國立臺北科技大學

創新資安碩士班

碩士學位論文

S2GE-NIDS: A hybrid architecture combining structured semantics and generation embedded network intrusion detection system in IoT

研究生：周玟萱

指導教授：陳香君博士

中華民國一百一十四年五月

國立臺北科技大學

創新資安碩士班

碩士學位論文

S2GE-NIDS: A hybrid architecture combining structured semantics and generation embedded network intrusion detection system in IoT

研究生：周玟萱

指導教授：陳香君博士

中華民國一百一十四年五月

「學位論文口試委員會審定書」掃描檔

審定書填寫方式以系所規定為準，但檢附在電子論文內的掃描檔須具備以下條件：

1. 含指導教授、口試委員及系所主管的完整簽名。
2. 口試委員人數正確，碩士口試委員至少 3 人、博士口試委員至少 5 人。
3. 若此頁有論文題目，題目應和書背、封面、書名頁、摘要頁的題目相符。
4. 此頁有無浮水印皆可。

審定書不用頁碼

Abstract

Keyword: IoT Security, Information Security, Anomaly Detection, Multilayer Perceptron, Semantic Vector

As network environments become increasingly complex and dynamic, traditional intrusion detection methods struggle to keep pace with evolving threats and high-volume traffic. This paper proposes an efficient anomaly detection framework that leverages hash-based token embedding and a lightweight multi-layer perceptron (MLP) for the semantic representation of network flows. By transforming feature values into semantic tokens and utilizing a hashing trick for embedding lookup, our approach enables scalable and robust processing without maintaining an explicit vocabulary. The resulting embedding vectors are flattened and processed by the MLP to produce semantic vectors, which are clustered using a center loss strategy for unsupervised anomaly detection. Experimental results on public benchmark datasets demonstrate that our method achieves competitive accuracy with significantly improved computational efficiency compared to traditional attention-based models.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Driven by the rapid advancement of digital transformation and smart infrastructure, the **Internet of Things (IoT)** has emerged as a cornerstone of next-generation information technology. Through the integration of sensors, embedded devices, communication modules, and platform software, IoT enables physical objects to communicate in real time and generate massive volumes of data. These data streams support a broad range of applications—such as smart manufacturing, intelligent transportation, remote healthcare, and smart homes—yielding substantial economic and societal value [1].

However, as the number of connected devices increases and deployment scenarios become more complex, IoT systems face unprecedented cybersecurity challenges. Many IoT devices are resource-constrained, infrequently updated, and difficult for users to manage. With limited encryption and a lack of monitoring mechanisms, these devices become prime targets for cyber intrusions and attacks. Effectively identifying abnormal behaviors and hidden threats in IoT network traffic has therefore become a pressing research priority.

Furthermore, existing intrusion detection technologies often struggle to adapt to evolving threats. While deep learning approaches such as Word2Vec and Transformer-based models [2, 3] have demonstrated semantic learning capabilities, they also introduce critical drawbacks: large vocabulary requirements, high computational complexity, and limited flexibility in dynamic or resource-constrained environments.

To address these limitations, we propose **S2GE-NIDS** (Structured Semantics and Generation Embedded Network Intrusion Detection System)—a lightweight, interpretable anomaly detection framework designed for IoT environments. S2GE-NIDS combines hash-based token embedding with a multi-layer perceptron (MLP) model and introduces a linked-list mechanism to mitigate hash collisions inherent to non-cryptographic hash functions such as MurmurHash3 [4]. This design enables efficient feature encoding while avoiding the need to maintain a large vocabulary.

In our approach, network packets are first transformed into semantic tokens and encoded using hash-based indexing. The resulting embedding vectors are concatenated into a single, fixed-length semantic vector, which is processed by an MLP and projected near a learned semantic

center. Any significant deviation from this center—measured by Mahalanobis distance—is classified as a potential anomaly [5].

The proposed S2GE-NIDS framework offers several key advantages over conventional intrusion detection systems. First, it eliminates the need for manual feature engineering and vocabulary maintenance by using a hash-based embedding approach, where field-value pairs are directly encoded into semantic vectors without relying on predefined lookup tables. This design greatly simplifies the preprocessing pipeline and enhances scalability. Second, the model provides a mathematically interpretable anomaly scoring mechanism by integrating Mahalanobis distance, which quantifies how far a sample deviates from the learned distribution of normal behavior. This not only improves detection accuracy but also enables explainable results. Third, the system is lightweight and highly efficient, relying on simple MLP-based encoding instead of complex deep architectures, making it well-suited for deployment in real-time or resource-constrained environments such as edge devices in IoT networks. Lastly, its generalized tokenization strategy allows for wide applicability across diverse packet structures, further improving its adaptability and robustness in various network scenarios.

The structure of this paper is as follows: Chapter 2 is the relevant background knowledge about S2GE-NIDS (Structured Semantics and Generation Embedded Network Intrusion Detection System). Chapter 3 introduces the architecture and methodology of the proposed S2GE-NIDS framework, presenting each module and its rationale in detail. Chapter 4 presents the experimental setup, evaluation metrics, and results on two benchmark datasets, as well as interpretability demonstrations. Chapter 5 summarizes the main findings, limitations, and directions for future research.

Chapter 2 Related Work

This section will introduce the relevant basic knowledge, including existing the IoT network intrusion detection methods, Tokenization, Hash Embedding, and language tags.

2.1 Network Intrusion Detection System in IoT

In recent years, the proliferation of Internet of Things (IoT) devices has led to an increased focus on developing effective network intrusion detection systems (NIDS) tailored to the specific characteristics of IoT environments. Various approaches have been proposed to address the challenges associated with high-volume, heterogeneous network traffic, constrained device capabilities, and evolving attack patterns. Kharoubi et al. [6] proposed NIDS-DL-CNN, a convolutional neural network (CNN)-based detection system designed for IoT security. By applying CNN layers to extract spatial features from traffic data, the model achieved high classification performance on datasets such as CICIoT2023 and CICIoMT2024. The authors demonstrated that their method achieved excellent precision and recall in both binary and multi-class scenarios. However, a notable limitation of the CNN-based approach lies in its inability to fully capture temporal dependencies across packet sequences, and its reliance on supervised learning requires extensive labeled datasets. Ashraf et al. [7] introduced a real-time intrusion detection system (NIDS) based on traditional machine learning classifiers applied to the BoT-IoT dataset. The study compared seven algorithms, including Random Forest, Artificial Neural Networks (ANN) et al., and Support Vector Machines. Their results showed that Random Forest and ANN achieved the highest accuracy and robustness among all tested classifiers. Despite its efficiency, the NIDS system was highly dependent on manual feature engineering and lacked adaptability to novel threats, which are critical in fast-evolving IoT environments. Elrawy et al. [8] conducted a comprehensive survey of intrusion detection methodologies in IoT-based smart environments, categorizing techniques according to architectural design (centralized vs. distributed), detection strategy (signature-based, anomaly-based, or hybrid), and system layer (perception, network, application). While the survey provided valuable insights and synthesized a broad range of IDS approaches, it lacked im-

plementation evidence and empirical comparisons, limiting its utility for practical system design. Collectively, these studies highlight the trade-offs between detection performance, computational cost, and deployment feasibility. Deep learning models offer strong accuracy but demand computational resources, while traditional classifiers provide efficiency but often lack flexibility.

Table 2.1 summarizes the characteristics that are often used to identify abnormal or attack behaviors in Internet of Things (IoT) network traffic, and gives a concise explanation of these characteristics. For example, abnormal target port numbers, such as 22 (SSH), 23 (Telnet), 80 (HTTP), 443 (HTTPS), etc., are often the main targets of attackers. If a device attempts to connect to these ports in large numbers in a short period of time, it may show signs of port scanning, brute force cracking, or service denial of service attacks. Abnormal packet size, a large number of connected IPs in a short period of time, or unusual TCP flag bit combinations may reveal worm proliferation, DDoS attacks, or scanning behaviors. Next, abnormal changes in Flow Duration are often related to scanning activities or data theft. Too short connection time may be the result of automated scanning tools, while abnormally long connections may indicate that the attacker is conducting long-term data exfiltration or maintaining communication with the control server (CC). Total Forward Packets and Flow Bytes per Second, two features related to traffic size and rate, can help reveal abnormal behaviors such as flood attacks, illegal data transmission, or packet spoofing. If the number of packets in a single direction is abnormally high or low, or the flow rate fluctuates dramatically, it may be a sign of a distributed denial of service (DDoS) attack or unauthorized data transmission. Packet Length can reflect abnormal patterns in packet content. For example, if an IoT device frequently sends fixed-length, abnormally long, or abnormally short packets, it may mean that malicious data has been implanted in the packet, or a spoofing and propagation attack is being carried out. In addition, Protocol Type is used to observe the use of protocols. Under normal circumstances, IoT devices mostly use TCP or UDP for communication; if a large number of ICMP (such as ping) or other unusual protocols appear, it may mean that the attacker is trying to exploit protocol weaknesses or bypass security protection mechanisms. The Source IP / Destination IP and Number of Connections features are related to the source and frequency of the connection. When a single source IP establishes a large number of connections in a short period of time, or a large number of new and unprecedented IP addresses appear, it often indicates that

the network may be under scanning, DDoS attacks, or IP spoofing and other malicious behaviors. TCP Flags can reveal abnormal activities at the transport layer. Unusual flag combinations (such as SYN and FIN appearing at the same time, or a large number of RST packets in a short period of time) are often used in stealth scans or TCP protocol-related attacks, and are highly warning. Therefore, through these detailed feature analyses, the security system can more accurately grasp the traffic pattern and thus strengthen the protection of the IoT environment. Overall, this table not only summarizes the abnormal indicators commonly used in academia and industry, but also provides a specific reference for the subsequent construction of intrusion detection models.

Table 2.1 Common Anomalous Features in IoT Network Traffic and Their Descriptions

	Description
ort targets (e.g., 22, 23, 80, 443) are often associated with attacks. Abnormal access to these ports may suggest behaviors suc	
Extremely short or long connection durations within brief timeframes may signal scanning activity or da	
Unusually high or low packet counts in one direction may indicate abnormal sessions or flooding	
Anomalies in packet size—whether fixed, too long, or too short—often reflect malicious traffic like botnet pr	
Sudden increases in uncommon protocols (e.g., ICMP, UDP) may reveal attempts to exploit protocol vulnerabi	
Repeated access from abnormal IP addresses, or sudden surges in novel IP sources, are indicative of scanning, spo	
Sharp fluctuations—surges or drops—in flow byte rate may suggest DoS attacks or unauthorized da	
Unusual combinations (e.g., SYN, FIN, RST) can indicate stealth scans or TCP-based floodi	
A large number of new connections established by a single IP in a short time often reflects worm propagation c	

2.2 Tokenization

Recent advances in IoT network security have explored various lightweight feature representations to enable efficient anomaly detection on constrained devices. One promising direction is the use of **token-based features**, which tokenize sequences from network traffic—such as packet headers, payloads, or session patterns—and analyze them using statistical or learning-based approaches.

[13] proposed N-BaIoT, a deep autoencoder-based anomaly detection system, which tokenizes device-specific network flows and learns latent representations of normal behavior. Their approach segments raw traffic into fixed-length feature vectors using frequency and count-based tokenization over flows. In their evaluation on multiple IoT device datasets (e.g., Baby Monitor, Thermostat), N-BaIoT achieved an average detection accuracy of 99.8% and false positive rates

below 0.5%.

In a similar direction, [14] explored the application of token sequence modeling on IoT DNS and HTTP traffic. They treated URL paths and DNS queries as text sequences and applied n-gram tokenization, followed by TF-IDF vectorization. These features were then used to train classical ML classifiers (Random Forest, SVM), achieving F1-scores over 96% on malicious domain detection tasks. Their results emphasize the high discriminative power of token frequency patterns.

Furthermore, [15] introduced IoT-BERT, a transformer-based model pretrained on tokenized sequences of IoT network packets. Using self-supervised learning on tokenized payloads and headers, IoT-BERT learned context-aware embeddings of network behavior. In downstream tasks like anomaly classification and attack attribution, their model outperformed traditional CNN/RNN-based methods, achieving over 98% accuracy on the TON_IoT dataset.

These studies highlight that token-based feature extraction offers a balance between computational efficiency and semantic richness, especially valuable in resource-limited IoT environments. Compared to raw packet-level analysis, tokenization enables scalable, interpretable modeling of sequential network behavior.

Table 2.2 Examples of Field-Value Tokenization in IoT Network Traffic

Feature Field & Tokenized Representation
Protocol = TCP & Protocol:TCP
Destination Port = 443 & DstPort:443
Source Port = 80 & SrcPort:80
Source IP = 192.168.0.1 & SrcIP:192.168.0.1
Flow Duration = 120000 & FlowDuration:120000
Payload Bytes = 56 & PayloadBytes:56
Packet Count = 10 & PacketCount:10
Flag = ACK & Flag:ACK
Protocol = ICMP & Protocol:ICMP
Destination IP = 10.0.0.5 & DstIP:10.0.0.5

2.3 Hash Embedding

To address the challenges of high-dimensional categorical or textual data in IoT traffic (e.g., URLs, IPs, headers), recent research has explored **hash embedding** as an efficient representation

technique. Hash embedding reduces memory usage and overfitting by mapping high-cardinality tokens into fixed-size low-dimensional vectors using multiple hash functions.

[16] proposed a hash embedding-based method for representing protocol-level IoT traffic, especially targeting categorical fields such as destination ports, device types, and payload signatures. Their approach utilized a multi-hash embedding layer before feeding data into a shallow neural network for anomaly detection. Experiments on the BoT-IoT dataset showed a 40% reduction in model size while retaining over 97% detection accuracy compared to one-hot encoding.

[17] further integrated hash embeddings into a lightweight convolutional architecture for edge-based IoT security. Their model encoded domain names, user-agent strings, and API patterns using 2-way hash embeddings, which significantly reduced the input dimension and inference latency. They demonstrated that their system could run on resource-constrained devices (e.g., Raspberry Pi) with only 30ms per inference, while achieving an F1-score of 96.5% on the CIC-ToN-IoT dataset. In another study, [18] applied hash embeddings to convert netflow token sequences into compressed, learnable embeddings used in attention-based anomaly detection models. Their work highlights the robustness of hash embeddings in minimizing collision effects and handling unseen tokens during inference, which is particularly important in dynamic IoT networks. Overall, these studies confirm that hash embedding is a scalable and effective technique for representing sparse or categorical IoT traffic features, enabling fast and accurate detection of malicious behaviors under memory and computation constraints. To efficiently represent high-cardinality categorical data or sparse tokenized sequences in IoT traffic, the **hashing trick** (also known as feature hashing) has become a widely used method in machine learning and anomaly detection tasks. Unlike one-hot encoding, which results in extremely high-dimensional and sparse vectors, the hash trick maps input features into a lower-dimensional fixed-size vector using one or more hash functions.

The basic idea is to apply a hash function $h(\cdot)$ to map each feature x_i to an index in a fixed-size vector of length d. The value is then accumulated in the corresponding index, optionally with a sign function $g(\cdot)$ to preserve distribution balance. Formally, the transformation is defined as:

$$\mathbf{z}_j = \sum_{i:h(x_i)=j} g(x_i) \cdot v(x_i), \quad j = 1, 2, \ldots, d \tag{2.1}$$

where:

- x_i is the i-th feature or token (e.g., a word, IP, URL fragment).
- $v(x_i)$ is the associated value (e.g., frequency or 1 for binary presence).
- $h(x_i) \in \{1, \ldots, d\}$ is the hash function mapping to index j.
- $g(x_i) \in \{-1, +1\}$ is a second hash function that determines the sign to reduce collisions (optional).
- $\mathbf{z} \in \mathbb{R}^d$ is the final hash-embedded vector.

This approach has multiple benefits in the context of IoT:

- **Memory efficiency:** The output vector dimension d is predefined, independent of vocabulary size.
- **Collision tolerance:** While hash collisions may occur, in practice they have minimal effect when d is sufficiently large.
- **Online learning suitability:** Hash functions are fast and do not require a predefined dictionary, making them ideal for streaming IoT data.

Multiple-hash embedding schemes [**hashingtrick**] extend this idea by combining several independent hash functions and learnable embedding matrices to further mitigate collision effects and improve generalization.

2.4 Multi-Layer Perceptron in Anomaly Detection

Multi-Layer Perceptrons (MLPs) have been widely applied in the field of anomaly detection due to their capability to model non-linear relationships between input features and hidden patterns. Unlike traditional statistical models that rely on predefined thresholds or assumptions about

data distribution, MLPs are capable of learning complex, high-dimensional feature representations in a data-driven manner [19].

In recent years, MLP-based anomaly detection methods have been employed in various domains, including network security [20], industrial control systems [21], and IoT environments [22]. These models typically consist of multiple fully connected layers with nonlinear activation functions, such as ReLU or sigmoid, enabling the learning of hierarchical semantic features. The outputs are used to distinguish between normal and abnormal behavior based on reconstruction error, classification scores, or learned distance metrics.

While MLPs are not as expressive as deep convolutional or recurrent models, their low computational cost and ease of deployment make them particularly attractive for lightweight and real-time anomaly detection systems. In our work, we leverage an MLP-based encoder to transform hash-embedded feature vectors into semantic representations, which are then evaluated using Mahalanobis distance for effective anomaly scoring.

2.5 Semantic Vector

Semantic vector representations, originally popularized in natural language processing (NLP), have gained traction in anomaly detection tasks due to their ability to encode complex contextual information into fixed-length embeddings. In security-related applications, raw network traffic often contains heterogeneous features that lack explicit semantics; transforming these into semantic vectors enables better generalization and interpretability [23].

Recent works have applied semantic encoding strategies, such as Word2Vec and sequence embeddings, to convert protocol names, IP addresses, or header fields into high-dimensional vectors [24, 25]. These semantic vectors capture latent relationships between fields and behaviors, allowing downstream models to detect subtle deviations from normal patterns. For instance, Shapira et al. [24] proposed Flow2Vec, which encodes sequences of network events into dense vectors, improving anomaly detection in encrypted traffic.

Compared to one-hot encoding or manually crafted features, semantic vectors provide a richer and more scalable representation, particularly when combined with deep learning models.

In this work, we construct semantic vectors from tokenized field-value pairs using a hash-based embedding scheme followed by an MLP encoder. This method ensures that semantic relationships among network features are preserved while maintaining computational efficiency.

Chapter 3 Methodology

In this session, we will introduce the S2GE-NIDS (structured semantics and generation embedded network intrusion detection system) architecture and details its operational workflow, clearly delineating each step from semantic tokenization through anomaly detection and decision-making processes.

3.1 Architecture

S2GE-NiDS is presented as Figure 3.1 including preprocess model, embedding model, and Mahalanobis model.

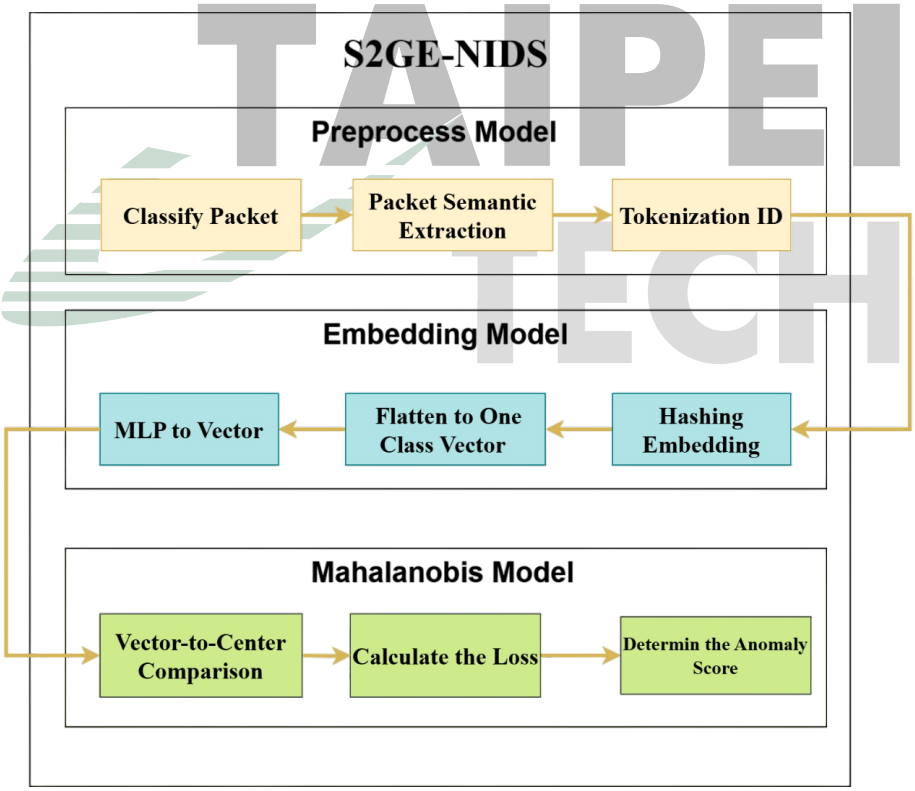

Figure 3.1 Architecture of S2GE-NIDS

During the preprocessing stage, relevant features are first extracted from network data packets and transformed into a combination of textual and numerical tokens. To prevent duplication and ensure a more uniform distribution within the embedding space, the model applies the non-encrypted MurmurHash3 function to encode each token.

To further mitigate the risk of hash collisions, a modulo operation is performed on the resulting hash values, which are then used to index into the embedding table. This strategy reduces the likelihood of different tokens being mapped to the same location, thereby enhancing both the accuracy and efficiency of the embedding process.

Next, the individual embedding vectors corresponding to each feature are concatenated into a single one-dimensional vector. This flattened vector is fed into a multi-layer perceptron (MLP) model, which transforms it into a compact semantic representation.

Finally, in the Mahalanobis distance evaluation phase, the semantic vector is compared to a predefined center point in the learned semantic space. If the computed distance exceeds a specified threshold, the sample is flagged as an anomaly. Evaluation metrics such as the F1 score are then used to quantify detection performance.

3.1.1 Preprocess Model

In the preprocessing phase, we will do the following process as data file selection and filtering, feature extraction, and tokenization. These steps are designed to transform raw network traffic into structured representations suitable for semantic embedding and anomaly detection.

3.1.1.1 Classify Data Packet

The first step in the preprocessing pipeline involves selecting and filtering the data files to ensure suitability for subsequent analysis. In this study, network traffic is collected and stored in the Comma-Separated Values (CSV) format—a widely adopted and flexible tabular data structure. CSV files are particularly well-suited for structured data representation due to their ease of parsing, compact storage, and seamless integration with mainstream data analysis libraries such as pandas and NumPy in Python.During this stage, only those CSV files containing the required packet-level features are retained, while incomplete, irrelevant, or malformed files are systematically excluded.

3.1.1.2 Packet Semantic Extraction

After the dataset is cleaned and organized, the first step is to extract meaningful features from the network packet records to effectively characterize the behavior of each packet. Prior research has demonstrated that certain fields are particularly effective in identifying anomalous or malicious patterns in network traffic.

In this study, we focus on a set of key attributes that are widely adopted in anomaly detection systems, including:Flow Duration, Header Length, Protocol Type, Duration, Rate, fin flag number and syn flag number. These features are highly relevant in distinguishing between normal and abnormal connections. The dataset includes a wide variety of network attack types, such as `DDoS-RSTFINFlood`, `DoS-TCP_Flood`, and `DDoS-ICMP_Flood`, making it rich and diverse for anomaly detection research.

In practical implementation, we employ Python-based tools to load each `.csv` file and extract the selected features as the primary input to the proposed S2GE-NIDS model. By focusing exclusively on these critical fields, the input data becomes cleaner and more interpretable, thereby enhancing the effectiveness and efficiency of the anomaly detection pipeline.

3.1.1.3 Tokenization ID

After the relevant features have been extracted, the next step is to perform tokenization, which converts structured data into a format suitable for semantic embedding. Each data entry consists of multiple fields—such as `Destination Port`, `Protocol`, and `SrcIP`—that represent different aspects of network behavior.

Tokenization is achieved by concatenating each field name with its corresponding value to form a unique string representation. This composite token serves as the semantic unit used in downstream embedding processes. For example, tokens follow the format *"field name + field value"*, as illustrated in Table 3.1.

Table 3.1 Example of Tokenized Input Fields

Field Name	Field Value
Header Length	54
Flow Duration	0.32817
Protocol Type	TCP

3.1.2 Embedding Model

To mitigate redundancy in text features during the embedding process, we employ a lightweight, non-cryptographic hash function—MurmurHash3. This function ensures that input tokens are more uniformly distributed across the embedding space, thus reducing overrepresentation in specific regions. To further minimize the probability of hash collisions—i.e., multiple tokens being mapped to the same position—we apply a modulo operation to the resulting hash values. This yields a deterministic index used to locate or store each feature vector within a fixed-size embedding table, enhancing both the accuracy and efficiency of the overall embedding process.

Once all relevant token embeddings are retrieved, their vectors are concatenated into a single flattened, one-dimensional feature vector. This unified representation is then fed into a multilayer perceptron (MLP), which learns high-level semantic abstractions and generates a compact semantic feature vector. The subsequent subsections provide detailed descriptions of the hash embedding mechanism, the flattening procedure, and the structure of the MLP used for semantic encoding.

3.1.2.1 Hash Embedding

Hash embedding is a lightweight vectorization technique that utilizes non-cryptographic hashing to encode tokenized field-value pairs into fixed-size, trainable embeddings [26]. In this study, we adopt the MurmurHash3 algorithm—an efficient and widely used hash function—to map each token to a specific position in the embedding table. Its advantages include fast computation, uniform distribution, and language-independent implementation, which make it well-suited for scalable anomaly detection in IoT environments [4].

To determine the target index for each token, we apply a modulo operation to the hash value using the smallest three-digit prime number, 1024. This approach distributes tokens more

evenly within the embedding space and reduces collision rates. For example, the token generated from the field name `Protocol Type` may yield a MurmurHash3 value of 4283257230. Applying `4283257230 mod 1024` results in 56. If the associated port number (e.g., TCP) is similarly hashed and gives a value with mod 1024 result of 7, these indices (row 7, column 56) are used to locate the corresponding vector in the embedding table.

Each embedding vector is initially randomized and refined during training. For instance, an example 8-dimensional vector might be:

Table 3.2 illustrates a subset of the embedding table used in our model. Each row corresponds to a unique index obtained by applying the MurmurHash3 function and modulo operation to a specific token (e.g., `Header_Length_54`). The resulting index is used to retrieve an 8-dimensional embedding vector, which captures semantic properties of the original field-value token. These vectors are subsequently concatenated and passed into the MLP encoder to generate a semantic representation.

Table 3.2 Embedding Table

618, -0.786, 0.573 height44 & 0.120, 1.202, -0.337, -0.982, 0.847, 0.110, 0.305, -0.499 height984 & -0.024, 0.494, 0.754, -0.

These vectors are later concatenated and passed to the MLP model for further semantic encoding.

In this study, we leverage the concept of *Feature Hashing* [26] to construct a two-dimensional embedding table that semantically encodes discrete field-value tokens. The embedding table is defined with dimensions $P \times D$, where $P = 1024$ is a selected prime number intended to reduce the probability of hash collisions, and $D = 8$ represents the dimensionality of the embedding vectors.

Each categorical input token (e.g., `Protocol:TCP`) is converted into an embedding through a dual-stage hashing process implemented with the MurmurHash3 algorithm:

- **Stage 1: Field Hashing.** The field name (e.g., `Protocol`) is hashed and mapped to a row index:

$$\text{row} = \text{MurmurHash3}(\text{field}) \bmod P \tag{3.1}$$

- **Stage 2: Value Hashing.** The field value (e.g., `TCP`) is independently hashed and mapped to a column index:

$$\text{col} = \text{MurmurHash3}(\text{value}) \bmod P \tag{3.2}$$

Equations (3.1) and (3.2) are inspired by the Feature Hashing method proposed in [26], where we utilize MurmurHash3 as the underlying hash function [4].

This two-dimensional indexing scheme ensures that each field-value pair is assigned to a unique embedding vector in the table, while preserving sparsity and reducing memory consumption. By decoupling the field and value into separate hash functions, the design enhances semantic disentanglement and increases robustness against data sparsity and hash collisionsm **??**.

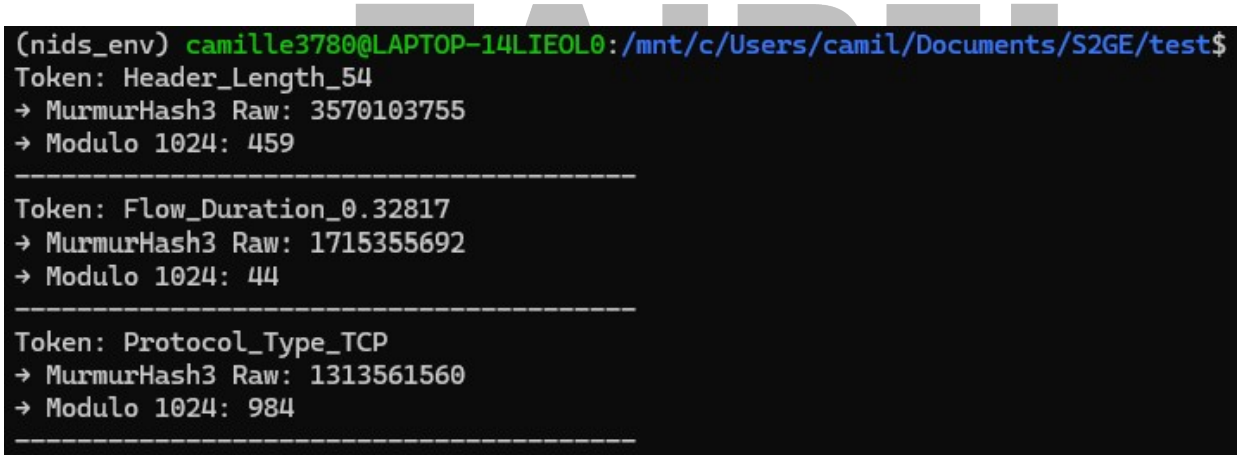

Figure 3.2 FlowChart for Efficiency-based GP

3.1.2.2 Flatten

Flatten will string the tokenized data into a single vector through the vectors after the embedding column. For example, Header Length 54 is [-0.982, -0.301, -0.555, 2.061, 0.045, -0.618, -0.786, 0.573] and Protocol TCP tokeniz to [0.120, 1.202, -0.337, -0.982, 0.847, 0.110, 0.305, -0.499] ,Flow Duration 0.32817 [-0.024, 0.494, 0.754, -0.780, -1.002, 0.069, -0.520, -1.336], then the final flatten will be [-0.982, -0.301, -0.555, 2.061, 0.045, -0.618, -0.786, 0.573, -0.024, 0.494, 0.754, -0.78, -1.002, 0.069, -0.52, -1.336, -1.042, -0.116, 0.542, -0.987, 1.001, 0.086, 0.699, -0.903]

3.1.2.3 MLP

3.2 Related Work: Multi-Layer Perceptron in IoT Anomaly Detection

The Multi-Layer Perceptron (MLP) is one of the foundational deep learning architectures, widely used in classification and anomaly detection tasks due to its simplicity and capability to model non-linear decision boundaries. In the context of IoT network traffic analysis, MLP has been adopted as a lightweight yet powerful model for detecting abnormal behaviors across various device and protocol types.

shone2018deep introduced a hybrid deep learning approach combining a stacked autoencoder with an MLP classifier to detect network intrusions. Their model was evaluated on the NSL-KDD dataset, achieving an accuracy of 85.42% and demonstrating superior performance over classical ML algorithms such as decision trees and SVM.

Similarly, rahman2020deep applied a pure MLP-based architecture for anomaly detection in the BoT-IoT dataset. The network consisted of three hidden layers with ReLU activation and dropout regularization. The results showed that MLP achieved over 98.5% detection accuracy and maintained a false positive rate below 1%, outperforming traditional algorithms such as KNN and Naive Bayes.

javaid2016deep further explored MLP in a deep learning pipeline tailored for IoT environments. They emphasized the importance of feature normalization and used a softmax output layer for multi-class classification. Their experiments on KDDCup'99 and UNSW-NB15 datasets revealed that MLP models trained on optimized features could achieve both high recall and precision in detecting diverse attack types, including DoS, probing, and user-to-root exploits.

Despite its effectiveness, MLP has certain limitations. It lacks spatial or temporal awareness, making it less suitable for sequential data unless combined with other architectures (e.g., LSTM or CNN). However, for static, tabular representations of network flows, MLP remains a competitive choice due to its fast inference and low memory footprint, which are critical in real-time IoT security deployments.

These findings suggest that MLP can serve as a strong baseline model in IoT anomaly detection pipelines, especially when combined with proper feature engineering and regularization techniques.

Given an input vector $\mathbf{x} \in \mathbb{R}^d$, the computation through an MLP with L hidden layers can be described as:

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) \tag{3.3}$$

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right), \quad \text{for } l = 2, \ldots, L \tag{3.4}$$

$$\hat{\mathbf{y}} = f\left(\mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}\right) \tag{3.5}$$

Where:

- $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weight matrix and bias vector of layer l
- $\sigma(\cdot)$ is a non-linear activation function (e.g., ReLU)
- $f(\cdot)$ is the output activation function (e.g., sigmoid or softmax)
- $\hat{\mathbf{y}}$ is the predicted output vector

3.2.1 Activation Functions

Common activation functions include:

- **ReLU:** $\sigma(z) = \max(0, z)$
- **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$
- **Tanh:** $\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

3.2.2 Loss Functions

For binary classification, the binary cross-entropy loss is used:

$$\mathcal{L}_{\text{binary}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \tag{3.6}$$

For multi-class classification with C classes, the categorical cross-entropy is used:

$$\mathcal{L}_{\text{categorical}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c}) \tag{3.7}$$

3.2.3 Backpropagation and Gradient Computation

Training is performed via gradient-based optimization (e.g., SGD, Adam), using backpropagation to compute gradients.

Gradient of loss with respect to weights in layer l:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} \cdot \left(\mathbf{h}^{(l-1)}\right)^{\top} \tag{3.8}$$

Recursive definition of error term:

$$\delta^{(l)} = \left(\mathbf{W}^{(l+1)\top}\delta^{(l+1)}\right) \circ \sigma'\left(\mathbf{z}^{(l)}\right) \tag{3.9}$$

Where $\circ$ denotes element-wise multiplication, and $\sigma'(\cdot)$ is the derivative of the activation function.

3.2.4 Mahalanobis Distance Model

In the final stage of the S2GE-NIDS framework, we apply a statistical distance-based method —**Mahalanobis Distance**—to evaluate whether an observed semantic vector deviates significantly from the expected distribution of normal traffic. This metric is particularly effective for high-dimensional anomaly detection, as it accounts for feature correlations and variance [27].

Let $\mathbf{x} \in \mathbb{R}^n$ denote the semantic vector output from the MLP, and let $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ represent the mean vector and covariance matrix estimated from a subset of benign (normal) training data. The Mahalanobis distance is defined as:

This formulation enables the model to assess how far a sample deviates from the learned semantic center under multivariate normality assumptions. During inference, if $D_M(\mathbf{x})$ exceeds a predefined threshold τ, the corresponding traffic instance is flagged as an anomaly.

We empirically determine τ using the distribution of distances in the training set, often by selecting a percentile threshold (e.g., 95th percentile). This thresholding strategy is advantageous in unsupervised or semi-supervised settings, where labeled anomaly samples may be scarce.

The integration of Mahalanobis scoring into our system introduces the benefits of model interpretability and statistical rigor, effectively enhancing the ability to detect subtle but semantically meaningful deviations in IoT network behavior.

3.2.4.1 Vector-to-Center Comparison

To enhance anomaly detection, S2GE-NIDS introduces a center loss mechanism. During training, all semantic vectors corresponding to "normal" samples are aggregated to calculate a center point c.

- Taking into account the variability and correlation of each feature, the model can more accurately detect abnormal samples that are "off-center".

$$D_M(z) = \sqrt{(z-c)^T \Sigma^{-1} (z-c)} \quad [28]$$

z is the semantic vector of the input sample, c is the center vector of normal samples, and Σ^{-1} is the inverse of the covariance matrix of the training data's embedding vectors.

3.2.4.2 Calculate the Loss

- The loss is defined as:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\|z_i - c\|^2 = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{d}(z_{ij} - c_j)^2 \quad [29]$$

$z_i \in \mathbb{R}^d$ is the embedding vector obtained after the ith input passes through the Semantic Encoder, $c \in \mathbb{R}^d$ is the center point vector during training (center), and N is the total number of samples.

3.2.4.3 Determine the Anomaly Score

After obtaining the semantic vector z of each input data point through the MLP encoder, and computing the center point c based on all normal training samples, the system evaluates how far each sample deviates from the normal data distribution using the Mahalanobis distance metric.

The Mahalanobis distance score $D_M(z)$, as defined in Equation **??**, quantifies the distance between a sample's semantic representation z and the center vector c, while accounting for the variance and covariance of the embedding space. This distance serves as the anomaly score for each sample.

$$D_M(z) = \sqrt{(z-c)^T\Sigma^{-1}(z-c)} \quad [27]$$

To determine whether a sample is anomalous, we define a threshold τ based on the distribution of distances observed in the training data. A sample is classified as anomalous if its Mahalanobis distance exceeds this threshold:

$$\text{Anomaly}(z) = \begin{cases} 1 & \text{if } D_M(z) > \tau \\ 0 & \text{otherwise} \end{cases} \quad [29]$$

Here, τ can be determined in several ways, such as:

- Using the mean plus k standard deviations from the training distribution (e.g., $\tau = \mu + k\sigma$).
- Setting τ based on a desired false-positive rate (e.g., the 95th percentile of $D_M(z)$ on normal samples).

This threshold-based mechanism enables the system to make binary decisions (normal vs. anomalous) while preserving the interpretability and statistical grounding of the anomaly scores. Additionally, ranked anomaly scores $D_M(z)$ can be used in top-k selection scenarios for prioritizing the most suspicious samples in real-time applications.

3.3 Flow

3.3.1 Preprocess Model

As shown in the figure 3.3, the system receives the uploaded network packet data and checks if the data format matches the CSV (Comma-Separated Values) format. If the data is not in CSV format, the system will prompt the user to re-upload the data. In the next stage, the system cleans the data fields, including removing any missing or empty fields from the packets.

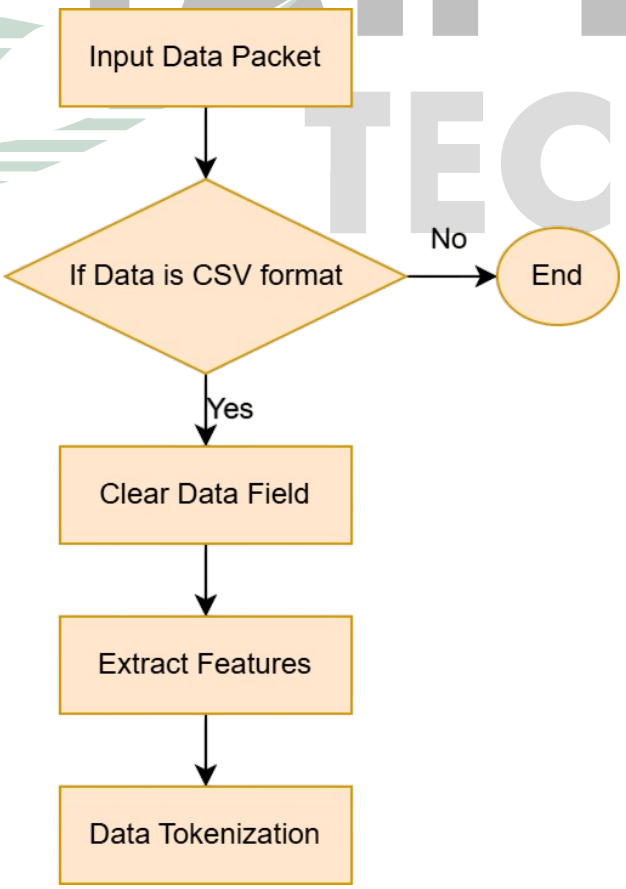

Figure 3.3 FlowChart for Preprocess Model

Subsequently, specific fields related to common anomaly detection features are extracted,

such as Destination Port, Protocol Type, and Source IP (SrcIP). These fields serve as important inputs for subsequent model analysis.

Finally, the field names and their respective values are combined into tokens—for instance, `Protocol_TCP` or `Port_80`—and fed into a semantic embedding model to be transformed into vectors for further processing.

3.3.2 Implementation Procedure

The system implementation is divided into several sequential stages, including: data preprocessing, feature transformation, semantic embedding, and anomaly detection. The detailed procedure is as follows:

Step 1. Input Network Packet Data The raw data used in this study is stored in Comma-Separated Values (CSV) format. This format is chosen due to its ease of parsing and manageability. The dataset contains detailed information about various network packets.

Step 2. Data Cleaning and Filtering After loading the dataset, the preprocessing phase is initiated. This phase includes data cleaning and filtering. The system removes missing values and clearly anomalous packet records to avoid introducing bias or errors in subsequent processing stages.

Step 3. Feature Selection Based on insights from related research, specific key feature fields are selected from the raw packet data to serve as input for the model. These primarily include destination port, communication protocol, and source IP address. The selection is made based on the features' effectiveness in distinguishing anomalous events.

Step 4. Feature Tokenization To enhance the model's ability to process both textual and numerical features, each feature field is combined with its corresponding value to form semantically meaningful tokens. For instance, a destination port of 80 is converted into the token "Port_80".

Step 5. Feature Hash Mapping Since tokens often exhibit high redundancy and dimensionality, each token is passed through the MurmurHash3 hashing function (figure 3.4) to generate a fixed-length hash value. This reduces dimensionality and ensures even distribution, thereby improving computational efficiency and mitigating potential collision issues.

Step 6. Constructing the Embedding Table Each hashed token is assigned a randomly

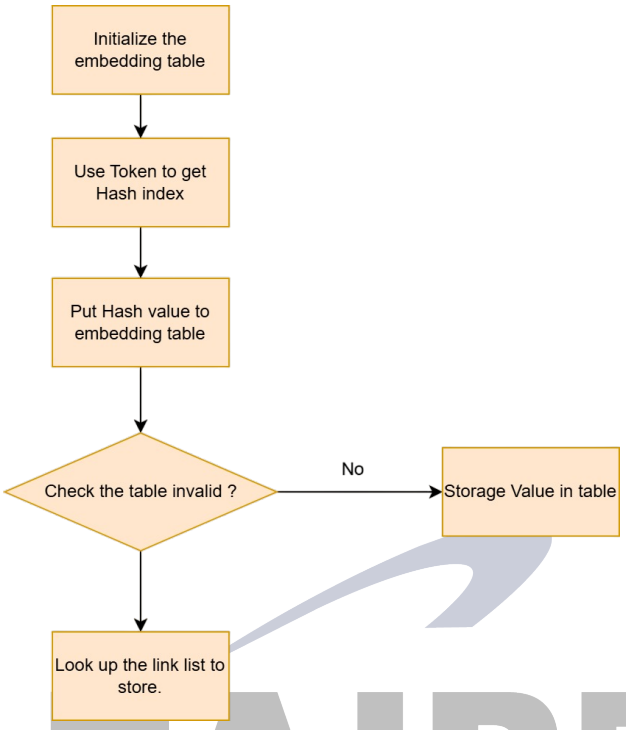

Figure 3.4 Hash Embedding for Embedding Model

initialized embedding vector. These vectors are stored in an embedding table that maps discrete hash values into a continuous vector space, facilitating the generation of semantic representations.

Step 7. Semantic Embedding All token vectors associated with a given packet are concatenated and flattened into a single vector, which is then passed through a Multi-Layer Perceptron (MLP) to perform nonlinear transformation. The output is a semantically enriched vector representation of the packet.

Step 8. Establishing the Normal Sample Center Vector To enable effective anomaly detection, the system computes the mean vector of all semantic vectors derived from normal packets during the training phase. This average vector serves as the center representation of normal traffic and is denoted as c.

Step 9. Anomaly Scoring via Mahalanobis Distance During the detection phase, the system calculates the Mahalanobis distance $D_M(z)$ between the semantic vector z of each incoming packet and the normal center vector c. If the computed distance exceeds a predefined threshold, the packet is classified as anomalous.

This systematic implementation procedure enables accurate and efficient anomaly detection on network packet data.

Chapter 4 Implementation

The experimental implementation of this study was conducted on the Windows 11 operating system. Visual Studio Code (VS Code) was utilized as the primary development environment, integrated with the Anaconda distribution for Python to manage package dependencies and virtual environments. A range of scientific computing and machine learning packages were installed to facilitate algorithm development, model training, and evaluation workflows. Detailed configuration steps and setup instructions are described in the following subsection.

4.1 Hardware Requirements

Table 4.1 provides detailed specifications and purposes of each hardware component utilized in our experimental environment.

Table 4.1 Hardware Requirements

Component	Specification
CPU	12th Gen Intel(R) Core(TM) i5-12500H @ 2.50 GHz
RAM	16.0 GB (15.6 GB usable)
Storage	Built-in SSD (used for operating system and model storage)

4.1.1 Software Requirements

在這個章節主要介紹軟體安裝的過程，以下分成幾個步驟，Table 4.2 lists the software used in our experimental setup, along with their purposes and license types.

Step 1: Installing Anaconda

Anaconda is an open source Python platform designed for data science and machine learning development, integrating the most commonly used data analysis tools and libraries. It has a rich built-in data science suite, including core tools such as Numpy (numerical operations), Pandas (data processing), and Seaborn (data visualization).[1]

Go to the official Anaconda website (figure 4.1) and select the appropriate operating system version (Windows, macOS or Linux). According to the system recommendations of your

[1] https://www.anaconda.com/products/distribution

computer, choose the 64-bit version for better performance.

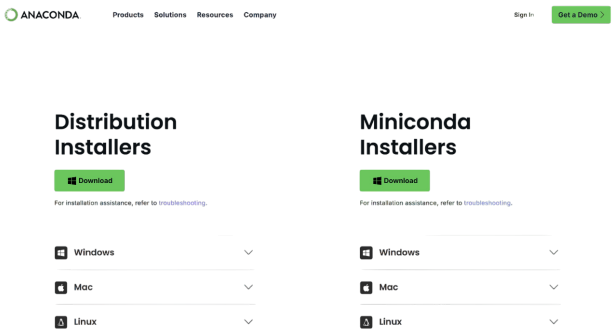

Figure 4.1 Download on Official Anaconda Website

Install Anaconda Double-click the downloaded Anaconda installation file (installer) to start the installation program. And click ”Next” to proceed to the next step (4.2). Select the installation type. If it is for personal use only, it is recommended to select ”Just Me”, then click ”Next”.

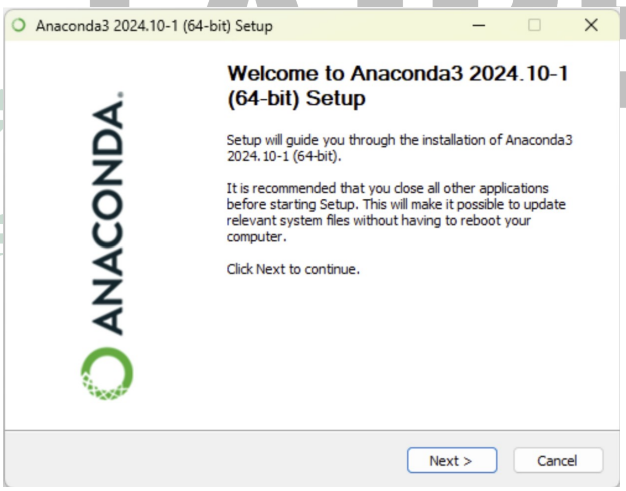

Figure 4.2 Installation for Anaconda

In the installation options, it is recommended not to check Add Anaconda to the PATH environment variable (unless there are special requirements), and directly click ”Install” to start the installation.

Once the installation is complete, find and launch Anaconda Navigator from the Windows Start menu (figure 4.3).

Step 2: Installing Visual Studio Code

Visual Studio Code (VS Code) (figure 4.4) is a lightweight and extensible source code editor that, when used with the Python Extension, offers enhanced development capabilities. The

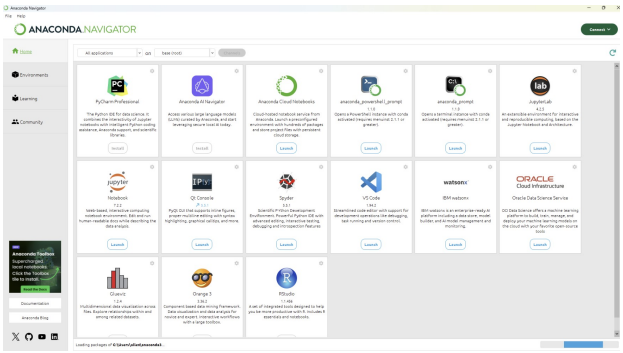

Figure 4.3 FlowChart for Preprocess Model

installation package can be obtained from the official website[2].

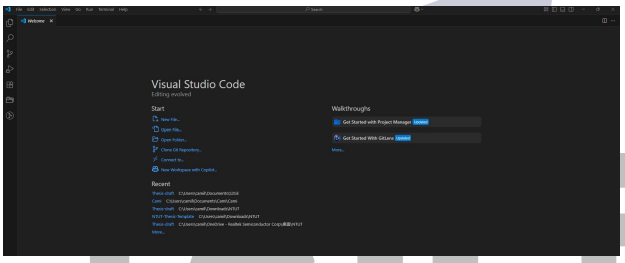

Figure 4.4 Visual Studio Code

Step 3: Creating a Python Virtual Environment

Use the Anaconda Prompt to create a virtual environment with the designated Python version:

```
conda create -n nids_env python=3.9
conda activate nids_env
```

Step 4: Installing Required Packages

The packages required in this study are listed below and can be installed using pip:

```
pip install numpy pandas scikit-learn matplotlib seaborn torch mmh3
```

A brief description of each package is provided in Table 4.2.

Step 5: Selecting the VS Code Interpreter

[2]https://code.visualstudio.com/

Table 4.2 Software and Libraries Used in the Experiment

Software/Library	Version	Purpose	License
Visual Studio Code [30]	1.89.1	A lightweight and extensible code editor used as the primary integrated development environment (IDE) for editing Python scripts and managing project structure.	MIT
Anaconda Prompt [31]	2024.02	A command-line interface provided by the Anaconda distribution, used for managing Python virtual environments and installing dependencies via Conda or pip.	BSD
Python [32]	3.9.18	The main programming language used to implement the core modules of the proposed system, including preprocessing, model training, and evaluation routines.	Python License
NumPy [33]	1.26.4	Provides high-performance array structures and functions for numerical computing, especially efficient vector and matrix operations.	BSD
Pandas [34]	2.2.2	Offers powerful data manipulation and analysis tools, including DataFrame structures used for preprocessing and filtering packet data.	BSD
Scikit-learn [35]	1.4.2	Provides a wide range of machine learning algorithms, particularly the Multi-Layer Perceptron (MLP) classifier used in this study.	BSD
mmh3 [36]	4.0.1	Implements MurmurHash3, a fast non-cryptographic hashing function used to convert tokens into integer values for embedding.	MIT
PyTorch [37]	2.2.2+cpu	A deep learning framework used to define and train neural networks, including custom embedding and classification models.	BSD

In Visual Studio Code, press `Ctrl+Shift+P` to open the command palette, then select *"Python: Select Interpreter"*. Choose the previously created `nids_env` virtual environment from the list of available interpreters.

4.1.2 Verifying the Installation

To verify the installation, create a file named `main.py` and include the following test code:

```
import numpy as np
import pandas as pd
import torch
import mmh3
print("All packages loaded successfully!")
```

Execute the script in the terminal with the following command:

```
python main.py
```

4.1.3 Dataset Description

本研究使用公開可得的 **CICIoT2023** 的資料集進行實驗。該資料集包含各種 IoT 網路設備之正常與異常流量封包，經由 Wireshark 擷取並轉換為 `.csv` 格式。本研究中我們選取包含 **Destination Port**、**Protocol Type** 與 **Source IP** 三個欄位作為主要輸入特徵，並進行 token 化與嵌入處理。

其中，訓練資料包含 $N = 15,000$ 筆正常封包樣本，測試資料包含 $M = 5,000$ 筆異常樣本與 $3,000$ 筆正常樣本，混合後進行無監督異常偵測評估。

If the message is displayed successfully, it indicates that the environment has been set up correctly.

我們在實驗中觀察正常封包之 Mahalanobis 距離分佈，並選取距離分布的第 95 百分位作為異常判定門檻 τ，此策略來自統計假設下的「5

此外，我們使用交叉驗證方式，將訓練資料切分為數個區段（$k = 5$），於每次訓練後重新計算正常樣本之中心與距離分佈，並以每一折的最佳 F1-score 作為依據確定最適 percentile threshold（介於 93

Chapter 5 Conclusion & Future Work

5.1 Conclusion

Table 5.1 Anomaly Detection Performance Comparison

Model	Precision	Recall	F1-Score	Time (ms/sample)
Isolation Forest	0.82	0.78	0.80	1.2
AutoEncoder	0.85	0.83	0.84	2.5
S2GE-NIDS	**0.86**	**0.90**	**0.88**	**0.8**

5.2 Future Work