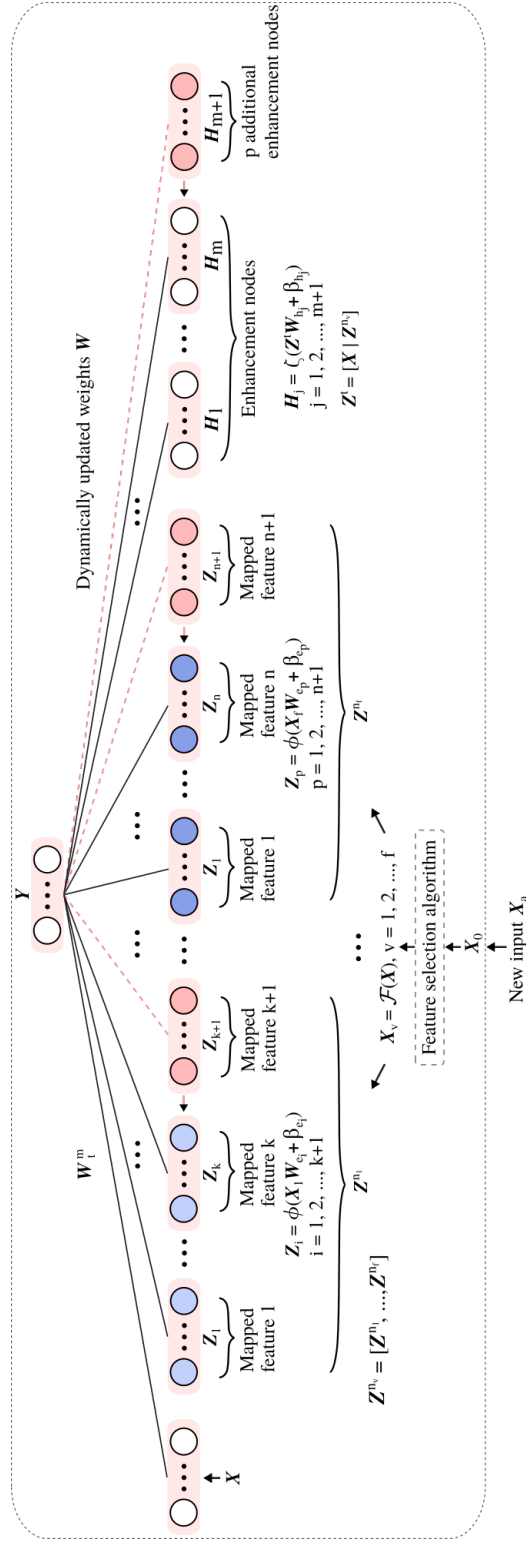
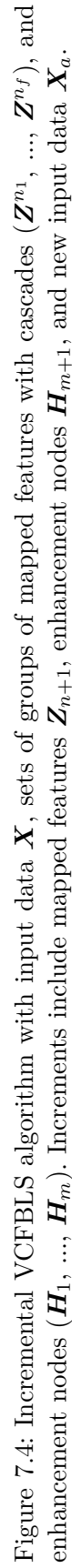


Figure 7.3: Incremental VFBL algorithm with input data  $X$ , sets of groups of mapped features without cascades ( $Z^{n_1}, \dots, Z^{n_f}$ ), and enhancement nodes ( $H_1, \dots, H_m$ ). Increments include mapped features  $Z_{n+1}$ , enhancement nodes  $H_{m+1}$ , and new input data  $X_a$ .



each step and ranked based on their importance. After the last step, all selected features are multiplied with weights proportional to the size of the dataset used in each step. They are then ranked and used for testing.

Data  $\mathbf{X}$  and features that are selected based on a feature selection algorithm are used as input:

$$\mathbf{X}_v = \mathcal{F}(\mathbf{X}), v = 1, 2, \dots, f, \quad (7.1)$$

where  $\mathbf{X}_v$  is a subset of  $\mathbf{X}$  with a selected set of features. Note that selecting all features in the input data to generate mapped features is a special case where  $\mathbf{X}_v = \mathbf{X}$ . Sets of groups of mapped features are generated as:

$$\mathbf{Z}^{n_v} \triangleq [\mathbf{Z}^{n_1}, \dots, \mathbf{Z}^{n_f}], \quad (7.2)$$

where  $\mathbf{Z}^{n_v}$  contains  $n_v$  mapped features. In the case of VFBLs groups of mapped features  $\mathbf{Z}^{n_v}$ ,  $\mathbf{X}_v$  and  $n_v$  correspond to  $\mathbf{X}$  and  $n$ , respectively (6.1) while VCFBLs groups of mapped features  $\mathbf{Z}^{n_v}$  are defined based on the previous group (6.4). Note that the first mapped feature  $\mathbf{Z}_1$  in each set is created from  $\mathbf{X}_1, \dots, \mathbf{X}_f$ . The number of mapped features and groups of mapped features may vary in sets  $\mathbf{Z}^{n_1}, \dots, \mathbf{Z}^{n_f}$ . The number of mapped features in each group of a set is constant. We improve performance by including properties of the original data and concatenating input data  $\mathbf{X}$  and sets of mapped features to create  $\mathbf{Z}^t$  similar to the case of random vector functional-link network [167]:

$$\mathbf{Z}^t = [\mathbf{X} | \mathbf{Z}^{n_v}]. \quad (7.3)$$

The enhancement nodes are:

$$\mathbf{H}_j = \xi(\mathbf{Z}^t \mathbf{W}_{h_j} + \beta_{h_j}), j = 1, 2, \dots, m. \quad (7.4)$$

The state matrix  $\mathbf{A}_t^m$  is the concatenation of  $\mathbf{Z}^t$  and  $\mathbf{H}^m$ . The ridge regression algorithm is then employed to compute the weights  $\mathbf{W}_t^m$  based on  $\mathbf{A}_t^m$  and given labels  $\mathbf{Y}$ . The error function, minimized during the training process, is defined as [4]:

$$\mathbf{E}(\mathbf{W}_t^m) = \|\mathbf{A}_t^m \mathbf{W}_t^m - \mathbf{Y}\|_2^2 + \lambda \|\mathbf{W}_t^m\|_2^2, \quad (7.5)$$

where  $\lambda$  is the sparse regularization coefficient. The term  $\lambda \|\mathbf{W}_t^m\|_2^2$  is used to control overfitting. The minimum of (7.5) can be found in closed-form by setting its derivative with respect to  $\mathbf{W}_t^m$  to 0. The output weights are defined as [3]:

$$\mathbf{W}_t^m = (\lambda \mathbf{I} + (\mathbf{A}_t^m)^T \mathbf{A}_t^m)^{-1} (\mathbf{A}_t^m)^T \mathbf{Y}. \quad (7.6)$$

Pseudocode for the VFBL and VCFBL algorithms and their incremental versions are listed in Algorithm 1 and Algorithm 2, respectively.

---

**Algorithm 1** VFBL and VCFBL algorithms: Pseudocode

---

```

1: procedure VFBL AND VCFBL
   (training dataset  $\mathbf{X}$  with labels  $\mathbf{Y}$ )
2:   Initialize:
3:   Number of subsets  $v$  of input data
4:   Number  $f$  of features to be selected in each subset
5:   Subsets of input data  $\mathbf{X}_v$ ,  $v = (1, \dots, f)$ 
6:   Sets of groups of mapped features  $\mathbf{Z}^{n_v}$ ,  $n_v = (n_1, \dots, n_f)$ 
7:   Groups of mapped features in each set  $\mathbf{Z}_i$ ,  $i = (k, \dots, p)$ 
8:   Number of mapped features in each group
9:   for each  $f$  in  $v$  do
10:    Calculate feature importance and create  $\mathcal{F}(\mathbf{X})$  by ranking
        features using a feature selection algorithm
11:    Generate subset  $\mathbf{X}_v = \mathcal{F}(\mathbf{X})$ ,  $v = 1, 2, \dots, f$ 
12:    for each set  $\mathbf{Z}^{n_i}$  in  $\mathbf{Z}^{n_v}$  do
13:      Initialize the set of groups of mapped features  $\mathbf{Z}^{n_i}$ 
14:      for each group  $\mathbf{Z}_i$  in set  $\mathbf{Z}^{n_i}$  do
15:        switch Algorithm do
16:          case VFBL
17:            Generate  $\mathbf{Z}_i$  based on  $\mathbf{X}_v$  and the number
              of mapped features
18:          case VCFBL
19:            Generate  $\mathbf{Z}_1$  based on  $\mathbf{X}_v$  and the number
              of mapped features
20:            Generate subsequent groups  $\mathbf{Z}_i$  based on
               $\mathbf{Z}_{i-1}$ 
21:            Insert  $\mathbf{Z}_i$  into  $\mathbf{Z}^{n_i}$ 
22:          end for
23:          Insert  $\mathbf{Z}^{n_i}$  into  $\mathbf{Z}^{n_v}$ 
24:        end for
25:      end for
26:    Construct matrix  $\mathbf{Z}^t = [\mathbf{X}|\mathbf{Z}^{n_v}]$ 
27:    Generate enhancement nodes  $\mathbf{H}^m = [\mathbf{H}_1, \dots, \mathbf{H}_m]$  based on
         $\mathbf{Z}^t$ 
28:    Concatenate  $\mathbf{Z}^t$  and  $\mathbf{H}^m$  to create the state matrix  $\mathbf{A}_t^m$ 
29:    Compute weights  $\mathbf{W}_t^m$  based on  $\mathbf{A}_t^m$  and labels  $\mathbf{Y}$  using
        the ridge regression algorithm
30: end procedure

```

---

A variety of feature selection algorithms may be employed. The autoencoder was employed to reduce the data dimension in case of VFBL and VCFBL algorithms. However, several autoencoders had to be trained in order to obtain low dimensional input data. Since this proved to be a time consuming task, the extra-trees algorithm [145] is used to rank features based on Gini importance thus leading to shorter execution time. The algorithm is an improved version of the decision tree and random forests used to select relevant features for generating subsets of the input data. It introduces additional randomness by randomly

---

**Algorithm 2** Incremental VFBLs and VCFBLs algorithms: Pseudocode

---

```
1: procedure INCREMENTAL VFBLs AND VCFBLs
   (training dataset  $\mathbf{X}$  with labels  $\mathbf{Y}$ )
2:   Extract initial input subset  $\mathbf{X}_0$  from dataset  $\mathbf{X}$ 
3:   Extract initial labels subset  $\mathbf{Y}_0$  from  $\mathbf{Y}$ 
4:   Initialize:
5:   Number of incremental learning steps:  $l$ 
6:   Number of data points per step:  $d$ 
7:   Number of enhancement nodes per step:  $e$ 
8:   Calculate feature weight vector  $\mathbf{W}_i = [w_0, w_1, \dots, w_l]$ :
    $w_0 = \mathbf{X}_0 / \mathbf{X}$ ;  $w_1, \dots, w_l = (1 - w_0) / l$ 
9:   for each step in  $l$  do
10:    Generate  $\mathbf{X}_a$  based on  $\mathbf{X}$ ,  $\mathbf{X}_0$ , and  $d$ 
11:    Generate  $\mathbf{Y}_a$  based on  $\mathbf{Y}$ ,  $\mathbf{Y}_0$ , and  $d$ 
12:    Calculate feature importance and create  $\mathcal{F}(\mathbf{X}_a)$  by
    ranking features using a feature selection algorithm
13:    Generate additional mapped features  $\mathbf{Z}_{n+1}$  and additional
    enhancement nodes  $\mathbf{H}_{m+1}$  using Algorithm 1
14:    Update  $\mathbf{A}_t^m$ 
15:    Update weights  $\mathbf{W}_t^m$ 
16:   end for
17:   Rank and select features to be used in testing based on:
    selected features and their importance in each step
    and the weight vector  $\mathbf{W}_i$ 
18: end procedure
```

---

splitting nodes in order to avoid over-fitting. Features with higher importance are more relevant for a given dataset and better capture its properties. They may have better spatial separation and, thus, enhance the model’s performance.

## 7.2 Experiments and Performance Evaluation

The experimental procedure consists of four steps: (1) Partitioning datasets for training and testing; (2) Processing data: converting categorical to numerical features, selecting features, and normalizing training and test datasets (features selected during training are applied in the test phase); (3) Using 10-fold cross-validation to train and tune parameters; and (4) Testing and evaluating the generated machine learning models based on accuracy, F-Score, and training time. Performance results were obtained using the same division of datasets for testing in all cases except the NSL-KDD dataset. (In the case of the NSL-KDD dataset, the training and test datasets were predefined.)

The BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets were used to create and evaluate performance of various machine learning models: deep learning RNNs (LSTM and GRU); BLS, RBF-BLS, CFBLs, CEBLS, and CFEBLS with and without incremental learning; and the newly proposed VFBLs and VCFBLs algorithms with and without incremental learning. We perform two-way classification to identify regular (0)

and anomalous (1) data. The datasets are first sorted based on time stamps and then partitioned. The training and test datasets consist of 60 % and 40 % of anomalous data, respectively. While performance of LSTM and GRU depends on the number of hidden layers and nodes [25,28], BLS classification performance is affected by BLS architecture as well as the number of mapped features, enhancement nodes, and groups of mapped features [29,30].

The number of selected features was used to evaluate their effect on BLS performance. (Note that RNNs do not require feature selection.) Features are ranked based on importance using the function `sklearn.ensemble.ExtraTreesClassifier()` [172] and by tuning and setting parameters  $n\_estimators = 100$  and  $random\_state = 1$ . For cross-validation of LightGBM models, we vary the number of estimators (10–200) and learning rate (0.01–0.1).

We develop deep learning and bidirectional RNN models having various number of hidden layers and input sequence lengths of 10 (Slammer, Code Red), 100 (Nimda), and 50 (NSL-KDD, CICIDS2017, CSE-CIC-IDS2018) data points. ReLU is used as the RNN activation function. Layers with 0.5 dropout probability are inserted after  $FC_2$  and  $FC_3$  layers. The optimization algorithm “Adam” is selected to train the RNN models using 30 (BGP RIPE) and 50 (NSL-KDD, CICIDS2017, CSE-CIC-IDS2018) epochs with learning rate  $lr = 0.001$ . The deep learning neural network model with four hidden layers consists of 37 (BGP RIPE)/109 (NSL-KDD)/78 (CICIDS2017, CSE-CIC-IDS2018) RNNs, 64  $FC_1$ , 32  $FC_2$ , and 16  $FC_3$  fully connected (FC) hidden nodes.

The CFBLs, CEBLs, and CFEBLs models were implemented by modifying the original BLS functions [30]. For the cross-validation of incremental and non-incremental BLS models, we vary the number of mapped features (10–400), groups of mapped features (1–50), and enhancement nodes (20–700). Parameters of the incremental BLS models are: *incremental learning steps* = 2 (Slammer, Nimda, Code Red, CICIDS2017, CSE-CIC-IDS2018), 3 (NSL-KDD); *enhancement nodes/step* = 50 (Slammer), 10 (Nimda), 60 (Code Red, NSL-KDD), 20 (CICIDS2017, CSE-CIC-IDS2018); and *data points/step* = 187 (Slammer), 290 (Nimda), 303 (Code Red), 3,000 (NSL-KDD), 55,680 (CICIDS2017), 49,320 (CSE-CIC-IDS2018). For the cross-validation of VFBLs and VCFBLs models, we vary mapped features (10–200), groups of mapped features (5–30), and enhancement nodes (40–700). In the case of incremental VFBLs and VCFBLs models, *feature weight for initial step* = 0.9 (BGP RIPE), 0.7 (NSL-KDD, CICIDS2017), 0.85 (CSE-CIC-IDS2018) while the remaining parameters are the same as for incremental BLS. Performance of models was controlled by tuning the parameter  $\lambda$  (7.6) for sparse regularization in the ridge regression algorithm that was used to calculate output weights during the training process.

Training parameters that generate the best performance results are listed in Table 7.1 to Table 7.5. We use the parameters shown in Table 6.1 for BLS and incremental BLS using CICIDS2017 and CSE-CIC-IDS2018 Datasets. Performance of LightGBM, RNN and Bi-RNN (LSTM and GRU) with 2 (LSTM<sub>2</sub> and GRU<sub>2</sub>), 3 (LSTM<sub>3</sub> and GRU<sub>3</sub>), 4 (LSTM<sub>4</sub>

and GRU<sub>4</sub>) hidden layers, and BLS models are shown in Figs. 7.5, 7.6, and 7.7. Training times for models leading to the best performance are listed in Table 7.6.

Table 7.1: LightGBM parameters:  
BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets.

Number of features	Dataset	Number of estimators	Learning rate
<b>16</b>	Slammer	30	0.1
<b>16</b>	Nimda	200	0.01
<b>8</b>	Code Red	30	0.05
<b>64</b>	NSL-KDD	200	0.01
<b>32</b>	CICIDS2017	200	0.01
<b>64</b>	CSE-CIC-IDS2018	100	0.02

Table 7.2: BLS and incremental BLS parameters: BGP RIPE datasets.

Number of features	Dataset	Model	Mapped features	Groups of mapped features	Enhancement nodes
<b>BLS</b>					
<b>8</b>	Slammer	BLS	100	50	100
	Nimda	CFBLS	300	10	50
	Code Red	RBF-BLS	200	3	300
<b>16</b>	Slammer	CFEBLS	200	50	100
	Nimda	CFBLS	100	10	200
	Code Red	CEBLS	100	1	700
<b>37</b>	Slammer	BLS	200	30	50
	Nimda	BLS	100	10	200
	Code Red	CEBLS	100	1	300
<b>Incremental BLS</b>					
<b>8</b>	Slammer	BLS	300	50	100
	Nimda	CEBLS	300	10	100
	Code Red	BLS	200	3	500
<b>16</b>	Slammer	CFEBLS	200	30	200
	Nimda	CFEBLS	300	50	200
	Code Red	CFBLS	300	3	300
<b>37</b>	Slammer	CEBLS	100	30	50
	Nimda	CEBLS	100	10	200
	Code Red	CFBLS	300	1	700

Experimental results show that BLS models offer comparable performance to deep learning GRU and LSTM RNN and Bi-RNN models while requiring shorter training time. Introduced VFBLs and VCFBLs models outperform other BLS models in most cases except models based on Slammer dataset. In the case of NSL-KDD datasets, the VFBLs models show 2 %-15 % improvement in accuracy and 4 %-12 % improvement in F-Score. Even

Table 7.3: BLS and incremental BLS parameters: NSL-KDD dataset.

Number of features	Model	Mapped features	Groups of mapped features	Enhancement nodes
<b>BLS</b>				
<b>32</b>	BLS	100	40	40
<b>64</b>	RBF-BLS	60	20	40
<b>109</b>	CFBLS	100	20	100
<b>Incremental BLS</b>				
<b>32</b>	CFEBLS	60	20	40
<b>64</b>	RBF-BLS	80	20	40
<b>109</b>	CFBLS	80	30	40

Table 7.4: VFBLs and VCFBLs parameters: BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets.

Number of features	Dataset	Mapped features	Groups of mapped features	Enhancement nodes
<b>VFBLs</b>				
<b>8, 16, 37</b>	Slammer	100, 30, 40	30, 20, 10	100
	Nimda	20, 40, 30	10, 20, 10	50
	Code Red	20, 50, 30	10, 10, 20	100
<b>32, 64, 109</b>	NSL-KDD	20, 40, 30	20, 20, 20	40
<b>32, 64, 78</b>	CICIDS2017	15, 10, 10	5, 10, 5	40
	CSE-CIC-IDS2018	10, 20, 10	10, 5, 10	40
<b>VCFBLs</b>				
<b>8, 16, 37</b>	Slammer	200, 30, 30	20, 10, 20	100
	Nimda	20, 30, 30	10, 20, 10	100
	Code Red	30, 40, 40	10, 10, 10	100
<b>32, 64, 109</b>	NSL-KDD	20, 40, 30	10, 20, 30	60
<b>32, 64, 78</b>	CICIDS2017	10, 20, 10	10, 5, 5	40
	CSE-CIC-IDS2018	10, 10, 20	5, 10, 5	40

though the LightGBM models have the fastest training time as shown in Table 7.6, its accuracy and F-Score are in some cases lower than the proposed VFBLs and VCFBLs models, as illustrated in Figs. 7.5, 7.6, and 7.7.

Performance of machine learning models heavily depends on the datasets used for training and test. In the case of BGP RIPE datasets, one of the Bi-GRU models has the best accuracy and F-Score. The proposed VFBLs models have the best accuracy and F-Score for KDDTest<sup>+</sup> and KDDTest<sup>-21</sup> and the best F-Score for CSE-CIC-IDS2018 datasets. The GRU models have the best accuracy for CICIDS2017 and CSE-CIC-IDS2018 datasets.

Performance results shown in Fig. 7.5 (top) illustrate that the extra-trees algorithm may not be able to select the top 8 features that better capture properties of the anomalous

Table 7.5: Incremental VFBLs and VCFBLs parameters:  
BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets.

Number of features	Dataset	Mapped features	Groups of mapped features	Enhance- ment nodes
<b>Incremental VFBLs</b>				
<b>8, 16, 37</b>	Slammer	40, 10, 20	10, 20, 10	100
	Nimda	50, 50, 40	30, 30, 20	30
	Code Red	30, 40, 30	10, 1, 2	100
<b>32, 64, 109</b>	NSL-KDD	20, 20, 50	10, 10, 10	40
<b>32, 64, 78</b>	CICIDS2017	10, 10, 10	5, 5, 5	40
	CSE-CIC-IDS2018	15, 10, 10	10, 10, 5	40
<b>Incremental VCFBLs</b>				
<b>8, 16, 37</b>	Slammer	40, 20, 20	10, 20, 20	50
	Nimda	50, 30, 20	30, 10, 10	20
	Code Red	30, 40, 20	5, 5, 5	100
<b>32, 64, 109</b>	NSL-KDD	30, 40, 30	20, 20, 20	40
<b>32, 64, 78</b>	CICIDS2017	15, 10, 10	5, 5, 5	20
	CSE-CIC-IDS2018	20, 10, 10	10, 5, 5	20

data. Hence, selecting 8 features is not sufficient for successfully classifying the Slammer dataset. Note that performance of BLS and incremental BLS remains comparable even without feature selection. While performance of incremental VFBLs and VCFBLs is often inferior to other algorithms, their advantage is that the models need not be retrained for new incoming data. These models exhibit poor performance due to inadequate feature selection. Occurrences of long consecutive regular or anomalous data points in the partitioned training datasets lead to low accuracy and F-Score. If a portion of the training dataset consists of one class only, generated feature importance is 0 and, hence, these features do not contribute to subsequent steps. In the case of BGP RIPE, CICIDS2017, and CSE-CIC-IDS2018 datasets, data points belong to a single class (regular or anomaly) over a long period of time and, hence, features selected during training are not relevant for the test dataset.

BLS and incremental BLS models offer comparable performance to deep learning models and have shorter training time [29] when used with large datasets such as NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018. Introduced VFBLs and VCFBLs models have much shorter training time compared to RNN and Bi-RNN models. The proposed algorithms have an order of magnitude (up to 250 times) shorter training time than Bi-GRU. Their training time is comparable to BLS and its extensions for smaller datasets such as BGP RIPE but it increases for larger datasets because it is mainly spent for selecting features. Incremental variants of VFBLs and VCFBLs require considerably shorter training time. Note that LightGBM models require a much shorter training time than the BLS models.

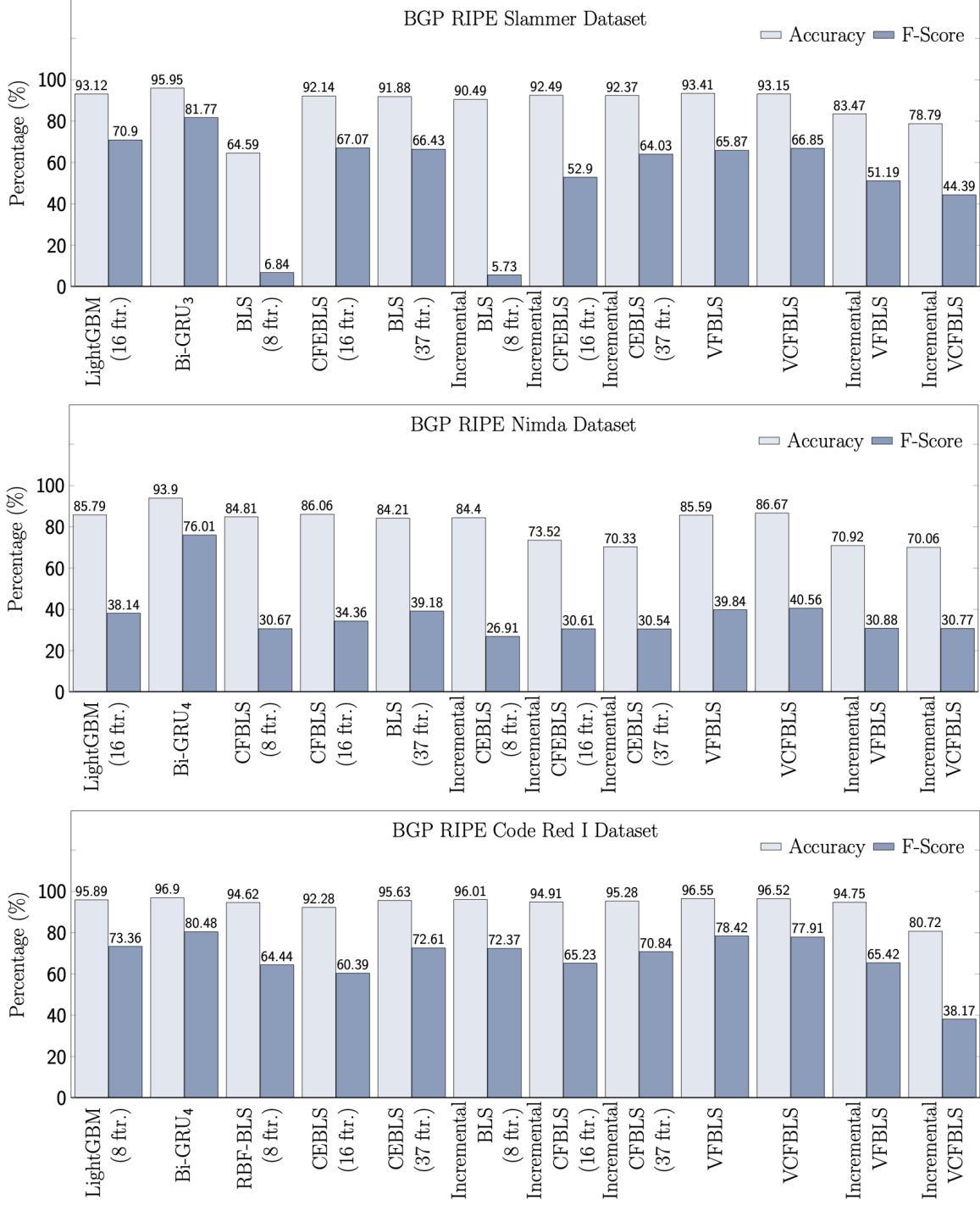


Figure 7.5: Best performance results for LightGBM, RNN and Bi-RNN (LSTM and GRU), BLS, Incremental BLS, VFBS, and VCFBS models: Slammer (top), Nimda (middle), and Code Red (bottom) datasets.

Note that times reported for training BLS and incremental BLS models do not account for the times spent for selecting features. The overheads for feature selection depend on the

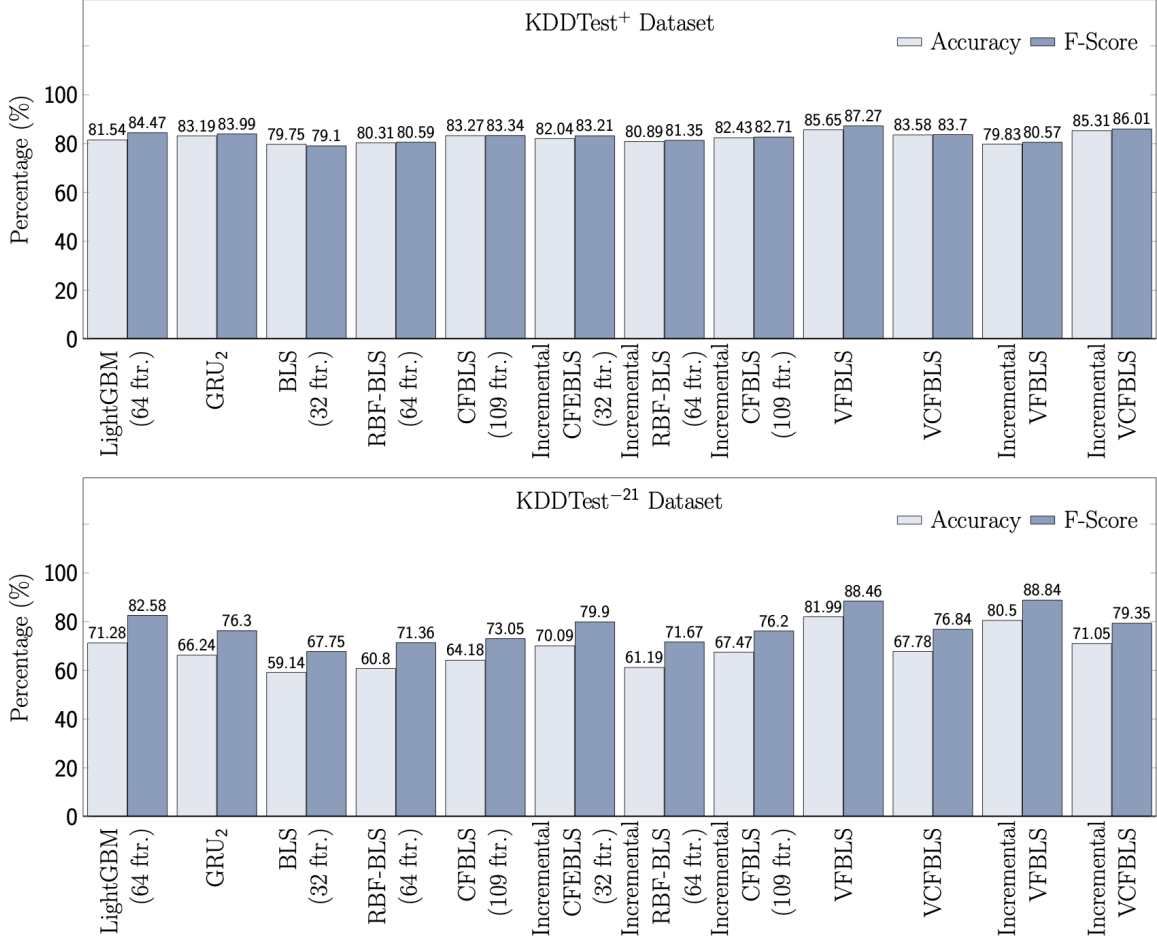


Figure 7.6: Best performance results for LightGBM, RNN and Bi-RNN (LSTM and GRU), BLS, Incremental BLS, VFBLs, and VCFBLs models: KDDTest<sup>+</sup> (top) and KDDTest<sup>-21</sup> (bottom) datasets.

size of the training datasets and the number of features. Times spent for selecting features in experiments with the extra-trees algorithm are: 0.13 s (Slammer), 0.22 s (Nimda), 0.24 s (Code Red), 10.24 s (KDDTrain<sup>+</sup>), 18.56 s (CICIDS2017), and 13.44 s (CSE-CIC-IDS2018).

VFBLs and VCFBLs algorithms may also be used for other classification tasks such as face recognition (image classification), sentiment analysis (natural language processing), and electroencephalogram (EEG) anomaly detection (Brain-Computer Interface (BCI)). For image datasets, a square matrix is often used to represent an image. Each element of the square matrix is a pixel of the image. The square matrix is associated with one label. When using VFBLs and VCFBLs, the square matrix needs to be transformed into a row vector. For example, in an image dataset [173] consisting of 60,000 images, the size of each image is  $28 \times 28$ . In the pre-processing phase, each image is reshaped into one  $1 \times 784$  row vector. The dataset of size  $60,000 \times 784$  is used as input to train VFBLs and VCFBLs models. In the case of the sentiment analysis dataset, each sentence corresponds

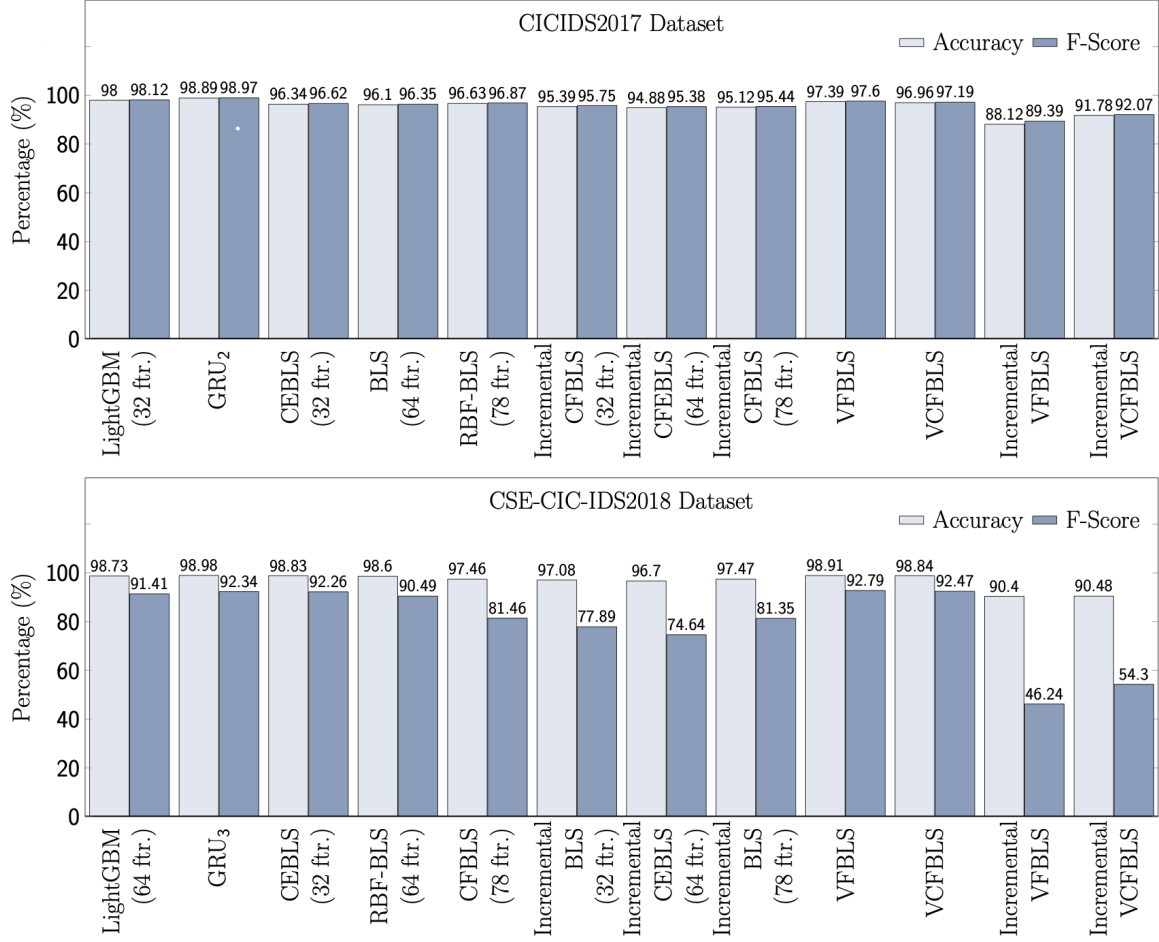


Figure 7.7: Best performance results for LightGBM, RNN and Bi-RNN (LSTM and GRU), BLS, Incremental BLS, VFBLs, and VCFBLs models: CICIDS2017 (top) and CSE-CIC-IDS2018 (bottom) datasets.

to a label. The input data is formed by using the word embedding technique that converts each word of the sentence into a row vector. A two-dimensional matrix is then generated for each sentence. In case of BCI anomaly detection, a label is also associated with multi-rows (two-dimensional matrix) of the dataset. These rows represent EEG signals generated at various brain locations. The input to the VFBLs and VCFBLs algorithms are transformed sentiment analysis and BCI datasets.

Table 7.6: Training time: BGP RIPE, NSL-KDD, CICIDS2017, and CSE-CIC-IDS2018 datasets.

Datasets	Model (No. of ftr.) Time (s)	LightGBM		RNN		BLS		Incremental BLS			Variable BLS		Incremental Variable BLS	
				Bi-GRU <sub>3</sub>	BLS (8)	CFEBS (16)	BLS (37)	BLS (8)	CFEBS (16)	CEBS (37)	VFBS (9.22)	VCFBLS (13.86)	VFBS (1.82)	VCFBLS (1.66)
Slammer		(16) 0.02		212.83	6.47	24.09	15.38	216.62	37.83	8.75	9.22	13.86	1.82	1.66
Nimda		(16) 0.11		Bi-GRU <sub>4</sub>	CFBLS (8)	CFBLS (16)	BLS (37)	CEBS (8)	CFEBS (16)	CEBS (37)	VFBS (2.12)	VCFBLS (1.97)	VFBS (10.37)	VCFBLS (5.98)
				219.50	3.51	1.29	2.01	8.27	43.41	13.35	2.12	1.97	10.37	5.98
Code Red		(8) 0.02		Bi-GRU <sub>4</sub>	RBF-BLS (8)	CEBS (16)	CEBS (37)	BLS (8)	CFBLS (16)	CFBLS (37)	VFBS (1.88)	VCFBLS (2.55)	VFBS (1.33)	VCFBLS (1.42)
				546.54	5.90	174.21	26.63	1.11	2.00	1.12	1.88	2.55	1.33	1.42
NSL-KDD		(64) 0.92		GRU <sub>2</sub>	BLS (32)	RBF-BLS (64)	CFBLS (109)	CFEBS (32)	RBF-BLS (64)	CFBLS (109)	VFBS (31.21)	VCFBLS (31.32)	VFBS (28.92)	VCFBLS (60.43)
				4,831.55	39.77	11.10	24.84	26.05	36.74	83.03	31.21	31.32	28.92	60.43
CICIDS2017		(32) 1.43		GRU <sub>2</sub>	CEBS (32)	BLS (64)	RBF-BLS (78)	CFBLS (32)	CFEBS (64)	CFBLS (78)	VFBS (25.25)	VCFBLS (26.05)	VFBS (25.55s)	VCFBLS (24.19)
				15,483.96	39.25	8.97	15.60	6.39	7.39	3.69	25.25	26.05	25.55s	24.19
CSE-CIC-IDS2018		(64) 0.99		GRU <sub>3</sub>	CEBS (32)	RBF-BLS (64)	CFBLS (78)	BLS (32)	CEBS (64)	BLS (78)	VFBS (21.30)	VCFBLS (21.38)	VFBS (24.83)	VCFBLS (14.86)
				26,887.14	33.46	4.65	4.13	5.65	11.59	6.78	21.30	21.38	24.83	14.86

## Chapter 8

# BGPGuard: BGP Anomaly Detection Tool

In this Chapter, we introduce a BGP anomaly detection tool named *BGPGuard*. The real-time detection and off-line classification are implemented using the Linux terminal-based and web-based applications. Network administrators may use the *BGPGuard* to detect the BGP suspicious events based on RRC routing records collected from major Internet exchange points worldwide. They may also create machine learning models based on the historical BGP anomalous events.

### 8.1 Architectures

The architecture of *BGPGuard* for real-time detection shown in Fig. 8.1 consists of six modules:

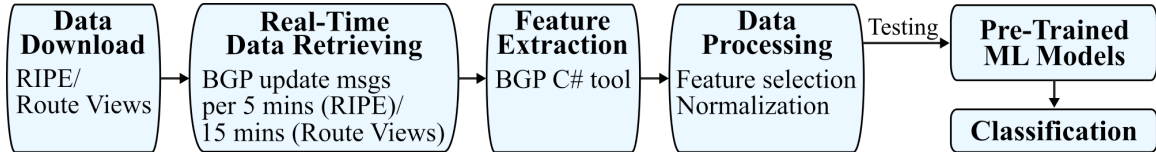


Figure 8.1: Architecture of *BGPGuard* for real-time detection and its modules: data download, real-time data retrieval, feature extraction, data processing, pre-trained models, and classification.

**Data Download:** The input to the data download module are the names of files with *update* messages, collection site (RIPE or Route Views), and collector name (For example, RIPE rrc04 or Route Views route-views2). The output are *update* messages in ASCII format. Raw data from RIPE and Route Views are organized in folders labeled by the year and month of the collection date. The format of the selected and downloaded BGP *update* messages is *updates.yyyymmdd.hhmm.gz* or *updates.yyyymmdd.hhmm.bz2* for RIPE and Route

Views datasets, respectively. <sup>1</sup> BGP *update* messages are initially collected in MRT format. They are transformed from MRT to ASCII format by using the *zebra-dump-parser* [174] tool written in Perl. GMT time is used for all *update* messages in order to synchronize RIPE and Route Views collection times.

**Real-Time Data Retrieval:** The module operates by retrieving the latest *update* message from RIPE or Route Views collection sites. BGP *update* messages are available for download from in RIPE and Route Views every 5 and 15 minutes, respectively.

**Feature Extraction:** A tool written in C# was used to generate datasets by extracting 37 numerical features from BGP *update* messages [175].

**Data Processing:** The module consists of feature selection and normalization steps. The most relevant features may be selected for both training and test datasets by using the extremely randomized trees (extra trees) [145] feature selection algorithm. The input to the module are: number of the most relevant features, file name, and labels of the dataset. By employing the *zscore* function, we generate datasets with *mean* = 0 and *standard deviation* = 1.

**Pre-Trained Models:** The pre-trained VFBLs and RNN models are used for real-time detection.

**Classification:** Accuracy, F-Score, precision, sensitivity (recall), confusion matrix, and training time may be used to evaluate performance of classification algorithms.

The architecture of *BGPGuard* for off-line classification shown in Fig. 8.2 consists of eight modules: four new modules (data partition, ML algorithms, parameters, and ML Models) and four modules (data download, feature extraction, data processing, and classification) from the real-time classification architecture.

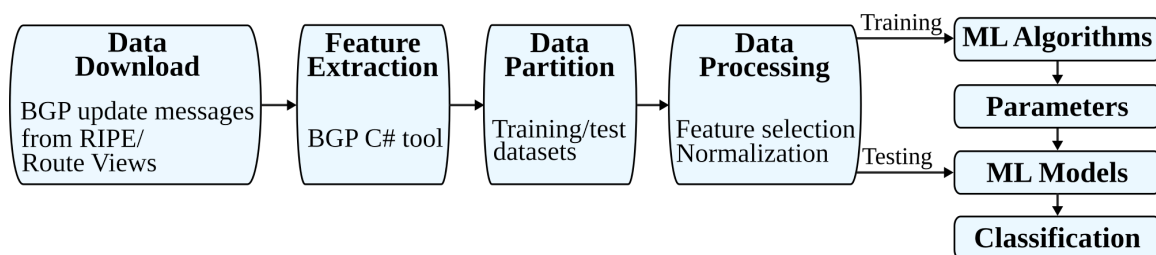


Figure 8.2: Architecture of *BGPGuard* for off-line classification and its modules: data download, feature extraction, data partition, data processing, machine learning (ML) algorithms, parameter selection, ML models, and classification.

**Data Partition:** This module is used to create the training and test datasets based on the percentages of anomalous data. Data are labeled based on the time intervals of collection.

---

<sup>1</sup>“yyyymmdd.hhmm” refers to year (yyyy), month (mm), day (dd), hour (hh), and minute (mm).

**ML Algorithms:** The module contains various deep learning RNN models with a number of hidden layers. Input parameters are: RNN algorithm, number of hidden layers and nodes, number of epochs, learning rate, activation function, and dropout rate.

**Parameters:** Parameters for cross-validation are stored in this module. The best set of parameters selected by cross-validation is used in the training process.

**ML Models:** This module generates machine learning models using training datasets.

## 8.2 Terminal-Based Application

A Linux terminal-based application (written in Python 3.6) has been developed to combine the anomaly detection steps. The application may be also executed on a low-power device such as Raspberry Pi [176]. The routing records are retrieved from either RIPE or Route Views for the real-time detection. A pre-trained model is then selected to perform prediction. Several options of the pre-trained models such as VFBLs and RNN models are available for low-memory and high-memory platforms. For the off-line classification mode, additional parameters include the start and end dates and times of the anomalous event, training and test partitioning option, and machine learning algorithms.

### 8.2.1 Structure

The hierarchical structure of the terminal-based *BGPGuard*:

```

BGPGuard_TerminalApp
├── LICENSE
├── README.md
├── main.py
├── app_offline.py
├── app_realtime.py
├── config.py
├── requirements.txt
├── src
│   ├── __init__.py
│   ├── dataDownload.py
│   ├── data_partition.py
│   ├── data_process.py
│   ├── featureExtraction.py
│   └── input_exp.txt
└── ...
...
├── label_generation.py
├── progress.py
├── progress_bar.py
├── subprocess_cmd.py
├── time_tracker.py
├── BGP_CSharp_Tool
├── STAT
├── VFBLs_v110
├── RNN_Running_Code
├── data_historical
├── data_ripe
├── data_routeviews
├── data_split
├── data_test
└── parmSel

```

The *src* directory contains the source code for the real-time detection and off-line classification tasks. Various Python functions have been developed to implement and incorporate the anomaly detection steps. Descriptions of developed Python functions and their variables are given in Appendix C. Listed are components of the *src* directory:

- *src/BGP\_CSharp\_Tool*: developed BGP C# tool [120].
- *src/STAT*: generated labels, number of the regular and anomalous data points in training and test datasets, and classification results.
- *src/VFBLS\_v110*: implementation of the VFBLS algorithm and its pre-trained models.
- *src/RNN\_Running\_Code*: implementation of the LSTM and GRU algorithms and their pre-trained models.
- *src/data\_historical*: datasets generated based on known BGP anomalous events. (The pre-trained models are created using the data in this directory.)
- *src/data\_ripe*: raw data downloaded from RIPE.
- *src/data\_routeviews*: raw data downloaded from Route Views.
- *src/data\_split*: training and test datasets after partition.
- *src/data\_test*: extracted feature matrix for the real-time task.
- *src/parmSel*: model parameters.

The external Python libraries are listed in *requirements.txt*. The following command is used to install the libraries:

---

```
> pip install -r requirements.txt
```

---

The Python file *main.py* is used to run the code.

### 8.2.2 External Libraries

The terminal-based application relies on several external libraries that are expected to be installed prior to executing the main files.

Listed are the Python libraries that are installed by using a package installer *pip* [177]:

- *NumPy* [178]: used to perform mathematical operations on multi-dimensional arrays and on matrices generated during the process.
- *SciPy* [179]: dependency of the *scikit-learn* library. *SciPy*'s *zscore*: function used to perform normalization.
- *Scikit-learn* [180]: employed for processing data and calculating performance metrics.
- *PyTorch* [165]: used for developing RNN models.

The C# compiler should be installed prior to executing the BGP C# tool:

- *Mono* [181]: an open source version of Microsoft .NET framework. Mono includes a C# compiler for several operating systems such as macOS, Linux, and Windows.

### 8.2.3 Sample Settings

We describe here sample settings for real-time detection and off-line classification. For real-time detection mode, variables “real-time” and “RIPE” are entered to start the program.

```
> python main.py
real-time or off-line?
Mode: real-time
Choose collection site: RIPE or RouteViews
Collection site: RIPE
```

The output is listed in Appendix D.

The file *input\_exp.txt* contains parameters for off-line classification. The main file (*main.py*) is executed to load the parameters from *input\_exp.txt* and start the experiment.

```
{
    "site": "RIPE",
    "start_date": "20050523",
    "end_date": "20050527",
    "start_date_anomaly": "20050525",
    "end_date_anomaly": "20050525",
    "start_time_anomaly": "0400",
    "end_time_anomaly": "1159",
    "cut_pct": "64",
    "rnn_seq": 10
}
```

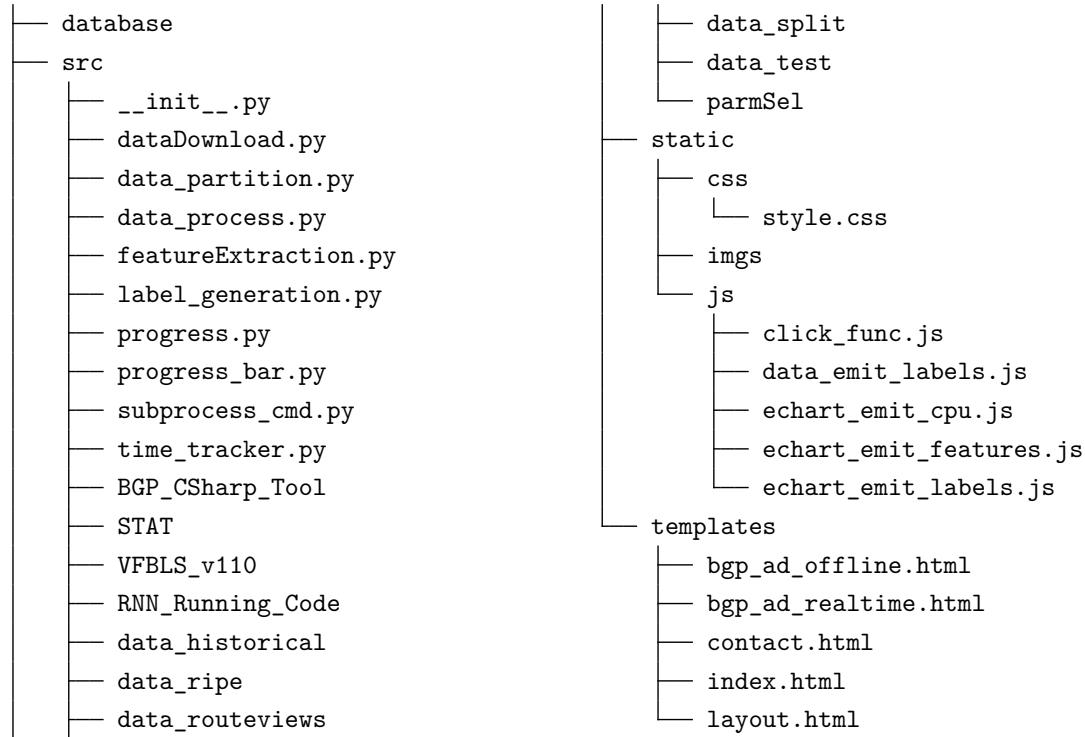
## 8.3 Web-Based Application

The web-based version of *BGPGuard* offers an interactive interface with a better view for monitoring and performing experiments. The web-based application includes several functions from the terminal-based application. It is developed using additional programming languages, frameworks, and functions. Its front-end is built based on HTML, CSS (*Bootstrap* [182]: an open-source CSS framework), and *Socket.IO* [183] (a transport protocol written in a JavaScript for real-time web applications). Its back-end is developed using *Flask* [184] (a micro web framework written in Python).

### 8.3.1 Structure

Shown is the hierarchical structure of the web-based *BGPGuard*:

```
BGPGuard_webApp
├── LICENSE
├── README.md
├── app.py
├── app_offline.py
├── app_realtime.py
├── config.py
└── requirements.txt
```



Descriptions of developed Python functions are given in Appendix C. Additional directories are created for establishing communication between the front-end and the back-end:

- *./database*: generated features and detection results in database format.
- *./static*: CSS style file, images, and JavaScript code.
- *./templates*: HTML files.

The Python file *app.py* is used to execute the application. The following command is used to start the application:

```

> export FLASK_APP=app.py
> export FLASK_ENV=development
> flask run
* Running on http://127.0.0.1:5000/

```

### 8.3.2 External Libraries

The web-based application relies on additional external libraries. The external CSS and JavaScript libraries provided by jsDelivr [185] have been included in *layout.html*.

The Python libraries installed by *pip* are:

- *Flask*: web framework based on *Werkzeug* [186] and *Jinja* [187]. (The Flask's functions are used to transfer variables and render web pages. Flask also processes the GET/POST requests from the front-end.)

- *Werkzeug*: web application library used to create a web server gateway interface (WSGI).
- *Jinja*: web template engine. Variables, statements, and expressions allowed to include in HTML files.
- *Flask-SocketIO* [188]: offers bi-directional communications with low latency between the clients (front-end) and the server (back-end) for Flask applications.
- *Python-engineio* [189]: implementation of the Engine.IO in Python.
- *Python-socketio* [190]: library for real-time communication between client and server based on WebSocket.
- *Eventlet* [191]: networking library for executing asynchronous tasks.

### 8.3.3 Web Pages

Four web pages illustrate the interface of the *BGPGuard* as shown in Figs. 8.3, 8.4, 8.5, and 8.6. For real-time detection web page shown in Fig. 8.4, results and statistics are shown: total number of the predicted regular points and anomalies (pie chart) and predicted labels/BGP announcements/BGP withdrawals vs. time (plots). The CPU usage of the implementation platform is also shown. For off-line anomaly classification, various parameters should be entered in order to execute experiments.

BGPGuard

[Home](#)
[Real-Time Detection](#)
[Off-Line Classification](#)
[Contact](#)

v1.1.0

# BGP ANOMALY DETECTION IN REAL-TIME & OFF-LINE

A BGP anomaly detection framework that integrates various stages of the anomaly detection process.

Get started >

## Framework

```

graph LR
    subgraph RealTime [Real-Time Detection]
        RTDD[Data Download  
RIPE/  
Route Views] --> RTRD[Real-Time Data Retrieving  
BGP update msgs  
per 5 mins (RIPE)/  
15 mins (Route Views)]
        RTRD --> RTE[Feature Extraction  
BGP C# tool]
        RTE --> RTP[Data Processing  
Feature selection  
Normalization]
        RTP -- Testing --> PTM[Pre-Trained ML Model]
        PTM --> C[Classification]
    end

    subgraph OffLine [Off-Line Classification]
        ODD[Data Download  
BGP update messages  
from RIPE/  
Route Views] --> OFE[Feature Extraction  
BGP C# tool]
        OFE --> ODP[Data Partition  
Training/test  
datasets]
        ODP --> ODP2[Data Processing  
Feature selection  
Normalization]
        ODP2 -- Training --> MA[ML Algorithms]
        MA --> P[Parameters]
        P --> MM[ML Models]
        MM -- Testing --> C2[Classification]
    end
  
```

## Detection and Classification Modes

Try it

Visualize the detection results

### Real-Time Detection

- Collection Sites: RIPE, Route Views
- Algorithm: VFBL

Try it

Customize your experiments

### Off-Line Classification

- Collection Sites: RIPE, Route Views
- Algorithms: VFBL, RNN

© Copyright 2022  
Communication Networks  
Laboratory

### Navigation

[Home](#)
[Real-Time Detection](#)

[Off-Line Classification](#)
[Contact](#)

### Contact us

Communication Networks Laboratory,  
 Simon Fraser University, 8888 University Dr.,  
 Burnaby, B.C. V5A 1S6 Canada  
 Email: [zhidal@sfu.ca](mailto:zhidal@sfu.ca)

Figure 8.3: The web-based *BGPGuard*: The index page.

118

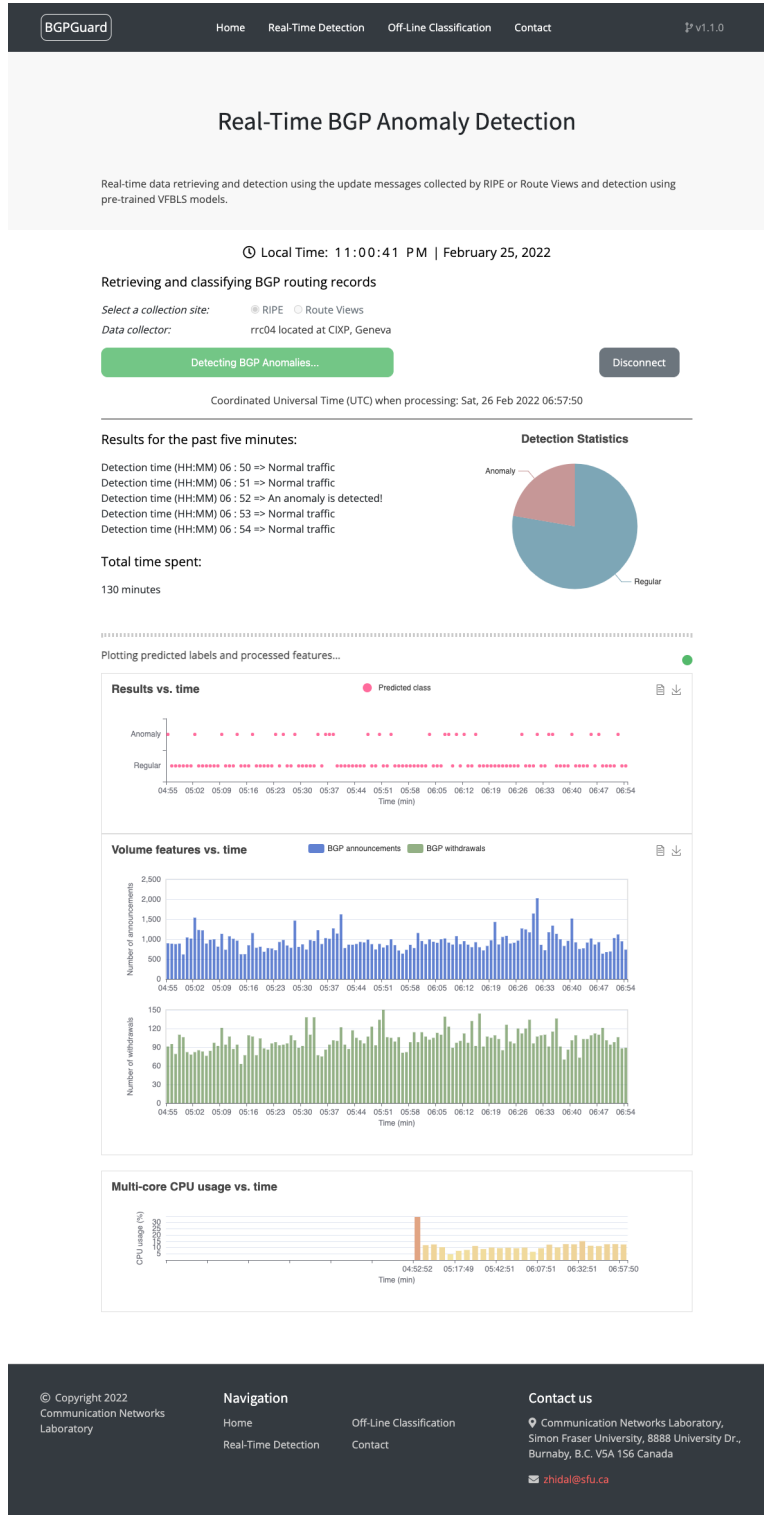


Figure 8.4: The web-based *BGPGuard*: The real-time detection view. The front-end displayed prediction results and data statistics are generated by the back-end modules: data download, real-time data retrieval, feature extraction, data processing, pre-trained models, and classification.

BGPGuard

[Home](#)
[Real-Time Detection](#)
[Off-Line Classification](#)
[Contact](#)

v1.1.0

## Off-Line BGP Anomaly Classification

Customizing off-line experiments by specifying the collection site, start and end dates and times of the anomalous event, partitioning the training and test datasets, and the choice of the RNN or VFBL algorithm.

Local Time: 04:43:18 PM | March 10, 2022

### Parameter Selections

Select a collection site: ☒ RIPE ☐ Route Views

Start date for the entire task:  Format: YYYYMMDD. Example: 20050523

End date for the entire task:  Format: YYYYMMDD. Example: 20050527

Start date for the anomalous event:  Format: YYYYMMDD. Example: 20050525

End date for the anomalous event:  Format: YYYYMMDD. Example: 20050525

Start time for the anomalous event:  Format: HHMM. Example: 0400

End time for the anomalous event:  Format: HHMM. Example: 1159

Partition percentage (train & test):  Format: Two integers; sum is 10. Example: 64

Sequence length for RNN algorithms if needed:  Format: Integer divisible by 10. Example: 20

Collection site selected:

Prediction:

### Download result files

Feature matrices, predicted labels


[^ Top](#)

© Copyright 2022  
 Communication Networks  
 Laboratory

**Navigation**  
[Home](#)  
[Real-Time Detection](#)  
[Off-Line Classification](#)  
[Contact](#)

**Contact us**  
 Communication Networks Laboratory,  
 Simon Fraser University, 8888 University Dr.,  
 Burnaby, B.C. V5A 1S6 Canada  
[zhidal@sfu.ca](mailto:zhidal@sfu.ca)


Figure 8.5: The web-based *BGPGuard*: The off-line classification view. The classification results are generated by the back-end modules: data download, feature extraction, data partition, data processing, ML algorithms, parameter selection, ML models, and classification.



[Home](#)
[Real-Time Detection](#)
[Off-Line Classification](#)
[Contact](#)

v1.1.0

## Introduction of the Author



**Zhida Li**

**Ph.D. Candidate | Research Assistant**

Zhida received the B.E. and M.Eng.Sc. degrees in Electrical Engineering and Microelectronic Design from the University College Cork, Ireland, in 2011 and 2013, respectively. He is currently pursuing a Ph.D. degree at Simon Fraser University.

His current research project deals with development of fast machine learning algorithms and a real-time system for detecting network anomalies. Zhida is an active IEEE volunteer and served as the Secretary and Vice-Chair of the IEEE SFU Student Branch in 2019-2020 and 2020-2021, respectively. He was a Program Committee Member of IEEE International Conference on High Performance Computing and Communications (HPCC) 2020 and Session Chair at IEEE International Conference on Cyber, Physical, and Social Computing (CPSCom) 2020. He serves as a Publicity Chair of CPSCom 2022.

## Join Our Email List

Kindly let us inform you of our latest research


© Copyright 2022  
Communication Networks  
Laboratory

**Navigation**

[Home](#)
[Off-Line Classification](#)

[Real-Time Detection](#)
[Contact](#)

**Contact us**

 Communication Networks Laboratory,  
Simon Fraser University, 8888 University Dr.,  
Burnaby, B.C. V5A 1S6 Canada


 [zhidal@sfu.ca](mailto:zhidal@sfu.ca)

Figure 8.6: The web-based *BGPGuard*: Introduction of the author.

## Chapter 9

# Conclusion and Future Work

This dissertation is focused on application of machine learning algorithms for classifying network anomalies and developing new BLS-based algorithms. We performed feature selection using Fisher, mRMR, OR, decision tree, extra-trees, and autoencoder algorithms. They were employed to select the most relevant features and, thus, may improve model performance and reduce the training time. We have evaluated performance of traditional machine learning, deep learning, and fast machine learning algorithms and the proposed variable features broad learning systems. The datasets that we considered were collected from deployed network traffic (BGP) and controlled testbed environments. BGP datasets used in experiments are extracted from routing records collected by RIPE and Route Views during several anomalous events: Internet worms (Slammer, Nimda, and Code Red), power outages (Pakistan power outage and Moscow blackout), and ransomware attacks (WestRock and WannaCrypt). Datasets collected from testbeds included NSL-KDD (improved version of KDD'99) generated in a simulated US Air Force base network and CIC (CICIDS2017, CSE-CIC-IDS2018, and CICDDoS2019) datasets generated by Canadian Institute for Cybersecurity. These datasets contain TCP/UDP network flows including regular data and various types of intrusions. Training time, accuracy, F-Score, precision, sensitivity, and confusion matrix were used to evaluate model performance.

We observed that SVM, HMM, naïve Bayes, decision tree, and ELM algorithms had advantages and limitations. They were feasible approaches to classify datasets of the smaller size (BGP). For larger datasets (NSL-KDD and CIC), deep learning algorithms were used to generate more robust models compared to the conventional machine learning algorithms. RNNs (LSTM and GRU) and Bi-RNNs (Bi-LSTM and Bi-GRU) outperformed CNN models in most cases because RNNs and Bi-RNNs are designed to process sequential data. Fast machine learning algorithms such as BLS and GBDT achieved comparable performance using the larger datasets. LightGBM achieved comparable performance to XGBoost and CatBoost. The shortest training time was required for LightGBM models. In case of CICDDoS2019 dataset having similar classification results (accuracy, F-Score, precision, and sensitivity), the confusion matrix was used to further compare the classification perfor-

mance. The lower number of misclassified anomalous (FN) and regular (FP) data points indicate better models.

Two new algorithms Variable Features Broad Learning Systems were aimed to achieve comparable performance to the existing fast machine learning and deep learning algorithms. The performance of VFBL was compared to LightGBM, RNN, Bi-RNN, and BLS algorithms. Experiments with BLS models were performed by varying the number of most relevant extracted features. The newly proposed algorithms employed variable number of mapped features and groups of mapped features without (VFBL) and with (VCFBL) cascades and a feature selection algorithm. Both algorithms also have incremental learning variants. Performance evaluation indicated that BLS with cascades of enhancement nodes required significantly longer training time. Note that in the case of incremental BLS, the model did not need to be retrained. As expected, larger numbers of mapped features, groups of mapped features, and enhancement nodes required additional memory and longer training time. In several cases, VFBL offered the best performance and shorter training time than RNNs and Bi-RNNs and comparable time to other BLS algorithms.

Classification performance of the developed VFBL and VCFBL algorithms may be further enhanced by implementing: (1) Multiple fast feature selection algorithms to create subsets of input data; (2) Recurrent networks with various hidden layers to replace enhancement nodes in order to capture dynamic characteristics of the time series data. Other machine learning algorithms may be enhanced using ESNs and transformers. ESNs are RNNs that use a reservoir to process the time series data. Various reservoirs may be incorporated in VFBL and VCFBL algorithms as a part of mapped features or enhancement nodes. Self-attention mechanism is a variant of the attention that is used in Transformers. The attention mechanism extracts the most important features from sparse data widely used for natural language processing tasks such as machine translation. The self-attention mechanism improves the attention mechanism by reducing the dependence on external information. It offers better performance by capturing the internal correlations of the data.

We extracted volume and AS-path features from the BGP routing records. They may not capture all properties from the collected *update* messages. The features capturing the topology of a communication network may also affect the classification of collected traffic. Hence, future work may include extracting additional features based on network topology for each data collection interval. These features include node degrees, betweenness, and closeness. BGP datasets may be also enhanced (enlarged) by using finer time scales (granularity) and extract features every second rather than every minute. (The time stamp of each *update* message from the collected BGP records is in seconds.) These extract features may or may not have the same label.

We also developed *BGPGuard* (Linux terminal-based and web-based) for classifying BGP anomalies for real-time detection and off-line classification. Detection and classification are performed based on routing records that are downloaded from RIPE and Route

Views collection sites. Anomaly detection steps have been incorporated to implement data retrieval, feature extraction, data processing, model training, and classification. The initial development phase of *BGPGuard* has been completed. The web-based *BGPGuard* will be deployed on a web server (Amazon Web Services [192] or Heroku [193]). The WebSocket protocol is essential for enabling asynchronous communication. The off-line classification may be deployed on the Amazon Elastic Compute Cloud [194] (Amazon EC2). However, Amazon EC2 does not support asynchronous tasks for real-time detection. Including NSL-KDD, CIC, and UNSW-NB15 datasets in the applications are also considered for future work. Existing algorithms may be combined to generate better performance results that are based on multiple models. Novel machine learning algorithms may be also implemented to improve detection performance.

# Bibliography

- [1] H. G. Kayacik, A. N. Zincir-Heywood, and M. I. Heywood, “Selecting features for intrusion detection: a feature relevance analysis on KDD 99 intrusion detection datasets,” in *Proc. 3<sup>rd</sup> Annu. Conf. Privacy Secur. Trust*, St. Andrews, NB, Canada, Oct. 2005, pp. 1–6.
- [2] (2022, Apr.) Route Views Collector Map. [Online]. Available: <http://www.routeviews.org/routeviews/index.php/map>.
- [3] C. M. Bishop, *Pattern Recognition and Machine Learning*. Secaucus, NJ, USA: Springer-Verlag, 2006.
- [4] C. L. P. Chen and Z. Liu, “Broad learning system: an effective and efficient incremental learning system without the need for deep architecture,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 1, pp. 10–24, Jan. 2018.
- [5] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, “Anomaly-based network intrusion detection: techniques, systems and challenges,” *Computers & Security*, vol. 28, no. 1–2, pp. 18–28, Feb.–Mar. 2009.
- [6] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: a survey,” *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, July 2009.
- [7] J. Du, C. Jiang, J. Wang, Y. Ren, and M. Debbah, “Machine learning for 6G wireless networks: carrying forward enhanced bandwidth, massive access, and ultrareliable/low-latency service,” *IEEE Veh. Technol. Mag.*, vol. 15, no. 4, pp. 122–134, Dec. 2020.
- [8] A. W. Moore and D. Zuev, “Internet traffic classification using bayesian analysis techniques,” in *Proc. Int. Conf. Measurement and Modeling of Comput. Syst.*, Banff, AB, Canada, June 2005, pp. 50–60.
- [9] C. Cortes and V. Vapnik, “Support-vector networks,” *J. Mach. Learn.*, vol. 20, no. 3, pp. 273–297, Sept. 1995.
- [10] J. R. Quinlan, “Induction of decision trees,” *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986.
- [11] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA, USA: The MIT Press, 2012.

- [12] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, “LightGBM: a highly efficient gradient boosting decision tree,” in *Proc. Int. Conf. Neural Inform. Process. Syst.*, Long Beach, CA, USA, Dec. 2017, pp. 3146–3154.
- [13] Y. LeCun, Y. Bengio, and G. E. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: The MIT Press, 2016.
- [15] C. L. P. Chen, Z. Liu, and S. Feng, “Universal approximation capability of broad learning system and its structural variations,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 4, pp. 1191–1204, Apr. 2019.
- [16] (2022, Apr.) RIPE NCC. [Online]. Available: <https://www.ripe.net/analyse>.
- [17] (2022, Apr.) University of Oregon Route Views projects. [Online]. Available: <http://www.routeviews.org>.
- [18] (2022, Apr.) NSL-KDD Data Set. [Online]. Available: <https://www.unb.ca/cic/datasets/nsl.html>.
- [19] (2022, Apr.) Intrusion Detection Evaluation Dataset (CICIDS2017). [Online]. Available: <https://www.unb.ca/cic/datasets/ids-2017.html>.
- [20] (2022, Apr.) A Realistic Cyber Defense Dataset (CSE-CIC-IDS2018). [Online]. Available: <https://registry.opendata.aws/cse-cic-ids2018>.
- [21] (2022, Apr.) DDoS Evaluation Dataset (CICDDoS2019). [Online]. Available: <https://www.unb.ca/cic/datasets/ddos-2019.html>.
- [22] (2022, Apr.) Python package index. [Online]. Available: <https://pypi.org>.
- [23] H. Liu and H. Motoda, *Computational Methods of Feature Selection*. Boca Raton, FL, USA: Chapman and Hall/CRC Press, 2007.
- [24] Y. Li, H. J. Xing, Q. Hua, X. Z. Wang, P. Batta, S. Haeri, and Lj. Trajković, “Classification of BGP anomalies using decision trees and fuzzy rough sets,” in *Proc. IEEE Trans. Syst. Man Cybern.*, San Diego, CA, USA, Oct. 2014, pp. 1331–1336.
- [25] Q. Ding, Z. Li, P. Batta, and Lj. Trajković, “Detecting bgp anomalies using machine learning techniques,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Budapest, Hungary, Oct. 2016, pp. 3352–3355.
- [26] G. Ditzler, J. LaBarck, J. Ritchie, G. Rosen, and R. Polikar, “Extensions to online feature selection using bagging and boosting,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 9, pp. 4504–4509, Sept. 2018.
- [27] Z. Li, Q. Ding, S. Haeri, and Lj. Trajković, “Application of machine learning techniques to detecting anomalies in communication networks: classification algorithms,” in *Cyber Threat Intelligence*, A. Dehghantanha, M. Conti, and T. Dargahi, Eds. Berlin: Springer, 2018, pp. 71–92.

- [28] P. Batta, Z. Li, and Lj. Trajković, “Evaluation of support vector machine kernels for detecting network anomalies,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Florence, Italy, May 2018, pp. 1–4.
- [29] Z. Li, P. Batta, and Lj. Trajković, “Comparison of machine learning algorithms for detection of network intrusions,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Miyazaki, Japan, Oct. 2018, pp. 4248–4253.
- [30] Z. Li, A. L. Gonzalez Rios, G. Xu, and Lj. Trajković, “Machine learning techniques for classifying network anomalies and intrusions,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Sapporo, Japan, May 2019, pp. 1–5.
- [31] A. L. Gonzalez Rios, G. X. Z. Li, A. D. Alonso, and Lj. Trajković, “Detecting network anomalies and intrusions in communication networks,” in *Proc. 23<sup>rd</sup> IEEE Int. Conf. Intell. Eng. Syst.*, Gödöllő, Hungary, Apr. 2019, pp. 29–34.
- [32] A. L. Gonzalez Rios, Z. Li, K. Bekshentayeva, and Lj. Trajković, “Detection of denial of service attacks in communication networks,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Seville, Spain, Oct. 2020.
- [33] Z. Liu and C. L. P. Chen, “Broad learning system: structural extensions on single-layer and multi-layer neural networks,” in *Proc. Int. Conf. Secur., Pattern Anal., Cybern.*, Shenzhen, China, Dec. 2017, pp. 136–141.
- [34] M. Xu, M. Han, C. L. P. Chen, and T. Qiu, “Recurrent broad learning system for time series prediction,” *IEEE Trans. Cybern.*, pp. 1–13, Sept. 2018.
- [35] L. Yang, S. Song, and C. L. P. Chen, “Transductive transfer learning based on broad learning system,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Miyazaki, Japan, Oct. 2018, pp. 912–917.
- [36] Z. Liu, C. L. P. Chen, T. Zhang, and J. Zhou, “Multi-kernel broad learning systems based on random features: a novel expansion for nonlinear feature nodes,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Bari, Italy, Oct. 2019, pp. 193–197.
- [37] M. Bhuyan, D. Bhattacharyya, and J. Kalita, “Network anomaly detection: methods, systems and tools,” *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 303–336, Mar. 2014.
- [38] H. Y. Shahir, U. Glässer, A. Y. Shahir, and H. Wehn, “Maritime situation analysis framework: Vessel interaction classification and anomaly detection,” in *Proc. Int. Conf. Big Data*, Santa Clara, CA, USA, Oct. 2015, pp. 1279–1289.
- [39] A. Y. Shahir, T. Charalampous, U. Glässer, and H. Wehn, “TripTracker: unsupervised learning of fishing vessel routine activity patterns,” in *Proc. Int. Conf. Big Data*, Orlando, FL, USA, Dec. 2021, pp. 1928–1939.
- [40] F. Lau, S. H. Rubin, M. H. Smith, and L. Trajković, “Distributed denial of service attacks,” in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Nashville, TN, USA, Oct. 2000, pp. 2275–2280.

- [41] C. Patrikakis, M. Masikos, and O. Zouraraki, "Distributed denial of service attacks," *The Internet Protocol*, vol. 7, no. 4, pp. 13–31, Dec. 2004.
- [42] H. Hajji, "Statistical analysis of network traffic for adaptive faults detection," *IEEE Trans. Neural Netw.*, vol. 16, no. 5, pp. 1053–1063, Sept. 2005.
- [43] M. Thottan and C. Ji, "Anomaly detection in IP networks," *IEEE Trans. Signal Process.*, vol. 51, no. 8, pp. 2191–2204, Aug. 2003.
- [44] S. Deshpande, M. Thottan, T. K. Ho, and B. Sikdar, "An online mechanism for BGP instability detection and analysis," *IEEE Trans. Comput.*, vol. 58, no. 11, pp. 1470–1484, Nov. 2009.
- [45] K. El-Arini and K. Killourhy, "Bayesian detection of router configuration anomalies," in *Proc. ACM SIGCOMM workshop on Mining Network Data*, Philadelphia, PA, USA, Aug. 2005, p. 221–222.
- [46] J. Zhang, J. Rexford, and J. Feigenbaum, "Learning-based anomaly detection in BGP updates," in *Proc. Workshop Mining Netw. Data*, Philadelphia, PA, USA, Aug. 2005, pp. 219–220.
- [47] J. Li, D. Dou, Z. Wu, S. Kim, and V. Agarwal, "An Internet routing forensics framework for discovering rules of abnormal BGP events," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 5, p. 55–66, Oct 2005.
- [48] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1153–1176, Second quarter 2016.
- [49] P. Mishra, V. Varadharajan, U. Tupakula, and E. S. Pilli, "A detailed investigation and analysis of using machine learning techniques for intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 686–728, First quarter 2019.
- [50] C. Wagner, J. Francois, R. State, and T. Engel, "Machine learning approach for IP-flow record anomaly detection," in *Lecture Notes in Computer Science: Proc. 10th Int. IFIP TC 6 Netw. Conf.*, J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, and C. Scoglio, Eds. Berlin: Springer, 2011, vol. 6640, pp. 28–39.
- [51] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long short term memory networks for anomaly detection in time series," in *Proc. Euro. Symp. Artif. Neural Netw. Comput. Intell. Mach. Learn.*, Bruges, Belgium, Apr. 2015, pp. 89–94.
- [52] M. Cheng, Q. Xu, J. Lv, W. Liu, Q. Li, and J. Wang, "MS-LSTM: a multi-scale LSTM model for BGP anomaly detection," in *Proc. IEEE 24th Int. Conf. Netw. Protocols*, Singapore, Nov. 2016, pp. 1–6.
- [53] D. Ariu, R. Tronci, and G. Giacinto, "HMMPayl: an intrusion detection system based on hidden Markov model," *Computers & Security*, vol. 30, no. 4, pp. 221–241, June 2011.
- [54] W. Zong, G. B. Huang, and Y. Chen, "Weighted extreme learning machine for imbalance learning," *Neurocomputing*, vol. 101, pp. 229–242, Feb. 2013.

- [55] K. Zhang, X. Z. A. Yen, D. Massey, S. Wu, and L. Zhang, "On detection of anomalous routing dynamics in BGP," in *Lecture Notes in Computer Science: Proc. Int. Conf. Research Netw.*, N. Mitrou, K. Kontovasilis, G. N. Rouskas, I. Iliadis, and L. Merakos, Eds. Berlin: Springer, 2004, vol. 3042, pp. 259–270.
- [56] K. Bekshentayeva and Lj. Trajković, "Detection of denial of service attacks using echo state networks," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Melbourne, Australia, Oct. 2021, pp. 1227–1232.
- [57] B. Al-Musawi, P. Branch, and G. Armitage, "BGP anomaly detection techniques: a survey," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 1, pp. 377–396, 2017.
- [58] M. C. Libicki, L. Ablon, and T. Webb, *The Defender's Dilemma: Charting a Course Toward Cybersecurity*. Santa Monica, CA, USA: RAND Corporation, June 2015.
- [59] R. Samrin and D. Vasumathi, "Review on anomaly based network intrusion detection system," in *Proc. Int. Conf. Elect., Electron., Commun., Comput., Optim. Techn.*, Mysuru, India, Dec. 2017, pp. 141–147.
- [60] J. Zhang and M. Zulkernine, "A hybrid network intrusion detection technique using random forests," in *Proc. First Int. Conf. Availability, Rel. Secur.*, Vienna, Austria, Apr. 2006, pp. 262–269.
- [61] Y. Jia, M. Wang, and Y. Wang, "Network intrusion detection algorithm based on deep neural network," *IET Inf. Secur.*, vol. 13, no. 1, pp. 48–53, Jan. 2019.
- [62] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, "A deep learning approach to network intrusion detection," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 2, no. 1, pp. 41–50, Feb. 2018.
- [63] C. Yin, Y. Zhu, J. Fei, and X. He, "A deep learning approach for intrusion detection using recurrent neural networks," *IEEE Access*, vol. 5, pp. 21 954–21 961, Nov. 2017.
- [64] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, "Deep learning approach for network intrusion detection in software defined networking," in *Proc. Wireless Netw. Mobile Commun.*, Fez, Morocco, Oct. 2016, pp. 258–263.
- [65] D. J. Weller-Fahy, B. J. Borghetti, and A. A. Sodemann, "A survey of distance and similarity measures used within network intrusion anomaly detection," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 70–91, 2015.
- [66] L. Li, Y. Yu, S. Bai, Y. Hou, and X. Chen, "An effective two-step intrusion detection approach based on binary classification and k-NN," *IEEE Access*, vol. 6, pp. 12 060–12 073, Mar. 2018.
- [67] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1993.
- [68] D. Jin, Y. Lu, J. Qin, Z. Cheng, and Z. Mao, "SwiftIDS: real-time intrusion detection system based on LightGBM and parallel intrusion detection mechanism," *Computers & Security*, vol. 97, no. 101984, pp. 1–12, Oct. 2020.

- [69] T. Shapira and Y. Shavitt, “AP2Vec: an unsupervised approach for BGP hijacking detection,” *IEEE Trans. Netw. Serv. Manag.*, Apr. 2022.
- [70] (2022, Apr.) BGProtect. [Online]. Available: <https://www.bgprotect.com>.
- [71] (2022, Apr.) Secure IPS (NGIPS). [Online]. Available: [https://www.cisco.com/c/en\\_ca/products/security/ngips/index.html](https://www.cisco.com/c/en_ca/products/security/ngips/index.html).
- [72] (2022, Apr.) FortiGuard IPS Security Service. [Online]. Available: <https://www.fortinet.com/products/ips>.
- [73] (2022, Apr.) Advanced Threat Prevention. [Online]. Available: <https://www.paloaltonetworks.com/network-security/advanced-threat-prevention>.
- [74] N. Chaabouni, M. Mosbah, A. Zemmari, C. Sauvignac, and P. Faruki, “Network intrusion detection for IoT security based on learning techniques,” *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2671–2701, Third quarter 2019.
- [75] L. N. Tidjon, M. Frappier, and A. Mammar, “Intrusion detection systems: a cross-domain overview,” *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3639–3681, Fourth quarter 2019.
- [76] B. Molina-Coronado, U. Mori, A. Mendiburu, and J. Miguel-Alonso, “Survey of network intrusion detection methods from the perspective of the knowledge discovery in databases process,” *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 4, pp. 2451 – 2479, Dec. 2020.
- [77] (2022, Apr.) KDD Cup 1999 Data. [Online]. Available: <http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>.
- [78] R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, and M. A. Zissman, “Evaluating intrusion detection systems: the 1998 darpa off-line intrusion detection evaluation,” in *Proc. DARPA Inform. Survivability Conf. Expo.*, Hilton Head, SC, USA, Jan. 2000, pp. 12–26.
- [79] S. J. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. K. Chan, “Cost-based modeling for fraud and intrusion detection: Results from the JAM project,” in *Proc. USENIX Workshop Tackling Comput. Syst. Problems Mach. Learn. Techn.*, Hilton Head, SC, USA, Jan. 2000, pp. 130–144.
- [80] (2022, Apr.) 2000 DARP intrusion Detection Scenario Specific Datasets. [Online]. Available: <https://www.ll.mit.edu/r-d/datasets/2000-darpa-intrusion-detection-scenario-specific-datasets>.
- [81] (2022, Apr.) CAIDA. [Online]. Available: <https://www.caida.org>.
- [82] (2022, Apr.) Border Gateway Protocol Routing Records from Réseaux IP Européens (RIPE) and BCNET. [Online]. Available: <http://ieee-dataport.org/1977>.
- [83] (2022, Apr.) UNIBS: Data sharing. [Online]. Available: <http://netweb.ing.unibs.it/~ntw/tools/traces>.

- [84] N. Moustafa and J. Slay, “UNSW-NB15: a comprehensive data set for network intrusion detection systems,” in *Proc. Military Commun. Inf. Syst. Conf.*, Canberra, ACT, Australia, Nov. 2015, pp. 1–6.
- [85] (2022, Apr.) Canadian Institute for Cybersecurity datasets. [Online]. Available: <https://www.unb.ca/cic/datasets/index.html>.
- [86] (2022, Apr.) MS SQL Slammer/Sapphire worm, SANS Institute GIAC Certifications. [Online]. Available: <https://www.giac.org/paper/gsec/3091/ms-sql-slammer-sapphire-worm/105136>.
- [87] (2022, Apr.) Attack of Slammer worm - a practical case study, SANS Institute GIAC Certifications. [Online]. Available: <https://www.giac.org/paper/gcih/414/attack-slammer-worm-practical-case-study/103632>.
- [88] (2022, Apr.) Responding to the Nimda worm: recommendations for addressing blended threats, Symantec, Cupertino, CA, USA. [Online]. Available: <https://vx-underground.org/archive/Symantec/nimda-worm-recommendations-blended-threats-01-en.pdf>.
- [89] (2022, Apr.) A challenging response to Nimda, SANS Institute GIAC Certifications. [Online]. Available: <https://www.giac.org/paper/gcih/273/challenging-response-nimda/102847>.
- [90] (2022, Apr.) The Code Red worm, SANS Institute Information Security Reading Room. [Online]. Available: <https://www.sans.org/reading-room/whitepapers/malicious/code-red-worm-85>.
- [91] (2022, Apr.) Intrusion detection evaluation dataset (ISCXIDS2012). [Online]. Available: <https://www.unb.ca/cic/datasets/ids.html>.
- [92] (2022, Apr.) CIC-Darknet2020. [Online]. Available: <https://www.unb.ca/cic/datasets/darknet2020.html>.
- [93] (2022, Apr.) RFC 1771 - a border gateway protocol 4 (BGP-4). [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc1771>.
- [94] D. P. Watson and D. H. Scheidt, “Autonomous systems,” *Johns Hopkins APL Technical Digest*, vol. 26, no. 4, pp. 368–376, Oct.-Dec. 2005.
- [95] (2022, Apr.) RFC 4271 - a border gateway protocol 4 (BGP-4). [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc4271>.
- [96] J. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach (sixth edition)*. Boston, MA, USA: Addison-Wesley, 2012.
- [97] H. Yan, R. Oliveira, K. Burnett, D. Matthews, L. Zhang, and D. Massey, “BGPmon: a real-time, scalable, extensible monitoring system,” in *Proc. Cybersecurity Appl. Technol. Conf. Homeland Secur.*, Washington, DC, USA, Mar. 2009, pp. 212–223.
- [98] K. Sriram, O. Borchert, O. Kim, P. Gleichmann, and D. Montgomery, “A comparative analysis of BGP anomaly detection and robustness algorithms,” in *Proc. Cybersecurity Appl. Technol. Conf. Homeland Secur.*, Washington, DC, USA, Mar. 2009, pp. 25–38.

- [99] B. Zhang, R. Liu, D. Massey, and L. Zhang, “Collecting the Internet AS-level topology,” *ACM Computer Communication Review*, vol. 35, no. 1, pp. 53–62, Jan. 2005.
- [100] (2022, Apr.) YouTube Hijacking: A RIPE NCC RIS case study. [Online]. Available: <http://www.ripe.net/internet-coordination/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study>.
- [101] (2022, Apr.) Quagga routing suite. [Online]. Available: <https://www.quagga.net/index.html>.
- [102] (2022, Apr.) Zebra. [Online]. Available: <http://www.zebra.org>.
- [103] (2022, Apr.) RIPE NCC routing information service. [Online]. Available: <https://www.ripe.net/publications/docs/ripe-200>.
- [104] (2022, Apr.) Malware 101 - Viruses, SANS Institute. [Online]. Available: <https://www.sans.org/reading-room/whitepapers/incident/malware-101-viruses-32848>.
- [105] (2022, Apr.) Threat chaos: making sense of the online threat landscape, Webroot Software, Inc. [Online]. Available: [https://www.webroot.com/pdf/WP\\_Threat\\_0105.pdf](https://www.webroot.com/pdf/WP_Threat_0105.pdf).
- [106] Y. V. Makarov, V. I. Reshetov, V. A. Stroev, and N. I. Voropai, “Blackout prevention in the United States, Europe, and Russia,” *Proc. IEEE*, vol. 93, no. 11, pp. 1942–1955, Nov. 2005.
- [107] (2022, Apr.) How ransomware attacks, SophosLabs. [Online]. Available: <https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/sophoslabs-ransomware-behavior-report.pdf>.
- [108] (2022, Apr.) Stemming the exploitation of ict threats and vulnerabilities, United Nations Institute for Disarmament Research (UNIDIR). [Online]. Available: <https://unidir.org/files/publications/pdfs/stemming-the-exploitation-of-ict-threats-and-vulnerabilities-en-805.pdf>.
- [109] (2022, Apr.) RAO “UES of Russia” annual report 2005. [Online]. Available: [http://www.rustocks.com/put.phtml/EESR\\_2005\\_sec.pdf](http://www.rustocks.com/put.phtml/EESR_2005_sec.pdf).
- [110] (2022, Apr.) Report on the investigation of the accident in the UES of Russia on May 25, 2005. [Online]. Available: [http://www.kef.ru/art\\_010.shtml](http://www.kef.ru/art_010.shtml).
- [111] (2022, Apr.) Moscow electroshock. [Online]. Available: <https://www.comnews.ru/content/13693>.
- [112] (2022, Apr.) North American Network Operators Group Mailing List Archive. [Online]. Available: [https://archive.nanog.org/maillinglist/mailarchives/old\\_archive/2005-05/index.html](https://archive.nanog.org/maillinglist/mailarchives/old_archive/2005-05/index.html).
- [113] (2022, Apr.) Protecting electricity networks from natural hazards. [Online]. Available: <https://www.osce.org/secretariat/242651>.

- [114] (2022, Apr.) Reverse Engineering of WannaCry Worm and Anti Exploit Snort Rules, SANS Institute. [Online]. Available: <https://www.sans.org/reading-room/whitepapers/malicious/reverse-engineering-wannacry-worm-anti-exploit-snort-rules-38445>.
- [115] (2022, Apr.) EternalBlue: a prominent threat actor of 2017–2018, Virus Bulletin. [Online]. Available: <https://www.virusbulletin.com/uploads/pdf/magazine/2018/201806-EternalBlue.pdf>.
- [116] N. Al-Rousan, S. Haeri, and Lj. Trajković, “Feature selection for classification of bgp anomalies using bayesian models,” in *Proc. Int. Conf. Mach. Learn. Cybern.*, Xi’an, China, July 2012, pp. 140–147.
- [117] (2022, Apr.) RFC 4384 - BGP communities for data collection. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc4384>.
- [118] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. F. Wu, and L. Zhang, “Observation and analysis of BGP behavior under stress,” in *Proc. 2nd ACM SIGCOMM Workshop on Internet Meas.*, New York, NY, USA, Nov. 2002, pp. 183–195.
- [119] Q. Ding, Z. Li, S. Haeri, and Lj. Trajković, “Application of machine learning techniques to detecting anomalies in communication networks: datasets and feature selection algorithms,” in *Cyber Threat Intelligence*, A. Dehghantanha, M. Conti, and T. Dargahi, Eds. Berlin: Springer, 2018, pp. 47–70.
- [120] (2022, Apr.) BGP C sharp tool. [Online]. Available: [https://github.com/communication-networks-laboratory/BGP\\_c\\_sharp\\_tool](https://github.com/communication-networks-laboratory/BGP_c_sharp_tool).
- [121] (2022, Apr.) BCNET. [Online]. Available: <http://www.bc.net>.
- [122] S. Lally, T. Farah, R. Gill, R. Paul, N. Al-Rousan, and Lj. Trajković, “Collection and characterization of BCNET BGP traffic,” in *Proc. IEEE Pacific Rim Conf. Commun., Comput., Signal Process.*, Victoria, BC, Canada, Aug. 2011, pp. 830–835.
- [123] T. Farah, S. Lally, R. Gill, N. Al-Rousan, R. Paul, D. Xu, and Lj. Trajković, “Collection of BCNET BGP traffic,” in *Proc. 23rd Int. Teletraffic Congress*, San Francisco, CA, USA, Sept. 2011, pp. 322–323.
- [124] J. McHugh, “Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln laboratory,” *ACM Trans. Inform. Syst. Secur.*, vol. 3, no. 4, pp. 262–294, Nov. 2000.
- [125] A. A. Olusola, A. S. Oladele, and D. O. Abosede, “Analysis of KDD’99 intrusion detection dataset for selection of relevance features,” in *Proc. World Congress Eng. Comput. Sci.*, San Francisco, CA, USA, Oct. 2010, pp. 162–168.
- [126] W. Heyi, H. Aiqun, S. Yubo, B. Ning, and J. Xuefei, “A new intrusion detection feature extraction method based on complex network theory,” in *Proc. 4<sup>th</sup> Int. Conf. Multimedia Inf. Netw. Secur.*, Nanjing, China, Nov. 2012, pp. 852–856.
- [127] W. Lee, S. J. Stolfo, and K. W. Mok, “Mining in a data-flow environment: experience in network intrusion detection,” in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery and Data Mining*, San Diego, CA, USA, Aug. 1999, pp. 114–124.

- [128] M. Tavallaei, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *Proc. IEEE Symp. Comput. Intell. Secur. Defense Appl.*, Ottawa, ON, Canada, July 2009, pp. 1–6.
- [129] I. Sharafaldin, A. Gharib, A. H. Lashkari, and A. A. Ghorbani, "Towards a reliable intrusion detection benchmark dataset," *Softw. Netw.*, vol. 2017, no. 1, pp. 177–200, July 2017.
- [130] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proc. 4<sup>th</sup> Int. Conf. Inform. Syst. Secur. Privacy*, Funchal, Portugal, Jan. 2018, pp. 108–116.
- [131] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, "Developing realistic distributed denial of service (DDoS) attack dataset and taxonomy," in *Proc. Int. Carnahan Conf. Secur. Technol.*, Chennai, India, Oct. 2019, pp. 1–8.
- [132] G. H. John, R. Kohavi, and K. Pfleger, "Irrelevant features and the subset selection problem," in *Proc. Int. Conf. Machine Learning*, New Brunswick, NJ, USA, July 1994, pp. 121–129.
- [133] M. N. A. Kumar and H. S. Sheshadri, "On the classification of imbalanced datasets," *Int. J. Comput. Appl.*, vol. 44, no. 8, pp. 1–7, Apr. 2012.
- [134] Y.-W. Chen and C.-J. Lin, "Combining SVMs with various feature selection strategies," *Strategies*, vol. 324, no. 1, pp. 1–10, Nov. 2006.
- [135] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. Hoboken, NJ, USA: Wiley-Interscience Publication, 2001.
- [136] Q. Gu, Z. Li, and J. Han, "Generalized fisher score for feature selection," in *Proc. Conf. Uncertainty in Artif. Intell.*, Barcelona, Spain, July 2011, pp. 266–273.
- [137] J. Wang, X. Chen, and W. Gao, "Online selecting discriminative tracking features using particle filter," in *Proc. IEEE Conf. Comput. Vis. Pattern Recog.*, San Diego, CA, USA, June 2005, pp. 1037–1042.
- [138] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 8, pp. 1226–1238, Aug. 2005.
- [139] J. Chen, H. Huang, S. Tian, and Y. Qu, "Feature selection for text classification with naive Bayes," *Expert Systems with Applications*, vol. 36, no. 3, pp. 5432–5435, Apr. 2009.
- [140] (2022, Apr.) mRMR feature selection (using mutual information computation). [Online]. Available: <https://www.mathworks.com/matlabcentral/fileexchange/14608-mrmr-feature-selection-using-mutual-information-computation>.
- [141] X.-Z. Wang, L. C. Dong, and J. H. Yan, "Maximum ambiguity based sample selection in fuzzy decision tree induction," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 8, pp. 1491–1505, Aug. 2012.

- [142] L. Breiman, “Bagging predictors,” *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996.
- [143] L. Rokach and O. Maimon, “Top-down induction of decision trees classifiers – a survey,” *IEEE Trans. Syst., Man, Cybern., Part C (Appl. Rev.)*, vol. 35, no. 4, pp. 476–487, Nov. 2005.
- [144] (2022, Apr.) Data Mining Tools See5 and C5.0. [Online]. Available: <https://www.rulequest.com/see5-info.html>.
- [145] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, Apr. 2006.
- [146] G. Louppe, L. Wehenkel, A. Suter, and P. Geurts, “Understanding variable importances in forests of randomized trees,” in *Proc. Int. Conf. Neural Inform. Process. Syst.*, Lake Tahoe, NV, USA, Dec. 2013, pp. 431–439.
- [147] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *Science*, vol. 313, pp. 504–507, July. 2006.
- [148] G. E. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, “Deep neural networks for acoustic modeling in speech recognition,” *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.
- [149] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.
- [150] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional LSTM and other neural network architectures,” *Neural Netw.*, vol. 18, no. 5–6, pp. 602–610, July/Aug. 2005.
- [151] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Oct. 1997.
- [152] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, “LSTM: a search space odyssey,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017.
- [153] K. Cho, B. van Merriënboer, C. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translations,” in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Doha, Qatar, Oct. 2014, pp. 1724–1734.
- [154] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han, “Enhanced network anomaly detection based on deep neural networks,” *IEEE Access*, vol. 6, pp. 48 231–48 246, Aug. 2018.
- [155] (2022, Apr.) Broadlearning. [Online]. Available: <https://broadlearning.ai>.
- [156] T. Chen and C. Guestrin, “XGBoost: a scalable tree boosting system,” in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, San Francisco, CA, USA, Aug. 2016, pp. 785–794.

- [157] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, “CatBoost: unbiased boosting with categorical features,” in *Proc. Int. Conf. Neural Inform. Process. Syst.*, Montreal, Quebec, Canada, Dec. 2018, pp. 6639–6649.
- [158] F. Provost, T. Fawcett, and R. Kohavi, “The case against accuracy estimation for comparing induction algorithms,” in *Proc. Int. Conf. Mach. Learn.*, Madison, WI, USA, July 1998, pp. 445–453.
- [159] C. Cortes and V. Vapnik, “Support-vector networks,” *Mach. Learn.*, vol. 20, pp. 273–297, Sept. 1995.
- [160] K. Morik, P. Brockhausen, and T. Joachims, “Combining statistical learning with a knowledge-based approach – a case study in intensive care monitoring,” in *Proc. Int. Conf. Mach. Learn.*, Bled, Slovenia, June 1999, pp. 268–277.
- [161] C.-W. Hsu and C.-J. Lin, “A comparison of methods for multiclass support vector machines,” *IEEE Trans. Neural Netw.*, vol. 13, no. 2, pp. 415–425, Mar. 2002.
- [162] D. Mladenic and M. Grobelnik, “Feature selection for unbalanced class distribution and naive Bayes,” in *Proc. Int. Conf. Mach. Learn.*, Bled, Slovenia, June 1999, pp. 258–267.
- [163] G. B. Huang, Q. Y. Zhu, and C. K. Siew, “Extreme learning machine: theory and applications,” *Neurocomputing*, vol. 70, no. 1, pp. 489–501, Dec. 2006.
- [164] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Mach. Learn.*, vol. 8, no. 3, pp. 229–256, May 1992.
- [165] (2022, Apr.) PyTorch. [Online]. Available: <https://pytorch.org>.
- [166] Z. Li, A. L. Gonzalez Rios, and Lj. Trajković, “Machine learning for detecting anomalies and intrusions in communication networks,” *IEEE J. Sel. Areas Commun.*, vol. 39, no. 7, pp. 2254–2264, July 2021.
- [167] Y.-H. Pao, G.-H. Park, and D. J. Sobajic, “Learning and generalization characteristics of the random vector functional-link net,” *Neurocomputing*, vol. 6, no. 2, pp. 163–180, Apr. 1994.
- [168] D. P. Kingma and J. Ba, “Adam: a method for stochastic optimization,” in *Proc. 3rd Int. Conf. Learn. Representations*, San Diego, CA, USA, May 2015, pp. 1–15.
- [169] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *Computing Research Repository (CoRR)*, vol. abs/1207.0580, pp. 1–18, July 2012.
- [170] J. Friedman, “Greedy function approximation: a gradient boosting machine,” *Ann. Statist.*, vol. 29, no. 5, p. 1189–1232, Apr. 2001.
- [171] (2022, Apr.) Cedar. [Online]. Available: <https://docs.computecanada.ca/wiki/Cedar>.
- [172] (2022, Apr.) Sklearn.ensemble.ExtraTreesClassifier. [Online]. Available: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>.

- [173] (2022, Apr.) The MNIST Database of Handwritten Digits. [Online]. Available: <http://yann.lecun.com/exdb/mnist>.
- [174] (2022, Apr.) zebra-dump-parser. [Online]. Available: <https://github.com/rfc1036/zebra-dump-parser>.
- [175] Q. Ding, Z. Li, S. Haeri, and L. Trajković, “Application of machine learning techniques to detecting anomalies in communication networks,” in *Cyber Threat Intelligence*, A. Dehghantanha, M. Conti, and T. Dargahi, Eds. Berlin: Springer, 2018, pp. 47–70 and pp. 71–92.
- [176] (2022, Apr.) Teach, learn, and make with Raspberry Pi. [Online]. Available: <https://www.raspberrypi.org>.
- [177] (2022, Apr.) Pip. [Online]. Available: <https://pip.pypa.io/en/stable/>.
- [178] (2022, Apr.) NumPy. [Online]. Available: <https://numpy.org>.
- [179] (2022, Apr.) SciPy. [Online]. Available: <https://scipy.org>.
- [180] (2022, Apr.) Scikit-learn: machine learning in Python. [Online]. Available: <https://scikit-learn.org/stable>.
- [181] (2022, Apr.) Mono. [Online]. Available: <https://www.mono-project.com>.
- [182] (2022, Apr.) Bootstrap. [Online]. Available: <https://getbootstrap.com>.
- [183] (2022, Apr.) Socket.IO. [Online]. Available: <https://socket.io>.
- [184] (2022, Apr.) Flask. [Online]. Available: <https://flask.palletsprojects.com/en/2.0.x>.
- [185] (2022, Apr.) jsDelivr. [Online]. Available: <https://www.jsdelivr.com>.
- [186] (2022, Apr.) Werkzeug. [Online]. Available: <https://werkzeug.palletsprojects.com/en/2.0.x>.
- [187] (2022, Apr.) Jinja. [Online]. Available: <https://jinja.palletsprojects.com/en/3.0.x>.
- [188] (2022, Apr.) Flask-SocketIO. [Online]. Available: <https://flask-socketio.readthedocs.io/en/latest>.
- [189] (2022, Apr.) Python-socketio. [Online]. Available: <https://python-socketio.readthedocs.io/en/latest>.
- [190] (2022, Apr.) Python-engineio. [Online]. Available: <https://github.com/miguelgrinberg/python-engineio>.
- [191] (2022, Apr.) Eventlet. [Online]. Available: <https://eventlet.net>.
- [192] (2022, Apr.) Cloud computing services – Amazon web services (AWS). [Online]. Available: <https://aws.amazon.com>.
- [193] (2022, Apr.) Cloud application platform | Heroku. [Online]. Available: <https://www.heroku.com>.
- [194] (2022, Apr.) Secure and resizable cloud compute – Amazon EC2. [Online]. Available: <https://aws.amazon.com/ec2>.

## Appendix A

# Anomaly Classification: Machine Learning Algorithms

Listed is Python code of the developed VFBLs and VCFBLs algorithms.

Listing A.1: VFBLs

```
1  """
2      @authors Zhida Li
3      @email zhidal@sfu.ca
4      @date July 10, 2020
5
6      @copyright Copyright (c) July 10, 2020
7          All Rights Reserved
8
9  """
10
11 #####
12 ##### VFBLs
13 #####
14
15 # Import the Python libraries
16 import time
17 import random
18 import numpy as np
19 import sys
20 import os
21
22 from scipy.stats import zscore
23 from scipy.linalg import orth
24 from numpy.linalg import inv
25 import math
26
27 #sys.path.append("..")
28 from bls.processing.result import result
29 from bls.processing.sparse_bls import sparse_bls
30
31 from sklearn import preprocessing
32 from bls.processing.mapminmax import mapminmax
33
34 from sklearn.metrics import f1_score
```

```

35 from sklearn.metrics import accuracy_score
36
37
38 # sys.path.append('../processing')
39 from bls.processing.feature_select_cnl import feature_select_cnl
40 from bls.processing.one_hot_m import one_hot_m
41
42 """
43     Function that creates the BLS model. It returns the training and test
44     accuracy, training and
45     testing time, and testing F-Score.
46     bls_train_fscore(train_x, train_y, test_x, test_y, s, C, N1, N2, N3)
47     'train_x' and 'test_x' are the entire training data and test data.
48     'train_y' and 'test_y' are the entire training labels and test
49     labels.
50     's' is the shrinkage parameter for enhancement nodes.
51     'C' is the parameter for sparse regularization.
52     'N1' is the number of mapped feature nodes.
53     'N2' are the groups of mapped features.
54     'N3' is the number of enhancement nodes.
55
56     Randomly generated weights for the mapped features and enhancement
57     nodes are
58     stored in the
59     matrices 'we' and 'wh,' respectively.
60 """
61 # Disable
62 def blockPrint():
63     sys.stdout = open(os.devnull, 'w')
64
65 # Restore
66 def enablePrint():
67     sys.stdout = sys.__stdout__
68
69
70 # rectified linear function
71 def relu(data):
72     return np.maximum(data,0)
73
74 # RBF function
75 def kerf( matrix ):
76     return np.exp( np.multiply(-1 * matrix, matrix) / 2.0 ) / np.sqrt(2 *
77     math.pi);
78
79 def vfbls_train_fscore(train_x, train_y, test_x, test_y, s, C, N1, N2, N3,
80     N1_bls_fsm1, N2_bls_fsm1, N1_bls_fsm2, N2_bls_fsm2, add_nfeature1,
81     add_nfeature2):
82     # blockPrint()
83
84     time_start=time.time()
85     num_add_vf = 2
86     ### feature selection - begin
87     # add_nfeature1 = larger
88     features = feature_select_cnl(train_x, train_y, -1)
89     feature1 = features[0:add_nfeature1]
90     train_x1 = train_x[:,feature1]
91     test_x1 = test_x[:,feature1]
92
93     # add_nfeature2 = smaller

```

```

87     # features = feature_select_cnl(train_x, train_y, add_nfeature2)
88     feature2 = features[0:add_nfeature2]
89     train_x2 = train_x[:,feature2]
90     test_x2 = test_x[:,feature2]
91     ### feature selection - end
92     train_x_vf = [1 ,train_x1, train_x2]
93     test_x_vf = [1 ,test_x1, test_x2]
94
95     N1_vfbcls = [1, N1_bls_fsm1, N1_bls_fsm2]
96     N2_vfbcls = [1, N2_bls_fsm1, N2_bls_fsm2]
97
98     # Training - begin
99     # torch.cuda.synchronize()
100    # time_start=time.time()
101    beta11 = [];
102
103    train_x = zscore(train_x.transpose(), axis = 0, ddof = 1).transpose();
104    # print(train_x.shape)
105    H1 = np.concatenate((train_x, 0.1 * np.ones((train_x.shape[0], 1))),
106    axis=1);
107
108    y = np.zeros((train_x.shape[0], N2 * N1));
109
110    max_list_set = [];
111    min_list_set = [];
112
113    ### Generation of mapped features
114    for i in range(0, N2):
115
116        #we = 2.0 * np.random.rand(train_x.shape[1] + 1, N1) - 1.0;
117        we = 2.0 * np.random.rand(N1, train_x.shape[1] + 1).transpose() -
118        1.0;
119
120        #we = np.loadtxt("we.csv", delimiter = ",");
121        #np.savetxt("we.csv", we, delimiter=",");
122
123        A1 = np.dot(H1, we);
124        [A1, max_list, min_list] = mapminmax(A1);
125        del we;
126
127        beta1 = sparse_bls(A1, H1, 1e-3, 50).transpose();
128        beta11.append(beta1);
129        T1 = np.dot(H1, beta1);
130
131        # print("Feature nodes in window ", i, ": Max Val of Output ", T1.
132        max(), " Min Val ", T1.min());
133
134        [T1, max_list, min_list] = mapminmax(T1.transpose(), 0, 1);
135        T1 = T1.transpose();
136
137        max_list_set.append(max_list);
138        min_list_set.append(min_list);
139
140        y[:, N1 * i: N1 * (i + 1)] = T1;
141
142    del H1;
143    del T1;

```

```

142 ##### NEW
143 #####
144 beta11_ivf = [1,1,1]
145 max_list_set_ivf = [1,1,1]
146 min_list_set_ivf = [1,1,1]
147 for ivf in range(1,num_add_vf+1):
148     beta11_fsm = [];
149
150     train_x_fsm = zscore(train_x_vf[ivf].transpose(), axis = 0, ddof =
151 1).transpose();
152     H1_fsm = np.concatenate((train_x_fsm, 0.1 * np.ones((train_x_fsm.
153 shape[0], 1))), axis=1);
154
155     y_fsm = np.zeros((train_x_fsm.shape[0], N2_vfbcls[ivf] * N1_vfbcls[ivf
156 ]));
157
158     max_list_set_fsm = [];
159     min_list_set_fsm = [];
160
161     ### Generation of mapped features
162     for i in range(0, N2_vfbcls[ivf]):
163
164         #we = 2.0 * np.random.rand(train_x.shape[1] + 1, N1) - 1.0;
165         we_fsm = 2.0 * np.random.rand(N1_vfbcls[ivf], train_x_fsm.shape
166 [1] + 1).transpose() - 1.0;
167
168         #we = np.loadtxt("we.csv", delimiter = ",");
169         #np.savetxt("we.csv", we, delimiter=",");
170
171         A1_fsm = np.dot(H1_fsm, we_fsm);
172         [A1_fsm, max_list_fsm, min_list_fsm] = mapminmax(A1_fsm);
173         del we_fsm;
174
175         beta1_fsm = sparse_bls(A1_fsm, H1_fsm, 1e-3, 50).transpose();
176         beta11_fsm.append(beta1_fsm);
177         T1_fsm = np.dot(H1_fsm, beta1_fsm);
178
179         # print("Feature nodes in window ", i, ": Max Val of Output ",
180 T1_fsm.max(), " Min Val ", T1_fsm.min());
181
182         [T1_fsm, max_list_fsm, min_list_fsm] = mapminmax(T1_fsm.
183 transpose(), 0, 1);
184         T1_fsm = T1_fsm.transpose();
185
186         max_list_set_fsm.append(max_list_fsm);
187         min_list_set_fsm.append(min_list_fsm);
188
189         y_fsm[:, N1_vfbcls[ivf] * i : N1_vfbcls[ivf] * (i + 1)] = T1_fsm;
190
191         del H1_fsm;
192         del T1_fsm;
193         beta11_ivf[ivf] = beta11_fsm
194         max_list_set_ivf[ivf] = max_list_set_fsm
195         min_list_set_ivf[ivf] = min_list_set_fsm
196         y = np.concatenate((y, y_fsm), axis=1);
197     y = np.concatenate((train_x,y), axis=1);
198 ##### NEW
199 #####

```

```

192     train_y = one_hot_m(train_y, 2);
193     test_y = one_hot_m(test_y, 2);
194
195     ### Generation of enhancement nodes
196     H2 = np.concatenate((y, 0.1 * np.ones((y.shape[0], 1))), axis=1);
197
198     if (N1 * N2 + N2_vfbcls[1] * N1_vfbcls[1] + N2_vfbcls[2] * N1_vfbcls[2] +
199         train_x.shape[1]) >= N3:
200         #wh = orth(2 * np.random.rand(N2 * N1 + 1, N3) - 1);
201         wh = orth(2 * np.random.rand(N3, (N1 * N2 + N2_vfbcls[1] * N1_vfbcls
202             [1] + N2_vfbcls[2] * N1_vfbcls[2] + train_x.shape[1]) + 1).transpose() - 1);
203     else:
204         #wh = orth(2 * np.random.rand(N2 * N1 + 1, N3).transpose() - 1).
205         transpose();
206         wh = orth(2 * np.random.rand(N3, (N1 * N2 + N2_vfbcls[1] * N1_vfbcls
207             [1] + N2_vfbcls[2] * N1_vfbcls[2] + train_x.shape[1]) + 1) - 1).transpose();
208
209     #wh = np.loadtxt("wh.csv", delimiter = ",");
210     #np.savetxt("wh.csv", wh, delimiter=",");
211
212     T2 = np.dot(H2, wh);
213     l2 = T2.max();
214     l2 = s * 1.0 / l2;
215
216     # print("Enhancement nodes: Max Val of Output ", l2, " Min Val ", T2.min
217     ());
218
219     T2 = np.tanh(T2 * l2);
220     T3 = np.concatenate((y, T2), axis=1);
221
222     del H2;
223     del T2;
224     # print((np.dot(T3.transpose(), T3) + np.identity(T3.transpose().shape
225     [0]) * C).shape)
226     # Moore-Penrose pseudoinverse (function pinv)
227     beta = np.dot(inv(np.dot(T3.transpose(), T3) + np.identity(T3.transpose
228     ().shape[0]) * C),
229         np.dot(T3.transpose(), train_y));
230
231     # T3_i1 = torch.mm(torch.from_numpy(T3.T), torch.from_numpy(T3))
232     # T3_i2 = torch.from_numpy(np.identity(T3.T.shape[0]) * C)
233     # T3_i = T3_i1 + T3_i2
234     # print(type(T3_i))
235     # T3_i = T3_i.cuda()
236     # T3_inv = torch.inverse(T3_i)
237     # beta = torch.mm(T3_inv, torch.mm((torch.from_numpy(T3.T)).cuda(), (
238     torch.from_numpy(train_y)).cuda()));
239     # print(type(T3_i))
240     # beta = beta.detach().cpu().numpy()
241
242     xx = np.dot(T3, beta);
243
244     del T3;
245
246     # torch.cuda.synchronize()
247     time_end=time.time()

```

```

242     Training_time = time_end - time_start
243
244     # Training - end
245
246     print("Training has been finished!");
247     print("The Total Training Time is : ", Training_time, " seconds");
248
249     ### Training Accuracy
250     yy = result(xx);
251     train_yy = result(train_y);
252
253     cnt = 0;
254     for i in range(0, len(yy)):
255         if yy[i] == train_yy[i]:
256             cnt = cnt + 1;
257
258     TrainingAccuracy = cnt * 1.0 / train_yy.shape[0];
259
260     print("Training Accuracy is : ", TrainingAccuracy * 100, " %");
261
262     ### Testing Process
263     # Testing - begin
264     time_start=time.time()
265     test_x = zscore( np.float128(test_x).transpose() ,axis = 0, ddof = 1).
transpose();
266
267     HH1 = np.concatenate((test_x, 0.1 * np.ones((test_x.shape[0], 1))), axis
=1);
268     yy1 = np.zeros((test_x.shape[0], N2 * N1));
269
270     ### Generation of mapped features
271     for i in range(0, N2):
272
273         beta1 = beta11[i];
274
275         TT1 = np.dot( np.float128(HH1), np.float128(beta1)) ;
276
277         max_list = max_list_set[i];
278         min_list = min_list_set[i];
279
280         [TT1, max_list, min_list] = mapminmax( TT1.transpose(), 0, 1,
max_list, min_list);
281         TT1 = TT1.transpose();
282
283         del beta1;
284         del max_list;
285         del min_list;
286
287         yy1[:, N1 * i: N1 * (i + 1)] = TT1;
288
289     del TT1;
290     del HH1;
291
292     ##### NEW
293     #####
294     for ivf in range(1,num_add_vf+1):
295         test_x_fsm = zscore( np.float128(test_x_vf[ivf]).transpose() ,axis =
0, ddof = 1).transpose();

```

```

295     HH1_fsm = np.concatenate((test_x_fsm, 0.1 * np.ones((test_x_fsm.
296     shape[0], 1))), axis=1);
297     yy1_fsm = np.zeros((test_x_fsm.shape[0], N2_vfbcls[ivf] * N1_vfbcls[
298     ivf]));
299
300     beta11_fsm = beta11_ivf[ivf]
301     max_list_set_fsm = max_list_set_ivf[ivf]
302     min_list_set_fsm = min_list_set_ivf[ivf]
303     ### Generation of mapped features
304     for i in range(0, N2_vfbcls[ivf]):
305
306         beta1_fsm = beta11_fsm[i];
307
308         TT1_fsm = np.dot( np.float128(HH1_fsm), np.float128(beta1_fsm))
309
310         max_list_fsm = max_list_set_fsm[i];
311         min_list_fsm = min_list_set_fsm[i];
312
313         [TT1_fsm, max_list_fsm, min_list_fsm] = mapminmax( TT1_fsm.
314         transpose(), 0, 1, max_list_fsm, min_list_fsm);
315         TT1_fsm = TT1_fsm.transpose();
316
317         del beta1_fsm;
318         del max_list_fsm;
319         del min_list_fsm;
320
321         yy1_fsm[:, N1_vfbcls[ivf] * i : N1_vfbcls[ivf] * (i + 1)] = TT1_fsm
322
323     ;
324
325     del TT1_fsm;
326     del HH1_fsm;
327
328     yy1 = np.concatenate((yy1, yy1_fsm), axis=1);
329     yy1 = np.concatenate((test_x, yy1), axis=1);
330     ##### NEW
331     #####
332
333     ### Generation of enhancement nodes
334     HH2 = np.concatenate((yy1, 0.1 * np.ones((yy1.shape[0], 1))), axis=1);
335     TT2 = np.tanh(np.dot(HH2, wh) * l2);
336
337     TT3 = np.concatenate((yy1, TT2), axis=1);
338
339     del HH2;
340     del wh;
341     del TT2;
342
343     x = np.dot(TT3, beta);
344
345     time_end=time.time()
346     Testing_time = time_end - time_start
347
348     # Testing - end
349
350     print("Testing has been finished!");

```

```

347     print("The Total Testing Time is : ", Testing_time, " seconds");
348
349     ### Testing accuracy
350     y = result(x);
351     test_yy = result(test_y);
352
353     cnt = 0;
354     for i in range(0, len(y)):
355         if y[i] == test_yy[i]:
356             cnt = cnt + 1;
357
358     TestingAccuracy = cnt * 1.0 / test_yy.shape[0];
359
360     label = test_yy;
361     predicted = y;
362
363     TestingAccuracy = accuracy_score(label, predicted)
364     f_score = f1_score(label, predicted)
365
366     del TT3;
367
368     print("Testing Accuracy is : ", TestingAccuracy * 100, " %");
369     print("Testing F-Score is : ", f_score * 100, " %");
370
371     return TrainingAccuracy, TestingAccuracy, Training_time, Testing_time,
        f_score;

```

Listing A.2: VCFBLS

```

1  """
2      @authors Zhida Li
3      @email zhidal@sfu.ca
4      @date July 10, 2020
5
6      @copyright Copyright (c) July 10, 2020
7          All Rights Reserved
8
9  """
10
11  #####
12  ##### VCFBLS
13  #####
14
15  # Import the Python libraries
16  import time
17  import random
18  import numpy as np
19  import sys
20  import os
21
22  from scipy.stats import zscore
23  from scipy.linalg import orth
24  from numpy.linalg import inv
25  import math
26
27  #sys.path.append("../")
28  from bls.processing.result import result
29  from bls.processing.sparse_bls import sparse_bls
30

```

```

31 from sklearn import preprocessing
32 from bls.processing.mapminmax import mapminmax
33
34 from sklearn.metrics import f1_score
35 from sklearn.metrics import accuracy_score
36
37
38 # sys.path.append('../processing')
39 from bls.processing.feature_select_cnl import feature_select_cnl
40 from bls.processing.one_hot_m import one_hot_m
41
42 """
43     Function that creates the BLS model. It returns the training and test
44     accuracy, training and
45     testing time, and testing F-Score.
46     bls_train_fscore(train_x, train_y, test_x, test_y, s, C, N1, N2, N3)
47     'train_x' and 'test_x' are the entire training data and test data.
48     'train_y' and 'test_y' are the entire training labels and test
49     labels.
50     's' is the shrinkage parameter for enhancement nodes.
51     'C' is the parameter for sparse regularization.
52     'N1' is the number of mapped feature nodes.
53     'N2' are the groups of mapped features.
54     'N3' is the number of enhancement nodes.
55
56     Randomly generated weights for the mapped features and enhancement
57     nodes are
58     stored in the
59     matrices 'we' and 'wh,' respectively.
60 """
61
62 # Disable
63 def blockPrint():
64     sys.stdout = open(os.devnull, 'w')
65
66 # Restore
67 def enablePrint():
68     sys.stdout = sys.__stdout__
69
70
71 # rectified linear function
72 def relu(data):
73     return np.maximum(data,0)
74
75 # RBF function
76 def kerf( matrix ):
77     return np.exp( np.multiply(-1 * matrix, matrix) / 2.0 ) / np.sqrt(2 *
78     math.pi);
79
80 def vcfbls_train_fscore(train_x, train_y, test_x, test_y, s, C, N1, N2, N3,
81     N1_bls_fsm1, N2_bls_fsm1, N1_bls_fsm2, N2_bls_fsm2, add_nfeature1,
82     add_nfeature2):
83     # blockPrint()
84
85     time_start=time.time()
86     num_add_vf = 2
87     ### feature selection - begin
88     # add_nfeature1 = larger
89     features = feature_select_cnl(train_x, train_y, -1)
90     feature1 = features[0:add_nfeature1]

```

```

83     train_x1 = train_x[:,feature1]
84     test_x1 = test_x[:,feature1]
85
86     # add_nfeature2 = smaller
87     # features = feature_select_cnl(train_x, train_y, add_nfeature2)
88     feature2 = features[0:add_nfeature2]
89     train_x2 = train_x[:,feature2]
90     test_x2 = test_x[:,feature2]
91     ### feature selection - end
92     train_x_vf = [1 ,train_x1, train_x2]
93     test_x_vf = [1 ,test_x1, test_x2]
94
95     N1_vfbbs = [1, N1_bls_fsm1, N1_bls_fsm2]
96     N2_vfbbs = [1, N2_bls_fsm1, N2_bls_fsm2]
97
98     # Training - begin
99     # torch.cuda.synchronize()
100    # time_start=time.time()
101    beta11 = [];
102
103    train_x = zscore(train_x.transpose(), axis = 0, ddof = 1).transpose();
104    # print(train_x.shape)
105    H1 = np.concatenate((train_x, 0.1 * np.ones((train_x.shape[0], 1))),
106    axis=1);
107
108    y = np.zeros((train_x.shape[0], N2 * N1));
109
110    max_list_set = [];
111    min_list_set = [];
112
113    wha_list = [];
114    ### Generation of mapped features
115    for i in range(0, N2):
116
117        #we = 2.0 * np.random.rand(train_x.shape[1] + 1, N1) - 1.0;
118        we = 2.0 * np.random.rand(N1, train_x.shape[1] + 1).transpose() -
119        1.0;
120
121        #we = np.loadtxt("we.csv", delimiter = ",");
122        #np.savetxt("we.csv", we, delimiter=",");
123
124        A1 = np.dot(H1, we);
125        [A1, max_list, min_list] = mapminmax(A1);
126        del we;
127
128        beta1 = sparse_bls(A1, H1, 1e-3, 50).transpose();
129        beta11.append(beta1);
130        # T1 = np.dot(H1, beta1);
131
132        # Cascades of mapped features
133        wha = orth( 2.0 * np.random.rand(N1, N1) - 1.0 );
134        wha_list.append(wha);
135
136        if i == 0:
137            T1 = np.dot(H1, beta1);
138            T1_he = T1;
139        else:
140            T1_he = np.dot(T1_he, wha);

```

```

139         T1 = T1_he;
140
141         # print("Feature nodes in window ", i, ": Max Val of Output ", T1.
max(), " Min Val ", T1.min());
142
143         [T1, max_list, min_list] = mapminmax(T1.transpose(), 0, 1);
144         T1 = T1.transpose();
145
146         max_list_set.append(max_list);
147         min_list_set.append(min_list);
148
149         y[:, N1 * i: N1 * (i + 1)] = T1;
150
151     del H1;
152     del T1;
153
154     ##### NEW
155     #####
156     beta11_ivf = [1,1,1]
157     max_list_set_ivf = [1,1,1]
158     min_list_set_ivf = [1,1,1]
159     wha_list_fsm_ivf = [1,1,1]
160     for ivf in range(1,num_add_vf+1):
161         beta11_fsm = [];
162
163         train_x_fsm = zscore(train_x_vf[ivf].transpose(), axis = 0, ddof =
1).transpose());
164         H1_fsm = np.concatenate((train_x_fsm, 0.1 * np.ones((train_x_fsm.
shape[0], 1))), axis=1);
165
166         y_fsm = np.zeros((train_x_fsm.shape[0], N2_vfbcls[ivf] * N1_vfbcls[ivf
]));
167
168         max_list_set_fsm = [];
169         min_list_set_fsm = [];
170
171         wha_list_fsm = [];
172         ### Generation of mapped features
173         for i in range(0, N2_vfbcls[ivf]):
174
175             #we = 2.0 * np.random.rand(train_x.shape[1] + 1, N1) - 1.0;
176             we_fsm = 2.0 * np.random.rand(N1_vfbcls[ivf], train_x_fsm.shape
[1] + 1).transpose() - 1.0;
177
178             #we = np.loadtxt("we.csv", delimiter = ",");
179             #np.savetxt("we.csv", we, delimiter=",");
180
181             A1_fsm = np.dot(H1_fsm, we_fsm);
182             [A1_fsm, max_list_fsm, min_list_fsm] = mapminmax(A1_fsm);
183             del we_fsm;
184
185             beta1_fsm = sparse_bls(A1_fsm, H1_fsm, 1e-3, 50).transpose();
186             beta11_fsm.append(beta1_fsm);
187             # T1_fsm = np.dot(H1_fsm, beta1_fsm);
188
189             # Cascades of mapped features
190             wha_fsm = orth( 2.0 * np.random.rand(N1_vfbcls[ivf], N1_vfbcls[ivf
]) - 1.0 );

```

```

190         wha_list_fsm.append(wha_fsm);
191
192         if i == 0:
193             T1_fsm = np.dot(H1_fsm, beta1_fsm);
194             T1_he_fsm = T1_fsm;
195         else:
196             T1_he_fsm = np.dot(T1_he_fsm, wha_fsm);
197             T1_fsm = T1_he_fsm;
198         # print("Feature nodes in window ", i, ": Max Val of Output ",
199         T1_fsm.max(), " Min Val ", T1_fsm.min());
200
201         [T1_fsm, max_list_fsm, min_list_fsm] = mapminmax(T1_fsm.
202         transpose(), 0, 1);
203         T1_fsm = T1_fsm.transpose();
204
205         max_list_set_fsm.append(max_list_fsm);
206         min_list_set_fsm.append(min_list_fsm);
207
208         y_fsm[:, N1_vfbcls[ivf] * i: N1_vfbcls[ivf] * (i + 1)] = T1_fsm;
209
210         del H1_fsm;
211         del T1_fsm;
212         beta1_ivf[ivf] = beta1_fsm
213         max_list_set_ivf[ivf] = max_list_set_fsm
214         min_list_set_ivf[ivf] = min_list_set_fsm
215         wha_list_fsm_ivf[ivf] = wha_list_fsm
216         y = np.concatenate((y, y_fsm), axis=1);
217     y = np.concatenate((train_x,y), axis=1);
218     ##### NEW #####
219     train_y = one_hot_m(train_y, 2);
220     test_y = one_hot_m(test_y, 2);
221
222     ### Generation of enhancement nodes
223     H2 = np.concatenate((y, 0.1 * np.ones((y.shape[0], 1)))), axis=1);
224
225     if (N1 * N2 + N2_vfbcls[1] * N1_vfbcls[1] + N2_vfbcls[2] * N1_vfbcls[2] +
226     train_x.shape[1]) >= N3:
227         #wh = orth(2 * np.random.rand(N2 * N1 + 1, N3) - 1);
228         wh = orth(2 * np.random.rand(N3, (N1 * N2 + N2_vfbcls[1] * N1_vfbcls
229         [1] + N2_vfbcls[2] * N1_vfbcls[2] + train_x.shape[1] + 1).transpose() - 1);
230
231     else:
232         #wh = orth(2 * np.random.rand(N2 * N1 + 1, N3).transpose() - 1).
233         transpose();
234         wh = orth(2 * np.random.rand(N3, (N1 * N2 + N2_vfbcls[1] * N1_vfbcls
235         [1] + N2_vfbcls[2] * N1_vfbcls[2] + train_x.shape[1] + 1) - 1).transpose());
236
237     #wh = np.loadtxt("wh.csv", delimiter = ",");
238     #np.savetxt("wh.csv", wh, delimiter=",");
239
240     T2 = np.dot(H2, wh);
241     l2 = T2.max();
242     l2 = s * 1.0 / l2;
243
244     # print("Enhancement nodes: Max Val of Output ", l2, " Min Val ", T2.min
245     ());

```

```

240     T2 = np.tanh(T2 * 12);
241     T3 = np.concatenate((y, T2), axis=1);
242
243     del H2;
244     del T2;
245     # print((np.dot(T3.transpose(), T3) + np.identity(T3.transpose().shape
246     [0]) * C).shape)
247     # Moore-Penrose pseudoinverse (function pinv)
248     beta = np.dot(inv(np.dot(T3.transpose(), T3) + np.identity(T3.transpose
249     ().shape[0]) * C),
250                   np.dot(T3.transpose(), train_y));
251
252     # T3_i1 = torch.mm(torch.from_numpy(T3.T), torch.from_numpy(T3))
253     # T3_i2 = torch.from_numpy(np.identity(T3.T.shape[0]) * C)
254     # T3_i = T3_i1 + T3_i2
255     # print(type(T3_i))
256     # T3_i = T3_i.cuda()
257     # T3_inv = torch.inverse(T3_i)
258     # beta = torch.mm(T3_inv, torch.mm((torch.from_numpy(T3.T)).cuda(), (
259     torch.from_numpy(train_y)).cuda())));
260     # print(type(T3_i))
261     # beta = beta.detach().cpu().numpy()
262
263     xx = np.dot(T3, beta);
264
265     del T3;
266
267     # torch.cuda.synchronize()
268     time_end=time.time()
269     Training_time = time_end - time_start
270
271     # Training - end
272
273     print("Training has been finished!");
274     print("The Total Training Time is : ", Training_time, " seconds");
275
276     ### Training Accuracy
277     yy = result(xx);
278     train_yy = result(train_y);
279
280     cnt = 0;
281     for i in range(0, len(yy)):
282         if yy[i] == train_yy[i]:
283             cnt = cnt + 1;
284
285     TrainingAccuracy = cnt * 1.0 / train_yy.shape[0];
286
287     print("Training Accuracy is : ", TrainingAccuracy * 100, " %");
288
289     ### Testing Process
290     # Testing - begin
291     time_start=time.time()
292     test_x = zscore( np.float128(test_x).transpose() ,axis = 0, ddof = 1).
293     transpose();
294
295     HH1 = np.concatenate((test_x, 0.1 * np.ones((test_x.shape[0], 1))), axis
296     =1);

```

```

293     yy1 = np.zeros((test_x.shape[0], N2 * N1));
294
295     ### Generation of mapped features
296     for i in range(0, N2):
297
298         beta1 = beta11[i];
299
300         # TT1 = np.dot( np.float128(HH1), np.float128(beta1)) ;
301         # Cascades of mapped features
302         wha = wha_list[i];
303
304         if i == 0:
305             TT1 = np.dot(HH1, beta1);
306             TT1_he = TT1;
307         else:
308             TT1_he = np.dot(TT1_he, wha);
309             TT1 = TT1_he;
310
311         max_list = max_list_set[i];
312         min_list = min_list_set[i];
313
314         [TT1, max_list, min_list] = mapminmax( TT1.transpose(), 0, 1,
max_list, min_list);
315         TT1 = TT1.transpose();
316
317         del beta1;
318         del max_list;
319         del min_list;
320
321         yy1[:, N1 * i: N1 * (i + 1)] = TT1;
322
323     del TT1;
324     del HH1;
325
326     ##### NEW
327     #####
328     for ivf in range(1,num_add_vf+1):
329         test_x_fsm = zscore( np.float128(test_x_vf[ivf]).transpose() ,axis =
0, ddof = 1).transpose();
329
330         HH1_fsm = np.concatenate((test_x_fsm, 0.1 * np.ones((test_x_fsm.
shape[0], 1))), axis=1);
331         yy1_fsm = np.zeros((test_x_fsm.shape[0], N2_vfbls[ivf] * N1_vfbls[
ivf]));
332
333         beta11_fsm = beta11_ivf[ivf]
334         max_list_set_fsm = max_list_set_ivf[ivf]
335         min_list_set_fsm = min_list_set_ivf[ivf]
336         wha_list_fsm = wha_list_fsm_ivf[ivf]
337         ### Generation of mapped features
338         for i in range(0, N2_vfbls[ivf]):
339
340             beta1_fsm = beta11_fsm[i];
341
342             # TT1_fsm = np.dot( np.float128(HH1_fsm), np.float128(beta1_fsm)
) ;
343
344             # Cascades of mapped features
345             wha_fsm = wha_list_fsm[i];

```

```

345
346         if i == 0:
347             TT1_fsm = np.dot(HH1_fsm, beta1_fsm);
348             TT1_he_fsm = TT1_fsm;
349         else:
350             TT1_he_fsm = np.dot(TT1_he_fsm, wha_fsm);
351             TT1_fsm = TT1_he_fsm;
352
353         max_list_fsm = max_list_set_fsm[i];
354         min_list_fsm = min_list_set_fsm[i];
355
356         [TT1_fsm, max_list_fsm, min_list_fsm] = mapminmax( TT1_fsm.
transpose(), 0, 1, max_list_fsm, min_list_fsm);
357         TT1_fsm = TT1_fsm.transpose();
358
359         del beta1_fsm;
360         del max_list_fsm;
361         del min_list_fsm;
362
363         yy1_fsm[:, N1_vfbls[ivf] * i: N1_vfbls[ivf] * (i + 1)] = TT1_fsm
;
364
365         del TT1_fsm;
366         del HH1_fsm;
367
368         yy1 = np.concatenate((yy1, yy1_fsm), axis=1);
369         yy1 = np.concatenate((test_x, yy1), axis=1);
370         ##### NEW
371         #####
372         ### Generation of enhancement nodes
373         HH2 = np.concatenate((yy1, 0.1 * np.ones((yy1.shape[0], 1))), axis=1);
374         TT2 = np.tanh(np.dot(HH2, wh) * l2);
375
376
377         TT3 = np.concatenate((yy1, TT2), axis=1);
378
379         del HH2;
380         del wh;
381         del TT2;
382
383         x = np.dot(TT3, beta);
384
385         time_end=time.time()
386         Testing_time = time_end - time_start
387
388         # Testing - end
389
390         print("Testing has been finished!");
391         print("The Total Testing Time is : ", Testing_time, " seconds");
392
393         ### Testing accuracy
394         y = result(x);
395         test_yy = result(test_y);
396
397         cnt = 0;
398         for i in range(0, len(y)):
399             if y[i] == test_yy[i]:

```

```

400         cnt = cnt + 1;
401
402     TestingAccuracy = cnt * 1.0 / test_yy.shape[0];
403
404     label = test_yy;
405     predicted = y;
406
407     TestingAccuracy = accuracy_score(label, predicted)
408     f_score = f1_score(label, predicted)
409
410     del TT3;
411
412     print("Testing Accuracy is : ", TestingAccuracy * 100, " %");
413     print("Testing F-Score is : ", f_score * 100, " %");
414
415     return TrainingAccuracy, TestingAccuracy, Training_time, Testing_time,
f_score;

```

## Appendix B

# BGPGuard: Main Modules

Listed are the main modules of the *BGPGuard*.

Listing B.1: Module of *Time Tracker*

```
1  """
2      @author Zhida Li
3      @email zhidal@sfu.ca
4      @date Apr. 28, 2020
5      @version: 1.1.0
6      @description:
7          Track the timestamp used for the filename from RIPE or Route
8          Views.
9
10     @copyright Copyright (c) Apr. 28, 2020
11         All Rights Reserved
12
13     This Python code (versions 3.6 and newer)
14 """
15 # =====
16 # time_tracker_single(), time_tracker_multi()
17 # =====
18 # Last modified: Feb. 19, 2022
19
20 #
21 import time
22 import datetime
23
24
25 # time.gmtime()
26 # time.struct_time(tm_year=2020, tm_mon=4, tm_mday=27, tm_hour=8, tm_min=10,
27 # tm_sec=10, tm_wday=0, tm_yday=118, tm_isdst=0)
28
29 # Enter RIPE or RouteViews
30 def time_tracker_single(site):
31     gmtime_now = time.gmtime()
32     year = gmtime_now.tm_year
33     year = str(year)
34
35     month = gmtime_now.tm_mon
```

```

36     month = str(month)
37     if len(month) == 2:
38         pass
39     else:
40         month = '0' + month
41
42     day = gmtime_now.tm_mday
43     day = str(day)
44     if len(day) == 2:
45         pass
46     else:
47         day = '0' + day
48
49     hour = gmtime_now.tm_hour
50     hour = str(hour)
51     if len(hour) == 2:
52         pass
53     else:
54         hour = '0' + hour
55
56     if site == 'RIPE':
57         minute = gmtime_now.tm_min
58         if minute % 5 != 0:
59             minute = minute - minute % 5
60
61         minute = str(minute)
62         if len(minute) == 2:
63             pass
64         else:
65             minute = '0' + minute
66
67         if minute == '00':
68             minute = '55'
69             hour = int(hour) - 1
70             hour = str(hour)
71             if len(hour) == 2:
72                 pass
73             else:
74                 hour = '0' + hour
75         else:
76             minute = int(minute)
77             minute = minute - 5
78             minute = str(minute)
79             if len(minute) == 2:
80                 pass
81             else:
82                 minute = '0' + minute
83     elif site == 'RouteViews':
84         minute = gmtime_now.tm_min
85         if minute % 15 != 0:
86             minute = minute - minute % 15
87
88         minute = str(minute)
89         if len(minute) == 2:
90             pass
91         else:
92             minute = '0' + minute
93

```

```

94         if minute == '00':
95             minute = '45'
96             hour = int(hour) - 1
97             hour = str(hour)
98             if len(hour) == 2:
99                 pass
100            else:
101                hour = '0' + hour
102        else:
103            minute = int(minute)
104            minute = minute - 15
105            minute = str(minute)
106            if len(minute) == 2:
107                pass
108            else:
109                minute = '0' + minute
110    else:
111        print('Site name is incorrect.')
112        exit()
113    return year, month, day, hour, minute
114
115
116    # Return the all the dates in a list.
117    # format YYYYMMDD
118    def time_tracker_multi(start_date, end_date):
119        # start_date='20030121'
120        # end_date='20030125'
121
122        # time.strptime(time_string[, format])
123        datestart = datetime.datetime.strptime(start_date, '%Y%m%d')
124        dateend = datetime.datetime.strptime(end_date, '%Y%m%d')
125
126        date_list = list()
127        while datestart <= dateend:
128            date_list.append(datestart.strftime('%Y%m%d'))
129            datestart += datetime.timedelta(days=1)
130
131        return date_list

```

Listing B.2: Module of *Data Download*

```

1  """
2      @author Zhida Li
3      @email zhidal@sfu.ca
4      @date May. 04, 2020
5      @version: 1.1.0
6      @description:
7          Download update messages from RIPE or RouteViews
8
9      @copyright Copyright (c) May. 04, 2020
10         All Rights Reserved
11
12         This Python code (versions 3.6 and newer)
13  """
14
15  # =====
16  # updateMessageName(), data_downloader_single(), data_downloader_multi()
17  # =====
18  # Last modified: Feb. 19, 2022

```

```

19
20 # Import the built-in libraries
21 import os
22 import sys
23
24 # Import customized libraries
25 sys.path.append('./src')
26 from progress_bar import progress_bar
27 from subprocess_cmd import subprocess_cmd
28 from time_tracker import time_tracker_multi
29
30
31 # Name of the update_message_file generation
32 def updateMessageName(year, month, day, hour, minute):
33     # updates.YYYYMMDD.HHMM.gz, minute: 5 minutes interval
34
35     data_date = "%s.%s" % (year, month) # used for data_link in Function
36     data_downloader().
37
38     # Code below also works
39     # update_message_file = 'updates.' + str(year) + str(month)+ str(day)+
40     # '.'+ str(hour)+ str(minute)+'.'+'gz'
41     update_message_file = "updates.%s%s.%s%s" % (year, month, day, hour,
42     minute)
43     return update_message_file, data_date # these two outputs are string
44
45
46 # Download specific file from RIPE or RouteViews
47 # rcc04: Geneva
48 def data_downloader_single(update_message_file, data_date, site,
49 collector_ripe='rcc04',
50 collector_routeviews='route-views2'):
51     data_file = update_message_file
52
53     if site == 'RIPE':
54         data_link = "https://data.ris.ripe.net/%s/%s/%s.gz" % (
55 collector_ripe, data_date, data_file)
56
57     # Update message files will be downloaded in "data_ripe" folder: --
58     directory-prefix
59     subprocess_cmd("chmod -R 777 ./src/;\
60 cd src/; wget -np --accept=gz %s \
61 --directory-prefix=data_ripe ; \
62 cd data_ripe ; \
63 chmod +x name-change-script.sh ;\
64 sh ./name-change-script.sh ; \
65 echo '=> >>>>>>>>> > > > > Extension changed (gz
66 to Z)' " % (data_link))
67
68     subprocess_cmd("cd src/; cd data_ripe ; \
69 chmod +x zebra-script.sh ;\
70 sh ./zebra-script.sh")
71
72     progress_bar(time_sleep=0.02, status_p='Converting')
73
74     subprocess_cmd("echo ' '; \
75 echo '=> >>>>>>>>> > > > > DUMP generated (MRT to
76 ASCII)' ")

```

```

69
70     # Move .Z file, then remove it
71     subprocess_cmd("cd src/; mv ./data_ripe/%s.Z ./data_ripe/temp/; \
72                     rm ./data_ripe/DUMPER; \
73                     rm ./data_ripe/temp/%s.Z" % (data_file, data_file))
74
75     # collector_routeviews is not finished (only use 'route-views2' now)
76     elif site == 'RouteViews':
77         data_link2 = "https://archive.routeviews.org/bgpdata/%s/UPDATES/%s.
78         bz2" % (data_date, data_file)
79
80         # Update message files will be downloaded in "data_routeviews"
81         folder: --directory-prefix
82         subprocess_cmd("wget -np --accept=bz2 %s \
83                         --directory-prefix=data_routeviews ; \
84                         cd data_routeviews ;" % (data_link2))
85
86         subprocess_cmd("cd data_routeviews ; \
87                         chmod +x zebra-script.sh ; \
88                         sh ./zebra-script.sh ; \
89                         echo '> >>>>>>>>> > > > > Decompressed bz2' ")
90
91         progress_bar(time_sleep=0.02, status_p='Converting')
92
93         subprocess_cmd("echo ' '; \
94                         echo '> >>>>>>>>> > > > > DUMP generated (MRT to
95         ASCII)' ")
96
97         # Move Decompressed file, then remove it
98         subprocess_cmd("mv ./data_routeviews/%s ./data_routeviews/temp/ ; \
99                         rm ./data_routeviews/DUMPER; \
100                        rm ./data_routeviews/temp/%s" % (data_file,
101                        data_file))
102     else:
103         print("Wrong name")
104         exit()
105
106 # Multiple files (days)
107 def data_downloader_multi(start_date, end_date, site, collector_ripe='rrc04',
108     collector_routeviews='route-views2'):
109     # updates.YYYYMMDD.HHMM.gz
110     # data_date = "%s.%s" % (year, month)
111     # update_message_file = "updates.%s%s.%s%s" % (year, month, day, hour,
112     minute)
113     date_list = time_tracker_multi(start_date, end_date)
114     print(date_list)
115
116     # RIPE
117     if site == 'RIPE':
118         for date_i in date_list:
119             # Create folders for each day
120             date_i_folder = './data_ripe/%s' % date_i
121             if not os.path.exists(date_i_folder):
122                 os.makedirs(date_i_folder)
123                 print("\n RIPE => >>>>>>>>> Folder %s has been created." %
124                 (date_i))

```

```

120         # Move dump parser to each folder
121         subprocess_cmd("cd data_ripe/ ; \
122             cp name-change-script.sh zebra-dump-parser-
modified.pl zebra-script.sh ./%s ; \
123                 cd %s/ " % (date_i, date_i))
124
125         # Download
126         year = date_i[0:4]
127         month = date_i[4:6]
128         data_date = "%s.%s" % (year, month)
129
130         data_link = "https://data.ris.ripe.net/%s/%s/" % (collector_ripe
, data_date)
131         data_file = 'updates.%s*.gz' % date_i
132         data_file_rm = 'updates.%s*.Z' % date_i
133
134         subprocess_cmd("cd data_ripe/%s/ ; \
135             wget -e robots=off -r -np -nd -A '%s' %s ;" % (
date_i, data_file, data_link))
136
137         # Generate DUMP using Zebra
138         subprocess_cmd("cd data_ripe/%s/ ; \
139             chmod +x name-change-script.sh ; sh ./name-
change-script.sh ; \
140                 echo '=> >>>>>>>>> > > > > Extension changed
(gz to Z)' ;\
141                 chmod +x zebra-script.sh ; sh ./zebra-script.sh
; \
142                 mv DUMP DUMP_%s" % (date_i, date_i))
143
144         progress_bar(time_sleep=0.02, status_p='Converting')
145
146         subprocess_cmd("echo ' ' ; \
147             echo '=> >>>>>>>>> > > > > DUMP generated (
MRT to ASCII)' " )
148
149         # Remove .Z file
150         subprocess_cmd("rm ./data_ripe/%s/%s" % (date_i, data_file_rm))
151
152         # RouteViews
153         # collector_routeviews is not finished (only use 'route-views2' now)
154         elif site == 'RouteViews':
155             for date_i in date_list:
156                 # Create folders for each day
157                 date_i_folder = './data_routeviews/%s' % date_i
158                 if not os.path.exists(date_i_folder):
159                     os.makedirs(date_i_folder)
160                     print("\n RouteViews => >>>>>>>>> Folder %s has been
created." % (date_i))
161
162                 # Move dump parser to each folder
163                 subprocess_cmd("cd data_routeviews/ ; \
164                     cp zebra-dump-parser-modified.pl zebra-script.sh
./%s ; \
165                         cd %s/ " % (date_i, date_i))
166
167                 # Download
168                 year = date_i[0:4]

```

```

169         month = date_i[4:6]
170         data_date = "%s.%s" % (year, month)
171
172         data_link = "https://archive.routeviews.org/bgpdata/%s/UPDATES/"
173         % (data_date)
174         data_file = 'updates.%s*.bz2' % date_i
175         data_file_rm = 'updates.%s*' % date_i
176
177         subprocess_cmd("cd data_routeviews/%s/ ; \
178             wget -e robots=off -r -np -nd -A '%s' %s ;" % (
179                 date_i, data_file, data_link))
180
181         # Generate DUMP using Zebra
182         subprocess_cmd("cd data_routeviews/%s/ ; \
183             chmod +x zebra-script.sh ; sh ./zebra-script.sh
184             ; \
185             echo '=> >>>>>>>>> > > > > Decompressed bz2'
186             ; \
187             mv DUMP DUMP_%s" % (date_i, date_i))
188
189         progress_bar(time_sleep=0.02, status_p='Converting')
190
191         subprocess_cmd("echo ' ' ; \
192             echo '=> >>>>>>>>> > > > > DUMP generated (
193             MRT to ASCII)' ")
194
195         # Remove .Z file
196         subprocess_cmd("rm ./data_routeviews/%s/%s" % (date_i,
197             data_file_rm))
198     else:
199         print("Wrong name")
200         exit()

```

Listing B.3: Module of *Feature Extraction*

```

1  """
2      @author Zhida Li
3      @email zhidal@sfu.ca
4      @date May. 01, 2020
5      @version: 1.1.0
6      @description:
7          Generate the matrix using BGP C# tool.
8
9      @copyright Copyright (c) May. 01, 2020
10         All Rights Reserved
11
12         This Python code (versions 3.6 and newer)
13  """
14
15  # =====
16  # feature_extractor_single(), feature_extractor_multi
17  # =====
18  # Last modified: Feb. 19, 2022
19
20  # Import the built-in libraries
21  # import time
22
23  # Import customized libraries
24  from progress_bar import progress_bar

```

```

25 from subprocess_cmd import subprocess_cmd
26 from time_tracker import time_tracker_multi
27
28
29 # Matrix generation
30 # Input: DUMP
31 # Output: DUMP_out.txt
32 def feature_extractor_single(site, file_name='DUMP'):
33     # Update message files will be downloaded in "data_ripe" or "
34     data_routeviews" folder
35     # line 19: &> ConsoleApplication1.out ; \
36     # Move DUMP
37     if site == 'RIPE':
38         subprocess_cmd("cd src/; \
39             mv ./data_ripe/%s ./CSharp_Tool_BGP/
40             ConsoleApplication1/bin/Release/ ; \
41             cd CSharp_Tool_BGP/ConsoleApplication1/bin/Release/
42             ; \
43             mono ConsoleApplication1.exe >/dev/null ; \
44             rm %s ; \
45             cd ../cd ../cd ../cd ../" % (file_name,
46             file_name))
47     elif site == 'RouteViews':
48         subprocess_cmd("mv ./data_routeviews/%s ./CSharp_Tool_BGP/
49             ConsoleApplication1/bin/Release/ ; \
50             cd CSharp_Tool_BGP/ConsoleApplication1/bin/Release/
51             ; \
52             mono ConsoleApplication1.exe >/dev/null ; \
53             rm %s ; \
54             cd ../cd ../cd ../cd ../" % (file_name,
55             file_name))
56     else:
57         print("Wrong name")
58         exit()
59
60 progress_bar(time_sleep=0.03, status_p='Generating')
61
62 subprocess_cmd("echo ' '; \
63     echo '=> >>>>>>>> > > > > Matrix generated (txt)' ;
64     \
65     echo 'Current Path: '; pwd ")
66
67 subprocess_cmd(" echo '+=====+' ; \
68     echo '| Data download      == done |'; \
69     echo '| MRT to ASCII        == done |'; \
70     echo '| Matrix generation == done |'; \
71     echo '| Next step:                |'; \
72     echo '|          Classification      |'; \
73     echo '+=====+' ; \
74     echo ' ' ")
75
76 output_file = "%s_out.txt" % file_name
77
78 # Move output_file
79 subprocess_cmd("cd src/; mv ./CSharp_Tool_BGP/ConsoleApplication1/bin/
80 Release/%s \
81             ./data_test/" % output_file)
82
83

```





```

19 def label_generator(start_date_anomaly, end_date_anomaly, start_time_anomaly
, end_time_anomaly, site, output_file_list):
20     # Maybe need a step to verify if end_date_anomaly > end_date_anomaly
21     # def label_generator():
22     print("-----Label Generation-Begin-----")
23     ## Verify if anomalous start and end dates are in the output_file_list.
24     if 'DUMP_%s_out.txt' % start_date_anomaly and 'DUMP_%s_out.txt' %
end_date_anomaly in output_file_list:
25         start_date_anomaly_list_index = output_file_list.index('DUMP_%s_out.
txt' % start_date_anomaly)
26         end_date_anomaly_list_index = output_file_list.index('DUMP_%s_out.
txt' % end_date_anomaly)
27         print("Input date verification: Pass")
28         pass
29     else:
30         print("Please re-enter anomalous start date or end date.")
31         exit()
32
33     ## Check the total number of data points
34     # Load all txt files
35     d = {} # use dictionary
36     for i, DUMPout in enumerate(output_file_list):
37         if i == 0:
38             d["DUMPout_%d" % i] = np.loadtxt("./data_split/%s" %
output_file_list[i])
39             matrix = d["DUMPout_%d" % i]
40         else:
41             d["DUMPout_%d" % i] = np.loadtxt("./data_split/%s" %
output_file_list[i])
42             matrix1 = d["DUMPout_%d" % i]
43             matrix = np.append(matrix, matrix1, axis=0)
44     # print('d keys: ', d.keys()) # class: dict_keys
45
46     # Select 37 features / 41 features
47     start_featrue = 5 - 1
48     matrix = matrix[:, start_featrue:]
49
50     # Check if individual file check needed
51     if matrix.shape[0] % 1440 == 0:
52         # print("matrix shape:", matrix.shape)
53         pass
54     else:
55         print("Check if individual out.txt file")
56         exit()
57
58     ## Initialize labels with zeros
59     labels = np.zeros((matrix.shape[0], 1))
60
61     # Create labels for anomaly == ones
62     # print(start_date_anomaly_list_index)
63     # print(end_date_anomaly_list_index)
64     # Check if the anomalous event is in one day or otherwise
65     start_point_anomaly = int(start_time_anomaly[0:2]) * 60 + int(
start_time_anomaly[2:])
66     end_point_anomaly = int(end_time_anomaly[0:2]) * 60 + int(
end_time_anomaly[2:])
67

```

```

68     labels_anomaly_index_start = start_date_anomaly_list_index * 1440 +
start_point_anomaly
69     labels_anomaly_index_end = end_date_anomaly_list_index * 1440 +
end_point_anomaly # This index should be included for anomaly
70
71     labels[labels_anomaly_index_start:(labels_anomaly_index_end + 1),
72     0] = 1 # +1 because labels_anomaly_index_end should be included
73
74     # print(labels_anomaly_index_start)
75     # print(labels_anomaly_index_end)
76
77     # Get the number of anomaly
78     # num_anomaly = len(np.where(labels==1)[0]) # May be returned
79     # print("Number of anomaly:", num_anomaly)
80     np.savetxt('./STAT/labels_%s.csv' % site, labels, delimiter=',', fmt='%d
')
81
82     print("-----Label Generation-end-----\n"
)
83
84     return labels

```

Listing B.5: Module of *Data Partition*

```

1 # Data partition for DUMP_out.txt
2 # csv files will be generated and saved to data_split folder.
3 # May. 06, 2020
4
5 import numpy as np
6 from label_generation import label_generator
7
8
9 # output_file_out_txt from feature_extractor_multi() function
10 # DUMP_YYYYMMDD_out.txt
11
12 # output_file_list = ["DUMP_20030123_out.txt", "DUMP_20030124_out.txt", "
DUMP_20030125_out.txt",
13 # "DUMP_20030126_out.txt", "DUMP_20030127_out.txt"]
14
15 # # Partition percentage
16 # cut_pct = '64'
17 # site = 'RIPE'
18 # rnn_seq = 20
19
20 def data_partition(cut_pct, site, output_file_list, labels, rnn_seq=1):
21     print("-----Data Partition-Begin-----")
22     # Load txt files for each day
23     d = {} # use dictionary
24
25     # matrix = np.array([])
26     for i, DUMPout in enumerate(output_file_list):
27         if i == 0:
28             d["DUMPout_%d" % i] = np.loadtxt("./data_split/%s" %
output_file_list[i])
29             matrix = d["DUMPout_%d" % i]
30         else:
31             d["DUMPout_%d" % i] = np.loadtxt("./data_split/%s" %
output_file_list[i])
32             matrix1 = d["DUMPout_%d" % i]

```

```

33         matrix = np.append(matrix, matrix1, axis=0)
34     # print('d keys: ', d.keys()) # class: dict_keys
35
36     # Select 37 features / 41 features
37     start_featrue = 5 - 1
38     matrix = matrix[:, start_featrue:]
39     print('matrix shape:', matrix.shape) # if shape, type = <class 'tuple'
    '>', if shape[1], type = <class 'int'>
40
41     # Statistics of the dataset
42     inds1 = np.where(labels == 1); # index of anomalies
43     dataset_Stat = [matrix.shape[0], len(inds1[0]), 0, 0, 0];
44     print("The processing dataset has %d data points, with %d anomaly inside
    ." % (dataset_Stat[0], dataset_Stat[1]))
45
46     # print("-----")
47     # Find the cutting point
48
49     dataset_Stat[2] = round(dataset_Stat[1] * float(int(cut_pct[0]) / 10));
    # np.round return float...percentage
50     # print("Dataset %s/%s cut at index %d of the anomaly set."% (cut_pct
    [0], cut_pct[1], dataset_Stat[2]-1)) # dataset_Stat[2] cut point in
    anomaly
51
52     # print("-----")
53
54     # Merge matrix and labels
55     dataset = np.concatenate((matrix, labels), axis=1)
56
57     # Cut
58     anomaly_index = inds1[0]
59     # print(anomaly_index)
60     cut_index = anomaly_index[dataset_Stat[2] - 1] # cut after cut_index,
    start from index 0, thats why -1
61     # print("Cutting index will be included: ",cut_index)
62
63     if (cut_index + 1) % rnn_seq == 0:
64         cut_index_fix = cut_index + 1
65     else:
66         cut_index_fix = (rnn_seq - (cut_index + 1) % rnn_seq) + cut_index +
    1;
67     print('Cutting data point:', cut_index_fix)
68
69     # train test
70     # train = dataset[0:(cut_index+1), :] # +1 because the point cut_index
    should be counted
71     # test = dataset[(cut_index+1):, :]
72     train = dataset[0:cut_index_fix, :] # +1 because the point cut_index
    should be counted
73     test = dataset[cut_index_fix:, :]
74
75     np.savetxt('./data_split/train_%s_%s.csv' % (cut_pct, site), train,
    delimiter=',') # ,fmt='%.4f')
76     np.savetxt('./data_split/test_%s_%s.csv' % (cut_pct, site), test,
    delimiter=',') # ,fmt='%.4f')
77
78     # Calculate number of regular and anomalous data points for train and
    test

```

```

79     num_regular_train = len(np.where(train[:, -1] == 0)[0])
80     num_regular_test = len(np.where(test[:, -1] == 0)[0])
81
82     num_anomaly_train = len(np.where(train[:, -1] == 1)[0])
83     num_anomaly_test = len(np.where(test[:, -1] == 1)[0])
84
85     toal_number_train = num_anomaly_train + num_regular_train
86     toal_number_test = num_anomaly_test + num_regular_test
87
88     train_test_stat = ["No. of data points in train:", toal_number_train, '\n',
89                        "No. of data points in test:", toal_number_test, '\n',
90                        ,
91                        "No. of regular data points in train:",
92                        num_regular_train, '\n',
93                        "No. of regular data points in test:",
94                        num_regular_test, '\n',
95                        "No. of anomaly data points in train:",
96                        num_anomaly_train, '\n',
97                        "No. of anomaly data points in test:",
98                        num_anomaly_test, '\n']
99
100     train_test_stat = np.array(train_test_stat)
101     np.savetxt('./STAT/train_test_stat.txt', train_test_stat, delimiter=',',
102               fmt='%s')
103
104     print("-----Data Partition-End-----Files saved-----\n")
105
106     '''
107     Has problem for dimension match:
108     result_array = np.array([])
109     for line in data_array:
110         result = do_stuff(line)
111         result_array = np.append(result_array, result)
112     '''

```

Listing B.6: Module of *Data Processing*

```

1  # Normalize data and feature selection
2  # May 06, 2020
3  import sys
4  import numpy as np
5  from scipy.stats import zscore
6  from sklearn.ensemble import ExtraTreesClassifier
7
8  sys.path.append('../VFBS_v110')
9  from VFBS_v110.bls.processing.replaceNan import replaceNan
10
11
12  # cut_pct = '64'
13  # site = 'RIPE'
14
15  # Normalization
16  def normTrainTest(cut_pct, site):
17      train = np.loadtxt('./data_split/train_%s_%s.csv' % (cut_pct, site),
18                        delimiter=',')
19      test = np.loadtxt('./data_split/test_%s_%s.csv' % (cut_pct, site),
20                       delimiter=',')

```

```

19
20     train_x = train[:, 0:-1]
21     train_x = zscore(train_x, axis=0, ddof=1); # For each feature, mean = 0
22         and std = 1
23     replaceNan(train_x);
24     test_x = test[:, 0:-1]
25     test_x = zscore(test_x, axis=0, ddof=1); # For each feature, mean = 0
26         and std = 1
27     replaceNan(test_x);
28
29     # train_y = train[:,train.shape[1] - 1 : train.shape[1]];
30     # test_y = test[:,test.shape[1] - 1 : test.shape[1]];
31     train_y = train[:, -1]
32     test_y = test[:, -1]
33     train_y, test_y = train_y.reshape(-1, 1), test_y.reshape(-1, 1)
34
35     # Merge matrix and labels for train and test
36     train_n = np.concatenate((train_x, train_y), axis=1)
37     test_n = np.concatenate((test_x, test_y), axis=1)
38
39     np.savetxt('./data_split/train_%s_%s_n.csv' % (cut_pct, site), train_n,
40         delimiter=',') # ,fmt='%.4f')
41     np.savetxt('./data_split/test_%s_%s_n.csv' % (cut_pct, site), test_n,
42         delimiter=',') # ,fmt='%.4f')
43
44 # Feature selections:
45 def featSel(featALGO, data, label, topFeatures=10):
46     np.random.seed(1);
47     model = ExtraTreesClassifier(n_estimators=100, random_state=1)
48     label = np.ravel(label)
49     model.fit(data, label)
50
51     importances = model.feature_importances_
52
53     f_indices = np.argsort(importances)[::-1]
54     # print(f_indices)
55
56     selected_features = f_indices[0:topFeatures]
57     # print(selected_features)
58     return selected_features

```

## Appendix C

# BGPGuard: Developed Python Functions

Listed are the developed Python functions for *BGPGuard*.<sup>1</sup>

---

Module:

`time_tracker.py`

Functions:

Time tracker function for real-time mode:

`time_tracker_single(site)`

This function was developed to generate year, month, day, hour, and minute in order to match the date and time of the latest update message collected by RIPE and Route Views.

Input:

- *site* (string): “RIPE” or “RouteViews”.

Output:

- Year, month, day, hour, and minute.

---

Module:

`dataDownload.py`

Functions:

Download function for real-time mode:

`data_downloader_single(update_message_file, data_date,  
 site, collector_ripe='rrc04',  
 collector_routeviews='route-views2')`

Download function for off-line mode:

`data_downloader_multi(start_date, end_date,`

---

<sup>1</sup>The description of each variable appears once.

```
site, collector_ripe='rrc04',
collector_routeviews='route-views2')
```

The functions are used to download the update messages and call the *zebra-dump-parser* to transform them from MRT to ASCII format.

Input of **data\_\_downloader\_\_single()**:

- *update\_message\_file* (string): filename of the BGP update message.  
Format: “updates.yyyymmdd.hhmm”.
- *data\_date* (string): URL path name for specifying year and month for a selected collector. Format: “yyyy.mm”.
- *site* (string);
- *collector\_ripe* (string, default=“rrc04”): collector name of RIPE.
- *collector\_routeviews* (string, default=“route-views2”): collector name of Route Views.

Input of **data\_\_downloader\_\_multi()**:

- *start\_date* (string): start date for the entire experiment.
- *end\_date* (string): end date for the entire experiment.
- *site*; *collector\_ripe*; *collector\_routeviews*.

Output of **data\_\_downloader\_\_single()** and **data\_\_downloader\_\_multi()**:

- The parsed files are moved into the directory *./data\_ripe*.

---

```
Module:
FeatureExtraction.py

Functions:
Feature extraction for real-time mode:
feature_extractor_single(site, file_name = 'DUMP')

Feature extraction for off-line mode:
feature_extractor_multi(start_date, end_date, site)
```

The functions are used to call the BGP C# tool [120] to extract 37 numerical features of each minute from the update messages.

Input of **feature\_\_extractor\_\_single()**:

- *site*;
- *file\_name* (string, default=“DUMP”): filename of the parsed update message in ASCII format.

Output of **feature\_extractor\_single()**:

- The output file (*DUMP\_out.txt*) is saved in the directory *./data\_test*.

Input of **feature\_extractor\_multi()**:

- *site*; *start\_date*; *end\_date*.

Output of **feature\_extractor\_multi()**:

- The output files (*DUMP\_yyyymmdd\_out.txt*) is saved in the directory *./data\_split*.

---

```
Module:
labelGeneration.py

Functions:
Label generation for off-line mode:
label_generator(start_date_anomaly, end_date_anomaly,
                start_time_anomaly, end_time_anomaly,
                site, matrix_file_list)
```

This function is used to generate regular (0) and anomalous (1) labels based on the specified date and time of an anomalous event.

Input of **label\_generator()**:

- *start\_date\_anomaly* (string): start date for the anomalous event.
- *end\_date\_anomaly* (string): end date for the anomalous event.
- *start\_time\_anomaly* (string): start time for the anomalous event.
- *end\_time\_anomaly* (string): end time for the anomalous event.
- *site*.
- *matrix\_file\_list* (list): list consists of the filenames of the entire experiment.

Output of **label\_generator()**:

- The output labels are saved in a list as well as put into the directory *./STAT*. Format: [*"DUMP\_yyyymmdd\_out.txt"*, ..., *"DUMP\_yyyymmdd\_out.txt"*].

---

```
Module:
dataPartition.py

Functions:
Data partition for off-line mode:
data_partition(cut_pct, site, matrix_file_list, labels, rnn_seq = 10)
```

This function is used to split the entire dataset into training and test datasets based on the percentages of anomalous data.

Input of **data\_partition()**:

- *cut\_pct* (string): partitioning percentage based on anomalous data.
- *site*; *matrix\_file\_list*;
- *labels* (list): labels for the entire experiment.
- *rnn\_seq* (integer, default=10): sequence length when using RNN algorithms.

Output of **data\_partition()**:

- The training and test datasets are saved in the directory *./data\_split*.
- The file *train\_test\_stat.txt*, including statistics of the regular and anomalous data points, is saved the directory *./STAT*.

---

Module:  
`dataProcess.py`

Functions:  
`normTrainTest(cut_pct, site)`  
`feature_select_ExtraTrees(cut_pct, site, topFeatures = 10)`

Normalization and feature selection are performed when using these functions.

Input of **normTrainTest()**:

- *cut\_pct*; *site*.

Output of **normTrainTest()**:

- The normalized dataset.

Input of **feature\_select\_ExtraTrees()**:

- *cut\_pct*; *site*;
- *topFeatures* (integer): number of the most relevant features.

Output of **feature\_select\_ExtraTrees()**:

- The indices of most relevant features from input dataset.
- The subset of the input dataset.

Listed are the developed functions for real-time detection and off-line classification.

---

Module:  
`app_realtime.py`

Function:  
`app_realtime_detection(ALGO='VFBS', site='RIPE', count=0)`

This function is used to process the variables emitted from the client (real-time classification) and return the detection results. The results are then emitted to the client.

Input:

- *ALGO* (string, default="VFBS"): pre-trained ML model.
- *site* (string);
- *count* (integer, default=0): the number of performed detection.

Output:

- The time stamps corresponding to the process update messages.
- The extracted features, predicted labels, and CPU usage.

---

Module:  
`app_offline.py`

Function:  
`app_offline_classification(ALGO='VFBS', site='RIPE', start_date, end_date,  
start_date_anomaly, end_date_anomaly,  
start_time_anomaly, end_time_anomaly,  
cut_pct, rnn_seq)`

This function is used to process the variables emitted from the client (off-line classification) and return the detection results.

Input:

- *ALGO*; *site*; *start\_date*; *end\_date*;  
*start\_date\_anomaly*; *end\_date\_anomaly*; *start\_time\_anomaly*; *end\_time\_anomaly*;  
*cut\_pct*; *rnn\_seq*.

Output:

- Results in CSV format are saved the directory `./STAT`.

## Appendix D

# BGPGuard: Sample output

Listed are the sample settings and output when executing the real-time detection mode of *BGPGuard*: The mode “real-time” and collection site “RIPE” are entered.

```
> python main.py
real-time or off-line?
Mode: real-time
Choose collection site: RIPE or RouteViews
Collection site: RIPE
```

Output from terminal:

```
=> >>>>>>>>> > > > > Processing update_message_file: updates
    .20220213.0430

--2022-02-12 20:39:04-- http://data.ris.ripe.net/rrc04/2022.02/updates
    .20220213.0430.gz
Resolving data.ris.ripe.net (data.ris.ripe.net)... 2001:67c:2e8:22::c100:68c
, 193.0.6.140
Connecting to data.ris.ripe.net (data.ris.ripe.net)|2001:67c:2e8:22::c100:68
c|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://data.ris.ripe.net/rrc04/2022.02/updates.20220213.0430.gz [
    following]
--2022-02-12 20:39:05-- https://data.ris.ripe.net/rrc04/2022.02/updates
    .20220213.0430.gz
Connecting to data.ris.ripe.net (data.ris.ripe.net)|2001:67c:2e8:22::c100:68
c|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 237928 (232K) [application/x-gzip]
Saving to: 'data_ripe/updates.20220213.0430.gz'

updates.20220213.0430 100%[=====>] 232.35K 337KB/s
    in 0.7s

2022-02-12 20:39:06 (337 KB/s) - 'data_ripe/updates.20220213.0430.gz' saved
    [237928/237928]

=> >>>>>>>>> > > > > Extension changed (gz to Z)
```

```

[=====] 100.0% ...
    Converting
=> >>>>>>>>> > > > > DUMP generated (MRT to ASCII)

[=====] 100.0% ...
    Generating
=> >>>>>>>>> > > > > Matrix generated (txt)

+=====+
| Data download      === done |
| MRT to ASCII       === done |
| Matrix generation  === done |
| Next step:         |
|      Classification |
+=====+

Classification results:
Test time (hour:minute) 04 : 30 => Normal traffic

Test time (hour:minute) 04 : 31 => Normal traffic

Test time (hour:minute) 04 : 32 => Normal traffic

Test time (hour:minute) 04 : 33 => Normal traffic

Test time (hour:minute) 04 : 34 => Normal traffic

```

---