



# Detecting Web-Based Attacks with SHAP and Tree Ensemble Machine Learning Methods

Ndichu, Samuel ; Kim, Sangwook ; Ozawa, Seiichi ; Ban, Tao ; Takahashi, Takeshi ; Inoue, Daisuke

---

(Citation)

Applied Sciences, 12(1):60

(Issue Date)

2022-01

(Resource Type)

journal article

(Version)

Version of Record

(Rights)

© 2021 by the authors. Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).


(URL)

<https://hdl.handle.net/20.500.14094/90009126>



## Article

# Detecting Web-Based Attacks with SHAP and Tree Ensemble Machine Learning Methods

Samuel Ndichu <sup>1,2,\*</sup> , Sangwook Kim <sup>1</sup>, Seiichi Ozawa <sup>1,3</sup>, Tao Ban <sup>2</sup>, Takeshi Takahashi <sup>2</sup> and Daisuke Inoue <sup>2</sup>

<sup>1</sup> Graduate School of Engineering, Kobe University, Kobe 657-8501, Japan; kim@eedept.kobe-u.ac.jp (S.K.); ozawasei@kobe-u.ac.jp (S.O.)

<sup>2</sup> National Institute of Information and Communications Technology, Tokyo 184-8795, Japan; bantao@nict.go.jp (T.B.); takeshi\_takahashi@nict.go.jp (T.T.); dai@nict.go.jp (D.I.)

<sup>3</sup> Center for Mathematical and Data Sciences, Kobe University, Kobe 657-8501, Japan

\* Correspondence: sammiendichu01@stu.kobe-u.ac.jp or ndichu@nict.go.jp

**Abstract:** Attacks using Uniform Resource Locators (URLs) and their JavaScript (JS) code content to perpetrate malicious activities on the Internet are rampant and continuously evolving. Methods such as blocklisting, client honeypots, domain reputation inspection, and heuristic and signature-based systems are used to detect these malicious activities. Recently, machine learning approaches have been proposed; however, challenges still exist. First, blocklist systems are easily evaded by new URLs and JS code content, obfuscation, fast-flux, cloaking, and URL shortening. Second, heuristic and signature-based systems do not generalize well to zero-day attacks. Third, the Domain Name System allows cybercriminals to easily migrate their malicious servers to hide their Internet protocol addresses behind domain names. Finally, crafting fully representative features is challenging, even for domain experts. This study proposes a feature selection and classification approach for malicious JS code content using Shapley additive explanations and tree ensemble methods. The JS code features are obtained from the Abstract Syntax Tree form of the JS code, sample JS attack codes, and association rule mining. The malicious and benign JS code datasets obtained from Hynek Petrak and the Majestic Million Service were used for performance evaluation. We compared the performance of the proposed method to those of other feature selection methods in the task of malicious JS code content detection. With a recall of 0.9989, our experimental results show that the proposed approach is a better prediction model.

**Keywords:** web-based attacks; feature selection; Shapley additive explanations; tree ensemble methods; machine learning



**Citation:** Ndichu, S.; Kim, S.; Ozawa, S.; Ban, T.; Takahashi, T.; Inoue, D. Detecting Web-Based Attacks with SHAP and Tree Ensemble Machine Learning Methods. *Appl. Sci.* **2022**, *12*, 60. <https://doi.org/10.3390/app12010060>

Academic Editor: Arcangelo Castiglione

Received: 18 November 2021

Accepted: 15 December 2021

Published: 22 December 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Websites are very popular; hence, cybercriminals find these platforms to be perfect tools for launching their attacks. Web-based attacks remain a significant challenge, as evasion techniques are continuously evolving. Attackers compromise Uniform Resource Locators (URLs) and their JavaScript (JS) content to perform malicious activities on the Internet. Such activities include phishing, URL redirection, spamming, social engineering, botnets, and drive-by-download exploits [1–3]. The attacks are delivered through emails, malware advertisements, texts, pop-ups, malicious scripts, and search results. The Domain Name System (DNS) [1–5] provides cybercriminals the flexibility to easily migrate their malicious servers, as they hide the IP addresses behind domain names [2,3]. Securing websites is vital for maintaining confidentiality, integrity, and availability, and an equally adaptive strategy is required to detect such attacks effectively.

Methods such as maintaining a blocklist [6], client honeypots, domain reputation inspection [4], and domain and web metrics analysis [5,7–9] can detect malicious URLs and their JS code content [4,10,11]. However, new URLs are registered every day, and a blocklist can be evaded through URL obfuscation, fast flux, cloaking, and URL shortening. Recently,

machine learning approaches that employ feature extraction and representation learning for malicious URLs and their JS code content detection have been proposed [2,3,12–14]. Machine learning algorithms learn a prediction function based on features such as lexical, host-based, URL lifetime, and content-based features that include HyperText Markup Language and JS code. A classifier is then used to predict whether given content is malicious or benign using a test dataset. Models such as these can be generalized to new data, unlike the blacklist approach. However, there are challenges in crafting fully representative features for model training.

In our previous work [15–17], we developed a system to detect malicious JS codes using fixed-length vectors and the Abstract Syntax Tree form of the JS code (AST-JS). This study extends our work by detecting malicious JS code content through feature selection. Using atomic features and allowing the model to determine the relationships between them is preferable instead of using composite features [2,3]. This study proposes using Shapley additive explanations (SHAP) values and tree ensemble methods. While other methods learn a set of manually extracted or engineered features, our approach adopts features that are automatically selected based on their contributions to the model's output.

Malicious URLs and their JS code content can stealthily perpetrate web-based attacks [15–18]. Other studies [12,19,20] have shown that malicious URLs and their JS code content exhibit features that are distinct from those of legitimate URLs. Therefore, we focused on JS code content-based features to detect malicious websites. Our premise is that malicious JS code content originating from attackers exhibits different AST-based features than legitimate content. A SHAP-based model would learn such features from the distribution of continuously evolving malicious JS code content.

The main contributions of this study are the following:

- This study proposes using SHAP and tree ensemble methods for detecting web-based attacks.
- We detail the process of obtaining features using AST-JS node sets and patterns, sample JS attack codes, and association rule mining.
- We compared the performances of different classifiers in malicious JS code detection using SHAP selected features and achieved good detection performance for the tree ensemble methods.
- We compared the performance of SHAP selected features to the performance of those selected by other feature selection methods: Boruta, ELI5, RandomForest, and SelectKBest.
- The proposed web-based attack detection method outperformed the other feature selection methods in all three evaluation metrics.

The remainder of this paper is organized as follows. Section 2 highlights the related work. We present our approach in Section 3. Section 4 presents the performance evaluations of our own and other models; and we present our discussion and conclusions in Sections 5 and 6, respectively.

## 2. Related Work

Researchers have put forward many prevention techniques to keep up with the increase in attacks and new attack methods. Blocklists are repositories of known malicious URLs and have long been employed to detect malicious URLs and their JS codes. Heuristic and signature-based systems search for signs of standard attack patterns. The weakness of these approaches is that new or variant URLs and their JS codes and zero-day attacks can easily evade detection. Further, cybercriminals widely use obfuscation to perpetrate attacks and evade detection.

Some studies [12–14] have proposed embedding the bag-of-words model in a URL or features of a URL and its JS code content. Ma et al. [12–14] proposed an approach to analyzing a URL to predict the maliciousness of websites using an online confidence-weighted algorithm for lexical feature learning. Individual feature vectors were manually engineered. Ma et al. [12] acknowledged that other potentially useful sources of information

for features could improve classification accuracy. However, they did not examine a web page's actual content, citing reasons such as user safety, the model's operation costs, the classifier's applicability to the URL context, and poor reliability when obtaining a malicious web page version. They also used bag-of-words features and therefore did not preserve token order. Conversely, our proposed approach uses JS code content features. Since AST-JS is resistant against perturbations in JS code content, it makes our model applicable for various JS code contexts and varying content.

Other systems [2,3,5,21,22] have been proposed that use constructed features or feature engineering. Bilge et al.'s [2,3] EXPOSURE conducted a passive DNS analysis to identify malicious domains. It is a dynamic reputation system based on passive recursive DNS monitoring. They extracted 15 behavioral features to train the J48 decision tree classifier. Their study assumed that large volumes of DNS data requests should exhibit sufficient behavioral differences to distinguish between benign and malicious domains. In another study, a system to detect malicious domains using DNS records and domain name features was proposed by Al Messabi et al. [5] using J48, a C4.5 decision tree. This approach studied prior DNS activities for each domain and the relationship between defective and legitimate domains' physical behavior [21], combining several existing DNS-based and domain name-based features from previous work [2,3,22]. They assumed that eight unique behavioral features could accurately identify malicious websites and proactively detect an attack. However, feature engineering is challenging, even for domain experts. We performed automatic feature selection to address these challenges.

Kuyama et al. [23,24] sought to detect targeted attacks by monitoring the communication between the Command and Control (C&C) server and the computers in the local area network. Their proposed method identified new C&C servers using supervised machine learning; extracted the WHOIS feature points and DNS information; and searched the site. For performance comparison, the malicious domains were detected separately using a neural network and a support vector machine. Since manual feature selection is a tedious task, even for domain experts, we propose an autonomous feature-selection method using SHAP values and tree ensemble methods.

Other studies have attempted to detect specific URL attack types, such as phishing [25–30], malicious advertisements, and click fraud [1,31–36]. These approaches are suited for specific attack types and do not generalize well to other attacks. Masri et al. [1] proposed a system for the automatic classification and detection of malicious advertisements. They used VirusTotal [37], URLVoid [38], and TrendMicro [39]. Their study relied heavily on other tools that employ signatures and blocklist services. Therefore, it was prone to problems inherent to signature and blocklist-based systems [4].

We propose an automatic feature selection method of malicious JS code content using SHAP values. A more thorough analysis of JS code content-based features may help detect threats [10,40]. These features provide more rich information for the feature learning process than URL-based ones, as much information is extracted from a web page. Safety concerns may arise if the JS code content is executed. However, we used AST-JS for the code structure representation. Further, AST-JS enables the capturing of more details regarding a web page. The assumption is that more information would lead to a better prediction model. We assumed that correlations between various web-based attack features exist, and a machine learning model can identify these relationships with minimal effort.

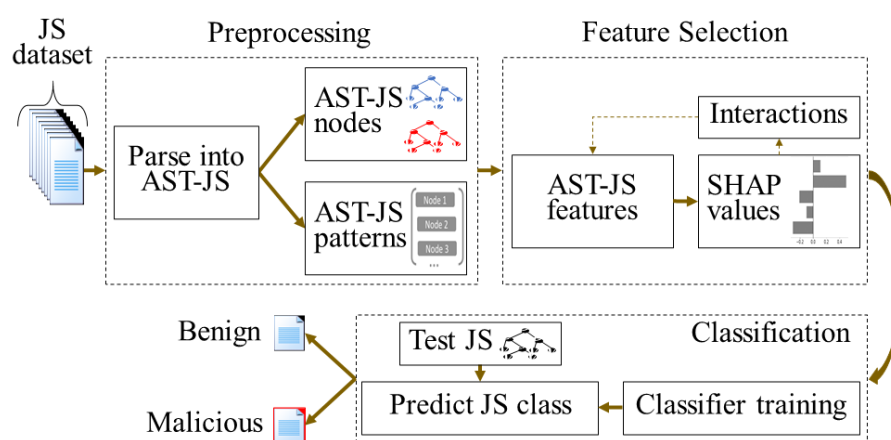
### 3. Proposed Method

When machine learning is employed to detect malicious URLs and their JS code content, certain considerations have to be addressed, such as privacy and safety concerns, feature learning methods, and the development of a fully representative feature set.

Given the common vulnerabilities, characteristics, and features employed by malicious JS codes, it is evident that malicious websites are different from benign ones, as they each have different objectives. For example, a malicious JS code contains functions or a combination of functions such as *document.write*, *location.replace*, and *document.getElementById*.

Among other things, these functions are used to perform cross-site scripting attacks by injecting malicious JS code into the document object model, direct concatenation of user input with the query string, and building a database query via string concatenation used to perform structured query language injections. Therefore, developing a malicious JS code detection system that can generalize well to detect different attack types is imperative. We propose automatic feature selection for AST-JS features, leveraging the global and local features that contribute to the model performance. Adequate details of a JS code structure are also presented without executing actual code.

This section details the proposed method for web-based attack detection using SHAP and tree ensemble methods. The method includes the following three main stages: preprocessing, feature selection, and classification, as shown in Figure 1. The following sections explain the JS code preprocessing, feature selection, the SHAP values used for feature selection, and the machine learning classifiers used for classification.



**Figure 1.** The proposed web-based attacks detection method.

### 3.1. Preprocessing

This section describes the AST-JS node sets and patterns, sample JS attack codes, and association rule mining.

#### 3.1.1. AST-JS Node Sets and Patterns

Preprocessing is essential to enhancing the feature learning process. To define our first input set, we parsed each JS code in our dataset to obtain the *expression*, *pattern*, *statement*, and *declaration* AST-JS nodes.

#### 3.1.2. Sample JS Attack Codes

Malicious JS code rarely uses raw values, as attackers endeavor to evade static analyzers. Therefore, it is essential to capture features that employ covert implementations to hide their values. To capture these features, we analyzed the feature importance of our first feature set using SHAP interaction values. This feature analysis exposed each AST-JS node and the combination's contribution to the malicious JS code detection model. To identify such combinations, we obtained sample JS attack codes and performed association rule mining using the Frequent Pattern growth (FP-growth) mlxtend library [41–43]. The sample JS attack codes are explained next.

Listing 1 shows a JS code that lacks a definition of the index of an object in the current scope. When *foo()* is called, *x = 0* is passed, which is a local index of the function. We capture this attack pattern using the AST node type *EmptyStatement\_ExpressionStatement\_CallExpression\_Identifier*.

Listing 2 shows a JS code where the argument of *setTimeout()* is a string concatenation with binary expression. We capture this attack pattern using the AST node type *CallExpression\_Identifier\_BinaryExpression*.

**Listing 1.** A snippet of a JS code with EmptyStatement\_ExpressionStatement\_CallExpression\_Identifier.

---

```

1 function foo (x){
2   var arr = ["1", "2", "3"];
3   eval (arr [x])
4 };
5 foo (0);
6 // AST for EmptyStatement\_ExpressionStatement\_CallExpression\_Identifier
7 "type": "EmptyStatement"
8 },
9 {
10  "type": "ExpressionStatement",
11  "expression": {
12    "type": "CallExpression",
13    "callee": {
14      "type": "Identifier",
15      "name": "foo"
16    },
17    "arguments": [
18      {
19        "type": "Literal",

```

---

**Listing 2.** A snippet of a JS code with CallExpression\_Identifier\_BinaryExpression.

---

```

1 function startOverflow (num) {
2 }
3 if (num == 255) setTimeout ("startOverflow (" + (num + 1) + ")", 2000);
4 // AST for CallExpression\_Identifier\_BinaryExpression
5 "type": "ExpressionStatement",
6 "expression": {
7   "type": "CallExpression",
8   "callee": {
9     "type": "Identifier",
10    "name": "setTimeout"
11  },
12  "arguments": [
13    {
14      "type": "BinaryExpression",
15      "operator": "+",
16      "left": {
17        "type": "BinaryExpression",
18        "operator": "+",

```

---

Listing 3 shows nested JS code function calls, where an attacker hides the function arguments in another function call to evade static analyses. We capture this attack pattern using the AST node *type CallExpression\_Identifier\_Literal\_CallExpression*.

Listing 4 shows variable declarations, where *e* in *line2* is assigned to a function *c*. We capture this attack pattern using the AST node *type AssignmentExpression\_Identifier\_FunctionExpression*.

Listing 5 shows a JS code where the attacker attempts to detect the user environment, such as the operating system or the browser version. We capture this attack pattern using the AST node *type VariableDeclarator\_Identifier\_LogicalExpression*.

Listing 6 shows the JS code sample string concatenation and function obfuscation codes. We capture this attack pattern using the AST node *type BinaryExpression\_BinaryExpression*.

**Listing 3.** A snippet of a JS code with CallExpression\_Identifier\_Literal\_CallExpression.

---

```

1 var p26 = new Array ();
2 function f85 (g55 , w27) {
3   p26 [g55] = w27;
4 };
5 function w25 (h51) {
6   return h51;
7 };
8 f85 (763 , w25 ('s'));
9 // AST for CallExpression\Identifier\_Literal\_CallExpression
10 "type": "ExpressionStatement",
11 "expression": {
12   "type": "CallExpression",
13   "callee": {
14     "type": "Identifier",
15     "name": "f85"
16   },
17   "arguments": [
18     {
19       "type": "Literal",
20       "value": 763,
21       "raw": "763"
22     },
23     {
24       "type": "CallExpression",

```

---

**Listing 4.** A snippet of a JS code with AssignmentExpression\_Identifier\_FunctionExpression.

---

```

1 eval (function (p, a, c, k, e, d) {
2   e = function (c) {
3     return (c < a ? '' : e(parseInt (c / a))) + ((c = c % a) > 35 ? String.
4       fromCharCode (c + 29) : c.toString (36) );
5     if (! '' . replace (/~/ , String )) {
6       while (c --) d[e(c)] = k[c] || e(c);
7     }
8   })
9   // AST for AssignmentExpression\_Identifier\_FunctionExpression
10  "type": "ExpressionStatement",
11  "expression": {
12    "type": "AssignmentExpression",
13    "operator": "=",
14    "left": {
15      "type": "Identifier",
16      "name": "e"
17    },
18    "right": {
19      "type": "FunctionExpression",

```

---

**Listing 5.** A snippet of a JS code with VariableDeclarator\_Identifier\_LogicalExpression.

---

```

1 var b = navigator.userAgent.match(/iPhone OS ([\ d_]+) /) ||
2   navigator.userAgent.match(/iPad OS ([\ d_]+) /) ||
3   navigator.userAgent.match(/CPU OS ([\ d_]+) /);
4 // AST for VariableDeclarator\_Identifier\_LogicalExpression
5 "type": "VariableDeclaration",
6 "declarations": [
7   {
8     "type": "VariableDeclarator",
9     "id": {
10      "type": "Identifier",
11      "name": "b"
12    },
13    "init": {
14      "type": "LogicalExpression",

```

---



**Listing 6.** A snippet of a JS code with BinaryExpression\_BinaryExpression.

---

```

1 // String concatenation
2 var a = "He" + "ll" + "o";
3 var b = " World !";
4 var c = a + " " + b;
5 // Function obfuscation
6 var x = eval;
7 var y = x("do"+"cu"+"ment");
8 // AST for BinaryExpression\_BinaryExpression
9 "type": "BinaryExpression",
10 "operator": "+",
11 "left": {
12   "type": "BinaryExpression",
13   "operator": "+",
14   "left": {

```

---

### 3.1.3. Association Rule Mining

Algorithm 1 shows AST-JS's association rule mining procedure using the FP-growth algorithm. The generated association rules for benign and malicious JS codes are transformed into a list (lines 14–17). For AST-JS feature selection, the lists are converted into individual benign and malicious data frames (lines 19–22).

---

**Algorithm 1** Mining frequent AST-JS node sets using the FP-growth algorithm.

---

**Input:**  $D$ —a database of benign and malicious JS codes defined as AST-JS nodes;  
 $min\_support$ —the minimum support count threshold.

**Output:** Benign and malicious DataFrames of AST-JS nodes and node combinations.

```

1: if  $Tree$  contains a single path  $P$  then
2:   for each combination  $\beta$  of the nodes in the path  $P$  do
3:     generate pattern  $\beta \cup \alpha$  with  $support = min\_support$  of nodes in  $\beta$ ;
4:   end for
5: else
6:   for each  $a_i$  in the Tree header do
7:     generate pattern  $\beta = a_i \cup \alpha$  with  $support = a_i.support$ ;
8:     construct  $\beta$ 's conditional pattern base and then  $\beta$ 's conditional FP_tree  $Tree_\beta$ ;
9:   end for
10:  if  $Tree_\beta \neq 0$  then
11:    call  $FP\_growth(Tree_\beta, \beta)$ ;
12:  end if
13: end if
14: for each benign 0 and malicious 1 frequent patterns do
15:   0.antecedents.apply(sorted(list(i)))+0.consequents.apply(sorted(list(i)));
16:   1.antecedents.apply(sorted(list(i)))+1.consequents.apply(sorted(list(i)));
17:   return list0, list1
18: end for
19: for each benign list0 and malicious list1 frequent patterns do
20:   TransactionEncode.fit(list0).transform(list0);
21:   TransactionEncode.fit(list1).transform(list1);
22:   pd.DataFrame(list0, list1, columns=TransactionEncode.columns_)
23: end for

```

---

Support is defined as the frequency of an AST-JS node or set in the JS code dataset, and confidence is the probability of an AST-JS node set's occurrence with its set of nodes. Confidence measures how often an association rule is found to be true.  $min\_support$  is the minimum support for an AST-JS node set to be identified as frequent.  $confidence's\ min\_threshold$  is the minimum confidence for generating an association rule.



Finally, we analyzed the feature importance of each AST-JS feature selected using SHAP values to understand how different features influence our malicious JS code detection model's performance. This process was achieved in two parts: first, we used our original AST-JS features and then the AST-JS combination features extracted through association rule mining. Next, we explain the feature selection process.

### 3.2. Feature Selection

The feature selection process comprises sample JS attack codes, association rule mining, and SHAP values. The sample JS attack codes are used to obtain AST-JS feature combinations based on expression, pattern, statement, and declaration nodes. AST-JS feature combinations augment the AST-JS node features by capturing JS-based attacks. The JS-based attacks include variable obfuscation, attacks using eval and unescape, attacks using the current context or browser objects, user environment detection, string obfuscation, string concatenation, encoding functions, nested function calls, function obfuscation, and attacks combining string concatenation with function obfuscation.

Association rule mining is used to measure how often AST-JS nodes appear together in benign and malicious JS codes. Features obtained using Association rule mining are used for performance comparison. SHAP values are used to determine AST-JS feature interactions, and a feature set is selected based on the contribution to the detection performance for malicious JS code. Next, we explain the SHAP features' importance.

### 3.3. Shapley Additive Explanations' Feature Importance

SHAP is a game-theoretic approach that explains any machine learning model's output that connects optimal credit allocation with local explanations using the classical Shapley values and their related extensions [44–46]. The values show the extent to which a feature is responsible for a change in the model's output. SHAP values either increase or decrease the model's prediction values and balance out the input's actual prediction. The prediction starts from the base value, the mean derived from the entire prediction. Our premise is that interpreting the malicious JS code detection model's output will guide AST-JS feature selection, further improving its performance. Identifying the most-impactful features may enable us to derive other features with additional information to improve the detection performance. This knowledge will also provide necessary insights into the distribution of specific benign and malicious JS code features.

A JS code input is represented as  $Z$  with a set of AST-JS features  $z_1, z_2, \dots, z_n$ , and its corresponding output  $Z'$  and predicted features  $z'_1, z'_2, \dots, z'_n$ , using the tree ensemble model  $g$ . Algorithm 2 shows the procedure for calculating tree SHAP values using tree ensemble methods that return AST-JS features (line 6) sorted in descending order of their importance. First, we obtained AST-JS features from the JS code dataset and saved them as AST-JS  $M$  features. For each feature  $z'_i$  in the AST-JS  $M$  features, we used tree SHAP to obtain the SHAP values. When predicting the observed feature  $z'_i$ , the SHAP importance values for each feature,  $z_1, z_2, \dots, z_n$ , excluding  $z_i$ , were calculated. Tree SHAP receives  $g$  and a background set with  $j$  instances to build the local explanation model and calculate the SHAP values. Then,  $g$  takes  $Z$  and  $i$  as input and predicts  $Z'$ ; the value in the  $i$ -th feature, a feature in the AST-JS  $M$  features, is returned by Algorithm 3. This results in a two-dimensional list of the *SHAPsortedM* features. Each row in the list represents the SHAP values one of each AST-JS feature in  $M$  features.

We divided the AST-JS features into those that enhanced the performance of the malicious JS code detection model and those that did not by determining whether their SHAP values moved the predicted value toward or away from the true value. Algorithm 4 shows how the SHAP values were used to select relevant AST-JS features. For each AST-JS feature (line 1), we checked whether the true feature value given by the input JS code was greater than the predicted value (line 2); a positive SHAP value indicates a contributing feature (line 3) and vice versa (line 4). If the predicted feature value was less than the input value for a JS code (line 5), the contributing features is indicated by a negative SHAP value

and vice versa. This algorithm returned two lists, the *SHAPSelected* and *SHAPnotSelected* features, containing the contributing and non-contributing AST-JS features, respectively, along with the SHAP values for each of the AST-JS  $M$  features.

---

**Algorithm 2** Calculating tree SHAP values for AST-JS  $M$  features.

---

**Input:**  $Z$ —a malicious or benign JS code we want to explain,  $Z_{1..j}$ —AST-JS instances that tree SHAP uses as background examples, the  $g$ —tree ensemble model.

**Output:** *SHAPsortedM* features - SHAP values for each AST-JS feature in JS code dataset sorted in descending order of their importance.

```

1:  $M$  features  $\leftarrow$  feature values based on AST-JS.
2: for each  $i \in M$  features do
3:   explainer  $\leftarrow$  shap.TreeExplainer( $g, Z_{1..j}$ )
4:   SHAPsortedM features[ $i$ ]  $\leftarrow$  explainer.shapvalues( $Z, i$ )
5: end for
6: return SHAPsortedM features

```

---



---

**Algorithm 3**  $g(Z, i)$ —predicting the  $i$ th AST-JS feature.

---

**Input:**  $Z$ —a malicious or benign JS code to explain,  $i$ —the AST-JS feature to get prediction for,  $g$ —tree ensemble model.

**Output:**  $z'_i$ —the value of the  $i$ th AST-JS feature in  $Z'$ .

```

1:  $z'_i \leftarrow g.predict(Z)[i]$ 
2: return  $z'_i$ 

```

---



---

**Algorithm 4** Selecting contributing AST-JS features' SHAP values.

---

**Input:** *SHAPsortedM* features—SHAP values for each AST-JS feature in JS code dataset sorted in descending order of their importance,  $Z$ —a malicious or benign JS code we want to explain,  $Z'$ —the prediction for  $Z$ .

**Output:** *SHAPSelected* features and *SHAPnotSelected* features.

```

1: for each  $i \in SHAPsortedM$  features do
2:   if  $z_i > z'_i$  then
3:     SHAPSelected[ $i$ ]  $\leftarrow SHAPsortedM$  features[ $i$ ]  $> 0$ 
4:     SHAPnotSelected[ $i$ ]  $\leftarrow SHAPsortedM$  features[ $i$ ]  $< 0$ 
5:   else
6:     SHAPSelected[ $i$ ]  $\leftarrow SHAPsortedM$  features[ $i$ ]  $< 0$ 
7:     SHAPnotSelected[ $i$ ]  $\leftarrow SHAPsortedM$  features[ $i$ ]  $> 0$ 
8:   end if
9: end for
10: return SHAPSelected features, SHAPnotSelected features

```

---

The benefits of computing SHAP values are global and local interpretability, which shows the manner and degree of each AST-JS feature's contribution to the prediction, and each observation obtains its own set of SHAP values, enabling the evaluation of the features' impacts.

### 3.4. Machine Learning Classifiers

For the classification task for web-based attack detection, seven machine learning classifiers from the Scikit-learn library [47–50] were used in the experiments, i.e., XGBoost, LightGBM, RandomForest, DecisionTree, LogisticRegression, KNeighbors, and GaussianNB. The best algorithm was selected based on the k-fold cross-validation evaluation results.

## 4. Experiments

This section presents the experimental setup and performance comparison results for web-based attack detection.

#### 4.1. Experimental Setup

We used a dataset of 39,443 malicious and 40,000 benign JS codes in the experimental setup. The malicious JS codes were obtained from Hynek Petrak's dataset [51], and the benign codes were obtained from the Majestic Million Service [7]. We used a JS code parser to preprocess the dataset into AST-JS nodes, resulting in 32,430 malicious and 38,891 benign JS codes. This resulted in 25 *expression* and *pattern* and 23 *statement* and *declaration* AST-JS features.

Next, we visualized and investigated each AST-JS feature's contribution to the malicious JS code detection model's performance using SHAP values. This analysis indicated the need to define more concrete features. Using sample JS attack codes, we added ten more AST-JS features based on combinations of the original ones. These features represent JS-based attacks and contribute to the model's detection performance. We then used association rule mining and a confidence metric to measure how often AST-JS nodes appear together in benign and malicious JS codes. This resulted in 33 features with parameters *min\_support* = 0.4 for malicious JS codes, *min\_support* = 0.53 for benign codes, and *confidence's min\_threshold* = 1 for both benign and malicious codes. This selection reduced our original features and formed the second input set. Finally, we selected a final set of AST-JS features using the SHAP values. This selection resulted in forty-four features, comprising thirty-four AS-JS nodes and ten node combinations.

Performance evaluation was conducted using the three-input feature sets, i.e., the 48 AST-JS features obtained after the initial preprocessing, the 33 features obtained using association rule mining, and the 44 features obtained by feature selection using the SHAP value. The parameters obtained using *GridSearchCV* for the two best-performing models, *XGBClassifier* and *LGBMClassifier*, using the three AST-JS feature sets, are listed in Table 1.

**Table 1.** *XGBClassifier* and *LGBMClassifier* parameters.

Parameter	AST-JS Nodes	Association Rule	SHAP Value
<b>XGBClassifier</b>			
<i>colsample_bytree</i>	0.5	0.7	0.4
<i>gamma</i>	0.0	0.1	0.0
<i>learning_rate</i>	0.2	0.2	0.2
<i>max_depth</i>	10	8	13
<i>min_child_weight</i>	1	1	1
<b>LGBMClassifier</b>			
<i>lambda_l1</i>	0	0	0
<i>lambda_l2</i>	0	0	0
<i>max_depth</i>	9	6	8
<i>n_estimators</i>	520	520	520
<i>num_leaves</i>	10	10	10

We found that these parameters are optimal for malicious JS code content detection using the three-input feature sets. We performed 10-fold cross-validation to evaluate the performance of our approach and computed precision, recall, and F1-score evaluation metrics. Precision is the classifier's ability to not label a benign JS code as malicious, and recall is the classifier's ability to find all malicious JS codes. The F1-score can be interpreted as the weighted harmonic mean of precision and recall. Given that *TP* is the number of malicious JS codes correctly classified as malicious, *TN* is the number of benign JS codes correctly classified as benign, *FN* is the number of malicious JS codes classified as benign, and *FP* is the number of benign JS codes classified as malicious. *Precision*, *recall*, and *F1-score* are given by:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

$$F1\text{-score} = 2 \frac{Precision \times Recall}{Precision + Recall} \quad (3)$$

#### 4.2. Performance Comparisons

The results of these experiments are presented in this section. Table 2 presents the precision, recall, and F1-score values obtained using features defined as AST-JS nodes and 10-fold cross-validation. Each model performed well on the recall metric, especially the tree ensemble methods. The best-performing model, XGBoost, could detect malicious JS code content with an error rate of 0.0019 for recall, 0.0302 for precision, and 0.0162 for F1. The lower precision metric revealed that the model misclassified some benign JS codes. Even though this scenario is not harmful, it may lead to threat-alert fatigue if users and security analysts receive many false alarms. Therefore, there is a need to improve this model further.

**Table 2.** Performance comparison using AST-JS node features.

Model	Recall	Precision	F1
XGBoost	0.9981 ± 0.0008	0.9698 ± 0.0033	0.9838 ± 0.0018
LightGBM	0.9979 ± 0.0008	0.9691 ± 0.0032	0.9833 ± 0.0019
RandomForest	0.9986 ± 0.0005	0.9702 ± 0.0032	0.9842 ± 0.0018
DecisionTree	0.9983 ± 0.0005	0.9687 ± 0.0029	0.9833 ± 0.0016
LogisticRegression	0.9839 ± 0.0021	0.9414 ± 0.0039	0.9622 ± 0.0025
KNeighbors	0.8448 ± 0.0062	0.9986 ± 0.0006	0.9153 ± 0.0037
GaussianNB	0.9968 ± 0.0013	0.5493 ± 0.0029	0.7083 ± 0.0025

We analyzed the AST-JS node features using the SHAP values. Figure 2 presents a SHAP summary plot that shows the relative impacts of AST-JS features on the JS code dataset.

The SHAP values are plotted on the x-axis for each AST-JS feature on a row sorted by the sums of their SHAP value magnitudes. The vertically piled points represent the feature density, and the colors show the feature values. The values give the distribution of each AST-JS feature's impact on the model's output. The red and blue colors represent high and low AST-JS feature values. The color allows us to visualize how changes in the value of an AST-JS feature would affect a change in prediction; for example, high SHAP values for the *SequenceExpression* feature would indicate a high risk of maliciousness for a JS code.

Using such plots, we can deduce that features such as the *ObjectExpression* would influence the model's prediction more than the *LabeledStatement*. A feature such as *SequenceExpression* has a significant, positive effect on AST-JS prediction, and therefore, a high *SequenceExpression* SHAP value may indicate a higher risk for maliciousness. On the contrary, a feature such as the *FunctionExpression* has a significant, negative effect on the AST-JS class prediction, and therefore, a low *FunctionExpression* SHAP value may indicate a higher risk for maliciousness.

Additionally, interesting patterns can be observed, such as high values of the *ExpressionStatement\_CallExpression\_Identifier* feature clustered in a very dense region represented by the red blob. Additionally, low values of the *BlockStatement* feature are clustered in a very dense region, as shown by the blue blob. However, features such as the *ReturnStatement* and *FunctionExpression* have a much more uniform distribution with high and low SHAP values, respectively, pushing the prediction to 1. The red and blue blobs on the left and right indicate an even distribution of that feature in the JS code dataset. Some features such as *EmptyStatement\_ExpressionStatement\_CallExpression* and *CallExpression\_Identifier\_Literal* are not crucial for most JS codes. However, these fea-

tures significantly impact a subset of JS codes in the dataset. This scenario highlights how a globally important feature is not necessarily the most critical feature for attack detection in JS codes.

Figure 3 shows the SHAP dependence plot for the top AST-JS feature before (a) and after (b) feature selection.

Every dot represents a JS code. Vertical dispersion at an AST-JS feature value results from interaction effects in the model. The color highlights the high or low forces behind the interactions. The y-axis represents the SHAP values. The SHAP summary plot is obtained by projecting the SHAP dependence plot points onto the y-axis and recoloring the value's feature. The *TryStatement* and *VariableDeclaration* features were automatically selected for coloring based on a potential interaction in the model. Plot (b) shows that low SHAP values of the *ObjectExpression* feature influence the model's output more significantly for observations where the *VariableDeclaration* feature has high SHAP values.

Figure 4 shows the SHAP interaction values plot for the top two AST-JS features before (a) and after (b) feature selection.

It shows the main effects and interaction effects for the *ObjectExpression* feature. These effects capture all vertical dispersions. Plot (a) shows that high SHAP values for the *ObjectExpression* and *FunctionExpression* feature significantly influence the model's output. Plot (b) shows that low SHAP values for the *ObjectExpression* feature and high SHAP values for the *ExpressionStatement\_CallExpression\_Identifier* feature significantly influence the model's output.

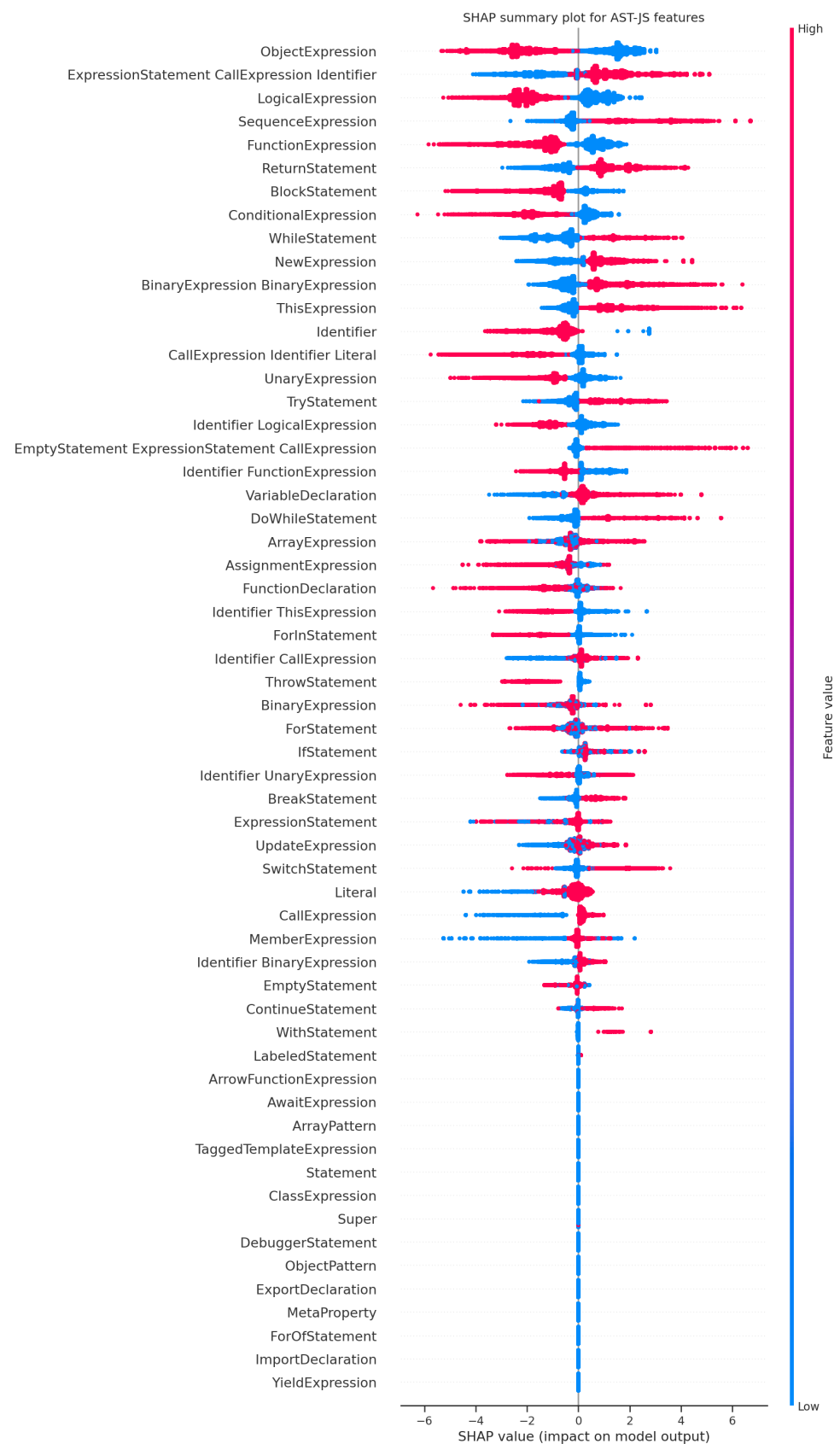
Table 3 presents precision, recall, and F1-scores obtained using features defined by association rule mining and 10-fold cross-validation.

By including these features, precision was improved by 0.0127, 0.0129, 0.0131, 0.0125, 0.0188, and 0.2352 for the XGBoost, LightGBM, RandomForest, DecisionTree, LogisticRegression, and GaussianNB, respectively. This improvement indicates a reduction in the number of misclassified benign JS codes. However, there was a reduction in the recall metric, which indicates an increase in misclassified malicious JS codes. Leveraging features from the malicious JS code samples and SHAP selected features based on their contributions to the model's output was pursued to improve the detection performance.

Table 4 presents precision, recall, and F1-scores obtained using features selected based on SHAP values and 10-fold cross-validation.

Compared to the association rule mining features, each model's detection performance was improved in all three evaluation metrics, with notable improvements in recall and F1-score. The best performing model, the XGBoost model, had error rates of 0.0011, 0.0168, and 0.0091 for recall, precision, and F1, respectively. The model outperformed the LightGBM and DecisionTree models by 0.03%, and the RandomForest model by 0.04% in the recall metric. A high recall rate translates to low misclassification and false-negative rates. AST-JS features selected using the SHAP values capture global and local feature importance. Therefore, these features enhance the machine learning model's feature learning process and lead to a better prediction model. As evidenced by the experiments, high performance was achieved by all the malicious JS code detection models, with tree ensemble methods yielding the best results. Consequently, these models had the lowest false positive and false negative rates.

Figure 5 shows the feature importance assigned to the AST-JS features by other feature selection methods: Boruta, ELI5, RandomForest, and SelectKBest [47,52].

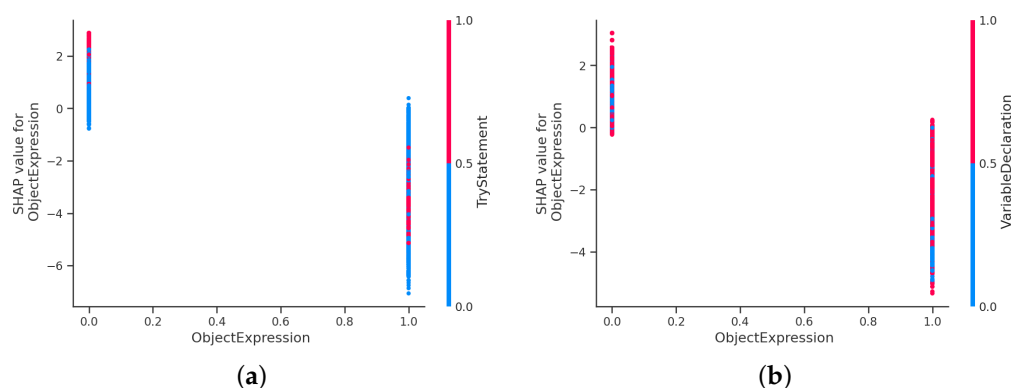


**Figure 2.** SHAP summary plot for AST-JS features.

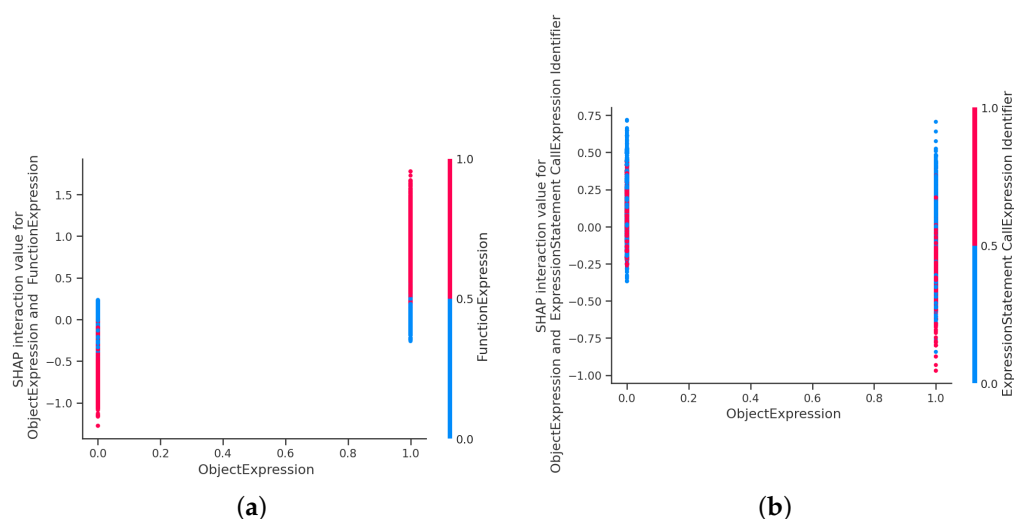
Each method resulted in different feature values for each AST-JS feature, and therefore, a different number of features was selected for each feature selection method. Boruta, a



wrapper around RandomForest, found all relevant features carrying information that can be used to predicting malicious JS. AST-JS features proven by a statistical test to be less relevant are rejected iteratively. ELI5 is used for explaining predictions. It is also referred to as permutation importance or Means Decrease Accuracy. The method measures how the score decreases when an AST-JS feature is eliminated. The RandomForest feature importance method measures each AST-JS feature's importance using the *entropy* function for the information gain. SelectKBest ranks AST-JS features by the k highest scores. This method measures the dependency between features using the mutual information score function.



**Figure 3.** SHAP dependence plot for the top AST-JS feature. (a) Before feature selection; (b) after feature selection.



**Figure 4.** SHAP interaction values plot for the top two AST-JS features. (a) Before feature selection; (b) after feature selection.

**Table 3.** Performance comparison using association rule mining AST-JS features.

Model	Recall	Precision	F1
XGBoost	0.9763 $\pm$ 0.0025	0.9825 $\pm$ 0.0024	0.9794 $\pm$ 0.0018
LightGBM	0.9762 $\pm$ 0.0023	0.9820 $\pm$ 0.0025	0.9791 $\pm$ 0.0018
RandomForest	0.9757 $\pm$ 0.0024	0.9833 $\pm$ 0.0025	0.9795 $\pm$ 0.0018
DecisionTree	0.9758 $\pm$ 0.0024	0.9812 $\pm$ 0.0025	0.9785 $\pm$ 0.0019
LogisticRegression	0.9600 $\pm$ 0.0033	0.9602 $\pm$ 0.0040	0.9601 $\pm$ 0.0025
KNeighbors	0.8516 $\pm$ 0.0044	0.9991 $\pm$ 0.0007	0.9195 $\pm$ 0.0026
GaussianNB	0.8221 $\pm$ 0.0048	0.7845 $\pm$ 0.0044	0.8029 $\pm$ 0.0038



**Table 4.** Performance comparison using SHAP selected AST-JS features.

Model	Recall	Precision	F1
XGBoost	0.9989 $\pm$ 0.0004	0.9832 $\pm$ 0.0024	0.9909 $\pm$ 0.0014
LightGBM	0.9986 $\pm$ 0.0007	0.9820 $\pm$ 0.0027	0.9902 $\pm$ 0.0015
RandomForest	0.9985 $\pm$ 0.0006	0.9840 $\pm$ 0.0024	0.9912 $\pm$ 0.0014
DecisionTree	0.9986 $\pm$ 0.0005	0.9815 $\pm$ 0.0024	0.9900 $\pm$ 0.0013
LogisticRegression	0.9895 $\pm$ 0.0013	0.9632 $\pm$ 0.0037	0.9762 $\pm$ 0.0017
KNeighbors	0.8779 $\pm$ 0.0056	0.9995 $\pm$ 0.0003	0.9347 $\pm$ 0.0031
GaussianNB	0.9373 $\pm$ 0.0056	0.6889 $\pm$ 0.0031	0.7941 $\pm$ 0.0030

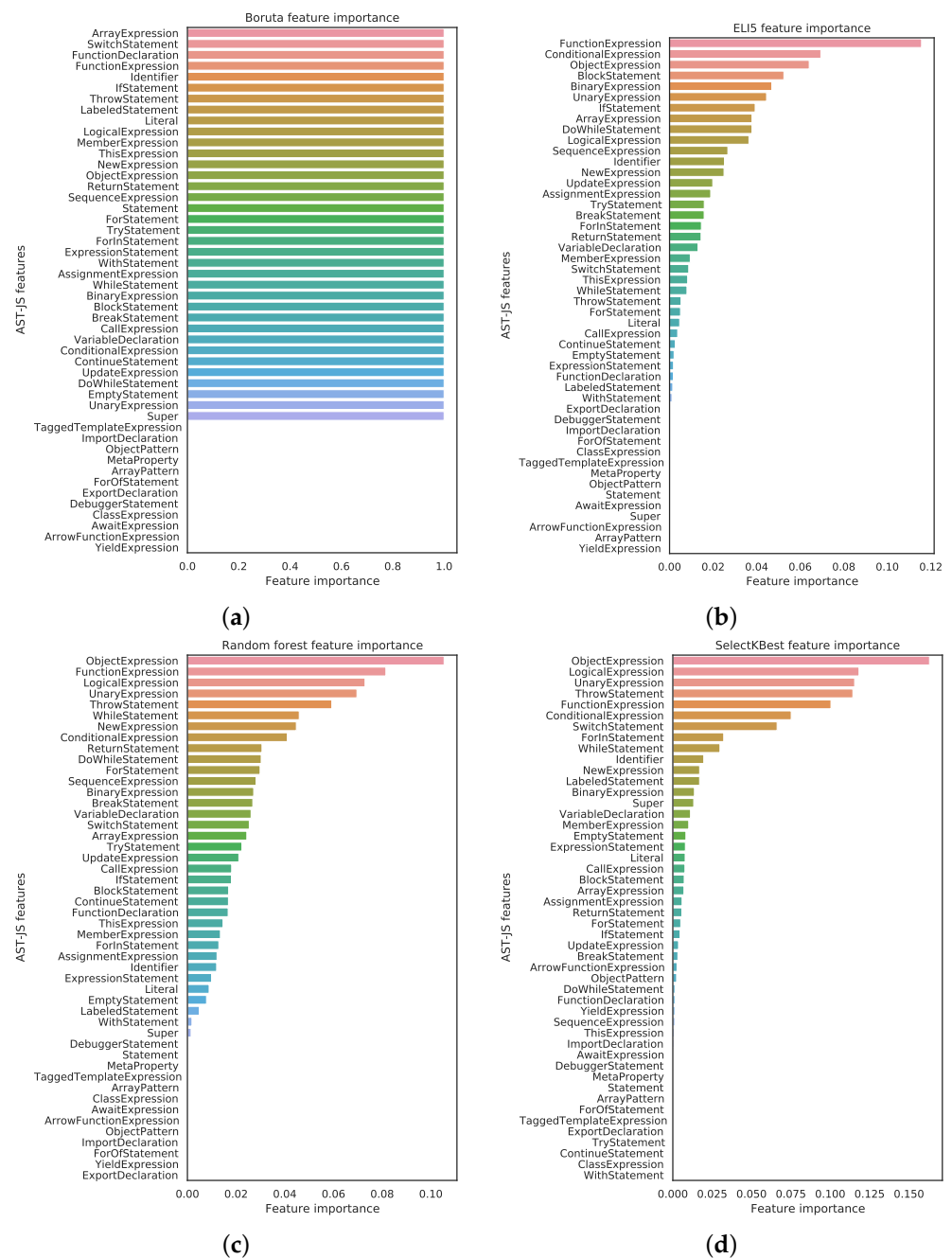
**Figure 5.** Feature importance assigned to AST-JS by other feature selection methods. (a) Boruta; (b) ELI5; (c) RandomForest; (d) SelectKBest.

Table 5 presents precision, recall, and F1-scores obtained using SHAP selected features compared to the other feature selection methods; Boruta, ELI5, RandomForest, and SelectKBest.

Our proposed detection model performed better than the other feature selection methods in all three evaluation metrics, with notable differences in precision and F1. The other feature selection methods have limitations because different model iterations assign different feature values to the AST-JS features. Additionally, Boruta assigns a *True* or *False* value to each AST-JS feature, as shown in Figure 5a. The permutation-based methods are computationally expensive and can have problems with highly-correlated AST-JS features, resulting in loss of important information. SHAP-selected AST-JS features have consistency in the feature values assigned to each feature. Unlike Boruta, SHAP values show the degree and manner of each AST-JS feature's contribution to the model prediction and are model-agnostic. These features also provide interaction graphs that are instrumental in getting information on AST-JS feature combinations, further boosting the model's detection performance.

**Table 5.** Performance comparison of SHAP and other feature selection methods.

Model	Recall	Precision	F1
SHAP	$0.9989 \pm 0.0004$	$0.9832 \pm 0.0024$	$0.9909 \pm 0.0014$
Boruta	$0.9983 \pm 0.0008$	$0.9698 \pm 0.0033$	$0.9839 \pm 0.0019$
ELI5	$0.9982 \pm 0.0008$	$0.9699 \pm 0.0033$	$0.9839 \pm 0.0019$
RandomForest	$0.9984 \pm 0.0007$	$0.9698 \pm 0.0032$	$0.9839 \pm 0.0018$
SelectKBest	$0.9982 \pm 0.0007$	$0.9687 \pm 0.0034$	$0.9832 \pm 0.0019$

Table 6 shows the training and detection times for the various classifiers on the JS code dataset using SHAP selected features. XGBoost yielded the highest detection performance and the third-fastest detection time, making it the best classifier for JS-based attack detection. KNeighbors yielded the lowest training time; however, it achieved the lowest recall rate, rendering it ineffective for this detection task. DecisionTree yielded the lowest detection time; however, XGBoost outperformed DecisionTree in all three evaluation metrics.

**Table 6.** Training and detection time.

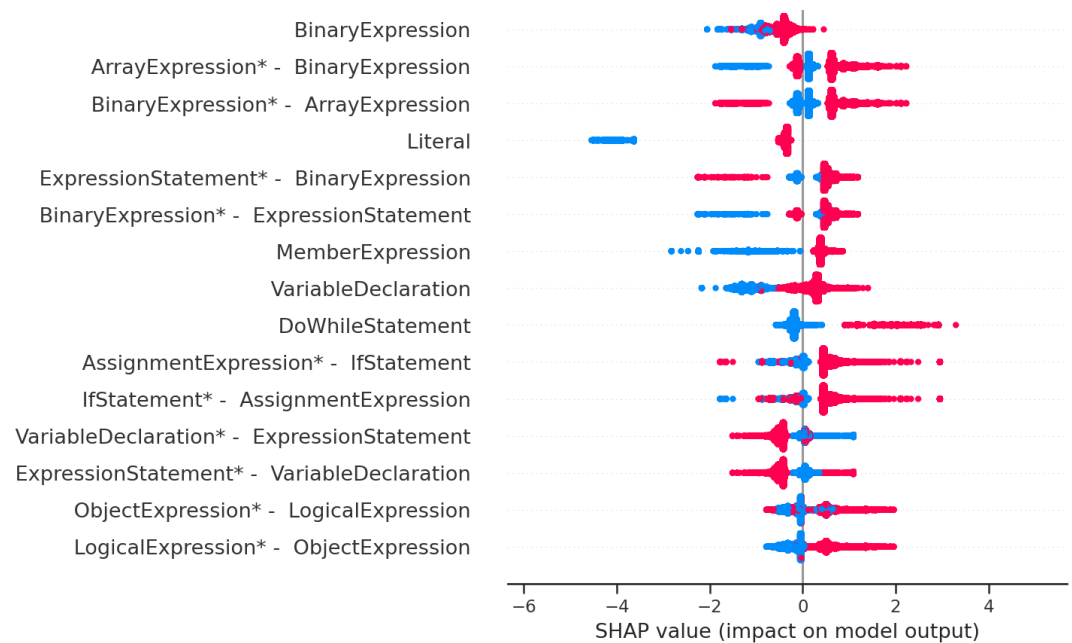
Model	Training Time (s)	Detection Time (s)
XGBoost	0.8685	$1.498 \times 10^{-6}$
LightGBM	0.5716	$2.114 \times 10^{-6}$
RandomForest	1.6350	$1.035 \times 10^{-5}$
DecisionTree	0.0895	$1.179 \times 10^{-6}$
LogisticRegression	0.6384	$1.871 \times 10^{-6}$
KNeighbors	0.0037	0.0015
GaussianNB	0.0425	$1.334 \times 10^{-6}$

## 5. Discussion

Blocklists, and heuristic- and signature-based systems are widely used to detect malicious URLs and JS codes. Current detection methods have various challenges stemming from fast-flux, cloaking, and zero-day attacks. Some services that can be used for copyright and privacy reasons in benign JS codes, such as obfuscation, can also create new or variant JS-based attacks. Therefore, it is biased to classify transformed URLs and JS codes as malicious without further detailed analysis. A system to analyze and accurately detect web-based attacks is needed.

As part of our research findings, feature selection using SHAP values resulted in a better prediction model. Other researchers limited their scope by studying specific malicious URLs and JS-based attacks, such as those containing phishing attacks, social engineering, malvertising attacks, and denial of service attacks, among others. For this,

they used manually engineered features unique to the type of attack they intended to identify. Such approaches have challenges in that they cannot be generalized to other attack types. Our premise is that features selected using SHAP values can overcome such challenges, as the analyst can easily identify which features significantly impact the overall model performance. Figure 6 shows SHAP interaction values with combinations of AST-JS features shown using *Feature\* - Feature* patterns.



**Figure 6.** SHAP interaction values. \* represents AST-JS feature combinations.

Using SHAP interaction values, it is easier to tell which AST-JS feature combinations or interactions contribute more toward the model's detection performance and in which direction. Some of these interactions are directly interpretable as attack types present in the dataset. Therefore, our proposed approach does not focus on specific categories of web-based attacks or threats.

## 6. Conclusions

Web-based attacks remain a significant challenge, as evasion techniques are continuously evolving. An equally adaptive strategy is required to detect such attacks effectively. This study proposes AST-JS feature selection using SHAP values and tree ensemble methods to detect these attacks. We also investigated how often AST-JS nodes appear together in benign and malicious JS codes using association rule mining. One expectation was that this approach would result in fully representative features that generalize well to other attacks. We used AST-JS nodes to represent the JS code structure and their SHAP values for feature selection. We experimented with features selected using AST-JS nodes, association rule mining, and SHAP values. We compared the performances of different machine learning classifiers in malicious JS code detection and achieved good detection performance with the tree ensemble methods. Additionally, we compared the performance of SHAP-selected features with the performances of those selected by other feature selection methods: Boruta, ELI5, RandomForest, and SelectKBest. The proposed web-based attack detection method outperformed the other feature selection methods in all three evaluation metrics. AST-JS nodes are resistant to perturbation in JS codes, and this ensures that the model is applicable in various JS code contexts, and therefore, to different JS-based attacks. We can extend this study to investigate malicious URL features in future work.

**Author Contributions:** Conceptualization, S.N., S.K. and S.O.; methodology, S.N., S.K., S.O., T.B. and T.T.; software, S.N.; validation, S.K., S.O., T.B., T.T. and D.I.; formal analysis, S.N.; investigation, S.N.; resources, S.O. and D.I.; data curation, S.N.; writing—original draft preparation, S.N.; writing—review and editing, S.N., S.K., S.O., T.B., T.T. and D.I.; visualization, S.N.; supervision, S.K., S.O., T.B., T.T. and D.I.; funding acquisition, D.I. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the Ministry of Education, Science, Sports, and Culture, Japan, grant number (B) 16H02874, and the Commissioned Research of the National Institute of Information and Communications Technology (NICT), Japan, grant number 190.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Acknowledgments:** This research was supported by the Ministry of Education, Science, Sports, and Culture, Japan, Grant-in-Aid for Scientific Research (B) 16H02874, and the Commissioned Research of the National Institute of Information and Communications Technology (NICT), Japan, grant number 190.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

- Masri, R.; Aldwairi, M. Automated malicious advertisement detection using virustotal, urlvoid, and trendmicro. In Proceedings of the 8th International Conference on Information and Communication Systems (ICICS'17), Irbid, Jordan, 4–6 April 2017; pp. 336–341. [\[CrossRef\]](#)
- Bilge, L.; Kirda, E.; Kruegel, C.; Balduzzi, M. EXPOSURE: Finding malicious domains using passive DNS analysis. In Proceedings of the 18th Annual Network and Distributed System Security Conference (NDSS'11), San Diego, CA, USA, 6–9 February 2011; pp. 1–17.
- Bilge, L.; Sen, S.; Balzarotti, D.; Kirda, E.; Kruegel, C. Exposure: A Passive DNS Analysis Service to Detect and Report Malicious Domains. *Assoc. Comput. Mach. Trans. Inf. Syst. Secur.* **2014**, *16*, 1–14. [\[CrossRef\]](#)
- Ghafir, I.; Prenosil, V. DNS traffic analysis for malicious domains detection. In Proceedings of the 2nd International Conference on Signal Processing and Integrated Networks (SPIN'15), Noida, India, 19–20 February 2015; pp. 613–918.
- Messabi, K.A.; Aldwairi, M.; Yousif, A.A.; Thoban, A.; Belqasmi, F. Malware detection using dns records and domain name features. In Proceedings of the 2nd International Conference on Future Networks and Distributed Systems (ICFNDS'18), New York, NY, USA, 26 June 2018; pp. 1–7. [\[CrossRef\]](#)
- LLC; OpenDNS. PhishTank: An Anti-Phishing Site. 2016. Available online: <https://www.phishtank.com> (accessed on 1 May 2020).
- Majestic SEO. The Majestic Million Service: The Million Domains We Find with the Most Referring Subnets. Available online: <https://majestic.com/reports/majestic-million> (accessed on 1 May 2020).
- Alexa Inc. The Top 500 Sites on the Web. Available online: <https://www.alexa.com/topsites> (accessed on 1 May 2020).
- myWOT. myWOT Web of Trust. Available online: <https://www.mywot.com/> (accessed on 1 May 2020).
- Sahoo, D.; Liu, C.; Hoi, S. Malicious URL detection using machine learning: A survey. *arXiv* **2019**, arXiv:1701.07179.
- Ferreira, M. Malicious URL detection using machine learning algorithms. In Proceedings of the Digital and Privacy Security Conference, Lusófona University of Porto, Porto, Portugal, 16 January 2019; pp. 114–122. [\[CrossRef\]](#)
- Ma, J.; Saul, L.; Savage, S.; Volker, G. Learning to Detect Malicious URLs. *ACM Trans. Intell. Syst. Technol.* **2011**, *2*, 1–24. [\[CrossRef\]](#)
- Ma, J.; Saul, L.; Savage, S.; Voelker, G. Beyond blacklists: Learning to detect malicious web sites from suspicious URLs. In Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD'09), Paris, France, 28 June–1 July 2009; pp. 1245–1254.
- Ma, J.; Saul, L.; Savage, S.; Voelker, G. Identifying suspicious URLs: An application of large-scale online learning. In Proceedings of the 26th Annual International Conference on Machine Learning (ICML'09), Montreal, QC, Canada, 14–18 June 2009; pp. 681–688.
- Ndichu, S.; Ozawa, S.; Misu, T.; Okada, K. A Machine Learning Approach to Malicious JavaScript Detection using Fixed Length Vector Representation. In Proceedings of the 2018 International Joint Conference on Neural Networks, (IJCNN'18), Rio de Janeiro, Brazil, 8–13 July 2018; pp. 1–8.
- Ndichu, S.; Kim, S.; Ozawa, S.; Misu, T.; Makishima, K. A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors. *Appl. Soft Comput. J.* **2019**, *84*, 1–11. [\[CrossRef\]](#)
- Ndichu, S.; Kim, S.; Ozawa, S. Deobfuscation, Unpacking, and Decoding of Obfuscated Malicious JavaScript for Machine Learning Models Detection Performance Improvement. *CAAI Trans. Intell. Technol.* **2020**, *5*, 184–192. [\[CrossRef\]](#)

18. Likarish, P.; Jung, E. A targeted web crawling for building malicious javascript collection. In Proceedings of the ACM First International Workshop on Data-Intensive Software Management and Mining (DSMM '09), New York, NY, USA, 6 November 2009; pp. 23–26. [\[CrossRef\]](#)
19. Chou, N.; Ledesma, R.; Teraguchi, Y.; Boneh, D.; Mitchell, J. Client-Side Defense against Web-Based Identity Theft. In Proceedings of The 11th Annual Network and Distributed System Security Symposium (NDSS '04), San Diego, CA, USA, 4–6 February 2004. Available online: <http://crypto.stanford.edu/SpoofGuard/webspooft.pdf> (accessed on 1 May 2020).
20. McGrath, D.; Gupta, M. Behind phishing: An examination of phisher modi operandi. In Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), San Francisco, CA, USA, 14 April 2008.
21. AlRoum, K.; Alolama, A.; Kamel, R.; Barachi, M.E.; Aldwairi, M. Detecting Malware Domains: A Cyber-Threat Alarm System. In Proceedings of the International Conference on Emerging Technologies for Developing Countries, Cotonou, Benin, 29–30 May 2018; Springer International Publishing: Cham, Switzerland, 2018; pp. 181–191.
22. NWang, W.; Shirley, K. Breaking bad: Detecting malicious domains using word segmentation. In Proceedings of the 9th IEEE Workshop on Web 2.0 Security and Privacy (W2SP'15), San Jose, CA, USA, 21 May 2015. Available online: <http://arxiv.org/abs/1506.04111> (accessed on 1 October 2021).
23. Kuyama, M.; Kakizaki, Y.; Sasaki, R. Method for Detecting a Malicious Domain by using WHOIS and DNS features. In Proceedings of the Third International Conference on Digital Security and Forensics (DigitalSec'16), Kuala Lumpur, Malaysia, 6–8 September 2016; pp. 74–80.
24. Kuyama, M.; Kakizaki, Y.; Sasaki, R. Method for detecting a malicious domain by using only well known information. *Int. J. Cyber-Secur. Digit. Forensics* **2016**, *5*, 166–174. [\[CrossRef\]](#)
25. Marchal, S.; Francois, J.; State, R.; Engel, T. Phishstorm: Detecting phishing with streaming analytics. *IEEE Trans. Netw. Serv. Manag.* **2014**, *11*, 458–471. [\[CrossRef\]](#)
26. Feroz, M.; Mengel, S. Phishing URL detection using URL ranking. In Proceedings of the IEEE International Congress on Big Data, BigData Congress, New York, NY, USA, 27 June–2 July 2015; pp. 635–638.
27. Moghimi, M.; Varjani, A. New rule-based phishing detection method. *Expert Syst. Appl.* **2016**, *53*, 231–242. [\[CrossRef\]](#)
28. Yuan, H.; Chen, X.; Li, Y.; Yang, Z.; Liu, W. Detecting Phishing Websites and Targets Based on URLs and Web page Links. In Proceedings of the 24th International Conference on Pattern Recognition (ICPR'18), Beijing, China, 20–24 August 2018; pp. 3669–3674. [\[CrossRef\]](#)
29. Anand, A.; Gorde, K.; AntonyMoniz, J.; Park, N.; Chakraborty, T.; Chu, B. Phishing URL detection with oversampling based on text generative adversarial networks. In Proceedings of the IEEE international conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; pp. 1168–1177. [\[CrossRef\]](#)
30. Jain, A.; Gupta, B. Towards detection of phishing websites on client-side using machine learning based approach. *Telecommun. Syst.* **2018**, *68*, 687–700. [\[CrossRef\]](#)
31. Ford, S.; Cova, M.; Kruegel, C.; Vigna, G. Analyzing and detecting malicious flash advertisements. In Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09), IEEE Computer Society, Honolulu, HI, USA, 6–10 December 2009; pp. 363–372.
32. Li, Z.; Zhang, K.; Xie, Y.; Yu, F.; Wang, X. Knowing your enemy: Understanding and detecting malicious web advertising. In Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS '12), New York, NY, USA, 16–18 October 2012; pp. 674–686.
33. Oentaryo, R.; Lim, E.P.; Finegold, M.; Lo, D.; Zhu, F.; Phua, C.; Cheu, E.Y.; Yap, G.E.; Sim, K.; Nguyen, M.; et al. Detecting click fraud in online advertising: a data mining approach. *J. Mach. Learn. Res.* **2014**, *15*, 99–140.
34. Xu, H.; Liu, D.; Koehl, A.; Wang, H.; Stavrou, A. Click fraud detection on the advertiser side. In Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS), Wroclaw, Poland, 7–11 September 2014; pp. 419–438. [\[CrossRef\]](#)
35. Kapravelos, A.Z.A.; Stringhini, G.; Holz, T.; Kruegel, C.; Vigna, G. The dark alleys of madison avenue: Understanding malicious advertisements. In Proceedings of the 2014 ACM Conference on Internet Measurement Conference (IMC'14), Vancouver, BC, Canada, 5–7 November 2014; pp. 373–380.
36. Akiyama, M.; Yagi, T.; Yada, T.; Mori, T.; Kadobayashi, Y. Analyzing the ecosystem of malicious URL redirection through longitudinal observation from honeypots. *Comput. Secur.* **2017**, *69*, 155–173. [\[CrossRef\]](#)
37. VirusTotal. Analyze Suspicious Files and URLs to Detect Types of Malware, Automatically Share Them with the Security Community. Available online: <https://www.virustotal.com/gui/home/url> (accessed on 1 May 2020).
38. URLVoid. Website Reputation Checker, This Service Helps You Detect Potentially Malicious Websites. Available online: <https://www.urlvoid.com/> (accessed on 1 May 2020).
39. TrendMicro. Site Safety Center, with One of the Largest Domain-Reputation Databases in the World, Trend Micro's Web Reputation Technology Is a Key Component of Trend Micro™ Smart Protection Network™. Available online: <https://global.sitesafety.trendmicro.com/> (accessed on 1 May 2020).
40. Canali, D.; Cova, M.; Vigna, G.; Kruegel, C. Prophiler: A fast filter for the large-scale detection of malicious web pages. In Proceedings of the 20th International Conference on World Wide Web (WWW'11), ACM, Hyderabad, India, 28 March–1 April 2011; pp. 197–206.
41. Hongtao, L.; Sergio, M.; Mahdi, C. *Amj: An Analyzer for Malicious Javascript*; Imperial College London, Department of Computing: London, UK, 2018.



42. Han, J.; Pei, J.; Yin, Y.; Mao, R. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Min. Knowl. Discov.* **2004**, *8*, 53–87. [[CrossRef](#)]
43. Raschka, S. MLxtend: Providing machine learning and data science utilities and extensions to Python’s scientific computing stack. *J. Open Source Softw.* **2018**, *3*, 638. [[CrossRef](#)]
44. Lundberg, S.M.; Erion, G.; Chen, H.; DeGrave, A.; Prutkin, J.M.; Nair, B.; Katz, R.; Himmelfarb, J.; Bansal, N.; Lee, S.I. From local explanations to global understanding with explainable AI for trees. *Nat. Mach. Intell.* **2020**, *2*, 2522–5839. [[CrossRef](#)] [[PubMed](#)]
45. Lundberg, S.M.; Nair, B.; Vavilala, M.S.; Horibe, M.; Eisses, M.J.; Adams, T.; Liston, D.E.; Low, D.K.W.; Newman, S.F.; Kim, J.; et al. Explainable machine-learning predictions for the prevention of hypoxaemia during surgery. *Nat. Biomed. Eng.* **2018**, *2*, 749. [[CrossRef](#)] [[PubMed](#)]
46. Lundberg, S.M.; Erion, G.G.; Lee, S. Consistent Individualized Feature Attribution for Tree Ensembles. *arXiv* **2018**, arXiv:1802.03888.
47. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
48. Buitinck, L.; Louppe, G.; Blondel, M.; Pedregosa, F.; Mueller, A.; Grisel, O.; Niculae, V.; Prettenhofer, P.; Gramfort, A.; Grobler, J.; et al. API design for machine learning software: Experiences from the scikit-learn project. *arXiv* **2013**, arXiv:1309.0238.
49. Ke, G.; Meng, Q.; Finley, T.; Wang, T.; Chen, W.; Ma, W.; Ye, Q.; Liu, T.Y. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In Proceedings of the 31st International Conference on Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; pp. 3149–3157.
50. Chen, T.; Guestrin, C. XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016. [[CrossRef](#)]
51. Petrak, H. JavaScript Malware Collection—A Collection of Almost 40.000 JavaScript Malware Samples. Available online: <https://github.com/HynekPetrak/javascript-malwarecollection> (accessed on 1 October 2020).
52. Kursa, M.B.; Rudnicki, W.R. Feature Selection with the Boruta Package. *J. Stat. Softw.* **2010**, *36*, 1–13. [[CrossRef](#)]