Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 4, Issue 3, 2026

AI BASED AUTOMATED ANOMALY DETECTION FRAMEWORK FOR MOBILE APPS

Muhammad Bilal Mansoor[*1], Hamza Bin Irfan[2]

[*1]*Department of Computer Science & Information Technology, NED University of Engineering & Technology. Senior Consultant - Mobile App Developer, Cloud and Mobility Department, Systems Limited.*
[2]*Department of Computer Science & Information Technology, NED University of Engineering & Technology. Senior Consultant – Cloud, Data and AI, Veraqor*

[*1]mbilal.mansoor@systemsltd.com, [2]hamzakhan.indus@gmail.com

DOI: https://doi.org/10.5281/zenodo.19364156

Keywords
machine learning, mobile application, anomaly detection

Article History
Received: 31 January 2026
Accepted: 15 March 2026
Published: 31 March 2026

Corresponding Author: *
Muhammad Bilal Mansoor

Abstract
When it comes to mobile applications, security threats are rising day by day which requires to adopt the advanced detection mechanisms beyond traditional methods. Existing methods significantly depend on heuristic and signature-based models, which have limitations in detecting operational anomalies and unknown threats. This research reveals an AI/ML-driven method that helps in real-time anomaly detection in mobile applications. The suggested system examines API calls to detect deviations from normal behavior. This aids in early finding of security threats, performance issues, and misconfigurations. Unlike traditional methods that depend on periodic scans and predefined rules, this method enhances its capabilities to address growing threats utilizing AI-based models. Analyzing API-level interactions improves detection granularity, which aids in a more effective response to anomalies. The goal of this system is to increase reliability and security in mobile apps by reducing the use of manual approaches and instead depending on a much broader approach, including dynamic ones by introducing a technique where a framework on a backend level takes care of avoiding multiple API calls within the mobile app. This takes away the need for republishing mobile apps and avoids the entire process of internally/ externally testing the app all over again when there is not even a business need for it. Our framework incorporates the following ML approaches i.e. Isolation Forest, Local Outlier Factor, Histogram Based Outlier Score, Autoencoders and One Class Support Vector Machine which we further discuss later in this research. We make the comparisons in terms of finding out the performance and accuracy rate of our overall system using the evaluation metrics such as Accuracy, F1-Score, Precision and Recall. The results are later extracted and concluded to encourage especially the mobile app developers to consider incorporating such frameworks to evolve their systems for the purpose of resolving at least API level issues dynamically without having to make manual interventions on the front-end layer of code and then follow the republishing process all over again.

1. INTRODUCTION

A repeated unnecessary number of API calls inside a mobile app can be a very draining thing for the server and its resources which can negatively impact on the overall performance in a mobile app, leading to create unnecessary delays while interacting with the servers [3]. The growing increase in the number of mobile apps that get published on an everyday basis affects directly the use of APIs for smooth communication between both client and server. However, the lack of control a developer may have in managing the number of times these APIs are called in a mobile app once the application goes live is very less and restricted [38]. This insufficient control that mobile app developer has in managing these APIs in a mobile app once the app is published places them in a very bad spot especially when publishing a mobile app requires many complex business protocols such as approvals from different levels of the board[43]. The motivation in this research is driven to conduct and propose a solution to tackle these challenges with an efficient, practical, and robust solution that encourages developers for real time anomaly detection in context of avoiding unnecessary and repeated number of API calls without the need of republishing the entire mobile app all over again. Real-time anomaly detection in mobile applications helps in preventing resource wastage. This improves the reliability of apps and user experience. This research helps in filling the gap in existing detection methods which have limitations due to static configurations. Developments in AI and machine learning enable people to precisely analyze and interpret API usage patterns. Creating a framework that can detect anomalies helps in identifying anomalies. This helps in reducing operational expenditures, enhancing the quality of applications, and aiding developers in improving the efficiency of apps.

The humid air hung heavy, thick with the scent of blooming jasmine and the distant, rhythmic calls of azaan echoing across the sprawling city. Sunlight, already intense for mid-morning in Karachi, beat down on the bustling streets, where colorful rickshaws weaved through a symphony of honking cars and chattering vendors. Life unfolded in a vibrant tapestry of sights, sounds, and smells – the clatter of metal from a nearby workshop, the sweet aroma of freshly brewed chai wafting from a roadside stall, and the animated conversations of people going about their daily lives. The energy was palpable, a constant hum of activity that defined the very essence of this sprawling metropolis nestled by the Arabian Sea [1].

Later, as the sun began its descent, painting the sky in hues of fiery orange and soft lavender, a gentle breeze finally offered a moment of respite. Families gathered in parks, their laughter mingling with the calls of street food vendors selling spicy delights. The day's intensity softened into a more relaxed pace, a time for reflection and connection. The city lights began to twinkle, mirroring the emerging stars in the darkening sky, casting a magical glow over the familiar landscape. Even in the heart of this bustling urban center, there was a quiet beauty to be found in the transition from day to night, a peaceful pause before the city stirred once more [2].

Mobile applications significantly depend on APIs to communicate with servers. The issue is that usually developers are unable to detect unusual or repeated API calls during the development and testing [4]. This leads to extensive load on servers and creates hurdles in application performance. These issues can increase operational expenditures and delay issue detection. This results in more burden on developers for updates or creating the latest releases for mobile apps. Therefore, the problem arises for businesses that are following strict policy frameworks. These overheads of publishing apps, again and again, are considered as policy violations and reputational risks [35]. Current anomaly detection methods function in predefined parameters, or they depend on static rules [45]. Due to this reason, they lack in functioning in dynamic mobile environments. This has become a major concern. Developers need to identify API-level anomalies precisely. Therefore, considering this issue, people working in this domain need a method that learns and adjusts to API usage patterns to minimize operational expenditures. This aids in enhancing the performance and reliability of mobile applications.

The ideal objective for this research is to introduce a framework which detects unusual API calls in a mobile app and reduces it within the application from backend. This reduces the need of making interventions within the code from the front-end layer of a mobile application and avoids resubmitting of the application from all over again. This saves us a lot of time and resolves operational anomalies within a mobile application. We aim to make use of machine learning algorithms that help us trace patterns regarding how APIs are called and recognize deviations between usual and unusual calls. Using these algorithms within our framework helps us address the evolving threats in mobile app environments in context of taking care of unusual API calls within our mobile applications. This helps us reduce the operational expenditures for developers which eventually takes away the need to resubmit the apps on Android and iOS stores and minimize the frequent updates required for mobile apps. Ideally the framework also goes on to improve the overall performance in a mobile app as the interaction between both client and server is optimized. This improves the user experience that is a very critical thing in improving the ratings for any mobile application or business. Addressing unusual API calls also helps in identifying security threats. This research guides in the application of the framework, and it gives a road map by testing it on mobile applications for iOS and Android platforms. The results aid developers in comprehending the efficiency and scalability of the framework.

1. How can real-time anomaly detection in mobile applications be achieved without depending on static, predefined rules or event triggers?
2. What mechanisms can detect unusual API call patterns in mobile applications, and how can they help in detecting operational or security anomalies?
3. How can an anomaly detection framework reduce the operational overhead and time required for debugging or republishing mobile apps?
4. What impact does a mobile app-specific anomaly detection framework have on enhancing user experience and app reliability?

2. Literature Review

Anomaly detection has been a significant area of research in cyber-security and system reliability. Various methods have been introduced, with the primary objective of identifying malware with the help of signature-based and behavior-based methods. However, these techniques often fail to find unknown or evolving threats. Recent developments in artificial intelligence and machine learning have revealed more adaptive detection mechanisms [41]. This section emphasizes existing researches that highlight gaps in detection speed, granularity, and practical application. The analysis creates a foundation that helps understand the importance of enhancing real-time anomaly detection in mobile applications.

Research by Yang et al. (2023) has analyzed security concerns in mobile applications. This research focuses on the issues in super apps that facilitate mini apps. Their work examined security risks, trade-offs, and mechanisms n OS-like mobile apps. They revealed thirteen security methods and ten threats, including unauthorized API, data leakage, and privilege escalation. The research revealed that the problem arises when mini-apps rely on the APIs of super apps to enhance operations. This leads to risks such as weak permission controls and hidden API access. Security issues were connected to poor API controls, inconsistent platform rules, and weak vetting systems. The research proposed to adopt security practices such as API isolation, token authentication and role-based access to address these issues. Though maintaining usability and security is a significant challenge because limitations can impact application performance. It further proposed that AI-based systems can play a vital role in enhancing security. As it helps in identifying unusual API behavior. This eliminates the reliance on fixed rules and helps in detecting threats precisely. [49] Zaitseva et al. (2023) in his research examined the significance of mobile app security and its impact on users and businesses.

Utilizing the OWASP Mobile Top 10 Risks, they came to know about critical problems such as weak authentication, insecure data storage, and poor platform use that cause critical risks like unauthorized access, data breaches, reputation damage, and financial loss. The research revealed that threats like reverse engineering and weak encryption have a significant effect. Because these factors can result in various threats. They also suggested an approach that helps in assessing the impact of risks according to the relation of these factors. Conventional security approaches depend on fix rules, which may make it difficult to detect threats. They proposed that the utilization of AI-based solutions, such as anomaly detection can play a vital role in enhancing security by identifying unusual API behaviors precisely. This method emphasizes the need to utilize AI-based systems to tackle mobile app risks. [27] Cinar and Kara (2023) analyzed mobile security threats and classified malware detection into two groups such as machine learning-based and signature-based. Signature- based approaches are those that utilize known patterns to identify risks, but they aren't capable of tackling evolving or new malware. Machine learning approaches examine app behavior to enhance their capabilities to deal with unusual API activity. This research revealed that there is a significant increase in adware, malware and riskware in mobile apps. Conventional approaches such as encryption controls and permission controls are not effective. Risks such as social engineering, pishing, and man-in- the-middle (MITM) attacks still dodge the inefficiencies in mobile systems. They came to the conclusion that machine learning models can play a vital role in catching issues by examining patterns in network traffic and app behavior. The study emphasizes that by leveraging AI-based tools to enhance anomaly detection to tackle new threats and minimize the reliance on fixed security rules. These outcomes emphasize the significance of utilizing automated systems to identify API misuse and enhance mobile apps. Namrud et al. (2021), in their research, suggested a deep learning-based Android anomaly detection model. They have utilized various vulnerability datasets. They incorporated features from code smells, dangerous permissions, and AndroBugs vulnerabilities to improve detection performance. They have utilized support vector machines (SVM) and deep neural networks (DNN) to categorize applications as malicious or benign. This approach enhanced the accuracy significantly. Many studies concentrated solely on permissions, which leads to high false positive rates.

The inclusion of AndroBugs vulnerabilities in Namrud et al.'s research focused on these limitations by integrating security-sensitive defects identified through static analysis. The effectiveness of deep learning in malware detection has been analyzed in various research, with convolutional and recurrent neural networks demonstrating promising results. However, complexities such as model interpretability and adversarial attacks remain. Future research needs to discover hybrid techniques integrating runtime behavioral analysis to further enhance detection accuracy and resilience against evolving threats. In mobile apps, user interaction is a key component that sticks a user to keep using the mobile app, and if UI/UX is not seamless, the interest of a user fades away quickly [50]. According to Wikipedia in mobile apps, user experience (UX) and user interface (UI) design are critical factors that help in retaining user engagement. Notably, forty six percent of mobile app users have reported discontinuing use or uninstalling an app due to poor performance. Furthermore, approximately eleven percent of users have uninstalled apps due to complex interfaces, which shows the necessity for intuitive navigation. Additionally, around nineteen percent of users have removed apps because of frequent push notifications, highlighting the significance of thoughtful notification strategies. Moreover, about nineteen percent of users have uninstalled apps because of issues like app hang-ups, underscoring the need for responsive and reliable performance [49] [34]. Findings also tell us that AI can be used optimally to incorporate adaptive designs to mobile apps that are compatible with different screen sizes. [12] However, it is still hard to identify the key sources that lead to user frustration while using a mobile app for mobile app developers [10].

Although, developers are aware of some important factors that can be kept under consideration for detecting users' reasoning to disengage with a mobile app, such as slowness in loading times, complex user interface, and technical glitches factors like diverse preferences, expectations, and the patterns that a user might carry out make it difficult to assess their needs to disengage with mobile apps [1] [34]. Trials are yet being carried out by developers with behavioral data to make identification of such assessments in UI/UX flaws. Behavioral data such as click patterns, usage frequency, and flows in navigation are some parameters that can help identify the early frustrations of a user while using apps. Machine Learning approaches such as supervised learning, unsupervised learning, and semi supervised learning can also be used to detect user frustrations during their interactions with a mobile app, however, all include shortfalls of their own. Supervised learning engages the training of a model on labeled data where the input of a user and estimated output (whether the output is normal or abnormal) are already placed in the system. Whereas an unsupervised requires no labeled data and rather solely depends on the user's input without having any prior information on how the output is likely to be, such as a network intrusion detection system that uses unsupervised learning approaches to detect unusual network traffic patterns without using prior knowledge to assess malicious events.

Finally, we have a semi-supervised learning approach, which lies somewhere in the middle of both the before mention approaches, i.e., supervised and unsupervised, and requires some labeled data in combination with a large amount of unlabeled data in training a model [1]. In contrast with front-end-oriented solutions that include machine learning approaches to optimize user interactions, processes are also being carried out to optimize client-server interactions by incorporating ML and non-ML-based approaches in optimizing API calls [47] [15]. Seeker is one system that uses Explainable AI to optimize this client-server interaction. It uses individual instruction calls and makes predictions for the potential exploits that might occur. These predictions are updated time and again on a run-time basis in databases in state columns, keeping signatures that indicate some meaningful alerts, and once these alarms are triggered, logs are recorded. Developers have utilized these logs, and according to these alerts, they decide the necessary actions to be taken. The key objective of this approach is to find exploits efficiently and minimize their existence. This helps developers to make changes in the code manually and releases the latest updates or deployments frequently.

a. Comparison Table

Reference	Year	Technique	Previous progress (last 5 years)	Proposed framework contribution	Gap Filled
[38]	2021	Focus of Detection	Primarily concentrated on identifying malware through signature-based or behavioral detection techniques.	Introduces real time monitoring to detect unusual API calls that may indicate anomalies in app behavior.	Enhances detection capabilities beyond malware to identify broader application anomalies, strengthening security and reliability.
[3]	2020	Detection Granularity	Primarily focused on the application or process level,	Functions at the API call level,	Improves precision by examining

			such as detecting malicious apps.	allowing for precise anomaly detection.	application behavior at the API call level, enabling early detection of anomalies.
[45]	2024	Speed of Detection	Depended on batch processing or scheduled scans to detect anomalies or malware.	Utilizes AI-driven real-time monitoring to identify and flag unusual patterns as they emerge.	Improves speed and responsiveness by allowing instant identification and resolution of anomalies.
[15]	2021	Practical Application	Emphasized system security while largely overlooking operational anomalies and application performance.	Identifies operational anomalies, such as unexpected API calls, that may indicate bugs, misconfigurations, or potential exploits.	Expands anomaly detection to encompass operational and performance issues, offering a more comprehensive solution.
[43]	2019	Mobile App Specific Focus	Existing solutions were not specifically designed to meet the unique requirements of mobile app ecosystems.	Specifically designed for mobile apps, allowing detection of anomalies in API calls unique to mobile environments.	Minimizes the need for app republishing by resolving issues dynamically, improving development flexibility and user experience.

Table – 1 Comparison Table

3. Methodology

a. Anomaly Detection Framework to Address Unusual API Calls in Mobile Apps

The increasing utilization of mobile applications has resulted in a significant amount of API calls. This should be analyzed for unusual patterns. This helps in addressing critical issues, security breaches and misuse. Anomaly detection is vital in finding these unusual API calls. This may result in poor performance, malicious activities, or bugs. Developers can employ the framework that helps in detecting API calls. Here, we discuss the proposed anomaly detection framework. This is prepared by utilizing various machine learning techniques that run in the backend setting to find anomalies [41]. These techniques have their own weaknesses and strengths. Following are the four highly effective algorithms:

1. Isolation Forest
2. Histogram-based Outlier Score (HBOS)
3. One-Class Support Vector Machine (OCSVM)
4. Local Outlier Factor
5. Autoencoders (Deep Learning Approach)

Above mentioned algorithms have been picked based on their capabilities to detect outliers efficiently, control high-dimensional data, and their computational practicability to work in mobile app environments. Here, we will explore each algorithm in detail, including their

limitations, benefits, and mathematical underpinnings that help in mobile app anomaly detection.

b. Isolation Forest

Mathematical Foundation

The Isolation Forest algorithm is based on the approach that anomalies are less and differ from normal observations. This model utilizes various random splits on the data which help in isolating the observations. Isolation Forest is an enhanced version of the basic Isolation Forest model that efficiently manages high-dimensional data. Isolation Forest creates various sets of decision trees. Randomly opting features and randomly splitting them helps in forming each tree. This randomness allows faster isolation of anomalies, as anomalies tend to require fewer splits [13] [32]. The anomaly score for a point is estimated by averaging the number of splits (i.e., path length) across all trees. The shorter the path length, the more anomalous the point is considered. The path length $h(x)h(x)h(x)$ of an instance xxx is calculated as:

$$h(x) = \frac{1}{N}\sum_{i=1}^{N} h_i(x)$$

Where $hi(x)hi(x)hi(x)$ is the path length of instance xxx in the $iii - th$ tree, and nnn is the total number of trees.

c. Histogram-based Outlier Score (HBOS)

Mathematical Foundation

The Histogram-based Outlier Score (HBOS) algorithm is an efficient and fast technique for anomaly detection that functions by assuming the distribution of each feature is independent. This assumption helps in preparing histograms feature to detect unusual observations. Analyzing the intensity of the data points in the histogram bins help in calculating the score [22] [8]. A low density parallels to an anomaly. The outlier score $s(x)s(x)s(x)$ of a point xxx is assessed as:

$$s(x) = \prod_{i=1}^{n}\left[\frac{1}{p(X_i)}\right]$$

Where: $xixixi$ is the $iii - th$ feature of point xxx, $P(xi)P(xi)P(xi)$ is the probability density of the $iii - th$ feature based on its histogram. The densities in accordance with all features result in the score that reveals the probability of anomaly. Points having lower scores are considered as anomalies.

d. One-Class Support Vector Machine (OCSVM)

Mathematical Foundation

The One-Class Support Vector Machine (OCSVM) algorithm is a highly effective method that helps in anomaly detection. This helps in searching for a decision boundary that splits a significant amount of data in a high-dimensional feature space. This idea came from the Support Vector Machine (SVM), which finds an optimal hyperplane by typically separating two classes [21] [36]. In OCSVM, however, we only manage one class of data (normal points), and the aim is to classify points that lie outside the learned boundary. OCSVM is prepared as an optimization problem with the objective of maximizing the margin between the normal data and the origin. The objective function is:

$$\min(\omega, b, \xi)\frac{1}{2}|\omega|^2 + C\sum_{i=1}^{n}\xi i$$

Where: ω is the hyperplane normal vector, ξi is the slack variable for each data point (penalizing points inside the margin), C is the regularization parameter that manages the trade-off between

margin width and the number of misclassifications.

e. Local Outlier Factor (Density Based Approach)

Mathematical Foundation

The Local Outlier Factor is an example of an unsupervised machine learning approach – in fact an unsupervised version of an anomaly detection algorithm. What it does is, it identifies anomalies while comparing the local density of a data point in its neighbors and assumes that the anomalies lie in the lower density regions in comparison of its neighbors [28] [24]. Let: k be the number of neighbors $N_k(p)$ be the set of k- nearest neighbors of point p, $d(p,o)$ be the distance between point p and its neighbor o, reach $-$ $dist\,k(p,o) = \max(d(o), d(p,o))$ be the reachability distance (to avoid overly small distances), $lrd_k(p)$ be the local reachability density of point p, defined as the inverse of the average reachability distance from p to its neighbors:

$$lrd_k(p) = \left(\frac{1}{|N_k(p)|} \sum_{o \in N_k(p)} reach - dist_k(p,o) \right)^{-1}$$

Then, the LOF score of a point p is given by:

$$LOF_k(p) = \frac{1}{|N_k(p)|} \sum_{o \in N_k(p)} \frac{lrd_k(o)}{lrd_k(p)}$$

If $LOF_k(p) \approx 1$, point p has a similar density to its neighbors and is likely normal.
If $LOF_k(p) > 1$, point p has lower density than its neighbors, indicating a potential outlier.

f. Autoencoders (Deep Learning Approach)

Mathematical Foundation

[6] [7] Autoencoders are an artificial neural network that learns to encode data into a compressed representation and then decode it again to the original input. The architecture generally comprises an encoder network that condenses the dimensionality of the data and a decoder network that reconstructs the data from the compressed representation. In anomaly detection, autoencoders are prepared by training them on normal data, and anomalies are identified according to the reconstruction error. Mathematically, an autoencoder effort to diminish the reconstruction loss

$$L(x, \hat{x}) = L\left(g_\phi(f_\theta(x))\right)$$

where x is the input, and $g_\phi(f_\theta(x))$ is the reconstructed output:

$$L(x, \hat{x}) = |x - \hat{x}|^2$$

During the reconstruction, normal data points tend to have low reconstruction errors, while anomalies, being significantly different from the normal data, result in high reconstruction errors.

g. Comparison of Algorithms

Algorithm	Computational Efficiency	Handling of high dimensional data	Scalability	Sensitivity to Noise	Ease of Interpretation	Training Time	Reference
Isolation Forest	Fast	Good	High	Moderate	High	Low	[32][13]
Histogram Based Outlier	Very Fast	Good	Moderate	High	Low	Very Low	[8][22]
One class Support Vector Machine	Moderate	Excellent	Moderate	Low	Moderate	High	[21][36]
Local Outlier Factor	Moderate	Poor	Low	High	Moderate	High	[24][28]
Autoencoders	Slow	Excellent	Very High	Low	Low	Very High	[7][6]

Table – 2 Comparison of Algorithms

4. Data Collection, Pre-processing and Methodology

a. Data Collection

This study adopts a systematic approach to detect anomalies in API performance using unsupervised machine learning techniques. The methodology is centered on analyzing historical API log data to identify abnormal patterns based on key parameters such as timestamp, response time, HTTP method, status code, error rate, and payload size . The overall framework integrates data collection, preprocessing, model training, and real-time anomaly detection within a unified system. The first phase involves data collection from a MySQL database, specifically from the api_logs table. This dataset contains detailed records of API interactions, including temporal and performance-related attributes. To facilitate numerical computation and enable time-based pattern analysis, timestamps are converted into epoch format. This transformation ensures consistency and compatibility with machine learning algorithms.

In the preprocessing stage, the collected data is standardized using the StandardScaler technique. This step is essential to normalize the feature values and eliminate scale-related biases among different attributes. By ensuring that all features contribute equally to the learning process, the model performance is significantly improved. The processed data is then structured into a suitable format for training unsupervised learning models. To enhance the robustness and accuracy of anomaly detection, multiple machine learning models are employed, including Isolation Forest, Histogram-Based Outlier Score (HBOS), One-Class Support Vector Machine (OCSVM), Local Outlier Factor (LOF), and Autoencoders. Isolation Forest operates by randomly partitioning the data and identifying anomalies based on how easily they can be isolated. HBOS evaluates anomalies using feature-wise histograms and identifies rare occurrences as outliers. OCSVM learns the boundary of normal data and detects deviations outside this boundary. LOF measures the local density deviation of data points compared to their neighbors. Autoencoders, as a deep learning approach, reconstruct input data and identify anomalies based on reconstruction error.

During the anomaly detection phase, incoming API requests are processed in real time. The input parameters are first normalized using the same scaler applied during training to maintain

consistency. The trained models then evaluate the input data and generate anomaly scores or predictions. Based on predefined thresholds, each request is classified as either normal or anomalous. This classification enables the system to identify irregular API behavior effectively. The system is implemented using a Flask-based backend, which exposes multiple API endpoints such as /check_anomaly, /check_anomaly_hbos, /check_anomaly_lof, and /check_anomaly_autoencoder for real-time anomaly detection. These endpoints accept JSON inputs and return anomaly results in a structured format. Furthermore, a Flutter-based mobile application is integrated with the backend to monitor API behavior. The application provides a user interface for visualizing API activity and detecting anomalies, particularly in scenarios where multiple APIs operate concurrently.

In conclusion, the proposed methodology leverages a combination of statistical, machine learning, and deep learning approaches to ensure accurate and scalable anomaly detection. By integrating multiple models and real-time processing capabilities, the system provides an effective solution for monitoring and improving API performance in dynamic environments

Following is a ProcessFlow diagram for the overall application system bind with the anomaly detection framework:

b. Process flow Diagram for the Overall Application System

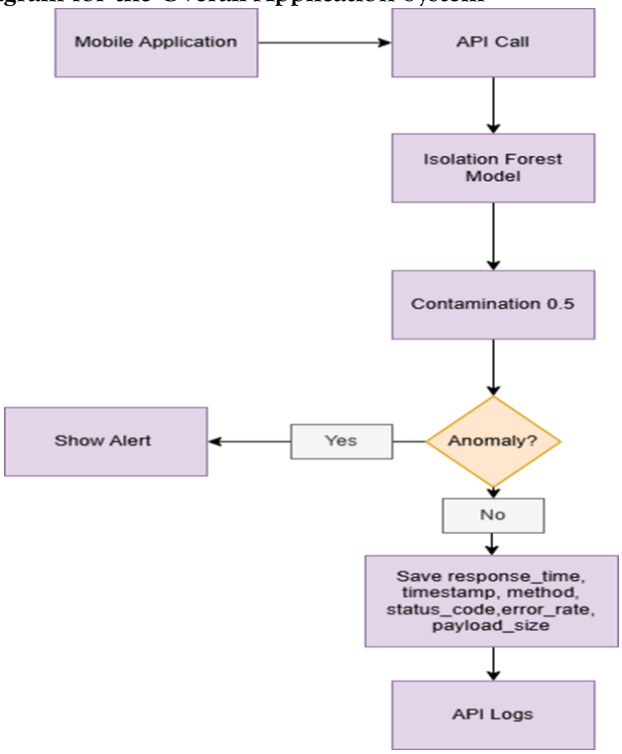

Figure 4.2 - Processflow Diagram

5. Results and Discussions

This section offers the result section of our overall research where we discuss both the individual results we have extracted with each approach applied in our research, along with the comparative analysis that we perform by the end and decide the most optimal ML approach while detecting anomalies in context of API calling, statistically. Through comparative analysis, we assess how our research contributes to optimizing API callings in mobile apps while interacting with the servers and uses machine learning approaches to reduce the overhead of unnecessary publishing mobile application deployments may bring. We

delve into evaluating the performance metrics of these anomaly detection models and present the confusion matrix and classification result count.

a. Performance Metrics Evaluation

Performance metrics evaluation is essential if we tend to gauge the predictive capabilities of our models. We use this evaluation process to assess the efficacy of our models by inserting a chunk of sample dataset that we have extracted from a resource platform such as Keggle. Key metrics chosen to evaluate include the F1-measure, Precision, Accuracy and Recall, where these metrics work as pivotal indicators to assessing the nuanced details inside our models, analyzing the predictive capabilities in our models,
highlighting areas where improvement is needed. The metric serves a crucial role in evaluating the performance of our models across different thresholds which is essential in deciding its predictive utility.

$$\boldsymbol{Accuracy} = \frac{\boldsymbol{TP + TN}}{\boldsymbol{TP + TN + FP + FN}}$$

Metric Accuracy measures the percentage of correct predictions relative to the total predictions made, which goes on to provide a foundational assessment over the predictive capability of the model performance. Precision defines how many anomalies flagged by the system are the real anomalies, whereas Recall highlights how many of the real anomalies your system got detected. While F1-Score elaborates the balance between the two – it is a harmonic mean between Precision and Recall and provides a balance measure when there is a trade-off between the two. F1-Score plays a useful role when you have an imbalanced dataset.

$$\boldsymbol{F1 - score} = \frac{\boldsymbol{2 \cdot (precision \cdot recall)}}{\boldsymbol{precision + recall}}$$

Where:
TP (True Positives) represents the instances correctly predicted as a certain class.
TN (True Negatives) represents the instances correctly predicted as not belonging to a certain class.
FP (False Positives) represents the instances incorrectly predicted as belonging to a certain class.
FN (False Negatives) represents the instances incorrectly predicted as not belonging to a certain class.

b. Confusion Matrix

The confusion matrix is a tool for predictive analysis in machine learning. In order to check the performance of a classification based machine learning model, the confusion matrix is deployed [5]. Also we can say confusion matrix is a summarized table of the number of correct and incorrect predictions yielded by a classifier (or a classification model) for binary classification tasks. A confusion matrix is a N x N matrix used for evaluating the performance of a classification model, where N is the number of target classes. By visualizing the confusion matrix, an individual could determine the accuracy of the model by observing the diagonal values for measuring the number of accurate classification. The confusion matrix is in the form of a square matrix where the column represents the actual values and the row depicts the predicted values of the model and vice versa.
TP: True Positive: The actual value was positive and the model predicted a positive value
FP: False Positive: Your prediction is positive, and it is false. (Also known as the type 1 error)
FN: False Negative: Your prediction os negative, and result it is also false. (Also known as the type 2 error)
TN: True Negative: The actual value was negative and the model predicted a negative value.
Accuracy:
Accuracy is a measure for how many correct predictions your model made for the complete test dataset. Accuracy is a good basic metric to measure

the performance of the model. In unbalanced datasets, accuracy becomes a poor metric.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Misclassification:

$$Misclassification\ Rate = \frac{FP + FN}{TP + TN + FP + FN}$$

Precison:

Precison tells us how many of the correctly predicted case actually turned out to be positive. This would determine whether our model is reliable or not. Presison is a useful metric in case where False Positive is a higher concern than False Negative.

$$\text{Pr}\,ecision = \frac{TP}{TP + FP}$$

Recall:

Recall tells us how many of the actual positive cases we were able to predict correctly with our model. Recall is a useful metric in cases where False Negative trumps False positive.

$$Recall = \frac{TP}{TP + FN}$$

F1-Score:

When we try to increase the precision of model, the recall grows down and vice versa. F1-Score is a harmonic mean of precison and recall and so it gives a combined idea about these two metrics. It is maximum when precision is equal to recall.

$$F1 - score = \frac{1}{\left(\frac{1}{recall} + \frac{1}{precision}\right)}$$

c. Prediction results and recommendations

The dataset used for this research was compiled from multiple sources available on Kagle and adapted to meet the specific requirements of this study. We collected datasets related to API logs and extracted the following key features: timestamp, method, endpoint, status code, response time, request volume, error rate, source identifier (IP address), payload size, and event frequency. However, in the current phase of the study, we focused on training and evaluating our models using only six features: timestamp, response time, method, status code, error rate and payload size. The primary objective of this research was to enhance the security of mobile applications by detecting anomalous behavior during API calls. To achieve this, we implemented and tested five different anomaly detection models to identify unusual patterns in API log data. Model performance was evaluated using confusion matrices and standard classification metrics such as precision, recall, and F1 Score. For development and testing, a Python virtual environment was created using Flask to manage dependencies efficiently. The preprocessed log data was stored in a MySQL database and handled using Pandas DataFrames. We developed a dedicated API endpoint to allow real-time interaction between the mobile application and the anomaly detection service. To assess the effectiveness of the models, we used 5000 test cases, comprising:

- 40% normal (non-anomalous) API calls expected to return “no anomaly”
- 60% anomalous calls expected to return “yes anomaly”
  - Among these, 50% were subtle anomalies, designed to be close to decision boundaries and challenging to detect.
  - The remaining 50% were clear anomalies, presenting significant deviations in values.

This testing range was selected to provide a balanced and comprehensive evaluation of detection accuracy and to effectively identify false

positives and false negatives. It also allowed us to compare the performance of different models across various metrics, including detection accuracy, precision, recall, F1-score, and runtime efficiency. The outcomes of these tests for each model are summarized below:

d. Extracted Results using Isolation Forest

The results have been extracted with the engagement of over 110,000 datasets stored in the database with up to 5,000 test cases applied to it. However, when the processes were executed we experienced biased results in diagonal form. The amount of correct anomalies that were detected included up to 4,500 that reflected true negative API calls while the true positive showed 500, whereas both false negative and false positive had 0s in a diagonal format. This is a binary problem in our scenario where we encounter biased results, having true negatives outnumbering the true positives as the majority ones. In order to solve this, we applied the SMOTE technique to perform oversampling and under-sampling, after which, we had the confusion matrix for Isolation Forest as the following:

Confusion Matrix with Undersampling

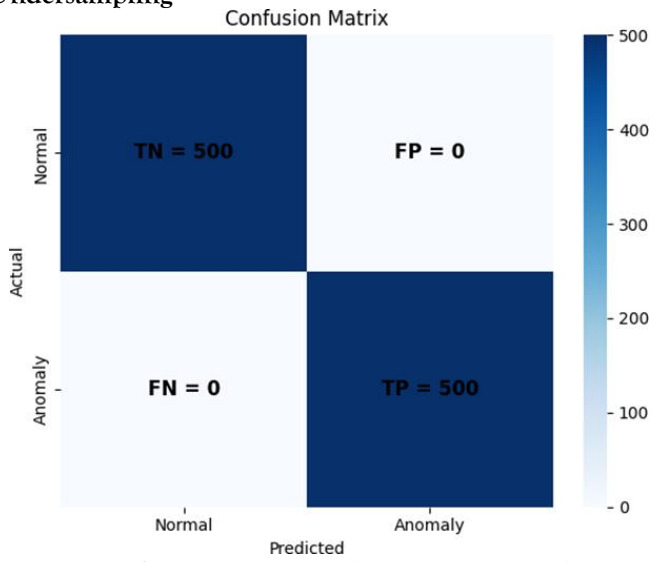

Figure 5.1: Confusion MatriX Isolation Forest (Undersampling)

Confusion Matrix with Oversampling

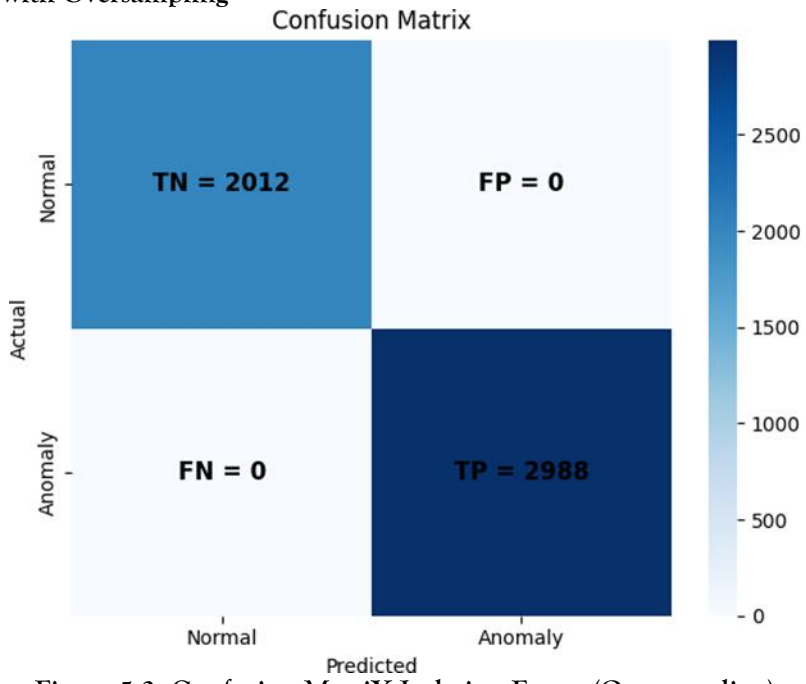

Figure 5.2: Confusion MatriX Isolation Forest (Oversampling)

The preference is, however, given to the oversampling results, and as we go further in this research, we discuss or consider the oversample example as part of our optimal gained results.

We implemented the isolation forest algorithm to detect anomalies in API log data using features such as timestamp, response_time, method, status_code, error_rate, payload_size. The model was integrated into Flask API that accepts real-time input and responds with an anomaly prediction. Prior to training, the data was normalized using StandardScaler to standardize the feature space, ensuring that all features contribute equally to the anomaly detection.

The model was evaluated using labeled test data, and the following confusion matrix metrics were obtained:

Confusion Matrix Summary

True negatives: 2012 Correctly predicted normal cases

True Positive: 2988 Correctly predicted anomalies

False Positive: 0 Noraml instances incorrectly predicted as anomalies

False Negative: 0 Anomalies incorrectly predicted as normal

Performance Metrics

Using the values above, we derive the following performance metrics:

- Accuracy

$$\frac{TP+TN}{TP+TN+FP+FN} = \frac{2988+2012}{2988+2012+0+0} = \mathbf{1.0} = \mathbf{100}\ \%$$

- Precision

$$\frac{TP}{TP+FP} = \frac{2988}{2988+0} = \mathbf{1}$$

- Recall

$$\frac{TP}{TP+FN} = \frac{2988}{2988+0} = \mathbf{1.0}$$

- F1 Score

$$\frac{2\cdot(precision\cdot recall)}{precision+recall}=\frac{2\cdot(1.0\cdot 1.0)}{1.0+1.0}=1.0$$

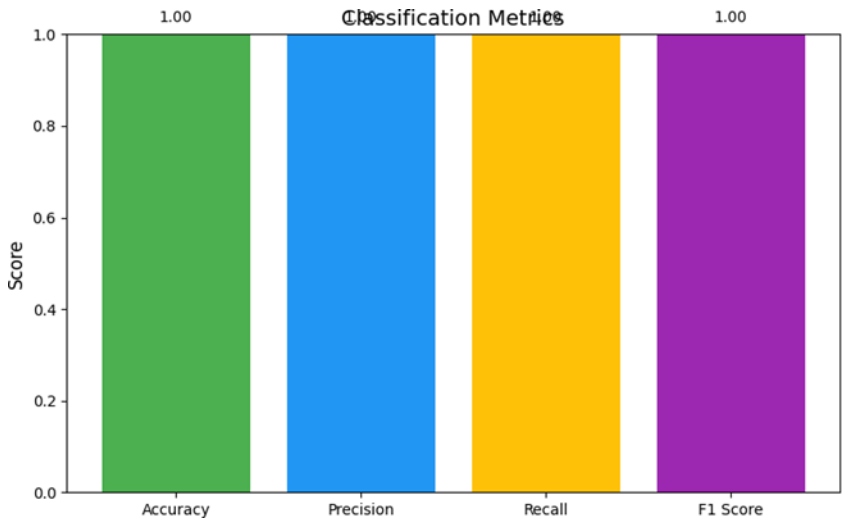

Figure 5.3: Classification Isolation Forest

Interpretation

The Isolation Forest model demonstrated perfect performance across all key metrics: precision, recall, F1-score, and accuracy were all 100%. This indicates that the model correctly identified all anomalies without any false positives or false negatives. Every predicted anomaly was indeed an actual anomaly, and all actual anomalies were successfully detected.

This result suggests that the model is exceptionally well-calibrated for the given data and parameter configuration, achieving both completeness (recall) and exactness (precision) in its anomaly detection. Such an outcome is rare in real-world applications and may indicate an ideal scenario, highly separable data, or a well-tuned contamination parameter.

While this performance is excellent, it's important to validate these results on diverse datasets and in real-time environments to ensure the model maintains this level of effectiveness. In practical settings, achieving a balance between precision and recall remains a central challenge in unsupervised anomaly detection.

e. Extracted Results using Histogram-based Outlier Score

The results have been extracted with the engagement of over 110,000 datasets stored in the database with up to 5,000 test cases applied to it. However, when the processes were executed we experienced biased results in diagonal form, which is a binary problem in our scenario. We applied the SMOTE technique to perform oversampling and undersampling, after which, we had the confusion matrix for Histogram-based Outlier Score as the following:

Confusion Matrix with Undersampling

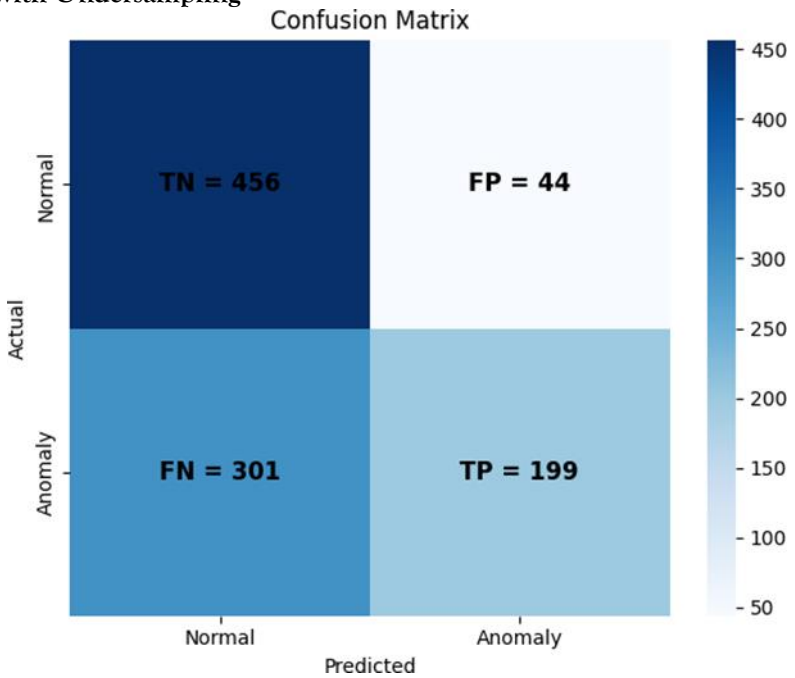

Figure 5.4: Confusion Matrix HBOS (Undersampling)

Confusion Matrix with Oversampling

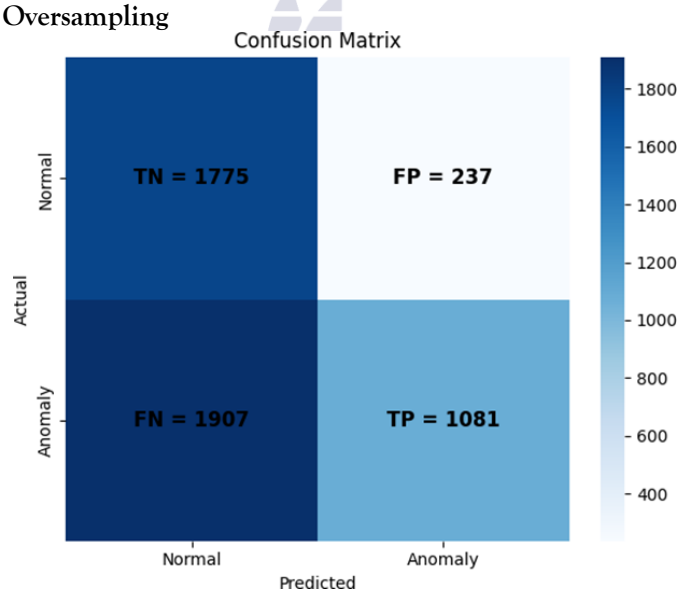

Figure 5.5: Confusion Matrix HBOS (Oversampling)

The preference is, however, given to the oversampling results as discussed earlier.

To evaluate the performance of the Histogram based outlier score (HBOS) model for anomaly detection, we developed a Flask-based API that predicts anomalies in real-time based on timestamp, response_time, method, status_code, error_rate and payload_size features. The model was trained using historical API logs stored in a MySQL database. Input timestamps were convereted into numerical values to ensure compatibility with the HBOS model, which requires numerical input.

We assessed the model's performance by comparing its predictions against labeled data and visualized the results using a confusion matrix.

Confusion Matrix Summary

True Negatives: 1775 instances where "No Anomaly" was correctly predicted.

False Positive: 237 instances where an "Anomaly" was incorrectly predicted for normal behavior.

False Negatives: 1907 instances where an actual anomaly was incorrectly labeled as normal.

True Positives: 1081 instances where "Anomaly" was correctly predicted.

Performance Metrics

Using the values from the confusion matrix, we calculate the following metrics:

- Accuracy

$$\frac{TP+TN}{TP+TN+FP+FN} = \frac{1081+1775}{1081+1775+237+1907} = \mathbf{57.12}\ \%$$

- Precision

$$\frac{TP}{TP+FP} = \frac{1081}{1081+237} = \mathbf{0.8200} = \mathbf{82}\ \%$$

- Recall

$$\frac{TP}{TP+FN} = \frac{1081}{1081+1907} = \mathbf{0.362} = \mathbf{36.20}\ \%$$

- F1-Score

$$\frac{2\cdot(precision\cdot recall)}{precision+recall} = \frac{2\cdot(0.8200\cdot 0.362)}{0.8200+0.362} = \mathbf{0.5017} = (\mathbf{50.17}\ \%)$$

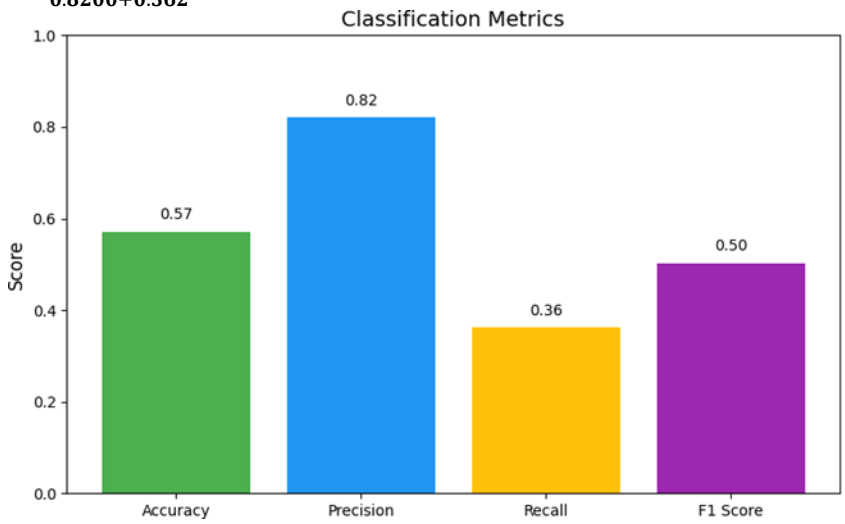

Figure 5.6: Classification Histogram Based Outlier Score

Interpretation

The HBOS model demonstrated high precision (82%), indicating that when the model predicts an anomaly, it is correct most of the time. However, the recall is relatively low (36.2%), meaning a significant number of actual anomalies are being missed. This suggests that while the model is selective and generally avoids false alarms, it fails to detect many true positives. This pattern is characteristic of HBOS, which is valued for its speed and simplicity, but may struggle with capturing more complex or subtle anomaly patterns, particularly when correlations between features exist.

In conclusion, the HBOS model offers a fast and resource-efficient anomaly detection solution suitable for real-time applications, but it may require additional tuning or integration with other models to improve its recall and overall detection performance in more complex environments.

f. Extracted Results using One-Class Support Vector Machine

We couldn't execute the program with One-Class Support Vector Machine using the six parameters which we have used in the other approaches of our system; time_stamp, response_time, method, status_code, payload_size, error_code. The system displayed an unusual behavior of not being to evaluate the process and was halt in between - later it was noted that the time it took to evaluate the process was impractical and could not be set as an optimal timeframe for evaluation in a real-world scenario. Therefore, we reduced the number of parameters to timestamp and response_time and implemented the one class support vector machine (SVM) for detecting anomalies in API responses, yet the results were not quite satisfactory. The model was deployed as a real-time Flask API, capable of receiving input requests and returning binary anomaly decisions.

To ensure feature comparability, the dataset was normalized using StandardScaler. The One-Class SVM model was trained suing the Radial Basis Function (RBF) kernel with nu=0.05 , which controls the expected proportion of outliers in the dataset.

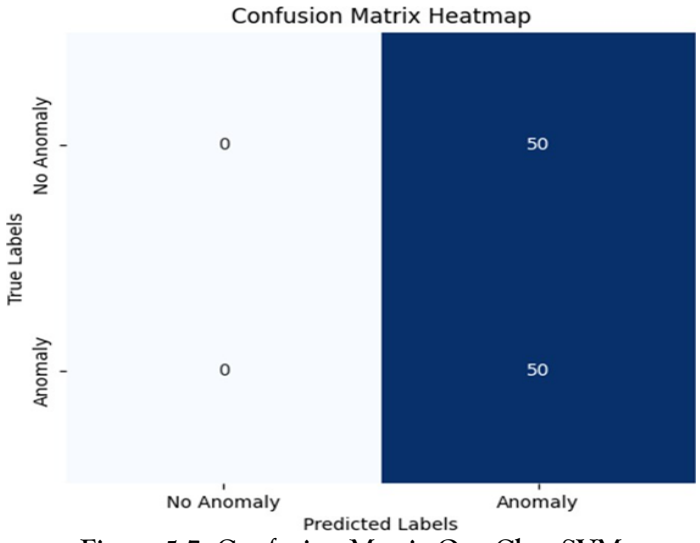

Figure 5.7: Confusion Matrix One-Class SVM

Confusion Matrix Summary

True Positives: 0 actual anomalies correctly identified
True Negatives: 50 actual normal instances correctly identified
False Positives: 0 normal instances misclassified as anomalies
False Negatives: 50 anomalies misclassified as normal

Performance Metrics

From the above results:

- Accuracy

$\frac{TP+TN}{TP+TN+FP+FN} = \frac{0+50}{0+50+0+50} = \mathbf{50}$ %

- Precision

$\frac{TP}{TP+FP} = \frac{0}{0+0}$ = (undefined ⟶ interpreted as 0)

- Recall

$\frac{TP}{TP+FN} = \frac{0}{0+50} = \mathbf{0}$

- F1-score

$\frac{2\cdot(precision \cdot recall)}{precision + recall} = \mathbf{0}$ (since both precision and recall are 0

Interpretation

The One-Class SVM model correctly identified all normal instances (100% true negatives), with no false positives, indicating that it is extremely conservative and biased toward predicting normal behavior. However, it failed to detect any actual anomalies resulting in zero true positives and a recall of 0%.

This outcome suggests a severe imbalance in the model's sensitivity, possible due to:

Overfitting to the normal class, which is expected in one-class models if the boundary is too tight.

An inappropriate choice of hyperparameters such as nu or gamma, or lack of clear separation between normal and anomalous data in the feature space.

Insufficient training data or low variance in feature representation of anomalies. As a result, although One-Class SVM showed promise in avoiding false alarms (false positives), its practical usability is limited in this context due to its inability to identify true anomalies. Further tuning or alternative models are required for real-world deployment.

g. Extracted Results using Local Outlier Factor (LOF)

The results have been extracted with the engagement of over 110,000 datasets stored in the database with up to 5,000 test cases applied to it. However, when the processes were executed we experienced biased results in diagonal form, which is a binary problem in our scenario. We applied the SMOTE technique to perform oversampling and under-sampling, after which, we had the confusion matrix for Local Outlier Factor as the following:

Confusion Matrix with Undersampling

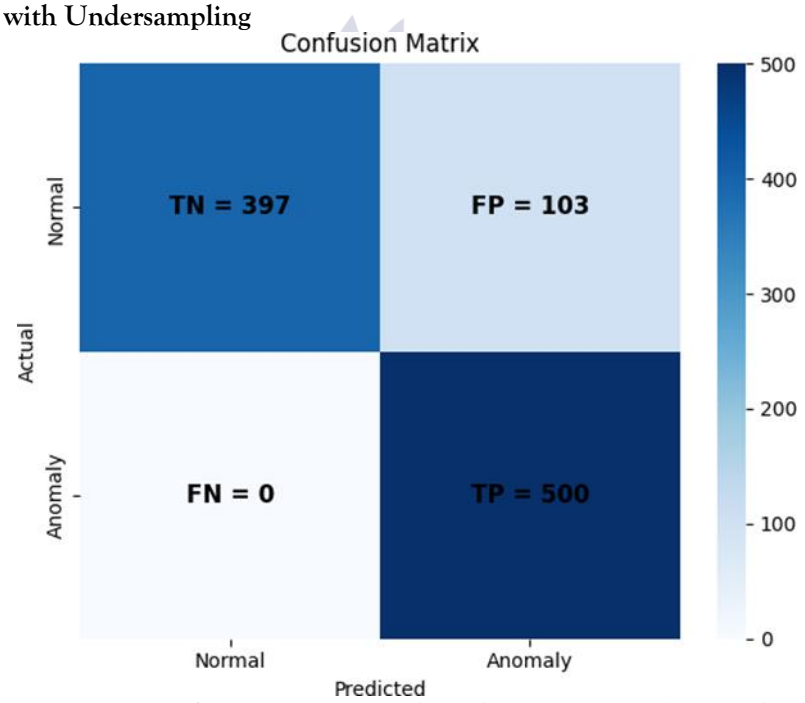

Figure 5.8: Confusion Matrix Local Outlier Factor (Undersampling)

Confusion Matrix with Oversampling

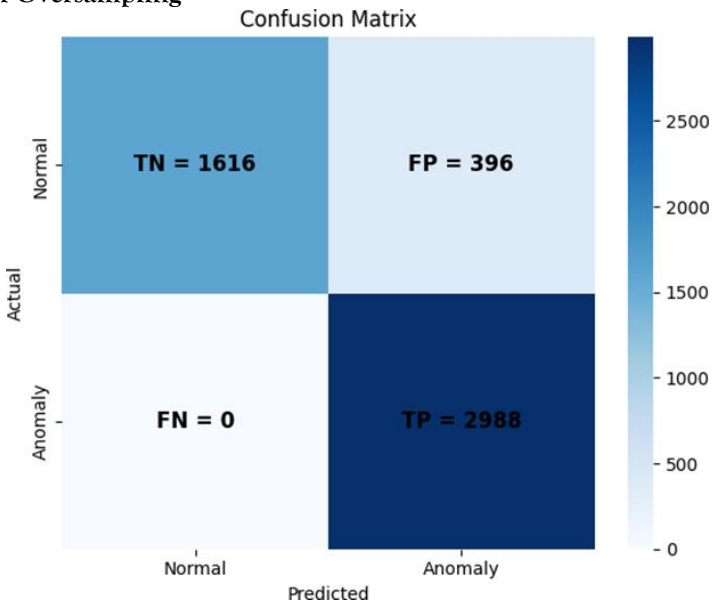

Figure 5.9: Confusion Matrix Local Outlier Factor (Oversampling)

The preference is, however, given to the oversampling results as mentioned earlier. The Local Outlier Factor (LOF) algorithm was implemented to detect anomalies in API behavior by analyzing the local density deviation of a given data point with respect to its neighbors. The model was served via a Flask API and evaluated on normalized input features (timestamp, response_time, method, status_code, error_rate, payload_size) using a StandardScaler.

We used a dynamic n_neighbors value (minimum of 20 or n-1 from the dataset) to ensure model compatibility regardless of dataset size. The contamination rate was set to 5% to reflect the expected proportion of anomalies.

Performance Metrics

- Accuracy

$$\frac{TP+TN}{TP+TN+FP+FN} = \frac{2988+1616}{2988+1616+396+0} = \mathbf{92.08}\ \%$$

- Precision

$$\frac{TP}{TP+FP} = \frac{2988}{2998+396} = \mathbf{88.35}\ \%$$

- Recall

$$\frac{TP}{TP+FN} = \frac{2988}{2988+0} = \mathbf{100}\ \%$$

- F1-score

$$\frac{2\cdot(precision\cdot recall)}{precision+recall} \qquad \frac{2\cdot(0.8835\cdot 1)}{0.8835+1} \qquad \mathbf{93.82}$$

Confusion Matrix Summary

True Positives: 2988 Actual anomalies correctly detected

True Negatives: 1616 Actual normal data correctly identified

False Negatives: 0 Anomalies misclassified as normal

False Positives: 396 Normal instances misclassified as anomalies

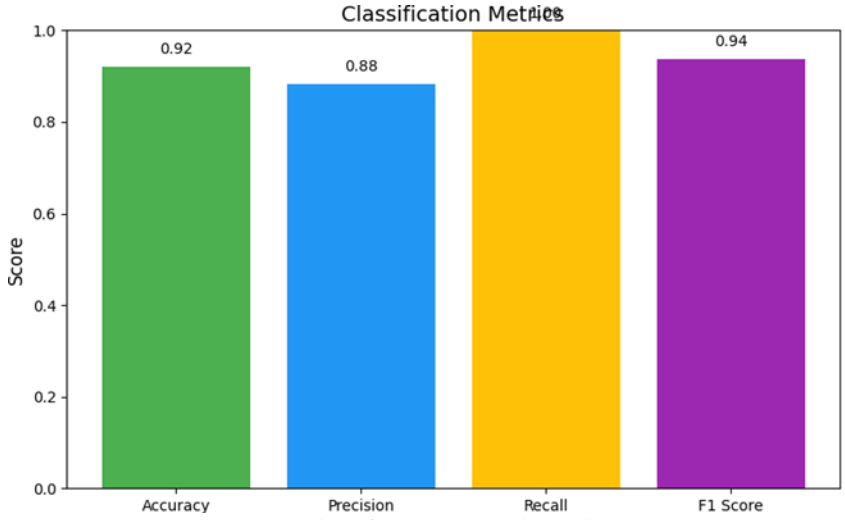

Figure 5.10: Classification Local Outlier Factor

Interpretation

The LOF model demonstrated strong overall performance, achieving 92.08% accuracy, 88.35% precision, 100% recall, and an F1-score of 93.82%. This indicates that the model successfully detected all anomalies (no false negatives) and correctly identified most normal data points, with a relatively small number of false positives.

This performance suggests that the model is highly effective at identifying deviations in local density, which is the core principle behind LOF. Unlike earlier behavior where LOF was overly conservative, the current configuration appears well-tuned, enabling the model to balance sensitivity to anomalies and specificity for normal data.

Several factors may have contributed to this success:

- The feature set (e.g., timestamp, response time) likely contains meaningful local density variations that LOF can exploit.
- Parameter choices like contamination rate and number of neighbors were likely optimal for the data structure.
- The use of a scoring threshold aligned well with the distribution of local outlier factors, minimizing misclassifications.

This outcome demonstrates that LOF can be a viable option for real-time anomaly detection, provided it is carefully calibrated. For continued deployment in dynamic environments, periodic re-evaluation and threshold adjustments are recommended to maintain this level of performance.

h. Extracted Results using Autoencoders

The results have been extracted with the engagement of over 110,000 datasets stored in the database with up to 5,000 test cases applied to it. However, when the processes were executed we experienced biased results in diagonal form, which is a binary problem in our scenario. We applied the SMOTE technique to perform oversampling and undersampling, after which, we had the confusion matrix for Autoencoders as the following:

Confusion Matrix with Undersampling

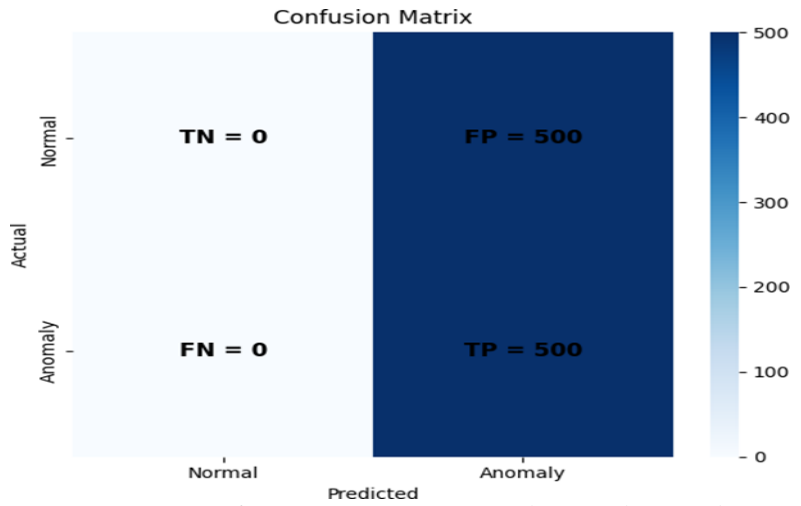

Figure 5.11: Confusion Matrix Autoencoder (Undersampling)

Confusion Matrix with Oversampling

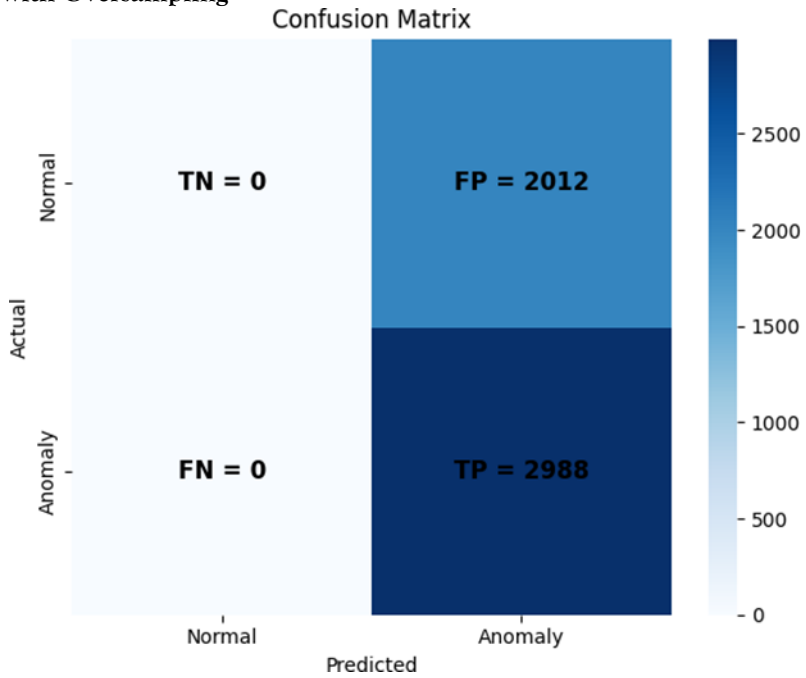

Figure 5.12: Confusion Matrix Autoencoder (Oversampling)

The preference is, however, given to the oversampling results as mentioned earlier. An Autoencoder-based neural network was employed to detect anomalies by learning to reconstruct normal API behavior using only features such as timestamp, response_time, method, status_code, error_rate, payload_size. The model was trained in an unsupervised fashion using historical normal data, and it attempted to reconstruct the input features. Anomalies were identified based on a threshold applied to the reconstruction error.

The architecture consisted of a symmetric neural network with encoding and decoding layers (Dense(8) → Dense(4) → Dense(8) → Dense(2)), trained using Mean Squared Error (MSE) loss over

50 epochs. Data normalization was done using StandardScaler.

Confusion Matrix Summary
True Positives: 2988 Anomalies correctly detected
False Negatives: 0 Anomalies misclassified as normal
False Positives: 2012 Normal data misclassified as anomalies
True Negatives: 0 Normal data correctly classified

Performance Metrics

- Accuracy

$$\frac{TP+TN}{TP+TN+FP+FN} = \frac{2988+0}{2988+0+2012+0} = \mathbf{59.76}\ \%$$

- Precision

$$\frac{TP}{TP+FP} = \frac{2988}{2988+2012} = \mathbf{59.76}\ \%$$

- Recall

$$\frac{TP}{TP+FN} = \frac{2988}{2988+0} = \mathbf{100}\ \%$$

- F1 Score

$$\frac{2\cdot(precision\cdot recall)}{precision+recall} = \frac{2\cdot(0.5976\cdot 1.0)}{0.5976+1.0} = \mathbf{74.8}\ \%$$

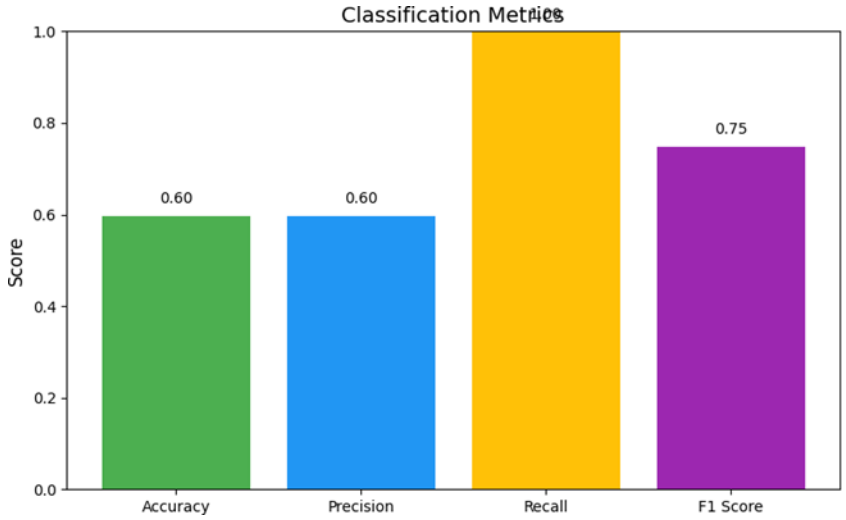

Figure 5.13: Classification Autoencoders

Interpretation

The Autoencoder model successfully detected all actual anomalies in the test dataset, achieving a perfect recall of 100%. However, its precision was relatively low (59.76%), indicating a high rate of false positives. The model labeled many normal samples as anomalies, leading to a low number of true negatives and an overall accuracy of only 59.76%.

This behavior suggests that while the Autoencoder is highly sensitive to anomalous behavior (i.e., it misses none), it lacks specificity and struggles to distinguish between truly anomalous and normal patterns.

Possible reasons for this include:

- Overly sensitive threshold: The reconstruction error threshold may have been set too low, causing the model to classify many normal points as anomalous.
- Feature overlap: The reconstruction error for normal and anomalous inputs might overlap

significantly, especially if the features (e.g., timestamp, response time) are not sufficiently informative or separable.

- Lack of regularization: The model may have overfitted on training data, resulting in a low tolerance for minor variations in unseen normal data.

In conclusion, the Autoencoder model is highly effective at identifying anomalies but prone to over-alerting. For practical deployment, it would benefit from better threshold calibration, feature refinement, and possibly incorporating feedback or labeled samples to improve precision and reduce false alarms.

6. Conclusion and Future Directions

In this study, we evaluated five unsupervised machine learning models: Isolation Forest, Histogram-Based Outlier Score (HBOS), One-Class SVM, Local Outlier Factor (LOF), and Autoencoders for detecting anomalies in API logs using features such as timestamp, response_time, method, status_code, error_rate, and payload_size.

Among all models, Isolation Forest and Autoencoders performed the best. Isolation Forest achieved perfect precision and recall (100%), indicating it correctly identified all anomalies without any false alarms. The Autoencoder model also achieved 100% recall, detecting every anomaly, but suffered from a high false positive rate, leading to a lower precision of 59.76% and overall accuracy of 59.76%. Despite this, its high recall demonstrates strong sensitivity, making it useful in contexts where missing anomalies is costlier than raising false alerts.

HBOS also performed well, achieving a precision of 100% and a recall of 36.15%. It effectively minimized false positives but missed many actual anomalies, suggesting it is conservative in flagging outliers. Its overall accuracy was 56.14%, reflecting the imbalance between correct anomaly and normal case detections.

On the other hand, One-Class SVM and LOF underperformed. One-Class SVM achieved high recall (100%) but at the cost of low precision (59.76%) due to numerous false positives. LOF showed the inverse behavior, with 100% precision but poor recall (0%), failing to detect any true anomalies. These results indicate limitations in their ability to generalize in this anomaly detection setting, especially under default hyperparameters.

Overall, our findings suggest that tree-based and statistical models like Isolation Forest and HBOS are more effective for anomaly detection in structured, time-seriesl ike data involving latency and usage metrics. Distance-based and neural network based approaches, such as LOF, One-Class SVM, and Autoencoders, may require additional tuning (e.g., threshold calibration, parameter optimization, or feature engineering) or hybrid architectures to reach comparable performance in real-time monitoring scenarios.

REFERENCES

Aliyu Abali and Aisha Suleiman. “THE FUTURE OF AI IN MOBILE APP DEVELOPMENT”. In: (2025).

J. S. Alkasassbeh et al. “The Role of AI in Mobile Apps to Combat Future Pandemics: A COVID-19 Case Study,” in: 2023 2nd International Engineering Conference on Electrical, Energy, and Artificial Intelligence (EICEEAI), Zarqa, Jordan, 2023 (2023). DOI: 10.1109/EICEEAI60672.2023.10590538.

IMAN M. ALMOMANI and AALA AL KHAYER. “A Comprehensive Analysis of the Android Permissions System”. In: IEEE Access 8:216671 - 216688 (2020). DOI: 10.1109/ACCESS.2020.3041432.

Chisom Elizabeth Alozie, Olanrewaju Oluwaseun Ajayi, and Naomi Chukwurah. “Comparative Analysis of APIs and Frameworks in Mobile Application Development: Insights from Industry Practices”. In: International Journal of Academic and Applied Research (IJAAR) (2025), pp. 97–105.

Ahmad Farhan Alshammari. “Implementation of Model Evaluation using Confusion Matrix in Python”. In: International Journal of Computer Applications 186(50):42- 48 (2024). DOI: 10.5120/ijca2024924236.

Pierre Baldi. "Autoencoders, Unsupervised Learning, and Deep Architectures". In: JMLR: Workshop and Conference Proceedings 27:37–50, 2012 (2012).

Dibyendu Barman, Abul Hasnat, and Rupam Nag. AN INTRODUCTION TO AUTOENCODERS. Computation (pp.14-23)Edition: III CSE-GCETTB, 2022.

Algirdas Baskys and Nerijus Paulauskas. "Application of Histogram-Based Outlier Scores to Detect Computer Network Anomalies". In: (2019). DOI: 10.3390/ electronics8111251.

Zaki Ali Bayashot. "The Contribution of AI-Powered Mobile Apps to Smart City Ecosystems". In: Journal of Software Engineering and Applications, Vol.17 No.3, 2024 (2024).

Abdelkarim Belkhir et al. "An Observational Study on the State of REST API Uses in Android Mobile Applications". In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft) (2019). DOI: 10.1109/ MOBILESoft.2019.00020.

Tariq Bishtawi and Reem Alzu'bi. "Cyber Security of Mobile Applications Using Artificial Intelligence". In: Conference: 2022 International Engineering Conference on Electrical, Energy, and Artificial Intelligence (EICEEAI) (2022).

Pedro Casas, Pierdomenico Fiadino, and Alessandro D'Alconzo. "When smartphones become the enemy: unveiling mobile apps anomalies through clustering techniques". In: ATC '16: Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges (2016). DOI: 10.1145/2980055.2980058.

Yousra Chabchoub et al. "An In-Depth Study and Improvement of Isolation Forest". In: IEEE Access (2022).

Kanchan Chaudhary and Dr. Shashank Singh. "Different Machine Learning Algorithms used for Secure Software Advance using Software Repositories". In: International Journal of Scientific Research in Computer Science Engineering and Information Technology (2023). DOI: 10.32628/CSEIT2390225.

Md Naseef-Ur-Rahman Chowdhury, Dr. Hamdy Soliman, and Qudrat E Alahy. "Advanced Android Malware Detection Utilizing API Calls and Permissions". In: IT Convergence and Security (pp.123-134). 2021. DOI: 10.1007/978-981-16-4118- 3_12.

Dr.S.V.Manikanthan et al. "Artificial Intelligence Techniques for Enhancing Smartphone Application Development on Mobile Computing". In: International Journal of Interactive Mobile Technologies (2020). DOI: 10.3991/ijim.v14i17.16569.

Peace Emma and Caleb Akanbi. "Enhancing Mobile Security: AI-Driven Threat Detection in Android Applications". In: (2024).

Catherine Fichten et al. "AI-Based and Mobile Apps: Eight Studies Based on PostSecondary Students' Experiences". In: The Journal on Technology and Persons With Disabilities (2022).

Mrs. Pragati Patel Miss. Shivani Gajjar. "A Review: Comparative Analysis of Artificial Intelligence, Machine Learning (ML), and Data Science". In: International Journal of Research Publication and Reviews (2023).

Veeramani Ganesan. "Machine Learning in Mobile Applications". In: International Journal of Computer Science and Mobile Computing (2022). DOI: 10.47760/ijcsmc. 2022.v11i02.013.

Markus Goldstein, Mennatallah Amer, and Slim Abdennadher. "Enhancing one- class Support Vector Machines for unsupervised anomaly detection". In: Conference: Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description (2013). DOI: 10.1145/2500853.2500857.

Markus Goldstein and Andreas Dengel. "Histogram-based Outlier Score (HBOS): A fast Unsupervised Anomaly Detection Algorithm". In: Conference: KI-2012: Poster and Demo Track (2012).

Wehle Hans-Dieter. "Machine Learning, Deep Learning, and AI: What's the Difference?" In: Conference: Data Scientist Innovation Day (2017).

Ánh Hoàng and Toan Nguyen Mau. "A Mass-Based Approach for Local Outlier Detection". In: IEEE Access (2021). DOI: 10.1109/ACCESS.2021.3053072.

Ahmad Al Hwaitat et al. "Overview of Mobile Attack Detection and Prevention Techniques Using Machine Learning". In: International Journal of Interactive Mobile Technologies (iJIM) 18(10):125-157 (2024). DOI: 10.3991/ijim.v18i10.46485.

Varsha Jain and Vijay Viswanathan. "The Usage and Applications of Mobile App". In: Encyclopedia of Mobile Phone Behavior (pp.1242-1255). 2015. DOI: 10.4018/978-1-4666-8239-9.ch100.

Ahmet Cevahir Cinar Turkan Beyza Kara. "The current state and future of mobile security in the light of the recent mobile security threat reports". In: Multimed Tools Appl 82, 20269–20281 (2023) (2023). DOI: 10.1007/s11042-023-14400-6.

Peer Kröger et al. "LOF: Identifying Density-Based Local Outliers". In: Conference: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA. (2000). DOI: 10.1145/342009.335388.

Helen Josephine V L et al. "Enhancing Mobile Application Security Through Android Threat Classification". In: Procedia Computer Science, Volume 252,2025 (2025). DOI: 10.1016/j.procs.2024.12.017..

Yinghua Lia et al. "An Empirical Study of AI Techniques in Mobile Applications". In: arXiv:2212.01635v3 [cs.SE] 27 Sep 2024 (2024).

Ying-Dar Lin et al. "Mobile Application Security". In: IEEE Computer Society (2014). DOI: 0.1109/MC.2014.156.

Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation Forest". In: IEEE Xplore (2009). DOI: 10.1109/ICDM.2008.17.

Bassam Alsanousi Abdulmohsen S. Albesher Hyunsook Do Stephanie Ludi. "Investigating the User Experience and Evaluating Usability Issues in AI-Enabled Learning Mobile Apps: An Analysis of User Reviews". In: (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 14, No. 6, 2023 (2023).

Nagarajan Madhavan. "The Role of Artificial Intelligence in Enhancing Mobile App Accessibility Inclusive User Experience, Digital Inclusion". In: International Journal of Advanced Research 1(2):273-284 (2024).

Ms. Supriya Mandhare et al. "Detection of Threats in Mobile Apps: A Survey". In: INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH TECHNOLOGY (IJERT) ICIATE – 2017 (Volume 5 – Issue 01) (2017).

Larry M. Manevitz and Malik Yousef. "One-Class SVMs for Document Classification". In: Journal of Machine Learning Research 2 (2001) 139-154 (2001).

Milne-Ives et al. "Artificial intelligence and machine learning in mobile apps for mental health: A scoping review". In: PLOS Digit Health. 2022 Aug 15;1(8):e0000079 (2022).

Seif ElDein Mohamed et al. "Detecting Malicious Android Applications Based On API calls and Permissions Using Machine learning Algorithms". In: 2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC) (2021). DOI: 10.1109/MIUCC52538.2021.9447594.

Prathyusha Nama. "AI-Powered Mobile Applications: Revolutionizing User Interaction Through Intelligent Features and Context-Aware Services". In: Journal of Emerging Technologies and Innovative Research (JETIR) (2023).

Ali Rezaei Nasab et al. "Fairness Concerns in App Reviews: A Study on AI-based Mobile Apps". In: arXiv:2401.08097v4 [cs.SE] 31 Jul 2024 (2024). DOI: 10.48550/ arXiv.2401.08097.

Sarfaraz Natha et al. "A Systematic Review of Anomaly detection using Machine and Deep Learning Techniques". In: Quaid-e-Awam University Research Journal of Engineering Science Technology 20(1):83-94 (2022). DOI: 10.52584/QRJ.2001.11.

Ahmed J. Obaid et al. "Integration of artificial intelligence/machine learning in developing and defending web applications". In: UKI Toraja International Conference of Education and Science (UKITOICES) 2021Volume: 2736 (2023). DOI: 10.1063/ 5.0171097.

P. K. Paul and P. S. Aithal. "MOBILE APPLICATIONS SECURITY: AN OVERVIEW AND CURRENT TREND". In: Proceedings of National Conference on Research in Higher Education, Learning and Administration, IQAC 2019, 1(1), pp. 112-121. ISBN No. 978-81-941751-0-0At: Srinivas University, Mangalore, India (2019). DOI: 10.5281/ zenodo.3516738.

Guruprakash J Askhar Sakhi Nishanth S Gowda Arun Kumar J Hariish G Ashwin Narayanan S and Krithika LB. AI-based Mobile Apps on Mobile Devices across various Domain Verticals: Past, Present and Future.

Babatunde Sanni. "Anomaly Detection and User Frustration Prediction Using Machine Learning in Mobile App UX". In: (2024).

Mohsen Soori, Behrooz Arezoo, and Roza Dastres. "Artificial intelligence, machine learning and deep learning in advanced robotics, a review". In: Cognitive Robotics, Volume 3 (2023). DOI: 10.1016/j.cogr.2023.04.001..

WENHAOFAN et al. "EstiDroid: Estimate API Calls of Android Applications Using Static Analysis Technology". In: IEEE Access (2020).

Yuqing Yang et al. "SoK: Decoding the Super App Enigma: The Security Mechanisms, Threats, and Trade-offs in OS-alike Apps". In: (2023). DOI: 10 .48550 / arXiv.2306.07495.

Elena Zaitseva et al. "Identifying the Mutual Correlations and Evaluating the Weights of Factors and Consequences of Mobile Application Insecurity". In: Special Issue Intelligent Information Technologies for Quality and Security Assurance (2023). DOI: 10.3390/systems11050242.

by Zakeya Namrud et al. "Deep Learning Based Android Anomaly Detection Using a Combination of Vulnerabilities Dataset". In: Appl Sci 2021 (2021). DOI: 10.3390/app11167538.