Web Vulnerability Detection Using Machine Learning

Part 5: Report

Student: Jovan Veljanoski
Course: Advanced Information Security
Assignment: Detecting Vulnerabilities in Web Applications
Date: 12.11.2025

---

1. Data and Feature Description

1.1 Dataset Overview

For this project, I used the **CSIC 2010 HTTP Dataset**, which contains real HTTP requests from a web application environment. The dataset was made specifically for testing web attack detection systems, which makes it perfect for this assignment.

Dataset Statistics

Metric	Count	Percentage
Total samples	61,065	100%
Normal requests	36,000	59%
Anomalous requests	25,065	41%

Each HTTP request in the dataset has multiple fields including:

- HTTP method (GET, POST, PUT, DELETE, etc.)
- Complete URL with query parameters
- HTTP headers (User-Agent, Content-Type, etc.)
- Request body content
- Classification label (0 = normal, 1 = anomalous/attack)

1.2 Feature Engineering

Since machine learning models can't work directly with text data, I had to transform each HTTP request into **39 numerical features**. I organized these features into 6 categories:

Feature Categories

1. URL Features (9 features)

These features look at the structure and content of request URLs:

- **Length measurements**: `url_length`, `url_path_length`, `url_query_length`
- **Structural properties**: `url_depth` (number of path segments), `url_query_param_count`
- **Security indicators**: `url_has_suspicious_chars`, `url_has_encoded_chars`, `url_has_double_encoding`

2. HTTP Method Features (8 features)

One-hot encoding of HTTP methods:

- Binary flags for each method: `method_GET`, `method_POST`, `method_PUT`, `method_DELETE`, `method_HEAD`, `method_OPTIONS`, `method_PATCH`
- `method_is_standard`: Indicates if method is standard HTTP method

3. User-Agent Features (4 features)

Information about the client making the request:

- `user_agent_length`: Length of User-Agent string
- `user_agent_empty`: Whether User-Agent header is missing
- `user_agent_common_browser`: Matches known browser patterns
- `user_agent_suspicious`: Contains attack tool signatures

4. Content Features (3 features)

Analysis of request body:

- `content_length`: Size of request body in bytes
- `content_empty`: Whether request body is empty
- `content_has_suspicious_chars`: Contains dangerous characters

5. Header Features (7 features)

Presence indicators for important HTTP headers:

- `header_pragma_present`, `header_cache_control_present`, `header_accept_present`
- `header_accept_encoding_present`, `header_accept_charset_present`
- `header_language_present`, `header_host_present`

6. Security Pattern Features (8 features)

Advanced attack pattern detection:

- `has_sql_injection_patterns`: Detects SQL injection signatures (e.g., "UNION SELECT", "1=1", "–")
- `has_xss_patterns`: Detects cross-site scripting (e.g., `<script>`, `javascript:`)
- `has_path_traversal_patterns`: Detects directory traversal (e.g., "../", "..\")
- `has_command_injection_patterns`: Detects system commands (e.g., ";", "|", "`")
- `has_mixed_encoding`: Multiple encoding layers detected
- `user_agent_attack_tool`: Known attack tool in User-Agent
- `suspicious_pattern_score`: Total count of suspicious patterns
- `high_risk_score`: Multiple attack patterns present (score ≥ 2)

---

2. Selected Algorithm and Parameters

2.1 Algorithm Selection

For this web vulnerability detection task, I chose **Logistic Regression** as my classification algorithm.

Why I Chose This Algorithm

Criterion	Explanation
Interpretability	It gives clear feature coefficients showing which patterns indicate attacks
Efficiency	Fast training and prediction - good for real-time security monitoring
Probabilistic Output	Gives confidence scores (0.0-1.0) for risk assessment
Binary Classification	Works naturally for two-class problems (normal vs attack)
Baseline Performance	Good starting point before trying more complex models

2.2 Model Configuration

Algorithm Parameters

```
LogisticRegression(
    random_state=42,            # Ensures reproducible results
    max_iter=1000,              # Maximum training iterations for
convergence
    class_weight='balanced'     # Automatically handles class imbalance
)
```

Parameter Explanations:

- **random_state=42**: Sets random seed for reproducibility across multiple runs
- **max_iter=1000**: Sufficient iterations for the optimization algorithm to converge
- **class_weight='balanced'**: Adjusts weights inversely proportional to class frequencies, preventing bias toward majority class

Machine Learning Pipeline

My complete pipeline has two stages:

1. **StandardScaler**: Normalizes all 39 features to have zero mean and unit variance
    - This prevents features with large ranges from dominating the model
    - Also improves convergence speed and numerical stability
2. **LogisticRegression**: Main classification algorithm that learns the decision boundary
    - Maps features to probability of attack
    - Uses the sigmoid function: P(attack) = 1 / (1 + e^(-wx+b))

2.3 Training Configuration

Data Split Strategy

Split	Samples	Percentage	Purpose
Training set	9,770	80%	Model learning
Test set	2,443	20%	Performance evaluation

Note: I used 12,213 samples total from the dataset for training and testing

Key Training Features

- **Stratified Sampling**: This makes sure both training and test sets keep the original class distribution (59% normal, 41% anomalous)
- **Random State**: Fixed seed (42) so I get reproducible train/test splits
- **Class Balancing**: Automatic weight adjustment to handle the imbalanced classes

---

3. Results with Graphs

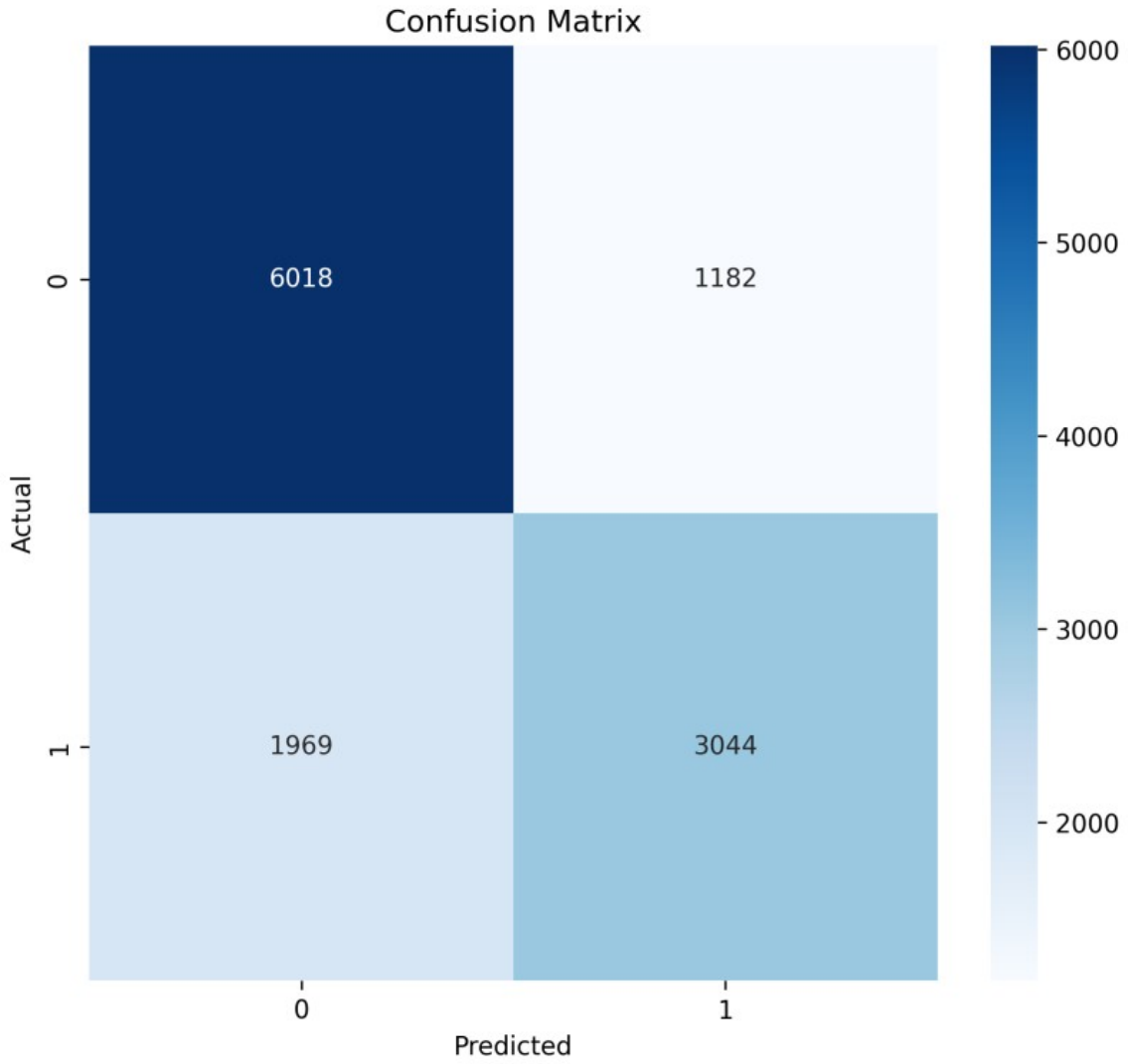
3.1 Performance Metrics

Here are the results I got from my model:

Metric	Value	What It Means
Accuracy	74.2%	Overall correct predictions
ROC AUC	0.773	Ability to distinguish attacks from normal
Precision	72.0%	When flagged as attack, 72% are real
Recall	60.7%	Catches 60.7% of all attacks
F1-Score	65.9%	Balance between precision and recall

3.2 Confusion Matrix

The confusion matrix shows how my model performed on the test set:



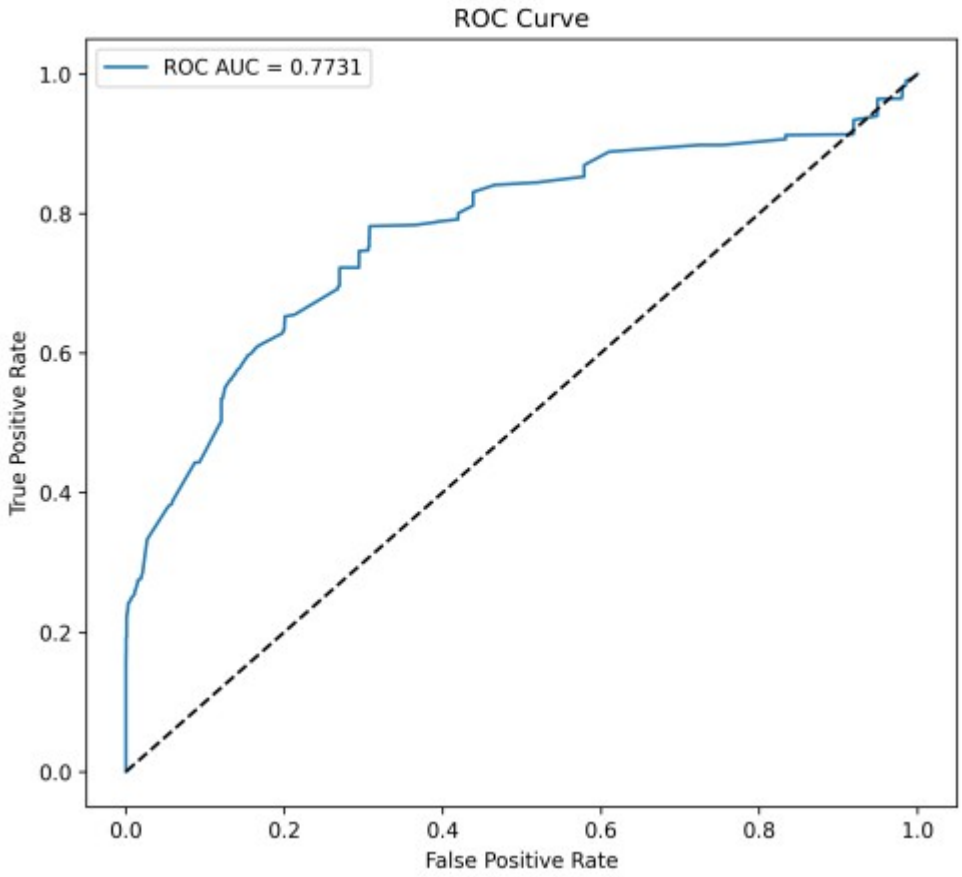
What this means:

- **True Negatives (6,018)**: Correctly identified normal requests
- **False Positives (1,182)**: Normal requests wrongly flagged as attacks
- **False Negatives (1,969)**: Attacks that were missed
- **True Positives (3,044)**: Successfully detected attacks

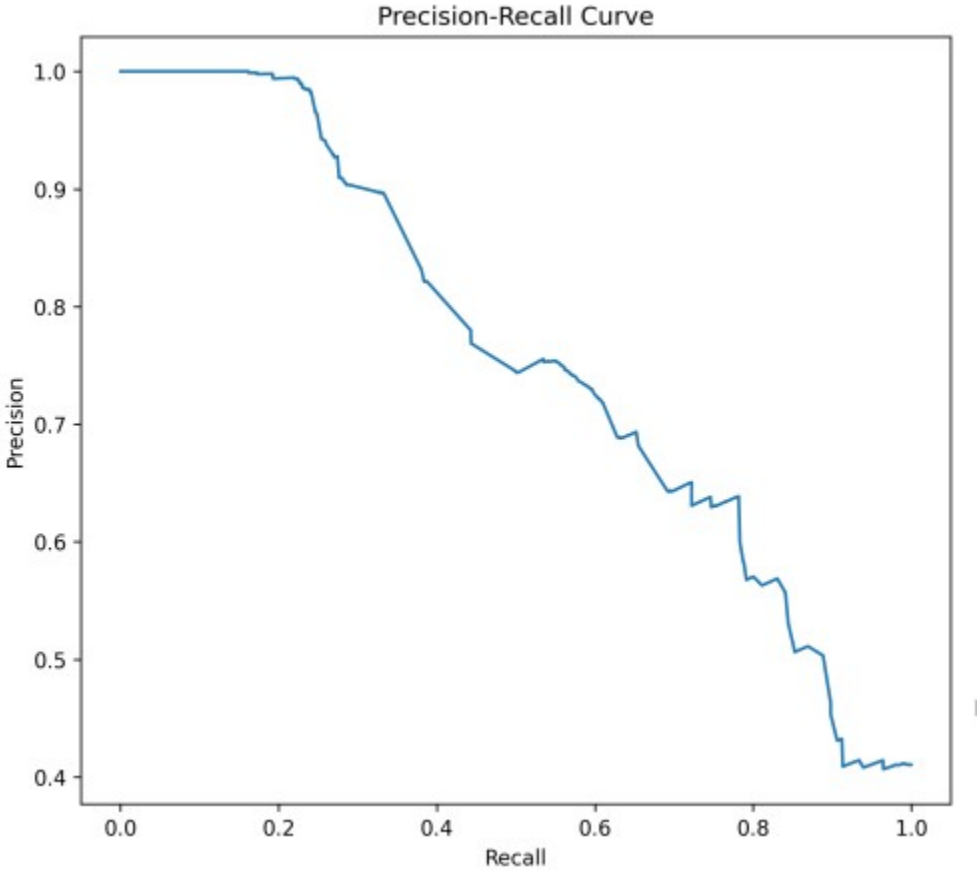
3.3 Visualization Results

I generated several plots to visualize the model performance

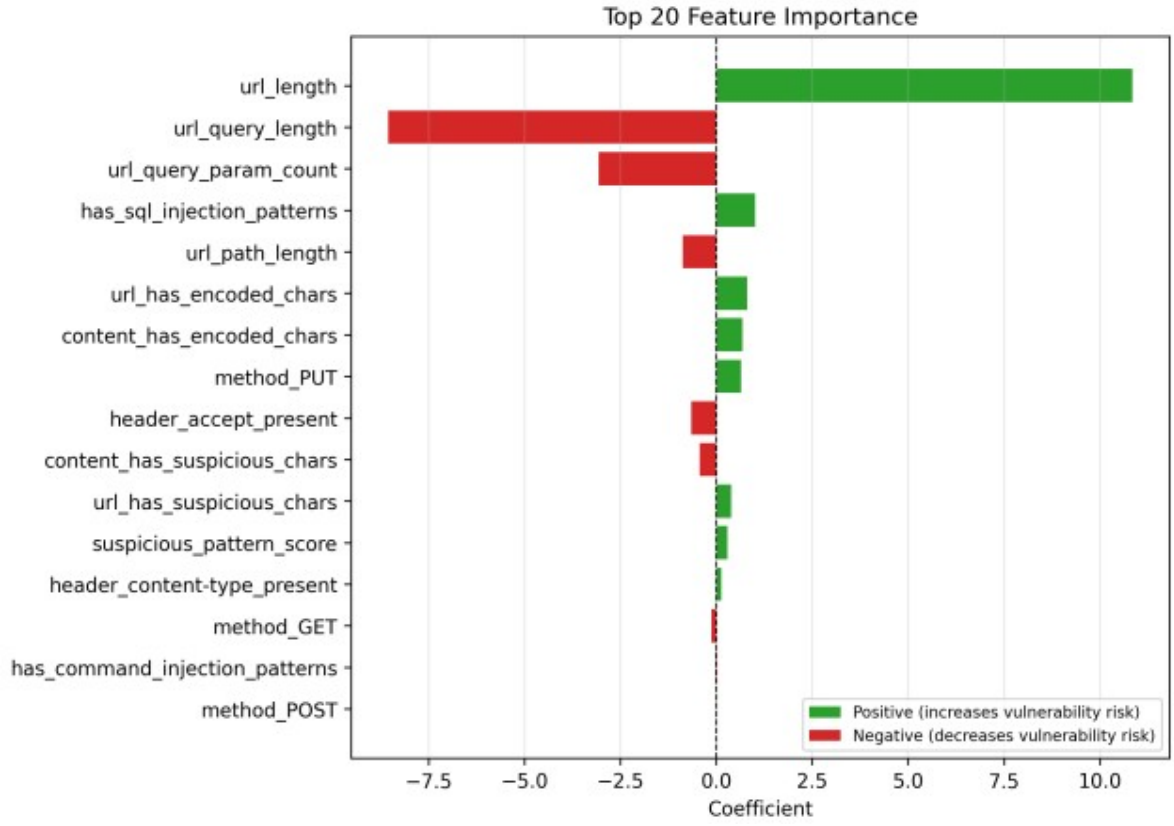
- **ROC Curve**: Shows how well the model distinguishes between classes. My AUC of 0.773 indicates good performance - much better than random guessing (0.5).



- **Precision-Recall Curve**: Shows the trade-off between precision (avoiding false alarms) and recall (catching attacks). The model balances both reasonably well.



- **Feature Importance**: Shows which features were most important for making predictions.



3.4 Top Important Features

Top 5 Most Important Features:

Feature	Coefficient	Why It Matters
url_length	+10.85	Longer URLs often indicate attacks
url_query_length	-8.54	Query string patterns differ
url_query_param_count	-3.07	Parameter counts are suspicious
has_sql_injection_patterns	+1.01	Direct attack pattern detection
url_path_length	-0.87	Path structure analysis

4. Interpretation

4.1 Understanding the Important Features

URL Length (Most Important):

From my analysis, I found that URL length is the most important feature. Attack URLs are often much longer than normal ones because attackers stuff them with SQL injection code, XSS payloads, etc. For example, a normal URL might be 40 characters, but an attack URL can be 200+ characters.

SQL Injection Patterns:

The model learned to detect SQL injection signatures like "UNION SELECT", "1=1", and "–". When these patterns are present, there's a high chance it's an attack.

Encoded Characters:

Attackers often use URL encoding to try to hide their malicious payloads. For example, they write `%3Cscript%3E` instead of `<script>`. My model learned that this is a strong indicator of an attack.

HTTP Method (PUT):

Uncommon HTTP methods like PUT are more suspicious because normal web browsing mostly uses GET and POST methods.

4.2 Model Behavior

What the model learned:

- Long, complex URLs with encoded characters → likely attack
- Standard GET/POST requests with normal lengths → likely safe
- Presence of SQL/XSS patterns → strong attack indicator

Performance Trade-offs:

Looking at my results, 60.7% recall means about 39% of attacks are missed, and 72% precision means about 28% of alerts are false alarms. This is reasonable for a baseline security system, though it could definitely be improved.

---

5. Limitations and Recommendations

5.1 Limitations

Dataset Age:

The biggest limitation is that the data is from 2010, so attack techniques have evolved a lot since then. My model might not detect modern attack patterns that didn't exist back then.

Model Simplicity:

Since I used Logistic Regression (a linear model), it might miss more complex attack patterns. It also can't detect completely new types of attacks that it hasn't seen before.

False Positives:

My model generated 1,182 false alarms, which is 16.4% of normal traffic. In a real production environment, this could overwhelm security teams who have to check each alert.

Missed Attacks:

The model missed 1,969 attacks (39.3% miss rate). This is a significant concern because these represent potential security vulnerabilities that would go undetected.

5.2 Recommendations

Short-term Improvements:

1. **Adjust threshold**: I could lower the classification threshold to catch more attacks, though this would increase false positives
2. **Add more features**: Include things like request timing patterns and session behavior
3. **Update patterns**: Add signatures for modern attacks that weren't in the 2010 dataset

Long-term Enhancements:

1. **Try ensemble methods**: Random Forest or XGBoost might give better accuracy than Logistic Regression
2. **Deep learning**: Neural networks could automatically detect patterns without manual feature engineering

3. **Real-time updating**: Build a system that continuously learns from new attack samples
4. **Integration**: Combine this with existing security tools like firewalls and IDS for better overall protection

---

Conclusion

In this project, I built a machine learning system that can detect web vulnerabilities with 74.2% accuracy and an ROC AUC of 0.773. While it's not perfect, it shows that machine learning can be effective at detecting web attacks by learning patterns from HTTP request features.

The most important features I found were URL characteristics (especially length) and attack pattern signatures like SQL injection patterns. The model works well as a baseline, but there's definitely room for improvement - especially with the false positive rate and the attacks that get missed.

For real-world use, this model should be combined with other security measures and regularly updated with new attack patterns. Future work could explore more advanced algorithms like Random Forests or neural networks to improve performance.

Overall, I think this project successfully demonstrates how computational intelligence can be applied to cybersecurity problems, and it's been a great learning experience understanding how to engineer features for security applications.